LBNL-40319
UCB//CSD-97-945

Measurements and Analysis of End-to-End Internet Dynamics

Vern Paxson
Ph.D. Thesis

Computer Science Division
University of California, Berkeley

and

Information and Computing Sciences Division
Lawrence Berkeley National Laboratory
University of California
Berkeley, CA 94720

April, 1997

This work was supported by the Director, Office of Energy Research, Scientific Computing Staff, of the United States Department of Energy under Contract No. DE-AC03-76SF00098.

Measurements and Analysis of End-to-End Internet Dynamics

by

Vern Edward Paxson

B.S. (Stanford University) 1985
M.S. (University of California, Berkeley) 1991

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Prof. Domenico Ferrari, Chair
Prof. Michael Luby
Prof. John Rice

1997

Measurements and Analysis of End-to-End Internet Dynamics

Abstract

Measurements and Analysis of End-to-End Internet Dynamics

by

Vern Edward Paxson
Doctor of Philosophy in Computer Science

University of California at Berkeley

Prof. Domenico Ferrari, Chair

Accurately characterizing end-to-end Internet dynamics—the performance that a user actually obtains from the lengthy series of network links that comprise a path through the Internet—is exceptionally difficult, due to the network's immense heterogeneity. It can be impossible to gauge the generality of findings based on measurements of a handful of paths, yet logistically it has proven very difficult to obtain end-to-end measurements on larger scales.

At the heart of our work is a "measurement framework" we devised in which a number of sites around the Internet host a specialized measurement service. By coordinating "probes" between pairs of these sites we can measure end-to-end behavior along $O(N^2)$ paths for a framework consisting of N sites. Consequently, we obtain a superlinear scaling that allows us to measure a rich cross-section of Internet behavior without requiring huge numbers of observation points. 37 sites participated in our study, allowing us to measure more than 1,000 distinct Internet paths.

The first part of our work looks at the behavior of end-to-end routing: the series of routers over which a connection's packets travel. Based on 40,000 measurements made using our framework, we analyze: routing "pathologies" such as loops, outages, and flutter; the stability of routes over time; and the symmetry of routing along the two directions of an end-to-end path. We find that pathologies increased significantly over the course of 1995, indicating that, by one metric, routing *degraded* over the year; that Internet paths are heavily dominated by a single route, but that routing lifetimes range from seconds to many days, with most lasting for days; and that, at the end of 1995, about half of all Internet paths included a major routing asymmetry.

The second part of our work studies end-to-end Internet packet dynamics. We analyze 20,000 TCP transfers of 100 Kbyte each to investigate the performance of both the TCP endpoints and the Internet paths. The measurements used for this part of our study are much richer than those for the first part, but require a great degree of attention to issues of *calibration*, which we address by applying *self-consistency checks* to the measurements whenever possible. We find that packet filters are capable of a wide range of measurement errors, some of which, if undetected, can significantly taint subsequent analysis. We further find that network clocks exhibit adjustments and skews relative to other clocks frequently enough that a failure to detect and remove these effects will likewise pollute subsequent packet timing analysis.

Using TCP transfers for our network path "measurement probes" gains a number of advantages, the chief of which is the ability to probe fine time scales without unduly loading the network. However, using TCP also requires us to accurately distinguish between connection dy-

namics due to the behavior of the TCP endpoints, and dynamics due to the behavior of the network path between them. To address this problem, we develop an analysis program, `tcpanaly`, that has coded into it knowledge of how the different TCP implementations in our study function. In the process of developing `tcpanaly`, we thus in tandem develop detailed descriptions of the performance and congestion-avoidance behavior of the different implementations. We find that some of the implementations suffer from gross problems, the most serious of which would devastate overall Internet performance, were the implementations ubiquitously deployed.

With the measurements calibrated and the TCP behavior understood, we then can turn to analyzing the dynamics of Internet paths. We first need to determine a path's *bottleneck bandwidth*, meaning the fastest transfer rate the path can sustain. Knowing the bottleneck bandwidth then lets us determine which packets a sender transmits must necessarily *queue* behind their predecessors, due to the load the sender imposes on the path. This in turn allows us to determine which of our probes are perforce *correlated*. We identify several problems with the existing bottleneck estimation technique, "packet pair," and devise a robust estimation algorithm, PBM ("packet bunch modes"), that addresses these difficulties. We calibrate PBM by gauging the degree to which the bottleneck rates it estimates accord with known link speeds, and find good agreement. We then characterize the scope of Internet path bottleneck rates, finding wide variation, not infrequent asymmetries, but considerable stability over time.

We next turn to an analysis of packet loss along Internet paths. To do so, we distinguish between losses of "loaded" data packets, meaning those which necessarily queued behind a predecessor at the bottleneck; "unloaded" data packets, which did not do so; and the small "acknowledgement" packets returned to a TCP sender by the TCP receiver. We find that network paths are well characterized by two general states, "quiescent," in which no loss occurs, and "busy," in which one or more packets of a connection are lost. The prevalence of quiescent connections remained about 50% in both our datasets, but for busy connections, packet loss rates increased significantly over the course of 1995. We further find that loss rates vary dramatically between different regions of the network, with European and especially trans-Atlantic connections faring much worse than those confined to the United States.

We also characterize: loss symmetry, finding that loss rates along the two directions of an Internet path are nearly uncorrelated; loss "outages," finding that outage durations exhibit clear Pareto distributions, indicating they span a large range of time scales; the degree to which a connection's loss patterns predict those of future connections, finding that observing quiescence is a good predictor of observing quiescence in the future, and likewise for observing a busy network path, but that the proportion of lost packets does not well predict the future proportion; and the efficacy of TCP implementations in dealing with loss efficiently, by retransmitting only when necessary. We find that most TCPs retransmit fairly efficiently, and that deploying the proposed "selective acknowledgement" option would eliminate almost all of their remaining unnecessary retransmissions. However, some TCPs incorrectly determine how long to wait before retransmitting, and these can suffer large numbers of unnecessary retransmissions.

We finish our study with a look at variations in packet transit delays. We find great "peak-to-peak" variation, meaning that maximum delays far exceed minimum delays. Delay variations along the two directions of an Internet path are only lightly correlated, but correlate well with loss rates observed in the same direction along the path. We identify three types of "timing compression," in which packets arrive at their receiver spaced more closely together than when originally transmitted. The prevalence of none of the three is such as to significantly perturb network performance, but all three occur frequently enough to require judicious filtering by network measurement procedures to avoid deriving false timing conclusions.

We then look at the question of the time scales on which most of a path's queueing variations occur. We find that, overall, most variation occurs on time scales of 100–1000 msec, which means that transport connections might effectively adapt their transmission to the variations, but only if they act quickly. However, as with many Internet path properties, we find wide ranges of behavior, with not insignificant queueing variations occurring on time scales as small as 10 msec and as large as one minute.

The last aspect of packet delay variations we investigate is the degree to which it reflects an Internet path's *available bandwidth*. We show that the ratio between the delay variations packets incur due to their connection's own loading of the network, versus the total delay variations incurred, correlates well with the connection's overall throughput. We further find that Internet paths exhibit wide variation in available bandwidth, ranging from very little available to virtually all. The degree of available bandwidth diminished markedly over the course of 1995, but, as for packet loss rates, we also find sharp geographic differences, so the overall trend cannot be summarized in completely simple terms. Finally, we investigate the degree to which the available bandwidth observed by a connection accurately predicts that observed by future connections, finding that the predictive power is fairly good for time scales of minutes to hours, but diminishes significantly for longer time periods.

We argue that our work supports several general themes:

- The N^2 scaling property of our measurement framework serves to measure a sufficiently diverse set of Internet paths that we might plausibly interpret the resulting analysis as accurately reflecting general Internet behavior.

- To cope with such large-scaled measurements requires attention to calibration using self-consistency checks; robust statistics to avoid skewing by outliers; and automated "micro-analysis," such as that performed by `tcpanaly`, that we might see the forest as well as the trees.

- With due diligence to remove packet filter errors and TCP effects, TCP-based measurement provides a viable means for assessing end-to-end packet dynamics.

- We find wide ranges of behavior, so we must exercise great caution in regarding any aspect of packet dynamics as "typical."

- Some common assumptions such as in-order packet delivery, FIFO bottleneck queueing, independent loss events, single congestion time scales, and path symmetries are all sometimes violated.

- The combination of path asymmetries and reverse-path noise render sender-only measurement techniques markedly inferior to those that include receiver-cooperation.

Finally, we believe an important aspect of this work is how it might contribute towards developing a "measurement infrastructure" for the Internet: one that proves ubiquitous, informative, and sound.

To Lindsay —

For making it both possible
and worthwhile

— *with all my love*

Contents

List of Figures

List of Tables

Acknowledgements

This work has its roots in the teaching, help, patience, and inspiration of a great number of people, to whom I wish to express my heartfelt gratitude.

Simply put, Van Jacobson is the reason I have studied networking; the reason I embarked on this study; and the reason I had faith that the work would, with sufficient diligence, yield a host of new insights. I am delighted that, having known him for nearly twenty years, I still find he has much to teach me.

Likewise, this work drew inspiration and invaluable support from Domenico Ferrari. The energy and respect that he affords to both his students' efforts, and to his students themselves, has made it a privilege to be advised by him.

I have also been delighted to have Sally Floyd as my mentor, colleague, and friend. She has listened to countless half-baked ideas of how to analyze and interpret various measurements, and has always patiently separated the promising from the harebrained. This calibration of ideas, and her suggestions on how to then pursue the more promising ones, has proved invaluable for fostering my sense of how to conduct sound research.

A number of others played major roles in shaping this work. I would particularly like to thank John Rice and Mike Luby for their industrious efforts in serving on my dissertation committee, which led to the work being much more solid than it would otherwise have been.[1]

My heartfelt thanks to Greg Minshall, for his detailed, insightful comments on nearly every page of the work (and for his willingness to burn an entire Friday evening discussing some of them); and to Amit Gupta, John Hawkinson, Kurt Lidl, Craig Partridge, and anonymous SIGCOMM and *IEEE/ACM Transactions on Networking* referees, all of whom contributed very helpful comments on earlier versions of the work.

I would like to also thank my colleagues at the Network Research Group: Kevin Fall, Craig Leres, and Steve McCanne, for their much appreciated ideas, support, and feedback.

Special thanks to Kathryn Crabtree, for her untiring help in surmounting innumerable administrative hurdles along the dissertation trail. She is an invaluable asset to UCB computer science.

This work would not have been possible without the efforts of the many volunteers who installed the Network Probe Daemon at their sites. In the process they endured debugging headaches, `inetd` crashes, software updates, and a seemingly endless stream of queries from me regarding their site's behavior. I am indebted to:

Guy Almes and Bob Camm (`adv`);
Jos Alsters (`unij`);
Jean-Chrysostome Bolot (`inria`);
Hans-Werner Braun, Kim Claffy, and Bilal Chinoy (`sdsc`);
Randy Bush (`rain`);
Jon Crowcroft and Atanu Ghosh (`ucl`);
Peter Danzig and Katia Obraczka (`usc`);
Mark Eliot (`sri`);
Robert Elz (`austr`);

[1] Particular thanks to Mike for throwing down the glove, and for knowing which glove to use.

Teus Hagen (oce);
Steinar Haug and Håvard Eidnes (sintef1, sintef2);
John Hawkinson (near and panix);
TR Hein (xor);
Tobias Helbig and Werner Sinze (ustutt);
Paul Hyder (ncar);
Alden Jackson (sandia);
Kate Lance (austr2);
Craig Leres (lbl);
Kurt Lidl (pubnix);
Peter Linington, Alan Ibbetson, Peter Collinson, and Ian Penny (ukc);
Steve McCanne (lbli);
John Milburn (korea);
Walter Mueller (umann);
Evi Nemeth, Mike Schwartz, Dirk Grunwald, Lynda McGinley (ucol, batman);
Francóis Pinard (umont);
Jeff Polk and Keith Bostic (bsdi);
Todd Satogata (bnl);
Doug Schmidt and Miranda Flory (wustl);
Sorell Slaymaker and Alan Hannan (mid);
Don Wells and Dave Brown (nrao);
Gary Wright (connix);
John Wroclawski (mit);
Cliff Young and Brad Karp (harv); and
Lixia Zhang, Mario Gerla, and Simon Walton (ucla).

I am likewise indebted to Keith Bostic, Evi Nemeth, Rich Stevens, George Varghese, Andres Albanese, Wieland Holfelder, and Bernd Lamparter for their invaluable help in recruiting NPD sites. Thanks, too, to Peter Danzig, Jeff Mogul, and Mike Schwartz for feedback on the design of NPD.

This work also benefited from discussions with Guy Almes, Tom Anderson, Robert Elz, Teus Hagen, John Krawczyk, Kate Lance, Dun Liu, Paul Love, Jamshid Mahdavi, Matt Mathis, Dave Mills, Pravin Varaiya, Curtis Villamizar, and Walter Willinger.

A preliminary analysis of the $\mathcal{R}_1$ routing dataset was done by Mark Stemm and Ketan Patel.

Often to understand the behavior of particular routers or to determine their location, I asked personnel from the organization responsible for the routers. I was delighted at how willing they were to help, and in this regard would like to acknowledge:

Vadim Antonov, Tony Bates, Michael Behringer, Per Gregers Bilse, Bjorn Carlsson, Peggy Cheng, Guy Davies, Sean Doran, Bjorn Eriksen, Amit Gupta, Tony Hain, John Hawkinson again!, Susan Harris, Ittai Hershman, Kevin Hoadley, Scott Huddle, James Jokl, Kristi Keith, Harald Koch, Craig Labovitz, Tony Li, Martijn Lindgreen, Ted Lindgreen, Dan Long, Bill Manning, Milo Medin, Keith Mitchell, Roderik Muit, Chris Myers, Torben Nielsen, Richard Nuttall, Mark Oros, Michael Ramsey, Juergen Rauschenbach, Douglas Ray, Brian Renaud, Jyrki Soini, Nigel Titley, Paul Vixie, and Rusty Zickefoose.

Finally, this work would never have been realized without the ongoing support provided by the Lawrence Berkeley National Laboratory. I am deeply grateful. In particular, I would like to thank Stu Loken and Ed Theil for their efforts and encouragement.

This work was supported by the Director, Office of Energy Research, Scientific Computing Staff, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

Chapter 1

Introduction

As the Internet grows larger, measuring and characterizing its dynamics grows harder. Part of the difficulty is how quickly the network changes. Depending on the figure of interest, the network between 80% and 100% each year, and has sustained this growth for well over a decade. Furthermore, the dominant protocols and their patterns of use can change radically over just a few years, or even a few months [Pa94b, CBP94].

Another difficulty, though, is the network's incredible—and increasing—heterogeneity. It is more and more difficult to measure a plausibly representative cross-section of its behavior. It is this latter concern that we attempt to address in this work. In this chapter, we develop the context for the rest of the study, by discussing different types of traffic studies, and how research efforts of those types have addressed heterogeneity problems. Our study falls in the category with perhaps the greatest heterogeneity difficulties, that of the "end-to-end" performance of entire paths through the network.

Our work has two distinct parts: a study of end-to-end routing behavior in the Internet (Part I), and a study of end-to-end Internet packet dynamics (Part II). These two are united by the common measurement framework used to gather the data analyzed in each part (described in Chapter 4). In addition, some of the results used in each part are incorporated into analysis in the other part. However, in many ways the two parts are distinct and self-contained. A reader particularly interested in one or the other topic might profitably just read the relevant part. Consequently, we defer an overview of each part to later chapters (Chapter 2 of Part I, and Chapter 9 of Part II). We summarize both parts, and what we perceive as the themes of the work, in Chapter 17, at the end of Part II. For the remainder of this introduction, we give an overview of the general problem of measuring large networks,

We can classify measurement studies into several basic types. Each faces the problem of heterogeneity to varying degrees, as follows.

Exhaustive studies analyze properties of a significant fraction of the entire network. Examples are Kleinrock's study of the ARPANET's behavior on time scales of hours to days [Kl76]; the series of "ping" experiments conducted by Mills to evaluate the effectiveness of the TCP retransmission-timeout algorithm [Mi83]; Claffy et al.'s study characterizing traffic on the T1 NSFNET backbone [CPB93b]; and Chinoy's study of the dynamics of routing information within the NSFNET backbone [Ch93]. While these studies can convincingly characterize the full range of behavior one might expect to observe from the network, they become impractical as the network grows in size.[1]

Site studies characterize the aggregate traffic patterns observed for entire sites. They focus on the connections sizes, durations, and interarrival times. An early site study, by Danzig and colleagues, identified large heterogeneities in the traffic "mix" at each site, meaning that the proportion of total traffic (total connections, total packets, or total bytes) due to different applications varies greatly [DJCME92]. Our subsequent work extended this finding to the characteristics of the connections made by each type of application. We found that the distributions of a particular application's connection sizes and durations varied greatly from site to site [Pa94a], in agreement with much earlier findings, in a different communications context, by Fuchs and Jackson [FJ70].

Another type of study focuses on *server* behavior, for services that are distributed over the Internet. The heterogeneity issues faced by these studies vary greatly, depending on the service. For example, Danzig and colleagues analyzed requests arriving at a "root" name server, finding a variety of performance problems [DOK92]. However, there are only a handful of root name servers. Because clients divide their requests between them, studying a single server yields results plausibly representative for all of the servers. On the other hand, a recent study of World Wide Web servers had to grapple with the issue of differences among the various servers studied [AW96], and did so by developing its central theme around the search for behavioral "invariants" among the six Web servers analyzed.

Related to server studies are *client* studies, which analyze the different ways in which clients access servers. From a heterogeneity standpoint, client studies are more difficult than server studies, since usually there are many more clients than servers. One approach is to study the behavior of all clients located at a particular site [CB96]. Doing so, however, incurs problems similar to those of site studies: it is difficult to gauge the generality of the findings. These problems, however, can be tempered based on the nature of the service. For example, we might expect request from one site's Web clients to more closely resemble those of another site's Web clients, than for the site's aggregate traffic to resemble that of another site.

Another type of study analyzes the aggregate traffic seen on network *links*. These studies have focussed on the dynamics of packet arrivals on the link [FL91, LTWW94, PF95, WTSW95], the characteristics of packet "flows" [JR86, He90, CBP95], or on traffic patterns over particularly singular links, such as the trans-Atlantic link connecting the U.S. and the U.K. [CW91, WLC92].

For link studies of *local area* networks [JR86, FL91, LTWW94, WTSW95], heterogeneity presents less of a problem than for those of *wide area* networks, because the latter encompass a much broader range of traffic sources and path characteristics than the former. Some wide area link studies attempt to address heterogeneity issues by analyzing traces from multiple links. However, gathering link traces is difficult (and becoming more so due to security, privacy, and business concerns), and such studies have not to date analyzed more than two dozen or so traces.[2]

Our work falls in still another class, that of *end-to-end* studies. These studies concern

[1] At the time of the later of the first two studies, the Internet comprised about 600 hosts. As of this writing, it comprises about 16 million hosts [Lo97]. At the time of the Claffy study, the backbone consisted of 15 nodes and two dozen links. Today, it is much larger, though sources of accurate statistics on its size have virtually disappeared with the commercialization of the Internet infrastructure.

[2] Most site studies are conducted as link studies, too, since an extremely convenient way to capture an entire site's Internet traffic is to monitor what is usually a single link connecting the site to the rest of the Internet. Some server studies can also be conducted in the context of a link study, by analyzing all of the server requests seen on a highly aggregated link [EHS92, DHS93].

how the network performs from the perspective of an end user. To users, a network like the Internet is simply a black box that somehow forwards packets between their host and hosts with which they wish to communicate. End-to-end studies face extreme heterogeneity problems because they strive to characterize the behavior of *paths* through the network. Not only does the Internet contain millions of distinct paths, but the dynamics of each path reflects the concatenation of the dynamics of each forwarding element along the path, and hence can be highly complex.

The few studies to date of end-to-end packet dynamics—Mogul's look at TCP dynamics such as ack compression [Mo92], Bolot's analysis of patterns of packet loss and delay [Bo93], and Claffy et al.'s characterization of one-way latencies [CPB93a]—have all been confined to measuring a handful of Internet paths, because of the great logistical difficulties, *and also analysis difficulties*, presented by large-scale measurement.[3] Consequently, it is hard to gauge how representative these end-to-end findings are for today's Internet.

As a result, even basic Internet path questions such as "how often do routes change?" and "how often are packets dropped?" remain unanswered in any sort of general way. It is towards answering these questions that we now embark.

[3]Mogul's study was actually conducted as a link study. Doing so let him observe behavior from a fairly large number of Internet paths, albeit ones that all had the single link in common. A drawback of this approach, however, is that it is difficult to infer from the perspective of a link the full end-to-end behavior as perceived by the endpoints, an issue we discuss further in § 10.4.

Part I

End-to-End Routing Behavior in the Internet

Chapter 2

Overview of the Routing Study

The large-scale behavior of routing in the Internet has gone virtually without any formal study, the exception being Chinoy's analysis of the dynamics of Internet routing information [Ch93]. In this part of our thesis we analyze 40,000 route measurements conducted using repeated "traceroutes" between 37 Internet sites. The main questions we strive to answer are:

1. What sort of pathologies and failures occur in Internet routing?
2. Do routes remain stable over time or change frequently?
3. Do routes from A to B tend to be symmetric (the same in reverse) as routes from B to A?

Our framework for answering these questions is the measurement of a large sample of Internet routes between a number of geographically diverse hosts. We argue that the set of routes is representative of Internet routes in general, and analyze how the routes changed over time to assess how Internet routing in general changes over time.

We begin by giving an overview of the routing literature in general and, more specifically, how routing works in the Internet (Chapter 3). We find that while routing *protocols* (mechanisms) have been heavily studied, the literature offers very few *measurement* studies of how routing *behaves* in practice.

We then discuss our experimental methodology (Chapter 4). This includes our measurement apparatus, which is the `npd` "network probe daemon" and the `traceroute` utility for measuring Internet paths; the use of exponential sampling, which allows us to apply the *PASTA Principle* [Wo82] as the basis for the generalizations we derive from our measurements; and the use of the *Fisher's exact test* [Ri95] to test for significant differences between different sets of observations. We also discuss which aspects of our measurements are plausibly representative of Internet routing behavior in general (namely, aggregate observations of Internet paths), and which are not (those depending on the behavior of individual sites).

In Chapter 5 we give an overview of the 37 sites participating in the study, and details of the raw data and of the failures encountered when attempting to capture it. We also discuss how we assigned geographic locations to all of the 1,531 routers appearing in the paths we measured.

We performed two separate sets of measurements. The first, $\mathcal{R}_1$, consisted of 6,991 attempted measurements of 689 different paths through the Internet (i.e., distinct source/destination pairs). The $\mathcal{R}_1$ measurements were made with an average interval of 1-2 days between samples.

Upon analyzing the $\mathcal{R}_1$ data, we realized that we could not accurately answer crucial questions regarding routing stability without higher frequency sampling, nor could we unambiguously assess routing symmetry without simultaneous measurements of both directions of an Internet path. To resolve these difficulties, we conducted a second set of measurements, $\mathcal{R}_2$, consisting of 37,097 attempted measurements of 1,056 Internet paths. These measurements were made in two groups, one with an average interval of about 2.75 days between samples, and the other with an average measurement interval of 2 hours. The latter suffices for accurately assessing routing stability. We also *paired* the bulk (80%) of the measurements, conducting back-to-back measurements of the different directions of each Internet path. Pairing allows us to accurately assess routing asymmetries, and also to reduce a source of measurement error (§ 5.2).

Before analyzing the data for routing stability and symmetry, we needed to categorize any anomalies present in order to prevent them from skewing the analysis. In Chapter 6 we classify a number of routing pathologies:

- unresponsive routers, routing loops, routing changes in the middle of measurement, erroneous routes, omission of TTL decrement, and infrastructure failures, all of which were rare;
- host and stub network outages, which were fairly common (but for which our samples are probably not representative);
- and "fluttering," in which the path rapidly alternated between two different routes. In $\mathcal{R}_1$, fluttering was quite common, and sometimes had great impact on the routes of consecutive packets sent by a host. But, like outages, our samples are not persuasively representative, and fluttering was rare in $\mathcal{R}_2$.

Because $\mathcal{R}_1$ and $\mathcal{R}_2$ were made a year apart, we can analyze the relative prevalence of pathologies in each (§ 6.10). We find that the likelihood of encountering a major routing pathology more than doubled between the end of 1994 ($\mathcal{R}_1$) and the end of 1995 ($\mathcal{R}_2$), rising from 1.5% to 3.4%.

After removing anomalous measurements, we analyze the remainder to investigate routing stability and symmetry. This analysis is primarily done using the $\mathcal{R}_2$ data, for the reasons given above. We begin in Chapter 7 by reviewing the importance of routing stability for a variety of network applications. This review reveals that there are two distinct types of stability that are of interest. The first is *prevalence*: whether we are likely to observe the same route in the future as at the present. The second is *persistence*: whether the route we observe at the present is likely to remain *unchanged* for a considerable period of time.

We show that it is easy to assess routing prevalence, and find that Internet paths are strongly dominated by a single prevalent route. But routing persistence is much more difficult to assess, because we have no *a priori* reason for assuming that observing a route at time T_1 and then again at time T_2 tells us anything about whether it changed (and changed back) in between those two measurements.

We tackle this difficulty by first analyzing those measurements we made that were spaced only minutes apart. Doing so reveals that a minority of the paths have routes that persist for only tens of minutes, while the majority persist for significantly longer. After eliminating the quickly changing paths, we repeat the analysis at time scales of 1 hour, 6 hours, and days. We find that, at each time scale, some paths are prone to changes and others are not. Overall, about two thirds of the paths have routes persisting for days or weeks.

A final question concerning routing stability is how an endpoint can determine that its route has changed. We investigate a simple method based on observing changes in the Time To Live (TTL) field. We find that this method provides a useful heuristic, having an overall accuracy of about 95%, but is prone to false negatives (missing the fact that the route has changed), which limits its utility.

In Chapter 8 we turn to the question of routing symmetry. As with routing stability, we first discuss the importance of symmetry for a number of networking applications. We also look at different mechanisms that can introduce asymmetry into network routing. Of these, one in particular ("hot potato" routing between different Internet service providers) is expected to grow in the future, leading to a greater prevalence of routing asymmetry, and the differences in asymmetry between the $\mathcal{R}_1$ and $\mathcal{R}_2$ measurements suggest that this is happening.

Our first attempt at defining whether two routes are symmetric founders on the difficulties of determining whether two Internet addresses do indeed correspond to the same host. In the face of this problem, we revise our definition to consider two routes symmetric only if they visit exactly the same cities. If two routes are *asymmetric* according to this definition, then they visit at least one different city. Such asymmetries are *major* because they likely imply different path characteristics, such as propagation times and congestion levels.

We find that *half* of all Internet paths in $\mathcal{R}_2$ contained a major asymmetry, while only 30% in $\mathcal{R}_1$ did. About 20% of the $\mathcal{R}_2$ paths differed in two or more cities, and about 30% differed in the autonomous systems they visited.

The presence of pathologies, short-lived routes, and major asymmetries highlights the difficulties of providing a consistent topological view in an environment as large and diverse as the Internet. Furthermore, the findings that the prevalence of pathologies and asymmetries greatly increased during 1995 show in no uncertain terms that *Internet routing has become less predictable in major ways*.

A constant theme running through our study is that of widespread diversity. We repeatedly find that different sites or pairs of sites encounter very different routing characteristics. This finding matches that of our previous work [Pa94a], which emphasizes that the variations in Internet traffic characteristics between sites are significant to the point that there is no "typical" Internet site. Similarly, there is no "typical" Internet path. But we believe the scope of our measurements gives us a solid understanding of the breadth of behavior we might expect to encounter—and how, from an endpoint's view, routing in the Internet actually works.

Chapter 3

Related Research

The problem of routing traffic in communications networks has been studied for well over twenty years [Sc77, SS80]. The subject has matured to the point where a number of books have been written thoroughly examining the different issues and solutions [Pe92, St95, Hu95].

A key distinction we will make concerning the study of routing is that between routing *protocols*, by which we mean mechanisms for disseminating routing information within a network and the particulars of how to use that information to forward traffic, and routing *behavior*, meaning how in practice the routing algorithms perform. This distinction is important because, while routing protocols have been heavily studied, routing behavior has not.

3.1 Studies of routing protocols

The literature contains many studies of routing protocols. In addition to the books cited above, see, for example, McQuillan et al.'s discussion of the initial ARPANET routing algorithm [MFR78] and the algorithms that replaced it [MRR80, KZ89]; the Exterior Gateway Protocol used in the NSFNET [Ro82, Re89], and the Border Gateway Protocol that replaced it [RL95, RG95, Tr95a, Tr95b]; the related work by Estrin et al on routing between administrative domains [BE90, ERH92]; Awerbuch's technique of reducing asynchronous networks to synchronous ones to simplify routing algorithms [Aw90]; Perlman and Varghese's discussion of difficulties in designing routing algorithms [PV88]; Deering and Cheriton's seminal work on multicast routing [DC90]; Perlman's comparison of the popular OSPF and IS-IS protocols [Pe91]; and Baransel et al.'s survey of routing techniques for very high speed networks [BDG95].

3.2 Studies of routing behavior

For routing behavior, however, the literature contains considerably fewer studies. Some of these studies are based on pure analysis, such as Bertsekas' study of routing dynamics for different topologies [Be82]; or on simulation, such as Zaumen and Garcia-Luna Aceves' studies of routing behavior on several different wide-area topologies [ZG-LA91, ZG-LA92], and Sidhu et al.'s simulation of OSPF [SFANC93]. In only a few studies do measurements play a significant role: Rekhter and Chinoy's trace-driven simulation of the tradeoffs in using inter-autonomous system routing information to optimize routing within a single autonomous system [RC92]; Chinoy's study of the dynamics of routing information propagated inside the NSFNET infrastructure [Ch93]; and Floyd and Jacobson's analysis of how periodicity in routing messages can lead to global synchronization among the routers [FJ94].

This is not to say that studies of routing protocols ignore routing behavior. But the presentation of routing behavior in the protocol studies is almost always qualitative, such as the discussion of the poor performance of the original ARPANET routing algorithm [MFR78] or the tendency for the revised algorithm to oscillate under heavy load [KZ89]. Even [MRR80], which presents the revised algorithm, and notes that to test it the authors subjected the network during off-hours to a greater volume of test traffic than users generated during peak hours, discuss this stress-testing in general terms, rather than delving into any measurement specifics.

Of the measurement studies mentioned above, [RC92] and [FJ94] are both devoted to examining a tightly focussed question. Only Chinoy's study is devoted to characterizing routing behavior in-the-large, and it remains the only formal measurement study of routing in wide-area networks of which we are aware.[1]

Chinoy found wide ranges in the dynamics of routing information: For those routers that send updates periodically regardless of whether any connectivity information has changed, the vast majority of the updates contain no new information. Most routing changes occur at the edges of the network and not along its "backbone." Outages during which a network is unreachable from the backbone span a large range of time, from a few minutes to a number of hours. Finally, most networks are nearly quiescent, while a few exhibit frequent connectivity transitions.

3.3 End-to-end routing dynamics

Chinoy's study concerns how routing information propagates *inside* the network. It is not obvious, though, how these dynamics translate into the routing dynamics seen by an end user. One of the areas noted by Chinoy as ripe for further study is "the end-to-end dynamics of routing information."

We will use the term *path* to denote the network-level abstraction of a "virtual link" between two Internet hosts. For example, when Internet host A wishes to establish a network-level connection to host B, A need not have any knowledge of the routing infrastructure upon which the Internet is built. As far as A is concerned, the network layer provides it with a link, or *path*, directly to B. Similarly, B has a *path* to A. We will sometimes abbreviate the notion of the path from A to B as $A \Rightarrow B$.

At any given instant in time, the path $A \Rightarrow B$ is realized at the network layer by a single *route*, which is a sequence of Internet routers along which packets sent by A and destined for B are forwarded. We will refer to a single *hop* of a particular route for the path as $R_1 \rightarrow R_2$, indicating that after arriving at router R_1, packets are next forwarded to R_2.

The path $A \Rightarrow B$ may oscillate very rapidly between different routes, or it may be quite stable (an issue we explore in Chapter 7). So Chinoy's suggested research area can be viewed as:

[1] Since publishing some of the results from this part of our thesis [Pa96b], we have learned of a very interesting study of Internet routing, similar in spirit to that of Chinoy's, by Jahanian, Labovitz and Malan [JLM97]. We will discuss this new work in the version of [Pa96b] presently undergoing revision for publication in *IEEE/ACM Transactions on Networking*. We unfortunately learned of the work too late to include discussion of it here.

given two hosts A and B at the edges of the network, how does the path $A \Rightarrow B$ between them behave over time? This is the question we attempt to answer in our study.

3.4 Routing in the Internet

For routing purposes, the Internet is partitioned into a disjoint set of *autonomous systems* (AS's), a notion first introduced in [Ro82]. Originally, an AS was a collection of routers and hosts unified by running a single "interior gateway protocol." Over time, the notion has evolved to be essentially synonymous with that of *administrative domain* [HK89], in which the routers and hosts are unified by a single administrative authority. Within the domain or AS are one or more *routing domains*, which are hosts and routers that communicate using the same routing protocol.

Routing between autonomous systems provides the highest-level of Internet interconnection. RFC 1126 [Li89] outlines the goals and requirements for inter-AS routing (of particular interest for our study are the goals of infrequent loops and stable routes). [Re95] gives an overview of how inter-AS routing has evolved.

When the NSFNET formed the "backbone" of the Internet, inter-AS routing was done using the Exterior Gateway Protocol (EGP) [Ro82, Re89]. A major constraint of EGP, however, is that it requires a tree-like topology between the AS's (with the NSFNET backbone at the root), and, if the topology is violated, loops can result. EGP has since been replaced with the Border Gateway Protocol (BGP), currently in its fourth version [RL95, RG95]. BGP is now used between all significant AS's [Tr95a]. BGP removes the EGP topology restrictions, allowing arbitrary interconnection topologies between AS's. It also provides a mechanism for preventing routing loops between AS's, which we discuss in § 6.3.1 and § 6.3.3.

The key to whether use of BGP will scale to a very large Internet lies in the *stability* of inter-AS routing [Tr95b]. If routes between AS's vary frequently—a phenomenon termed "flapping" [Do95]—then the BGP routers will spend a great deal of their time updating their routing tables and propagating the routing changes. Daily statistics concerning routing flapping are available from [Me95b] (see also [Co91-95]).

It is important to note that stable inter-AS routing does *not* guarantee stable end-to-end routing, because AS's are large entities capable of significant internal instabilities. In our study we focus on end-to-end routing behavior at the granularity of individual routers, though we also note where appropriate how the behavior changes when the granularity is shifted to that of autonomous systems (where the route for the path $A \Rightarrow B$ is viewed as a sequence of AS's rather than a sequence of routers).

One final note: since the publication of Chinoy's study, the Internet has undergone a major topological and administrative change. Inter-AS routing now uses BGP rather than EGP, as discussed above; and the network topology is no longer constrained to a tree with the NSFNET backbone at the root, but has switched to a number of commercial network service providers supporting a potentially arbitrary interconnection topology. Our measurements spanned this transition, with the first dataset taken at the end of 1994, before the NSFNET backbone was decommissioned in Spring 1995, while the second was taken at the end of 1995. Thus, our measurements give us an opportunity to determine whether Internet routing changed significantly during the year separating them. As discussed in § 6.10 and § 8.5, we find significant increases in the prevalence of routing "pathologies" and in routing asymmetry. These changes are not, however, necessarily due to the

NSFNET transition; in particular, two thirds of the routes measured in the first dataset already did not transit the NSFNET, traversing instead commercial providers such as SprintLink and MCINET, or networks outside the U.S.

Chapter 4

Methodology

In this chapter we discuss the methodology used to make our routing measurements. We begin with the software we used: the `npd` network probe daemon, the `npd_control` program used to drive the measurements, and the `traceroute` utility for measuring Internet paths. We then discuss the utility of sampling at exponentially distributed intervals, including the "PASTA Principle," which provides the underlying statistical validity of our measurements. In § 4.4 we then address which aspects of our data are plausibly representative of Internet traffic and which are not.

In our analysis we also attempt to draw some conclusions as to which differences between our datasets reflect significant changes in Internet conditions over time. To do so, we give in § 4.5 an overview of *Fisher's exact test* for determining whether the frequencies with which a property is observed in two different datasets is consistent with the null hypothesis of a single underlying probability of observing the property. If the frequencies observed are inconsistent with this hypothesis, then we conclude that the probability of observing the property *changed* between the two datasets, reflecting a corresponding change in Internet conditions. Finally, in order to use Fisher's test, we need to make an independence assumption that is not entirely accurate. § 4.6 discusses why this assumption remains tenable.

4.1 Experimental apparatus

We conducted our experiment as follows. First we recruited a number of Internet sites (detailed in Tables I and II) to participate in the study. Each site ran a "network probe daemon" (`npd`) that provides measurement services, as described in the Appendix. To measure the route from Internet host A to host B, a program called `npd_control`, running on our local workstation, would connect to `npd` on host A and request that it trace the route to host B using `traceroute`. The `npd` on A would then do so and send the results back to `npd_control`. In this fashion, we could run a single script on our local workstation to orchestrate any number of simultaneous route measurements. The script (which we programmaticly generated) would run `npd_control` in the background to conduct a single measurement, sleep until the time for the next measurement, run another `npd_control` in the background to conduct that measurement, and so on. Each measurement comprised a single `traceroute` from a randomly selected site to another randomly selected site.

This setup gave our experiment a single point of failure, namely our local workstation, but also the benefit of a single point of administration, which greatly simplified the task of keeping the experiment running correctly as we added new participating sites. Fortunately, during the entire measurement period the workstation never crashed or required rebooting, so the measurements proceeded uninterrupted.

For our first set of measurements, termed $\mathcal{R}_1$, we tuned the script driving the measurements so that each site would measure routes at an average rate of one every two hours, to minimize network load. Two exceptions were the `austr` and `korea` sites. They instead made measurements at lower rates of one every four hours and one every eight hours, in deference to the heavily loaded trans-Pacific network links that their traffic had to cross.

While using the same rate for each site meant each site had a consistent measurement load, as we added new participating sites to our study, the sampling rate of *pairs* of sites decreased. This inhomogeneity, however, does not present any particular difficulties for our sampling methodology, a point we address in § 4.3.

For the second set of measurements, $\mathcal{R}_2$, we made measurements at two different, fixed rates. The majority (60%) of the measurements were made with a mean inter-measurement interval of 2 hours, while the remainder were made with a mean interval of about 2.75 days. The bulk of the $\mathcal{R}_2$ measurements were also *paired*, meaning we would measure the path $A \Rightarrow B$ and then immediately measure the path $B \Rightarrow A$. We discuss the reasons for these changes in methodology in § 7.4 and § 8.4.

4.2 The `traceroute` Utility

`Traceroute` is a program written by Van Jacobson to trace the different hops comprising a route through the Internet [Jac89]. In this section we discuss the operation of the tool, as its particulars have direct impact on our routing measurements.

4.2.1 The Time To Live field

All packets sent using the Internetwork Protocol (IP) contain in their headers a *Time To Live* (TTL) field [Po81a]. In the original IP design, this field was meant to limit the amount of time that a packet could exist inside the network, to prevent packets from endlessly circulating around routing loops (and eventually clogging up the entire network). The TTL header field is 8 bits wide and is interpreted as the time in seconds remaining until the packet must be discarded. Each internetwork router must decrement the field by the amount of time required to process the packet (including queueing), or by 1 second, whichever is larger. Thus, the TTL limits packets to at most 255 hops through the network,[1] and a lifetime of at most 255 seconds.

If upon decrementing the TTL field a router observes that the TTL has reached zero, then it must not forward the packet but instead discard it as being too old. When it discards a packet for this reason, it must[2] then send back an Internet Control Message Protocol (ICMP; [Po81b]) message informing the sender of the packet that it was dropped due to an expired lifetime.

[1] This is plenty in today's Internet. Routes of more than 30 hops are rare (§ 6.7.5). But if much longer routes became commonplace, then the limited size of the TTL field could render parts of the Internet unable to communicate with other parts.

[2] This "must" is actually a very strong "should." [Ba95] states that the router must generate the message, but can provide a per-interface option to disable generation, provided the option defaults to generation enabled.

While the original IP standard states that TTL is a *time* [Po81a], in reality virtually all Internet routers only decrement the TTL by 1 per hop, regardless of the processing time, often for reasons of performance. Acknowledging this *de facto* behavior, the current standard for Internet routers only requires that routers decrement the TTL by 1 per hop, while allowing them the option to decrement by more to account for processing time [Ba95]. Part of the motivation for this relaxation of the TTL requirement is to aid the workings of `traceroute`.

4.2.2 How `traceroute` works

The heart of `traceroute` is clever exploitation of the TTL field, as follows. To trace the route to a remote host H, `traceroute` first constructs a packet with H as its destination but with the TTL field initialized to 1. When this packet reaches the first hop in the path to H, the router decrements the TTL field, notices that it is zero, and sends back an ICMP message to this effect. The ICMP message includes in its own header the address of the router sending the message, which lets `traceroute` identify the hop 1 router as that address.

`Traceroute` then sends a packet to H with the TTL field initialized to 2, and, similarly, gets back an ICMP message identifying the hop 2 router. It proceeds in this fashion until it receives a reply from H itself, and at that point it has elucidated the entire path to H. (Note that it has *not* also elucidated the path from H to the host running `traceroute`. The two are not necessarily the same, as we demonstrate in Chapter 8.)

We will refer to the packets `traceroute` sends with adjusted TTL's as *probes*, and those with an initial TTL of n as "hop n" probes. Here is an example of the output from `traceroute`, tracing the path from a host at the University of Colorado at Boulder (`ucol`, as explained in Table I) to one at the San Diego Supercomputing Center (`sdsc`).

```
traceroute to rintrah-fddi.sdsc.edu (198.17.46.57),
  30 hops max, 40 byte packets
 1  128.138.209.2  2 ms  2 ms  2 ms
 2  128.138.138.1  14 ms  4 ms  3 ms
 3  144.228.73.113  44 ms  39 ms  53 ms
 4  144.228.73.82  218 ms  207 ms  147 ms
 5  134.24.66.100  234 ms *  85 ms
 6  198.17.46.57  85 ms  63 ms  67 ms
```

By default, `traceroute` sends three probes for each hop. The probes are sent serially, each waiting until `traceroute` receives an answer for the previous one. For each hop, `traceroute` reports the number of hop, the IP address of the corresponding router, and the time in milliseconds it took to receive the reply. We note, however, that these times are often exceptionally noisy, because part of the total round-trip time includes the delay incurred at the router in generating an ICMP response to the exceptional event of an expired TTL. This delay can be quite large of the router is busy with other, higher priority tasks.

A reply time of "`*`," such as shown for hop 5, corresponds to a *lost* packet. Either the `traceroute` probe or the corresponding ICMP message was dropped by the network (or perhaps the ICMP message was not generated—see § 6.1, and also below). `Traceroute` waits 5 seconds for a reply before deciding that it will not be getting one.[3]

[3] Most versions of the `traceroute` documentation erroneously give this time as 3 seconds.

The first line of the output indicates "`30 hops max`," meaning that `traceroute` will stop sending probes after trying to elicit the 30th hop. This behavior is important because, as we will see in § 6.3.1, the Internet sometimes contains routing loops that would allow packets to circulate all the way up to the maximum of 255 hops, wasting considerable network resources. For our study we always used the default of 30 hops maximum (only very rarely did this prevent us from measuring the full path between sites in our study; see § 6.7.5), and the default of three probes per hop.

We can translate the IP addresses to hostnames in order to visualize the route more clearly:

```
 1  cs-gw-discovery.cs.colorado.edu  2 ms  2 ms  2 ms
 2  cu-gw.colorado.edu  14 ms  4 ms  3 ms
 3  sl-ana-3-s2/4-t1.sprintlink.net  44 ms  39 ms  53 ms
 4  sl-univ-ca-1-s0-t1.sprintlink.net  218 ms  207 ms  147 ms
 5  sdsc-ucop-mci.cerf.net  234 ms *  85 ms
 6  rintrah.sdsc.edu  85 ms  63 ms  67 ms
```

We see that the first two hops occur inside the University of Colorado at Boulder; then the packets are forwarded on to SprintLink, traveling first to Anaheim, CA, then up to Oakland, California (the University of California Office of the President), and finally back down along CERFNET to San Diego.

4.2.3 `Traceroute` limitations

When using `traceroute` there are several limitations and measurement difficulties that one must bear in mind. In the previous section we showed an example of a `traceroute` from Colorado to San Diego that went quite smoothly, suffering only a single packet loss. In contrast, consider the following `traceroute`, between the same two hosts:

```
traceroute to rintrah.sdsc.edu (198.17.47.57),
  30 hops max, 40 byte packets
 1  128.138.209.2  10 ms  0 ms  0 ms
 2  128.138.138.1  0 ms  0 ms  0 ms
 3  129.19.248.61  10 ms 129.19.254.45  10 ms 129.19.248.61  30 ms
 4  192.52.106.1  60 ms  60 ms  70 ms
 5  140.222.96.4  60 ms *  50 ms
 6  140.222.88.1  70 ms  60 ms  60 ms
 7  140.222.8.1  60 ms  50 ms  60 ms
 8  140.222.16.1  70 ms  70 ms  70 ms
 9  140.222.135.1  60 ms  70 ms  70 ms
10  198.17.47.2  4720 ms !H *  5100 ms !H
```

Here are the corresponding hostnames:

```
traceroute to rintrah.sdsc.edu (198.17.47.57),
  30 hops max, 40 byte packets
 1  cs-gw-discovery.cs.colorado.edu  10 ms  0 ms  0 ms
 2  cu-gw.colorado.edu  0 ms  0 ms  0 ms
 3  129.19.248.61 10 ms ncar-cu.co.westnet.net 10 ms 129.19.248.61 30 ms
 4  enss.ucar.edu  60 ms  60 ms  70 ms
 5  t3-3.cnss96.denver.t3.ans.net  60 ms *  50 ms
```

```
 6  t3-0.cnss88.seattle.t3.ans.net  70 ms  60 ms  60 ms
 7  t3-0.cnss8.san-francisco.t3.ans.net  60 ms  50 ms  60 ms
 8  t3-0.cnss16.los-angeles.t3.ans.net  70 ms  70 ms  70 ms
 9  t3-0.enss135.t3.ans.net  60 ms  70 ms  70 ms
10  enss.sdsc.edu  4720 ms !H *  5100 ms !H
```

The first thing we notice is that this route is longer than the previous one, and more circuitous, traveling over ANSNET (instead of SprintLink) through Denver and Seattle before arriving in California.

We also notice that the router at hop 3, `129.19.248.61`, does not have a corresponding hostname registered in the Domain Name System (DNS; [MD88]). While most routers have hostnames associated with their IP addresses, we found that not all do. In this case, we could identify the router's location from its network prefix (`129.19`), as Colorado State University in Boulder, Colorado.

Furthermore, for hop 3 `traceroute` reports not just one IP address but *multiple* addresses. What happened was that the first hop 3 probe was routed via the router with IP address `129.19.248.61`, while the second one went via a *different* router, `129.19.254.45` (this one has a hostname, `ncar-cu.co.westnet.net`). The third one went via the same router as the first one, `129.19.248.61`. Routing variation such as this can occur due to "load balancing," in which the upstream router (hop 2 in this case) alternates the downstream links it uses to forward packets in an effort to spread load among them and avoid overloading either one. We investigate the effects of such routing, which we term "fluttering," in detail in § 6.6.

Hop 3 also illustrates the more general principle that *packets do not always take the same route*. It also can be difficult to determine whether two routes are equivalent. For example, it may be that `129.19.248.61` is indeed an interface on the same `ncar-cu.co.westnet.net` router, but one that happens not to have a hostname associated with it. Or it may be a physically distinct router.

Because Internet routes can change between successive probe packets, we need to also realize that *we have no guarantee that probes of different hops take the same route as previous probes.* For example, from the above we might conclude that the first hop 3 probe took the route `cs-gw-discovery.cs.colorado.edu` → `cu-gw.colorado.edu` → `129.19.248.61`, and the second took the route `cs-gw-discovery.cs.colorado.edu` → `cu-gw.colorado.edu` → `ncar-cu.co.westnet.net`. But for all we know the upstream route could have changed between the end of the hop 2 probes and the beginning of the hop 3 probes, and the hop 3 packets may have been routed via Alaska at the first two hops! The only "guarantees" we can have that the route has not changed are: (1) consistency with other measurements of the same path (for example, in multiple measurements we always see the same routers for hop 2 and hop 3), and (2) self-consistency within the route. For example, if we find that hop $n + 1$ is geographically distant from hop n, and we know the network lacks a link between those two locations, then we would conclude that a routing change occurred upstream from hop $n + 1$. Some examples of this behavior are given in § 6.5 and § 6.6.1.

In general, if a route appears self-consistent and shows no sign of multiple routing for any of its hops, then we assume that it is indeed self-consistent, and treat the route as a valid measurement of the path to the remote host.

Another anomaly to discuss in the example above is the 10th hop:

```
10  enss.sdsc.edu  4720 ms !H *  5100 ms !H
```

Here "`!H`" indicates that `traceroute` received an ICMP "Host unreachable" message from the router `enss.sdsc.edu`. This means that the router knows that the host cannot be presently reached. Another diagnostic `traceroute` can generate is "`!N`," indicating that it received an ICMP "Network unreachable," the counterpart message indicating an entire network is unreachable (e.g., due to a failed link). We observed only two of these in all of our measurements.

Note also that the 3rd probe packet reports a round-trip time (RTT) of 5,100 msec, even though `traceroute` supposedly only waits 5 seconds to receive a reply. `Traceroute`'s timer, however, is not fine-grained, so due either to the timer's granularity, or to delays in scheduling the `traceroute` process for execution, `traceroute` received the reply before it decided to time out the probe.

Another limitation to keep in mind is that `traceroute` elicits the route as seen at the *IP network layer*. Each hop reported gives the next IP router in the path from the source to the destination. Often, IP routers are connected to one another using simple "link layer" technologies such as Ethernets or point-to-point links, with trivial topologies. Increasingly, however, the link layer technologies, for example ATM or Frame Relay, themselves have more complicated topologies, and are capable of routing packets within a link layer mesh that itself has multiple hops. `traceroute` cannot measure routing at this layer, because the TTL mechanism (§ 4.2.2) is present only at the higher (IP) layer. For example, in our second dataset we found a route with the following two successive hops:

```
gw1.scl1.alter.net
107.hssi4/0.gw1.mia1.alter.net
```

The first hop is in Santa Clara, California, and the second in Miami, Florida. It turns out that there is no direct physical connection between these two routers, but rather a Frame Relay mesh [Lid96], a fact that we could not have surmised from the `traceroute` measurement of the route.

Another potential source of measurement error arises in older (4.3 BSD-derived) routers incorrectly setting the TTL in their ICMP replies. As explained in the `traceroute` documentation ([Jac89]), these routers would erroneously use for the ICMP reply the TTL of the incoming packet that triggered the reply. For `traceroute` probes, this is a disaster, because the reply being triggered is precisely "TTL expired," so the ICMP replies would be sent back using a TTL of 0, too (and thus never reach us). Since such routers consistently fail to return an ICMP reply to the sender, they are a form of "unresponsive" router, for which we analyze our measurements in § 6.1.

A more subtle measurement problem occurs due to routers that are configured to *rate limit* generation of ICMP messages. For example, some routers will send at most one ICMP message each second. Such behavior is specifically encouraged in §4.3.2.8 of [Ba95], as a means of conserving both network bandwidth and router resources. In § 6.2 we analyze our measurements for the presence of rate-limiting routers, and find that, in general, only endpoint hosts (and not routers internal to the Internet) appear to be presently limiting their ICMP generation rate.

Another issue regarding `traceroute` concerns its use of the User Datagram Protocol (UDP; [Po80]). In order to associate the ICMP replies it receives with the probe packets it previously generated, `traceroute` must construct packets that manage to record identifying information in just the first 8 bytes of the transport layer header, as that is all of the original packet returned in an ICMP message. It does this by using for its probe a UDP packet, which it sends to a (hopefully) nonexistent port on the remote host H. The information `traceroute` needs to record the identifying information is coded in the port number in the UDP header.

Some network sites, however, have "firewalls" in place to filter incoming network traffic for security purposes [CB94]. These firewalls may decide that the incoming UDP packet does not appear destined to any of the services the site wishes to make publicly available to the Internet, so the firewall drops the packet without returning an ICMP Time Exceeded message. Thus, firewalls can generate an effect similar to lost packets (`traceroute` never receives a reply for a given hop, or beyond it). It was easy to identify such sites, as `traceroutes` to them consistently stopped short at the same router (§ 6.7.4). For our analysis of the data, we considered any `traceroute` reaching a firewall router as having successfully reached the host.

`Traceroute`'s use of UDP packets raises another measurement issue. When `traceroute` traces the route to an IP address A, it determines that it has elicited the full route whenever it receives a "UDP Port Unreachable" ICMP reply, *even if the reply did not come from a router identifying itself as address A*. Some hosts (and indeed all routers) have multiple IP addresses associated with them, so it is possible when tracing the route to address A to receive a reply from address B. When this happens, it indicates that A and B are both addresses for the same host (even though their associated hostnames might not reveal this).[4]

It is sometimes possible to use this `traceroute` feature to determine whether two IP addresses correspond to the same host. For example, the name associated with `134.55.12.231` is `llnl3-e-stub.es.net`, while the name associated with `134.55.6.71` is `llnl-lc3-3.es.net`. Both of these names have DNS "A" records for the corresponding addresses, and no extra records, so *a priori* we might assume that the two addresses/hostnames refer to two separate machines. However, depending on the state of ESNET routing, it is possible for a traceroute to `llnl3-e-stub.es.net` to be "answered" by `llnl-lc3-3.es.net`, indicating that they are indeed the same machine. This test is not guaranteed to work, though. It depends on the machine's algorithm for deciding what IP address to put in its ICMP reply, and on which interface the incoming UDP probe packet arrives (which in turn depends on the current routing).

4.3 Exponential sampling

We use the term "measurement" to denote the full process of running the `traceroute` utility; that is, the attempted tracing of the entire route between a source host and a destination host. In our experiment we devise our measurements of Internet routes so that the time intervals between consecutive measurements are independent and exponentially distributed.

Using independent and exponentially distributed intervals between measurements gains two important (and related) properties. The first is that the measurements correspond to *additive random sampling* [BM92]. Such sampling is unbiased because it samples all instantaneous signal values with equal probability.

The second important property is that the measurement times form a Poisson process. This means that Wolff's *PASTA principle*—"Poisson Arrivals See Time Averages"—applies to our measurements: asymptotically, the proportion of our measurements that observe a given state is equal to the amount of time that the Internet spends in that state [Wo82].

[4] Note also that sometimes the route to address A is different than the route to address B! For our measurements, this only occurred for `mbone.ucar.edu`, for which the route to one of its addresses is one hop longer (and a strict superset) of the route to the other address. We accommodated this difference in our analysis by considering a `traceroute` that reached the endpoint of the shorter route as having traveled successfully to the host.

Two important points regarding Wolff's theorem are (1) the observed process does *not* need to be Markovian; and (2) the Poisson arrivals need not be *homogeneous*[5] [Wo82, § 3]. This second point is particularly important for our study, because our measurement rate varied, as discussed in § 4.1.

The only requirement of the PASTA theorem is that the observed process cannot *anticipate* observation arrivals. For any interarrival distribution other than independent exponentials, the process can anticipate observation times to some degree because the instantaneous probability of an arrival changes with the length of time since the last observation. For the exponential distribution, however, the probability remains constant, a consequence of the distribution's "memoryless" property. Thus, the theorem fundamentally requires independent exponential intervals between measurements, which argues strongly for the use of exponential sampling in practice.

There is one respect in which our measurements fail the "lack of anticipation" requirement. Even though we schedule our observations to come at independent, exponentially distributed intervals, the network *can* anticipate arrivals to a certain extent. In particular, *when the network has lost connectivity between the site running* `npd_control` (§ 4.1) *and a site potentially conducting a* `traceroute`*, the network can predict that* no *measurement will occur.* Thus, while the times at which we *attempted* to measure the network satisfy the PASTA requirements, the times for which we *successfully* measured the network do not in this regard. The effect of this imperfect sampling is a tendency to *underestimate* the prevalence of network connectivity problems, as discussed further in § 5.2.

The main use we make of the PASTA theorem is as follows. If we make n observations of Internet routing, of which k find state S and $n - k$ find some other state, then because of PASTA we are on firm ground making the assumption that the unconditional probability of observing state S is approximately k/n. Furthermore, if $k \ll n$, we argue that we can consider the observations as independent, and hence can apply a Fisher's exact test (§ 4.5) to test for significant differences among sets of observations. We discuss this independence assumption further in § 4.6.

4.4 Which observations are representative?

In this section we discuss what sort of observations we can make of the Internet for which our samples are plausibly representative of Internet behavior in general, and those for which we would not consider our samples representative.

37 Internet hosts participated in our routing study. This is a miniscule fraction of the estimated 6.6 million Internet hosts as of July, 1995 [Lo95], so clearly the behavior we observe that is due to the particular endpoint hosts in our study is not representative.[6]

The 37 endpoint hosts were from 34 different networks, again a miniscule fraction of the more than 50,000 known to the NSFNET in April, 1995 [Me95a]. So, again, any behavior we observe due to the particular endpoint ("stub") networks in our study is not persuasively representative.

On the other hand, we argue that the *routes* between the 37 hosts are plausibly representative, because they include a non-negligible fraction of the *autonomous systems* (AS's) which

[5] That is, the arrival rate can vary over time, as long as the interarrival distribution remains exponential and the arrivals remain independent of each other and of the observed process.

[6] Furthermore, the sites were self-selected (usually, though not always, because someone at the site had an interest in wide-area networking) and skewed to universities.

together comprise the Internet. Recall that AS's are administrative entities that manage routing for a collection of networks, using unspecified protocol(s), and that routing *between* AS's is done using the Border Gateway Protocol. We expect the different routes within an AS to have similar characteristics (e.g., prevalence of pathologies, or routing stability), because they fall under a common administration. We therefore argue that sampling a significant number of AS's lends representational weight to a set of measurements.

To determine the number of AS's in the Internet, we proceeded as follows. In January, 1996, we obtained a BGP routing table dump from the AS border router `kasina.sdsc.edu`, located at the San Diego Supercomputer Center (SDSC) [7]. The routing table lists all the destinations (networks, more or less) known to the router, i.e., its view of the Internet. For each of those destinations, the table includes a list of AS's over which routing information for the destination traveled to `kasina.sdsc.edu`. The view of Internet routing given directly by this table is skewed by SDSC's particular location in the Internet. However, virtually all of the routing reflects disparate AS's connecting to SDSC's network service provider, MCI, at many different points. So, if we exclude MCI itself from our subsequent analysis, then the remainder of the routing gives us a much broader view, namely that seen by MCI at its many interconnection points.

All in all, the routes in the table included 1,031 AS's for 33,824 distinct destinations. From this we estimate that the Internet presently has about 1,000 active AS's. (As of August, 1995, about 6,600 had been assigned [DISA95].) The routes in our study traversed 85 of these, or about 8%.

An important point, however, is that not all AS's are equal—some are much more prominent in Internet routing than others. We devised a "weight" to associate with AS's as follows. For each AS, we counted the number of times it occurred in the BGP table in a path to a remote destination. The AS's weight then is the ratio of the number of its occurrences to the total number of occurrences of any AS.[8]

The weights obtained in this fashion are skewed towards the view of the Internet as seen by SDSC, and indeed two AS's had weight 25%: AS 145 ("NSFNET-CORE") and AS 3561 ("MCI-RESTON"), because virtually every route known to the SDSC router goes through these two. But the next AS has a weight of only 5% (AS 1239, "SprintLink"), because the majority of the routes do not go through it. So we adjusted for the SDSC-skewed perspective by removing the first two AS's from the set and recomputing the weights. After this adjustment, we find that the AS's sampled by the routes we measured represent, by weight, about 52% of the Internet routes. We take this as an indication that we did indeed sample a significant subset of the large-scale variation in Internet routes, and our observations of those routes are plausibly representative of Internet routing as a whole.

4.5 Testing for significant differences

Because we have measurements taken at two points in time—the end of 1994 and the end of 1995—we have an opportunity to assess a number of aspects of the measurements in the two datasets for the degree to which they reflect significant differences. We can then interpret these

[7] Many thanks to Hans-Werner Braun of SDSC for suggesting and facilitating this.

[8] Better would probably be to weight by traffic volume. Unfortunately, the statistics necessary for doing so are not available.

differences (or lack of differences) as indicating how the Internet changed (remained unchanged) over the course of 1995. While having just two points in time offers only the most crude form of trend, it is still far better than simply assuming that characteristics of the Internet do not change, particularly given evidence of major changes over time as discussed in our previous work [Pa94b, Pa94a].

The potential changes we will attempt to assess concern the frequency with which we observe different Internet phenomena (for example, routing loops). Suppose that, out of two representative samples from $\mathcal{R}_1$ and $\mathcal{R}_2$ of n_1 and n_2 observations, respectively, we find that subsets of size k_1 and k_2 exhibit some property $\mathcal{P}$. We wish to gauge whether finding k_1 instances of $\mathcal{P}$ out of n_1 samples in $\mathcal{R}_1$ is statistically consistent with finding k_2 instances out of n_2 samples in $\mathcal{R}_2$. If consistent, then we do not have evidence of a significant change between $\mathcal{R}_1$ and $\mathcal{R}_2$. But if the findings are inconsistent, then we interpret the difference as due to a change in the prevalence of $\mathcal{P}$: either the likelihood of $\mathcal{P}$ increased during 1995, if $\frac{k_2}{n_2} > \frac{k_1}{n_1}$, or decreased, if $\frac{k_2}{n_2} < \frac{k_1}{n_1}$.

To test for statistically significant differences, we use *Fisher's exact test*. The discussion of the test we now present follows that of Rice [Ri95]. Let K_1 denote a random variable giving the number of instances of $\mathcal{P}$ observed in $\mathcal{R}_1$, N_1 the total number of observations in $\mathcal{R}_1$, and K_2 and N_2 the same for $\mathcal{R}_2$. Let $K = K_1 + K_2$ and $N = N_1 + N_2$ correspond to the totals across both datasets.

The key observation of Fisher's test is that, if the likelihood of observing $\mathcal{P}$ is the same in the two datasets, then we can view the problem as: for K total instances of $\mathcal{P}$ out of N observations, how likely is it that K_1 of them would have fallen into $\mathcal{R}_1$, given that $\mathcal{R}_1$ comprises N_1 observations? With this rephrasing of the problem, we have that

$$P[K_1 = k_1 | N_1 = n_1, K = k, N = n] = \frac{\binom{n_1}{k_1}\binom{n-n_1}{k-k_1}}{\binom{n}{k}}. \tag{4.1}$$

The numerator of Eqn 4.1 corresponds to the number of ways that k instances of $\mathcal{P}$ can be distributed, among a partition of n total observations, into two sets of n_1 and $n_2 = n - n_1$ observations, given that the first set of observations includes k_1 instances of $\mathcal{P}$. The denominator corresponds to the total number of ways that k instances can be distributed over n observations, not subject to any conditioning. The ratio then gives the probability of observing k_1 instances in $\mathcal{R}_1$, given the size of $\mathcal{R}_1$, the total number of instances of $\mathcal{P}$, the size of the combined sample pool, and the null hypothesis that $\mathcal{R}_1$ and $\mathcal{R}_2$ are constructed using independent draws without replacement from the combined sample pool.

Armed with Eqn 4.1 for the probability of observing exactly k_1 instances, we can then construct a *rejection region* corresponding to values of k_1 that we would be unlikely to observe if the null hypothesis is indeed correct. We use a *two-sided* region, meaning that it includes both values of k_1 that are too low to be likely, and values that are too high. To construct the region, we find the maximum k_l and minimum k_u for which

$$\begin{aligned} P[K_1 \le k_l | N_1 = n_1, K = k, N = n] &\le \frac{\alpha}{2} \\ P[K_1 \ge k_u | N_1 = n_1, K = k, N = n] &\le \frac{\alpha}{2}. \end{aligned}$$

Given these values, we then have

$$P[K_1 \le k_l \text{ or } K_1 \ge k_u | N_1 = n_1, K = k, N = n] \le \alpha.$$

So, given the null hypothesis, K_1 will fall into the rejection region by chance with probability α or smaller. By using $\alpha = 0.05$, using this test we will erroneously reject the null hypothesis at most 5% of the time. Consequently, if K_1 falls into the rejection region, we conclude with confidence 95% that the null hypothesis is incorrect, and indeed there was a significant change in the prevalence of $\mathcal{P}$ between $\mathcal{R}_1$ and $\mathcal{R}_2$.

All that remains to use this test is to specify how to find k_l and k_u. For a given κ, we have

$$P[K_1 \leq \kappa | N_1 = n_1, K = k, N = n] \;=\; \binom{n}{k}^{-1} \sum_{i=\max(0,k-n_2)}^{\kappa} \binom{n_1}{i}\binom{n-n_1}{k-i},$$

where $n_2 = n - n_1$. So to find k_l we simply carry out the summation for $\kappa = 0, \ldots, \min(n_1, k)$ and note the largest value of κ for which the probability is $\leq \frac{\alpha}{2}$.[9]

The procedure for finding k_u is analogous.

4.6 A note on independence

The argument in the previous section assumes that our measurements are observing independent events. This is not quite true for our measurements. Using Poisson sampling means that the measurement *arrivals* are independent. However, the observations *themselves* (what each independently scheduled measurement observes) are not independent: any temporal correlations in the observed process will be faithfully reflected in the observations.

However, we will be applying the methodology in § 4.5 to *rare* events, such as the observation of pathological routing conditions. These rare events are generally *not* clustered in time, so the approximation that observations of them are independent is a good one.

[9]Here and in the equation, the $\min$ and $\max$ operators are to exclude values of κ that are impossible because they require more than n_2 instances of $\mathcal{P}$ in the second set of the partition, or fewer than 0.

Chapter 5

The Raw Routing Data

In this chapter we discuss the sites that participated in our routing experiments, the duration of the experiments, and the preliminary reduction of the raw data we gathered.

5.1 Participating sites

The first routing experiment began the evening of Tuesday, November 8, 1994, and lasted until the morning of Saturday, December 24. During this time, we attempted 6,991 `traceroutes` between 27 sites. We refer to this collection of measurements as $\mathcal{R}_1$ (dataset #1). We will often refer to a single such measurement as a "`traceroute`."

The second experiment began the morning of Friday, November 3, 1995, and lasted until the afternoon of Thursday, December 21. It included 37,097 attempted `traceroutes` between 33 sites. We refer to this collection of measurements as $\mathcal{R}_2$. Details of the measurements and the sampling intervals are discussed in § 4.1. Both $\mathcal{R}_1$ and $\mathcal{R}_2$ are publicly available from the *Internet Traffic Archive*, at:

`http://www.acm.org/sigcomm/ITA`

under the name *NPD-Routes*.[1]

Table I lists the sites participating in $\mathcal{R}_1$, giving the abbreviation we will use to refer to each site, the site's Internet domain, the number of days it participated in the study, a brief description of the site, and its location. These sites also participated in $\mathcal{R}_2$, except for `batman`, `korea`, `usc`, and `xor`. Table II lists the additional sites participating in $\mathcal{R}_2$. In $\mathcal{R}_2$, all sites participated at least a month, except for `ukc`, which participated for 23 days, and 13 of the sites participated for the maximum of 48 days.

The sites include educational institutes, research labs, network service providers, and commercial companies, in 9 countries. Figures 5.1 and 5.2 show the geographic locations of the North American and European sites.

[1] At the time of this writing, the Archive is moving from its old location to the above URL. If the reader has any difficulty accessing the Archive, send email to `vern@ee.lbl.gov`.

Name	Domain	Days	Description	Location
`austr`	mu.oz.au	24	University of Melbourne	Melbourne, Australia
`batman`	batman.net	11	Experimental ATM network at National Center for Atmospheric Research	Boulder, CO
`bnl`	bnl.gov	37	Brookhaven National Lab	Brookhaven, NY
`bsdi`	bsdi.com	9	Berkeley Software Design, Inc.	Colorado Springs, CO
`connix`	connix.com	22	Caravela Software	Middlefield, CT
`harv`	harvard.edu	9	Harvard University	Cambridge, MA
`inria`	inria.fr	9	INRIA	Sophia, France
`korea`	postech.ac.kr	36	Pohang Institute of Science and Technology	Pohang, South Korea
`lbl`	lbl.gov	45	Lawrence Berkeley Lab	Berkeley, CA
`lbli`	lbl.gov	45	LBL home computer connected via ISDN	Berkeley, CA
`mit`	mit.edu	21	Massachusetts Institute of Technology	Cambridge, MA
`ncar`	ucar.edu	22	National Center for Atmospheric Research	Boulder, CO
`nrao`	cv.nrao.edu	44	National Radio Astronomy Observatory	Charlottesville, VA
`oce`	oce.nl	19	Oce-van der Grinten	Venlo, The Netherlands
`pubnix`	va.pubnix.com	11	Pix Technologies Corp.	Fairfax, VA
`sdsc`	sdsc.edu	24	San Diego Supercomputer Center	San Diego, CA
`sri`	sri.com	9	SRI International	Menlo Park, CA
`ucl`	ucl.ac.uk	24	University College	London, U.K.
`ucol`	colorado.edu	45	University of Colorado	Boulder, CO
`ukc`	ukc.ac.uk	24	University of Kent	Canterbury, U.K.
`umann`	uni-mannheim.de	19	University of Mannheim	Mannheim, Germany
`umont`	umontreal.ca	15	University of Montreal	Montreal, Canada
`unij`	kun.nl	9	University of Nijmegen	Nijmegen, The Netherlands
`usc`	usc.edu	45	University of Southern California	Los Angeles, CA
`ustutt`	uni-stuttgart.de	16	University of Stuttgart	Stuttgart, Germany
`wustl`	wustl.edu	33	Washington University	St. Louis, MO
`xor`	xor.com	30	XOR Network Engineering	East Boulder, CO

Table I: Sites participating in first experiment ($\mathcal{R}_1$)

Name	Domain	Description	Location
`adv`	advanced.org	Advanced Network & Services	Armonk, New York
`austr2`	newcastle.edu.au	University of Newcastle	Newcastle, Australia
`mid`	mid.net	MIDnet	Lincoln, Nebraska
`near`	near.net	NEARnet	Cambridge, Massachusetts
`panix`	nyc.access.net	Public Access Networks Corporation	New York, New York
`rain`	rain.net	RAINet, Inc.	Portland, Oregon
`sandia`	ca.sandia.gov	Sandia National Laboratories	Livermore, California
`sintef1`	sintef.no	University of Trondheim	Trondheim, Norway
`sintef2`	sintef.no	University of Trondheim	Trondheim, Norway
`ucla`	ucla.edu	University of California	Los Angeles, California

Table II: Additional sites participating in second experiment ($\mathcal{R}_2$)

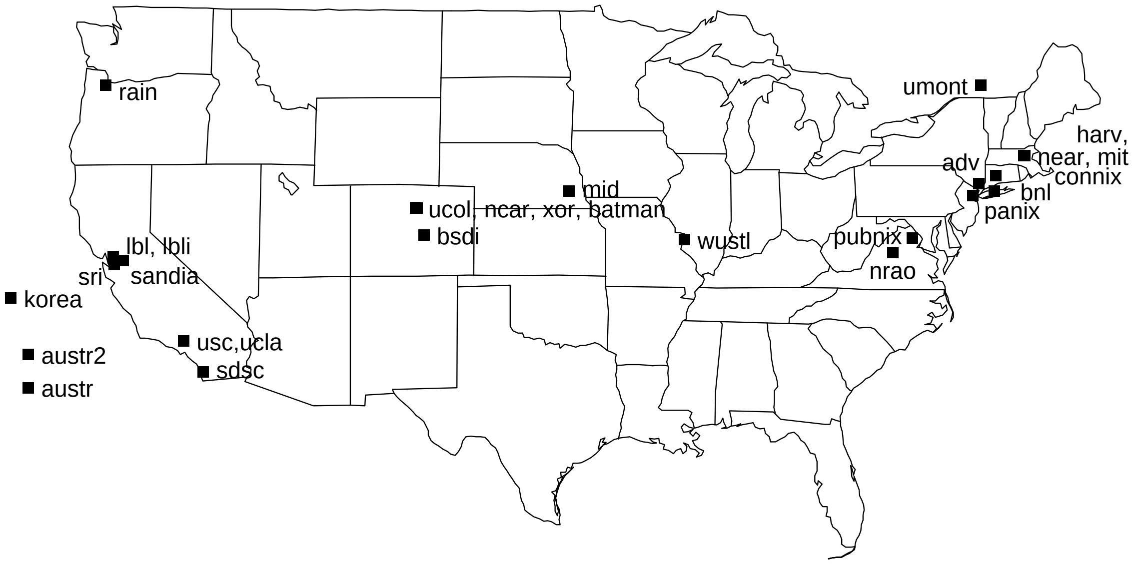

Figure 5.1: Sites participating in routing study, North America and Asia

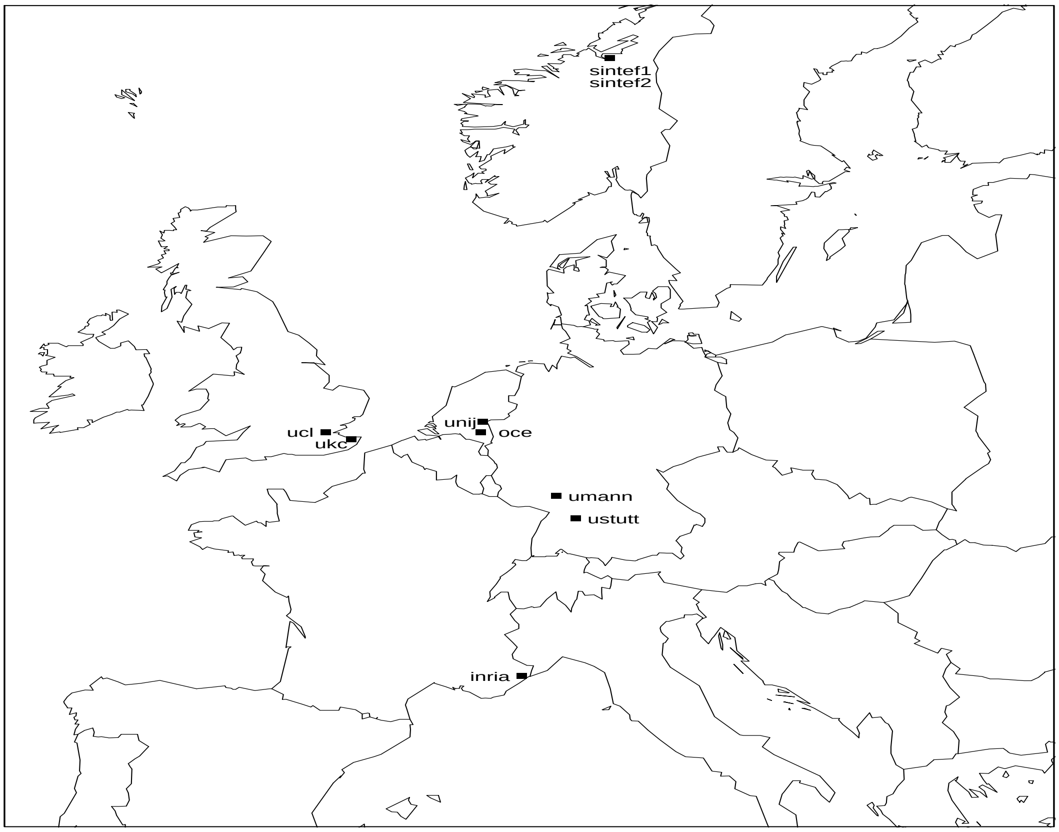

Figure 5.2: Sites participating in routing study, Europe

Status	Experiment 1 #	%	Experiment 2 #	%
Unable to contact daemon	495	7.1%	1,872	5.0%
Daemon configuration error	25	0.4%	15	0.04%
Host lookup failure	12	0.2%	101	0.3%
Total failures	532	7.6%	1,988	5.4%
Total successes	6,459	92.4%	35,109	94.6%
Total	6,991	100.0%	37,097	100.0%

Table III: Summary of routing experiment difficulties

5.2 Measurement breakdown

In the two experiments, between 5–8% of the `traceroutes` failed outright (i.e., we were unable to contact the remote `npd`, execute `traceroute` and retrieve its output). As shown in Table III, almost all of the failures were due to an inability of the `npd_control` process to contact the remote daemon. Some of these were failures involving `lbli`; that site, due to its ISDN link frequently being down (§ 6.7.4), was often unreachable. But for most of the failures we do not *a priori* know whether they represent the remote host being down or an Internet connectivity failure. It is important to note that, if the latter was frequently the case, then to some degree *the assumptions behind PASTA are invalid*, since an agent at the remote site with knowledge of current connectivity problems could reliably predict no sampling would occur in the near future (§ 4.3).

For our analysis, the effect of these failures to contact the remote daemon (`npd`) will lead to a bias towards *underestimating* Internet connectivity failures, because sometimes the failure to contact the remote daemon will result in losing an opportunity for a `traceroute` experiment to reveal the lack of connectivity between that site and another remote site that shares the same path as used between `npd_control` and the daemon.

When taking the $\mathcal{R}_2$ measurements, however, we somewhat corrected for this underestimation by *pairing* each measurement of the path $A \Rightarrow B$ with a measurement of the path $B \Rightarrow A$.[2] If `npd_control` was unable to reach one of either A or B, it still attempted to contact the other to measure the reverse route. In those circumstances where it was able to measure the reverse route, it still had an opportunity to observe the routing fault, if present in both directions.

`npd_control` was unable to reach one of either A or B 1,872 times. It was unable to contact the other host of the measurement pair, either, in only 5% of these instances. Thus, for the most part, the $\mathcal{R}_2$ measurements do not suffer from bias in observing bidirectional routing faults.

We could further reduce this measurement problem by introducing a "batch" design to `npd`, where the daemon would accept a list of measurements it should make at future points in time, and would email back the results when they were complete. We did not adopt this approach because one of our goals in the design of `npd` was to keep it simple enough that sites volunteering to run it could with reasonable ease inspect the code to see what they were running.

[2] About 20% of the measurements were not paired, because they were made in conjunction with the measurements discussed in Part II.

Destination \ Source	total	lbl	usc	nrao	ucol	lbli	ucl	bnl	wustl	xor	ukc	sdsc	mit	connix	ncar	oce	umont	umann	ustutt	harv	korea	batman	inria	sri	bsdi	unij	pubnix	austr
austr	100	4	6	4	9	1	4	3	8	3	9	5	3	5	2	6	4	2	5	2	3	1	1	4	1	3	2	
pubnix	98	3	4	5	2	5	6	5	1	5	6	2	3	4	1	8	1	4	4	5		3	4	4	5	7		1
unij	82	10	4		3	1	2	2	2	4	6	1	6	1	6	2	2	5	1	6		1	3	5	5		3	1
bsdi	87	3	3	3	3	4		5	6	4	4	1	1	1	1	1	7	3	5	6	2	7	5	5		3	3	1
sri	104	8	2	5	3	3	1	4	7	5	5	4	6	6	2	4	4	8	2	5		1	4		6	3	5	1
inria	106	7	6	4	6	1	4	3	5	6	4	3	4	4	4	4	2	6	4	8	1	4		5	1	3	6	1
batman	126	5	6	4	4	1	7	7	5	5	1	3	5	8	4	5	8	6	4	4			7	9	5	4	5	4
korea	88	7	10	6	8	8	5	7	6	7	3	3	2		2		1	1	1	2		1		1	1	2	2	2
harv	119	3	5	4	4	1	4	6	7	5	4	9	7	5	4	6	9	3	2		2	2	4	2	11	4	4	2
ustutt	142	5	4	8	8	11	7	7	8	10	4	3	13	5	9	3	7	1		1	1	6	4	1	7	1	5	3
umann	185	12	11	6	12	4	10	9	10	9	13	4	11	11	9	7	3		8	4	2	8	3	2	5	4	4	4
umont	168	7	6	11	5	10	3	8	14	10	5	8	7	11	8	10		5	7	2	2	5	7	4	5	3	3	2
oce	216	20	10	11	3	8	4	17	17	8	23	9	11	10	10		13	4	6	3	4	6	2	4	2	2	2	7
ncar	225	9	16	15	8	10	14	13	14	16	8	16	20	7		4	5	6	5	4	2	6	5	3	5	4	2	8
connix	222	15	17	13	12	5	7	13	14	14	10	14	11		6	9	12	8	2	8	6	5	3	2	5	4	3	4
mit	200	14	10	13	11	8	6	9	5	10	13	10		8	6	13	7	10	11	6	2	5	5	6	4	4	2	2
sdsc	233	11	16	11	14	10	9	21	10	19	16		12	16	8	10	11	9	5	5	2	4	2	4		1	6	1
ukc	228	14	11	13	14	13	13	16	11	10		10	16	10	16	11	5	4	8	3	4	4	5	6	3	3	3	2
xor	302	24	20	22	19	22	16	27	21		14	14	18	11	8	11	5	9	4	4	5	6	6	3	5	4	1	3
wustl	308	25	23	27	20	16	16	20		24	17	18	14	13	14	9	11	2	4	6	5	5	3	7	4	1	3	1
bnl	374	34	31	30	35	26	32		32	25	16	14	9	6	13	14	9	8	3	5	7	4	3	3	1	4	3	7
ucl	431	41	44	40	45	43		33	24	20	14	16	15	7	11	11	7	6	3	7	9	3	7	5	6	8	2	4
lbli	506	56	51	72	59		41	31	28	26	13	27	7	13	16	6	10	7	6	3	10	2	4	5	4	2	4	3
ucol	436	47	53	45		45	44	26	28	14	9	16	10	13	23	6	4	8	3	6	14	5	5	2	2	7		1
nrao	454	57	48		52	42	44	28	32	24	12	14	8	15	12	9	4	7	6	2	8	5	5	4	3	3	7	3
usc	467	60		49	47	52	49	37	25	18	11	17	12	13	6	11	7	9	4	4	7	4	2	5	2	5	7	4
lbl	452		56	35	50	46	37	25	40	18	12	10	10	17	13	12	8	4	11	4	11	6	7	2	2	4	5	7
total	6459	501	473	456	456	396	385	382	380	319	252	251	241	220	214	192	166	145	124	115	109	109	106	103	100	93	92	79

Figure 5.3: Number of measurements made for each Internet path, $\mathcal{R}_1$ dataset

Destination	total	connix	lbl	ucla	sintef2	sintef1	austr2	inria	mid	rain	adv	ucl	near	wustl	umont	austr	ustutt	mit	sri	bsdi	bnl	umann	pubnix	sandia	sdsc	unij	ucol	harv	oce	ncar	nrao	lbli	ukc	panix
panix	682	13	15	23	9	14	17	32	18	23	32	13	16	41	16	45	27	12	29	16	13	20	15	24	14	20	28	34	19	27	23	11	23	
ukc	564	17	27	15	22	38	17	19	14	23	16	38	9	21	15	16	16	8	30	12	13	21	30	17	33	14	13	18	5	11	7	8		1
lbli	688	24	37	36	23	17	30	21	28	22	19	26	35	14	23	22	27	16	26	28	36	16	15	11	11	17	19	27	13	24	15		7	3
nrao	864	46	21	20	39	28	17	32	32	33	23	14	25	25	43	32	44	36	26	29	32	38	14	34	20	29	13	20	27	28		27	11	6
ncar	844	32	24	28	21	44	14	30	28	39	28	35	19	9	19	31	26	30	42	33	27	49	13	21	27	37	31	17	33		24	19	9	5
oce	1006	33	30	27	34	37	32	39	25	38	38	24	32	54	43	40	62	32	29	39	18	37	18	12	19	25	31	27		43	37	19	18	14
harv	931	31	34	30	34	38	30	44	32	26	44	41	20	31	20	26	23	23	32	37	37	13	40	41	35	36	23		26	15	13	30	11	15
ucol	921	27	34	31	45	29	34	32	31	13	22	24	46	26	32	32	35	25	40	44	26	53	31	21	16	34		17	33	31	12	23	14	8
unij	910	33	28	26	35	23	50	19	27	35	32	23	28	25	52	22	26	33	22	31	23	32	36	36	31		35	33	15	37	28	18	13	3
sdsc	1078	48	32	46	30	54	47	37	44	33	45	43	34	28	43	48	33	51	18	26	38	33	21	29		43	19	33	21	21	27	9	38	6
sandia	973	41	37	30	29	29	38	27	19	50	32	27	26	42	31	31	33	32	70	30	39	24	12		39	44	29	36	15	19	32	8	15	7
pubnix	1021	35	57	59	27	45	22	42	41	47	41	35	47	31	46	54	26	26	18	29	29	31		24	20	34	33	25	20	13	10	20	25	9
umann	1018	32	26	48	49	25	25	24	40	39	37	29	30	42	29	29	40	37	44	27	28		20	24	30	51	50	10	29	48	36	17	14	9
bnl	1031	37	28	41	36	49	86	28	31	23	31	39	40	44	32	33	34	30	38	18		27	28	32	33	22	24	46	18	27	29	32	12	3
bsdi	1043	45	42	29	60	43	54	26	28	28	43	34	31	51	24	37	16	23	30		18	37	37	39	28	32	27	30	39	35	27	29	10	11
sri	1043	43	35	27	34	19	31	47	38	34	37	32	32	42	44	21	39	21		26	39	40	15	64	17	28	40	36	28	36	21	26	31	20
mit	1030	39	41	38	48	43	55	32	82	42	40	41	24	27	28	34	18		21	21	29	37	26	32	52	32	25	22	15	25	33	15	7	6
ustutt	1110	36	36	60	33	33	39	34	37	39	37	44	39	51	41	23		18	41	21	33	41	25	34	28	40	37	22	61	26	41	29	23	8
austr	1065	42	41	36	51	46	46	30	35	48	37	14	32	36	41		17	36	23	37	35	36	51	31	43	23	28	32	38	31	28	8	9	24
umont	1077	26	24	72	49	34	20	25	34	38	36	47	39	43		55	42	30	32	36	28	29	31	27	33	59	31	19	34	19	41	26	13	5
wustl	1063	29	50	37	64	24	31	42	18	39	44	41	22		43	35	62	31	31	55	33	38	31	38	24	22	30	17	40	10	22	9	26	25
near	1228	81	46	68	44	48	42	54	48	51	29	57		40	54	34	43	43	34	33	36	26	48	35	34	25	40	17	30	27	28	21	9	3
ucl	1210	72	50	78	54	51	26	47	52	34	26		60	43	40	15	43	41	42	33	20	29	47	43	51	34	22	31	20	34	22	20	27	3
adv	1165	52	50	53	26	52	67	40	59	33		18	29	61	43	42	44	40	25	41	31	31	40	29	38	30	22	39	41	33	19	6	15	16
rain	1260	37	66	48	61	54	45	64	45		57	37	51	48	51	44	37	42	38	33	24	35	43	57	29	28	23	25	16	39	26	13	21	23
mid	1260	53	64	48	54	49	46	47		45	54	56	53	27	45	38	35	82	40	46	34	42	40	23	30	18	29	31	19	30	33	36	8	5
inria	1282	46	82	55	50	67	76		45	64	47	61	47	34	22	36	33	34	35	33	34	24	46	30	21	22	30	47	39	31	16	44	15	16
austr2	1168	51	49	22	45	42		66	53	44	67	21	44	31	18	41	56	54	37	46	75	26	29	30	37	30	23	38	28	12	14	15	13	11
sintef1	1336	87	61	32	43		61	65	51	41	67	37	46	34	42	43	35	46	16	53	56	29	48	22	53	35	26	27	43	44	23	15	44	11
sintef2	1277	57	59	45		31	42	43	44	69	29	71	42	68	39	41	39	48	47	40	38	46	22	29	39	48	31	28	35	21	19	27	27	13
ucla	1287	43	58		46	32	26	43	51	41	52	79	59	36	66	38	53	39	29	31	49	39	53	25	45	39	27	30	27	28	21	40	24	18
lbl	1309	51		47	57	61	49	81	54	61	28	46	46	44	43	51	28	41	47	38	39	32	59	51	32	13	30	41	27	25	14	47	21	5
connix	1365		52	43	36	87	64	43	53	36	52	68	88	34	26	43	39	39	60	37	44	33	47	41	36	25	44	30	33	33	53	27	11	8
total	35109	1339	1336	1298	1288	1286	1279	1255	1237	1231	1222	1215	1191	1183	1154	1132	1131	1099	1092	1059	1054	1044	1031	1006	998	989	913	905	887	883	794	694	564	320

Source

Figure 5.4: Number of measurements made for each Internet path, $\mathcal{R}_2$ dataset

Site	Best Guess
`wvnet-wtn9-c1.sura.net`	Charleston, WV
`128.167.205.2`	Charlottesville, VA
`reynolds-ctv1-c1.sura.net`	Charlottesville, VA
`uva-ctv-c3mb.sura.net`	Charlottesville, VA
`38.2.213.16`	New York, NY
`core.net218.psi.net`	New York, NY
`leaf.net218.psi.net`	New York, NY
`38.1.2.14`	Washington, D.C.
`core.net222.psi.net`	Washington, D.C.
`137.209.1.1`	College Park, MD
`192.80.6.2`	College Park, MD
`198.25.80.1`	College Park, MD
`199.54.78.1`	College Park, MD

Table IV: Uncertain router sites

Figures 5.3 and 5.4 summarize the number of `traceroute` measurements between each pair of sites for each of the experiments.

5.3 Geography

To understand the Internet topology traversed by the experiment, and how each router relates to others, we undertook to identify the geographic locations of the 751 routers (distinct IP addresses) involved in $\mathcal{R}_1$ and the 1,095 routers in $\mathcal{R}_2$. The identification involved several steps:

1. Routers with an Internet hostname in the same domain as one of the participating sites (e.g., `colorado.edu`) were assumed to be located at that site.

2. Routers with a single geographic location in their name (e.g., `dallas1.tx.alter.net`) were assumed to reside at that location.

3. For still-unidentified routers, we sent email to the NIC “whois” contacts [HSF85] for the router's domain, asking if they could identify the router's location or the naming scheme used for routers in that domain. The various contacts proved remarkably helpful, willing to go to considerable efforts to aid in locating the sites. We also benefited from various “whois” servers, especially the European server `whois.ripe.net` and its corresponding WAIS server, and topology maps.

4. If any still-unidentified routers only occurred as a hop between two identified sites at the same location, we assumed the router was sited at that location too. For example, if we observed a partial network path of $A \to B \to C$, with A and C both sited in San Diego, then we assumed that B was sited in San Diego too.

5. For the remainder, we made a “best guess,” based on the locations of upstream and downstream routers. Table IV summarizes the sites for which we had to guess.

Thus, of the 1,531 routers traversed during the study, we were able to identify the location of all but 13.

After locating the routers, we reduced the topology traversed by the experiment to connections between cities, listed in Table V. Having developed a geographic database for the various routers, we then constructed maps showing the links traversed in the study.[3] Figure 5.5 shows these links from a North American perspective, where sites in Hawaii, Korea, and Australia are shown west of California, and sites in Europe and Israel are shown in the Atlantic. Figure 5.6 show the links from a European perspective; here, the only links extending outside of Europe were those to sites in the U.S., which is represented as a single site west of France.

[3] Doing so first required analyzing the `traceroutes` for routing pathologies (§ 6), because “fluttering” and midstream routing changes can easily introduce spurious links.

State or Country	City
California	Anaheim, Berkeley, Bloomington, Hayward, Livermore, Los Angeles, NASA-AMES (Moffett Field), Oakland, Palo Alto, Pasadena, Sacramento, San Diego, San Francisco, San Jose, Santa Clara, Stockton
Colorado	Boulder, Colorado Springs, Denver, East Boulder
Connecticut	Hartford, Middlefield
Florida	Miami
Georgia	Atlanta
Hawaii	Honolulu
Illinois	Batavia, Chicago, Willow Springs
Maryland	College Park
Massachusetts	Boston, Cambridge, Waltham
Michigan	Detroit
Missouri	Kansas City, St. Louis
Nebraska	Lincoln
New Jersey	Pennsauken, Princeton, West Orange
New Mexico	Albuquerque, Los Alamos
New York	Albany, Armonk, Brookhaven, Buffalo, Deer Park, Ithaca, New York, Syracuse
North Carolina	Greensboro, Raleigh
Ohio	Cleveland, North Royalton
Oregon	Portland
South Carolina	Greenville
Texas	Austin, Dallas, Fort Worth, Houston
Virginia	Charlottesville, Fairfax, Falls Church, Newport News, Norfolk, Vienna
Washington, D.C.	
Washington	Kent, Seattle
West Virginia	Charleston
Australia	Adelaide, Canberra, Melbourne, Newcastle, Sydney
Austria	Vienna
Belgium	Brussels
Canada	Vancouver, Montreal, Toronto
England	Cambridge, Canterbury, London, Manchester
Finland	Helsinki
France	Lyon, Marseilles, Montpellier, Nice, Paris, Poitiers, Sophia, Toulouse
Germany	Aachen, Duesseldorf, Heidelberg, Karlsruhe, Mannheim, Munich, Stuttgart
Italy	Milan
Israel	Jerusalem, Rehovot
Korea	Pohang, Seoul
Netherlands	Amersfoort, Amsterdam, Den Bosch, Eindhoven, Nijmegen, Venlo, Utrecht
Norway	Oslo, Trondheim
Spain	Madrid
Sweden	Stockholm
Switzerland	Geneva

Table V: Router cities

Figure 5.5: Links traversed during $\mathcal{R}_1$ and $\mathcal{R}_2$, North American perspective

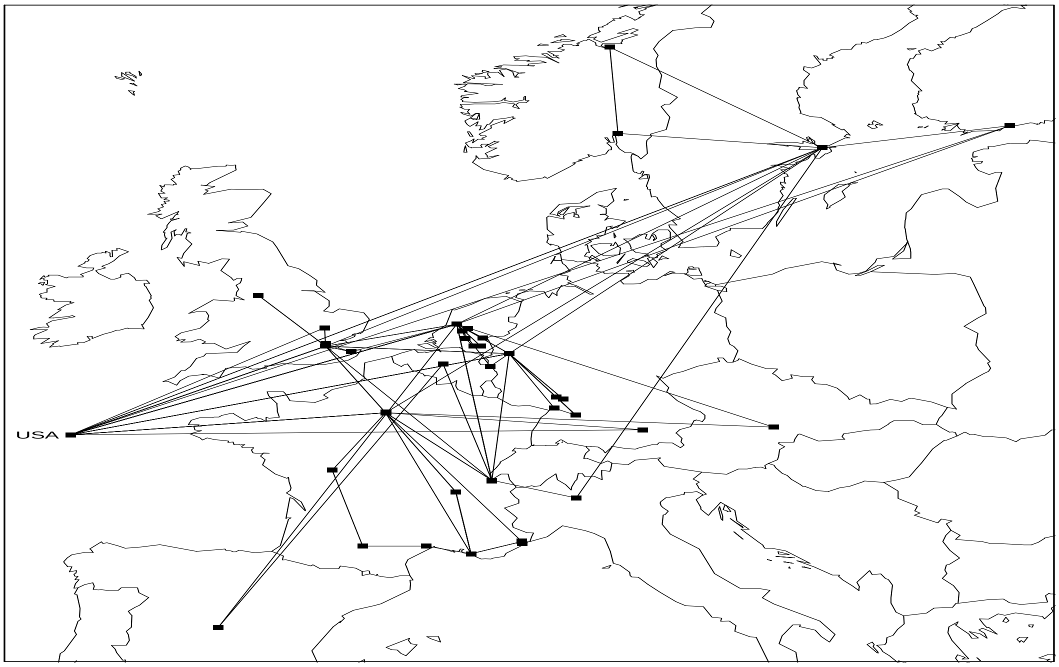

Figure 5.6: Links traversed during $\mathcal{R}_1$ and $\mathcal{R}_2$, European perspective

Chapter 6

Routing Pathologies

We begin our analysis by classifying occurrences of routing pathologies—those routes that exhibited either clear sub-standard performance, or out-and-out broken behavior.

6.1 Unresponsive routers

Some routers do not return the required ICMP messages in response to `traceroute` probes (§ 4.2.2), or do so with insufficient TTL's to make the return trip. We refer to these as *unresponsive* routers. If these routers are prevalent, they will add a great deal of noise to our measurements, making analysis difficult. This is especially the case because an unresponsive router looks identical to a router that had to drop all three probe packets due to congestion, a case we are interested in analyzing.

Fortunately, unresponsive routers are easy to spot. Unlike congested routers, unresponsive routers *consistently* fail to answer any of the `traceroute` probe packets. Because we measured multiple `traceroutes` between sites, we can look for just such consistency.[1]

Upon inspecting the `traceroutes` in $\mathcal{R}_1$, we found 4 unresponsive routers (which between them appeared in a total of 93 `traceroutes`): the last two hops prior to the `ukc` endpoint (repaired on December 8); the last hop prior to the `lbli` endpoint (frequently, but not always); and the 8th hop from `usc` to various destinations for traffic routed between CERFNET (hop 7) and AlterNet or MCINET (hop 9), consistently. This quantity of only 4 unresponsive routers contrasts with the 751 responsive routers in the first measurement set: clearly almost all Internet routers correctly return ICMP messages for expired TTL's. Furthermore, in $\mathcal{R}_2$ we did not identify *any* unresponsive routers, in contrast with 1,095 responsive routers. The previously unresponsive routers found in the first measurement set now were responsive, indicating they had been upgraded (except we were unable to determine if those on the `usc` paths had been upgraded since `usc` did not participate in the second set of measurements).[2]

---

[1] Recall that we use the term "`traceroute`" to refer to both the utility, and to an instance of a measurement made using the utility.

[2] In doing this analysis for $\mathcal{R}_2$, we encountered a strange anomaly: all of the `traceroutes` from `adv` to `ustutt` were missing the hop between `icm-dc-1-h1/0-t3.icp.net` and `amsterdam1.dante.net`. But this hop consistently appeared in other `traceroutes` to `ustutt`, identifying itself as `icm-dante-e0.icp.net`. It turned out that due to an administrative decision, `icm-dante-e0.icp.net` did not have a route to `adv`'s autonomous

6.2 Rate-limiting routers

Some routers limit the rate at which they generate ICMP messages, to conserve resources (§ 4.2.3). We can partially test for the presence of such routers in our measurements as follows. Recall that, for each hop n, `traceroute` sends three "probes" to elicit ICMP messages in reply. If the hop n router limits its ICMP generation rate, then in general it will reply to the first probe (unless it happens to already have been generating ICMP messages). This reply will lead to `traceroute` rapidly sending another probe, one whose ICMP reply will then be suppressed by the router due to rate-limiting. Since `traceroute` waits up to 5 seconds between probe packets, the third probe will not arrive until 5 seconds after the second, by which time rate-limiting again allows the router to reply. So rate-limiting routers that limit ICMP generation to on the order of one per 1-2 seconds will show up in our measurements as having a high proportion of first and third replies received, but no second reply received. We term such replies "R-*-R," reflecting their pattern.

We analyzed $\mathcal{R}_2$ to determine for each router the proportion ρ of "R-*-R" replies, limiting the analysis to routers for which we had at least 5 measurements. The distribution of ρ was sharply bimodal, with 8 routers exhibiting $\rho \geq 50\%$ and the remaining 701 all having $\rho \leq 20\%$. Of the 8 routers, 7 were endpoints: `inria`, `mid`, `nrao`, `sri`, `ustutt`, `ucl`, and `wustl`. These seven are all running the Solaris operating system, which by default is configured to do rate-limiting. The other router was `cs-gw.colorado.edu`, which, according to its DNS "HINFO" record, is a Cisco 7000. These routers support rate-limiting and apparently this one had the option activated; but we conclude that, in general, routers deployed today do not rate-limit their ICMP generation, at least not on time scales of one per 1-2 seconds.

Because we subsequently only undertake light analysis of dropped `traceroute` probes (and never endpoint drops), for simplicity we assume that all missing ICMP replies correspond to either a dropped `traceroute` probe packet or a dropped reply, and not to the effects of rate-limiting.

6.3 Routing loops

Suppose router R_1's routing tables indicate that, to forward a packet to host H, it should send the packet along a path that eventually includes router R_2. If, due to an inconsistency, R_2's tables indicate it in turn should forward the packet to H via a path that eventually includes R_1, the network contains a loop. The packet will circulate between R_1 and R_2 until either its TTL expires (§ 4.2.1), never reaching H, or the loop is broken by a routing update.

In general, routing algorithms are designed to avoid loops, provided all of the routers in the network share a consistent view of the present connectivity. Thus, loops are apt to form when the network experiences a change in connectivity and that change is not immediately propagated to all of the routers [Hu95]. One hopes that loops resolve themselves quickly, as they represent a complete failure. As long as the loop persists, end-to-end communication involving the path is impossible.

Some researchers have downplayed the significance of temporary routing loops [MRR80], and the ARPANET was subject to transitory looping "at the 1% level" [Co90]. Assuming that this

system, so its replies were always lost.

means that ARPANET paths on average contained a loop 1% of the time, then from the figures presented in this section and the next we will see that loops in the Internet occur much more rarely.

Other researchers have noted that loops can rapidly lead to congestion as a router is flooded with multiple copies of each packet it forwards [ZG-LA92], and minimizing loops is a major Internet design goal [Li89]. To this end, the Border Gateway Protocol (BGP) used between autonomous systems is designed to never allow the creation of inter-AS loops [RL95, Re95], which it accomplishes by tagging all routing information with the AS path it traversed. This technique is based on the observation that routing loops occur only when the propagation of routing *information* itself is subject to loops. The tagging allows a BGP router to determine if a peer is giving it information that the peer directly or indirectly derived from the router itself. If so, the router discards the information.

In this section we analyze our measurements for the prevalence of routing loops. We classify these loops as two types, "persistent" if they lasted longer than the `traceroute` measurement, or "temporary" if they resolved within the span of the `traceroute` observing them. The next two subsections look at these two types, and the final subsection comments on the location of the loops within the network.

6.3.1 Persistent routing loops

A persistent routing loop is easy to detect in a `traceroute`. Here is an example of a loop between `lbl` and `lbli`, ordinarily 6 hops apart:

```
 1  ir6gw.lbl.gov  1.853 ms  1.623 ms  2.358 ms
 2  er1gw.lbl.gov  7.165 ms  2.996 ms  3.098 ms
 3  ir2gw.lbl.gov  4.882 ms  3.516 ms  8.371 ms
 4  isdn1gw.lbl.gov  7.98 ms  4.393 ms  4.311 ms
 5  ascend49.lbl.gov  36.833 ms  32.772 ms  31.428 ms
 6  isdn1gw.lbl.gov  30.428 ms  30.502 ms  33.528 ms
 7  ascend49.lbl.gov  69.006 ms  59.429 ms  58.82 ms
 8  isdn1gw.lbl.gov  59.358 ms  63.734 ms  61.775 ms
 9  ascend49.lbl.gov  85.629 ms  84.168 ms  83.397 ms
10  isdn1gw.lbl.gov  83.374 ms  83.201 ms  83.349 ms
11  ascend49.lbl.gov  110.316 ms  120.243 ms  116.84 ms
12  isdn1gw.lbl.gov  109.221 ms  108.97 ms  109.242 ms
13  ascend49.lbl.gov  135.867 ms  136.797 ms  140.849 ms
14  isdn1gw.lbl.gov  137.359 ms  138.757 ms  137.028 ms
15  ascend49.lbl.gov  171.109 ms  167.197 ms  168.027 ms
16  isdn1gw.lbl.gov  187.18 ms  177.017 ms  165.499 ms
17  ascend49.lbl.gov  199.461 ms  193.441 ms  201.067 ms
18  isdn1gw.lbl.gov  191.205 ms  198.674 ms  192.041 ms
19  ascend49.lbl.gov  228.833 ms  219.05 ms  240.464 ms
20  isdn1gw.lbl.gov  213.537 ms  214.975 ms  220.435 ms
21  ascend49.lbl.gov  249.681 ms  254.247 ms  243.089 ms
22  isdn1gw.lbl.gov  239.341 ms  239.072 ms  243.516 ms
23  ascend49.lbl.gov  268.134 ms  270.585 ms  267.982 ms
24  isdn1gw.lbl.gov  273.742 ms  274.974 ms  265.043 ms
25  ascend49.lbl.gov  297.033 ms  293.392 ms  294.328 ms
26  isdn1gw.lbl.gov  348.844 ms  303.868 ms  291.552 ms
```

Source	Dest.	Loop	Location
ucol	bnl	129.19.253.18, 129.19.253.17	Col. State Univ.
austr mit	umann umann	mf-0.enss145.t3.ans.net, umd-rt1.es.net same	FIX-East
lbli	xor	icm-fix-e-h2/0-t3.icp.net, icm-dc-2b-h3/0-t3.icp.net	FIX-East, Washington D.C.
lbl	lbli	isdn1gw.lbl.gov, ascend49.lbl.gov (this loop occurred twice)	LBL
lbl	inria	llnl-e-llnl2.es.net, llnl2-e-llnl.es.net	Livermore, California
sdsc	ukc	gw.ukc.ac.uk, gw.ulcc.ja.net	London, Canterbury
sdsc	usc	mobydick.cerf.net, drzog.cerf.net	SDSC
harv	ucl	mf-0.cnss56.washington-dc.t3.ans.net, mf-0.cnss58.washington-dc.t3.ans.net	Washington, D.C.

Table VI: Persistent routing loops in $\mathcal{R}_1$

```
27  ascend49.lbl.gov  335.637 ms  324.15 ms  322.982 ms
28  isdn1gw.lbl.gov  328.654 ms  321.418 ms  316.452 ms
29  ascend49.lbl.gov  344.561 ms  351.843 ms  346.087 ms
30  isdn1gw.lbl.gov  358.938 ms  348.781 ms  355.01 ms
```

`isdn1gw.lbl.gov` is the Laboratory's ISDN gateway, and `ascend49.lbl.gov` is the other end of the ISDN link to `lbli`. Here, `ascend49.lbl.gov` apparently has lost track of the notion that `lbli` resides on its side of the ISDN point-to-point link, so it forwards any packets for `lbli` back to the ISDN gateway.

For our analysis, we considered any `traceroute` showing a loop that was not resolved by the end of the `traceroute` (i.e., after probing 30 hops) as a "persistent loop." Of the 6,204 `traceroutes` in $\mathcal{R}_1$,[3] 10 exhibited persistent routing loops. Table VI summarizes these.

Three of these loops appear to have formed *during* the `traceroute` probe. In the `harv` ⇒ `ucl` loop, for example, the probes made it to London and almost to the `ucl` endpoint before the loop appeared in Washington, D.C., at hop 16:

```
 1  glan-gw.harvard.edu  87 ms  3 ms  2 ms
 2  wjhgw1.harvard.edu  4 ms  2 ms  2 ms
 3  harvard-gw.near.net  8 ms  11 ms  4 ms
 4  prospect-gw.near.net  20 ms  20 ms  12 ms
 5  tang-gw.near.net  32 ms  6 ms  6 ms
 6  enss.near.net  6 ms  6 ms  3 ms
 7  t3-3.cnss48.hartford.t3.ans.net  7 ms  9 ms  11 ms
 8  t3-2.cnss32.new-york.t3.ans.net  9 ms  10 ms  10 ms
 9  t3-1.cnss56.washington-dc.t3.ans.net  18 ms  16 ms  20 ms
```

[3]This number represents the 6,459 total `traceroutes`, minus 255 `traceroutes` originating from `wustl`, which, as explained in § 6.6.2, suffered from a large degree of "fluttering," making it difficult to determine whether true routing loops were also present.

```
10  mf-0.cnss58.washington-dc.t3.ans.net  15 ms  17 ms  16 ms
11  washington2.dante.net  20 ms  15 ms  19 ms
12  icm-dc-1-e4/0.icp.net  75 ms  58 ms  77 ms
13  icm-london-1-s1-1984k.icp.net  144 ms  218 ms  127 ms
14  smds-gw.ulcc.ja.net  230 ms  161 ms  146 ms
15  smds-gw.ucl.ja.net  131 ms  155 ms  138 ms
16  cisco-pb.ucl.ac.uk 1566 ms
      * mf-0.cnss58.washington-dc.t3.ans.net 53 ms
17  mf-0.cnss56.washington-dc.t3.ans.net  58 ms  58 ms  55 ms
18  mf-0.cnss58.washington-dc.t3.ans.net  66 ms  61 ms  60 ms
19  mf-0.cnss56.washington-dc.t3.ans.net  62 ms  68 ms  68 ms
etc.
```

In the `sdsc` ⇒ `usc` loop, the loop formed just one hop from the SDSC source, after the probe had already made it from San Diego to Los Angeles:

```
 1  drzog.cerf.net  163 ms  2 ms  2 ms
 2  134.24.120.102  7 ms  8 ms  7 ms
 3  * ucla-la-smds.cerf.net  66 ms  19 ms
 4  * losnet.ucla.edu  16 ms  16 ms
 5  isi-ucla-gw.ln.net  57 ms  20 ms  18 ms
 6  * * mobydick.cerf.net  9 ms
 7  drzog.cerf.net  13 ms  9 ms  7 ms
 8  mobydick.cerf.net  9 ms  10 ms  9 ms
 9  drzog.cerf.net  10 ms  11 ms  21 ms
10  mobydick.cerf.net  13 ms  32 ms  11 ms
etc.
```

The presence of packet loss (`*`'s) prior to the loop forming at hops 6–7 may indicate connectivity deteriorating prior to a routing change (which led to an inconsistent state). A similar loss can be seen in the `harv` ⇒ `ucl` example above, at hop 16.

The `lbl` ⇒ `inria` loop entailed two separate loops:

```
 1  ir6gw.lbl.gov  1.858 ms  1.66 ms  1.546 ms
 2  er1gw.lbl.gov  3.68 ms  2.423 ms  2.244 ms
 3  lbl-lc2-1.es.net  3.252 ms  2.618 ms  2.645 ms
 4  llnl-lbl-t3.es.net  5.892 ms  4.634 ms  3.985 ms
 5  lanl-llnl-t3.es.net  34.728 ms  29.444 ms  30.195 ms
 6  snla-lanl-t3.es.net  61.712 ms  60.392 ms  60.347 ms
 7  pppl-fnal-t3.es.net  78.807 ms  79.19 ms  77.252 ms
 8  pppl-nis.es.net  79.454 ms  78.5 ms  78.166 ms
 9  umd-pppl.es.net  85.851 ms  105.744 ms  89.141 ms
10  icm-fix-e-f0.icp.net  129.442 ms  86.567 ms  88.157 ms
11  * * *
12  * * llnl-lanl-t3.es.net  321.099 ms
13  lanl-llnl-t3.es.net  577.496 ms  199.259 ms  134.383 ms
14  llnl-lanl-t3.es.net  134.854 ms  135.204 ms  134.909 ms
15  lanl-llnl-t3.es.net  160.895 ms  160.312 ms  162.187 ms
16  llnl-lanl-t3.es.net  161.882 ms  315.869 ms *
17  * * *
18  * * *
```

```
19  * * *
20  * * *
21  * * *
22  * * *
23  * * *
24  llnl2-e-llnl.es.net  17.051 ms  26.225 ms  22.082 ms
25  llnl-e-llnl2.es.net  21.823 ms  15.619 ms  21.804 ms
26  llnl2-e-llnl.es.net  16.693 ms  22.776 ms  26.126 ms
27  llnl-e-llnl2.es.net  23.758 ms  19.809 ms  22.475 ms
etc.
```

The first sign of trouble is at hop 11, where, after having made it to FIX-East in Maryland at hop 10, the network begins dropping probe packets (or their responses). At hop 12, a temporary routing loop forms between the ESNET routers in Livermore, California, and Los Alamos, New Mexico. This loop appears to lead to further problems at the end of hop 16,[4] where subsequent packets are lost for nearly 2 minutes (recall that each `*' represents a lost response, including a 5-second wait). Finally, at hop 24 the network comes back, but in an inconsistent state, with a consequent routing loop. Most likely the routing inconsistency leading to the first loop was propagated through ESNET to form the second loop.

In $\mathcal{R}_2$, 50 `traceroutes` showed persistent loops. Due to $\mathcal{R}_2$'s higher sampling frequency, for some of these loops we can place bounds on how long they persisted, by looking for surrounding measurements between the same hosts that do not show the loop. In addition, sometimes the surrounding measurements *do* show the loop—these allow us to put lower bounds on the loop's duration, too.

Table VII summarizes the loops seen in $\mathcal{R}_2$. The first two columns give the source and destination of the `traceroute`, the next column the date, and the fourth column the number of consecutive `traceroutes` that encountered the loop. The fifth and sixth columns give the routers involved in the loop and the geographic location. Note that only one of the loops spanned multiple cities (and multiple continents!), the last in the table.

The final column gives the bounds we were able to assess for the duration of the loop. Upper bounds indicate the difference in time between the two non-looping `traceroutes` bracketing the loop, if this difference was less than 1 day (otherwise the upper bound is potentially so lax that we omit it). Lower bounds, when present, indicate the difference in time between the first `traceroute` in a sequence observing the loop, and the last. For loops only observed during a single `traceroute`, this bound is omitted. Loops for which we were unable to assign any plausible bounds have their bounds marked as "?".

The loop durations appear to fall into two modes, those definitely under 3 hours (and possibly quite shorter), and those of more than half a day. The presence of persistent loops of durations on the order of hours to tens of hours is quite surprising, and suggests a lack of good tools for diagnosing network problems: neither the NOC's (Network Operation Centers) responsible for the looped routers, nor the customers, apparently discovered and repaired the loops for considerable periods of time, despite the total connectivity outage due to the loop.

We also note a tendency for persistent loops to come in clusters. Geographically, loops occurred much more often in the Washington D.C. area (MAE-East and College Park are only a

[4] So the loop persisted for about 2.5 seconds, as indicated by summing the return times for each of the probe packets.

Source	Dest.	Date	#	Loop	Location	Duration
inria	adv	Nov. 6	1	icm-dc-1-f0/0.icp.net, icm-dc-2b-f2/0.icp.net	Washington	?
inria	near	Nov. 11	1	same as above	Washington	$\leq$ 3 hr
wustl	inria	Nov. 24	1	same as above	Washington	?
inria	pubnix	Nov. 12	1	icm-dc-3-f2/0.icp.net, icm-dc-2b-f2/0.icp.net	Washington	?
inria	austr2	Nov. 15	1	same as above	Washington	?
sintef1	adv	Nov. 12	1	icm-pen-1-h1/0-t3.icp.net, icm-dc-2b-h0/0-t3.icp.net	Washington	?
pubnix	sintef1	Nov. 8	1	sl-ana-1-f0/0.sprintlink.net, sl-ana-2-f0/0.sprintlink.net	Anaheim	?
ustutt	ucl	Nov. 11	16	stuttgart1.belwue.de, stuttgart4.belwue.de	Stuttgart	16–32 hr
connix	bsdi	Nov. 14	1	sl-dc-8-h1/0-t3.sprintlink.net, sl-mae-e-h2/0-t3.sprintlink.net	MAE-East	$\geq$ 10 hr
ustutt	austr	Nov. 14	1	same as above		
pubnix	sintef1	Nov. 14	1	fddi0/0.cr1.dca1.alter.net, cisco1.washington.dc.ms.uu.net	Washington	$\leq$ 5.5 hr
austr	nrao	Nov. 15	1	cpk8-cpk-cf.sura.net, cpk9-cpk-cf.sura.net	College Park	?
many	oce	Nov. 23	12	amsterdam.nl.net, wgm01.nl.net	Amsterdam	14–17 hr
ucol	ustutt	Nov. 24	1	borderx1-hssi3-0.sanfrancisco.mci.net pacbell-nap-atm.sanfrancisco.mci.net	San Francisco	?
ucol	inria	Nov. 27	1	stamand1.renater.ft.net, stamand3.renater.ft.net	Paris	$\leq$ 14 hr
mid	bsdi	Nov. 28	1	sl-dc-6-f0/0.sprintlink.net, sl-dc-8-f0/0.sprintlink.net	Washington	$\leq$ 3 hr
mid	austr	Dec. 6	1	sl-chi-6-h3/0-t3.sprintlink.net, sl-chi-nap-h1/0-t3.sprintlink.net	Chicago	$\leq$ 3 hr
mit	wustl	Dec. 10	1	starnet2.starnet.net, starnet8.starnet.net	St. Louis	?
umann	nrao	Dec. 13	1	heidelberg1.belwue.de, heidelberg2.belwue.de	Heidelberg	?
ucl	mit	Dec. 14	1	mci-its.near.net, w91-rtr-external-fddi.mit.edu	Cambridge	$\leq$ 3 hr
near	ucla	Dec. 16	1	ln-gw.cs.ucla.edu, ucla-isi-gw.ln.net	Los Angeles	?
sri	near	Dec. 17	1*	su-a.bbnplanet.net, su-b.bbnplanet.net	Palo Alto	?
near	sri	same	1*	barrnet.sanfrancisco.mci.net, borderx1-hssi2-0.sanfrancisco.mci.net	San Francisco	?
bsdi	sintef1	Dec. 21	1	icm-pen-2-h2/0-t3.icp.net, icm-uk-1-h0/0-t3.icp.net	Pennsauken, London	$\leq$ 10 hr

Table VII: Persistent routing loops in $\mathcal{R}_2$

few miles away), perhaps because the very high degree of interchange between different network service providers in that area offers ample opportunity for introducing inconsistencies.

Loops involving separate pairs of routers also are clustered in time. The `pubnix` ⇒ `sintef1` loop, involving two AlterNet routers sited in Washington D.C., was measured at the same time between the `connix` ⇒ `bsdi` and `ustutt` ⇒ `austr` observations of a SprintLink loop, at nearby MAE-East. The `sri` ⇒ `near` and `near` ⇒ `sri` loop observations were made back-to-back. They do *not* observe the same loop, but rather two separate loops between closely related routers (the typical path from `near` to `sri` proceeds from MCINET in San Francisco immediately to BARRNET at Stanford (Palo Alto), and then at the next hop to BBN Planet at Stanford). Thus, it appears that the inconsistencies that lead to long-lived routing loops are not confined to a single pair of routers but also affect nearby routers, tending to introduce loops into their tables too. This in turn suggests that any persistent loop encountered in the network is very serious, as it may reflect a substantially larger outage than just the two looped routers initially observed.

6.3.2 Temporary routing loops

Fortunately, routing loops do not always persist for long periods of time. In addition to analyzing the `traceroute` data for persistent loops, we also looked for temporary loops. We define a temporary loop as one during which a router was visited at different hops, yet eventually the `traceroute` probe traveled beyond the loop. This definition requires manual inspection of the candidates, to remove spurious "loops" that are in reality due instead to other factors, such as "fluttering" (rapidly-variable routing; § 6.6.2) or midstream route changes (§ 6.5).

The `lbl` ⇒ `inria` example in the previous section shows both a temporary loop and a permanent loop, both involving ESNET routers. In addition to the `lbl` ⇒ `inria` example above, $\mathcal{R}_1$ exhibited one other case of a temporary routing loop, occurring between `ucl` and `wustl`:

```
 1  cisco.cs.ucl.ac.uk  12 ms  5 ms  5 ms
 2  cisco-pb.ucl.ac.uk  11 ms  4 ms  4 ms
 3  cisco-b.ucl.ac.uk  5 ms  4 ms  5 ms
 4  gw.lon.ja.net  20 ms  22 ms  19 ms
 5  eu-gw.ja.net  60 ms  21 ms  19 ms
 6  icm-lon-1.icp.net  20 ms  25 ms  37 ms
 7  icm-dc-1-s3/2-1984k.icp.net  177 ms  191 ms  168 ms
 8  * sl-dc-7-f0.sprintlink.net  1174 ms  183 ms
 9  sl-starnet-1-s0-t1.sprintlink.net  220 ms  216 ms  233 ms
10  * * *
11  * * *
12  stl2-e0.starnet.net  506 ms  775 ms  262 ms
13  stl3-e0.starnet.net  218 ms * *
14  stl2-e0.starnet.net  919 ms *  237 ms
15  * stl3-e0.starnet.net  193 ms  191 ms
16  * * *
17  * * *
18  * * *
19  * * *
20  * * *
21  * tango.cs.wustl.edu  260 ms *
```

Here, at hops 12-15, the STARnet routers engage in a short-term routing loop that evidently is resolved during hops 16-20 (an outage of about 80 seconds).[5]

While in $\mathcal{R}_1$ we only observed two temporary loops, in $\mathcal{R}_2$ we found 23. We confine ourselves here to a look at two of the more seriously pathological, as these illustrate the degree to which routing can degrade.

The first of these was from `rain` to `inria`:

```
 1  r0.pdx.rain.rg.net  3.212 ms  2.903 ms  2.348 ms
 2  border1-serial2-5.seattle.mci.net  8.119 ms  7.509 ms  8.303 ms
 3  core-fddi-0.seattle.mci.net  10.255 ms  11.472 ms  9.087 ms
 4  core2-hssi-3.denver.mci.net  42.005 ms  45.637 ms  41.765 ms
 5  core1-aip-4.denver.mci.net  180.353 ms  210.453 ms  222.771 ms
 6  core2-hssi-2.westorange.mci.net  192.796 ms  224.263 ms  257.99 ms
 7  core2-hssi-2.washington.mci.net  96.183 ms  90.611 ms  90.897 ms
 8  borderx2-fddi-1.washington.mci.net  88.917 ms  98.286 ms  99.512 ms
 9  mae-east-plusplus-two.washington.mci.net
      95.96 ms  111.302 ms  121.937 ms
10  icm-dc-e-f0/0.icp.net  91.077 ms  102.348 ms  95.265 ms
11  * * *
12  * * *
13  * * borderx2-fddi-1.washington.mci.net  269.431 ms
14  mae-east-plusplus-two.washington.mci.net
      440.782 ms  293.266 ms  166.355 ms
15  mae-east-plusplus.washington.mci.net
      89.681 ms  94.609 ms  90.987 ms
16  borderx1-hssi2-0.washington.mci.net  91.661 ms  89.673 ms  96.562 ms
17  core2-fddi-0.washington.mci.net  137.351 ms  174.362 ms  204.639 ms
18  borderx2-fddi-1.washington.mci.net  95.169 ms  90.19 ms  94.371 ms
19  mae-east-plusplus-two.washington.mci.net
      97.839 ms  91.079 ms  97.236 ms
20  mae-east-plusplus.washington.mci.net  92.483 ms  91.213 ms  91.38 ms
21  borderx1-hssi2-0.washington.mci.net  92.318 ms  92.662 ms  95.358 ms
22  * * *
23  r0.pdx.rain.rg.net  3.343 ms !H * *
24  * t8-gw.inria.fr  779.58 ms *
25  tom.inria.fr  657.659 ms * *
```

The `traceroute` begins without any problems, traveling to ICP (the Sprint/NSF International Connectivity Project) in Washington via Seattle, Denver, West Orange (New Jersey), Washington, and MAE-East. At hop 11, however, we observe a 40 second outage. Evidently the outage was due to the loss of the link between `mae-east-plusplus-two.washington.mci.net` and `icm-dc-e-f0/0.icp.net`, because when the outage finished, we find ourselves in a routing loop between five different routers:

```
borderx2-fddi-1.washington.mci.net
mae-east-plusplus-two.washington.mci.net
mae-east-plusplus.washington.mci.net
```

[5] As discussed in § 6.6.2 below, these STARnet routers also suffered from route "fluttering," though that problem was apparently fixed on December 12, and this trace is from December 15, after the repair.

```
borderx1-hssi2-0.washington.mci.net
core2-fddi-0.washington.mci.net
```

This is one of only two times in either $\mathcal{R}_1$ or $\mathcal{R}_2$ that we observed a loop involving more than two routers. (The other is discussed in § 6.4.) The loop persists from hop 13 to hop 21 (at least). At hop 22 we suffer a 15 second outage, and when it resolves we find ourselves all the way back to where we started at hop 1. The router there has returned an "ICMP unreachable" message (the `!H`), indicating it is convinced that it cannot reach `inria`, presumably because it has lost its link to `border1-serial2-5.seattle.mci.net`. After another 15 second outage, however, we suddenly find ourselves in France, at `inria`'s doorstep: either both of the previous problems had resolved themselves, or an alternate path was discovered.

The second seriously pathological `traceroute` was from `ucol` to `umann`:

```
 1  cs-gw-srl.cs.colorado.edu  3 ms  3 ms  2 ms
 2  cu-gw-fddi.colorado.edu  5 ms  2 ms  4 ms
 3  ncar-cu.co.westnet.net  13 ms  4 ms  8 ms
 4  ml-t3-gw.ucar.edu  11 ms  24 ms  34 ms
 5  border2-hssi1-0.denver.mci.net  73 ms  141 ms  87 ms
 6  core-fddi-1.denver.mci.net  80 ms  22 ms  24 ms
 7  * core2-hssi-2.westorange.mci.net  47 ms  64 ms
 8  core2-hssi-2.washington.mci.net  58 ms  63 ms  59 ms
 9  borderx2-fddi-1.washington.mci.net  73 ms  98 ms  111 ms
10  mae-east-plusplus-two.washington.mci.net  60 ms  64 ms  60 ms
11  icm-dc-e-f0/0.icp.net  112 ms  99 ms  91 ms
12  icm-dc-1-h1/0-t3.icp.net  81 ms  94 ms  105 ms
13  icm-dante-e0.icp.net  115 ms  150 ms *
14  * amsterdam1.dante.net  205 ms *
15  nl-s1.dante.bt.net  177 ms  166 ms  151 ms
16  nl-f0-0.eurocore.bt.net  172 ms  190 ms  176 ms
17  de-s1-1.eurocore.bt.net  206 ms  247 ms  227 ms
18  de-f0.dante.bt.net  251 ms  181 ms  227 ms
19  * * *
20  * * *
21  * icm-dc-2b-f2/0.icp.net  151 ms  138 ms
22  icm-dc-1-f0/0.icp.net  97 ms  86 ms  64 ms
23  icm-dc-2b-f2/0.icp.net  98 ms  85 ms  107 ms
24  icm-dc-1-f0/0.icp.net  109 ms  92 ms umd2-pppl2.es.net  251 ms
25  * mae-east-plusplus-two.washington.mci.net  178 ms  251 ms
26  pppl2-umd2.es.net  702 ms * *
27  core-hssi-3.sanfrancisco.mci.net  158 ms !H *
    core-fddi-1.denver.mci.net  34 ms !H
```

Everything is fine up until hop 18, with the path traversing from Boulder to Denver, in Colorado; then over MCINET to West Orange and down to MAE-East, then across to Amsterdam and over to Duesseldorf—almost there! But a 35 second outage at hops 19–21 is the beginning of trouble. When the network begins responding again, we have fallen back to a temporary loop between `icm-dc-1-f0/0.icp.net` and `icm-dc-2b-f2/0.icp.net` in Washington, D.C., a position similar to that we had achieved at hops 11-12 earlier. At hop 25 we again visit `mae-east-plusplus-two.washington.mci.net`, already visited at hop 10. Note two things about this hop. First, we have now backtracked twice, once to `icm-dc-2b-f2/0.icp.net`, and then again to MAE-East, which is an earlier hop than ICM in Washington. Second, we have acquired an additional *15 hops* to our route *upstream* of MAE-East, so along with the routing loop in Washington, there is also a major change closer to `ucol`. At hop 26 we find ourselves on ESNET, but at hop 27 we initially are rerouted to San Francisco on MCINET, indicating *another* upstream change (since ESNET does not have a link from Princeton to MCI in San Francisco). This router indicates that it knows of an immediate outage by flagging the hop using `!H`. But only five seconds later we lose connectivity even to San Francisco—we are back in Denver again, as we were at hop 6, and unable to make any further progress (the router flags `!H`).

Clearly at least two different major failures occurred in this example, one the routing loop at `icm-dc-2b-f2/0.icp.net`, and the other the rapidly changing (and lengthening) path upstream from MAE-East. In the previous example, the same applies: we observed both a routing loop in Washington, and a connectivity outage between Portland and Seattle. A very interesting question is whether these failures were actually reflections of a single underlying catastrophe that propagated through the network at large.

All in all we observed 20 instances of multiple large-scale changes such as illustrated in this example, suggesting that either the propagation of a single fault's effects through the network sometimes leads to widespread, temporary instability, or that a mechanism separate from the exchange of routing information is producing widespread faults. Determining which of these is the case and how the fault propagates would make for interesting future work.

6.3.3 Location of routing loops

We analyzed the routers involved in temporary and persistent loops to see whether any of the loops involved more than one AS. As mentioned above, the design of BGP in theory prevents any inter-AS loops, by preventing any looping of routing information. We found that only three of the $\mathcal{R}_1$ loops spanned more than one AS, and only two of those in $\mathcal{R}_2$. We also learned that at least one of the inter-AS loops in $\mathcal{R}_2$ occurred due to the presence of a static route, and thus clearly was not the fault of BGP. It may be that the others have similar explanations. In any event, it appears clear from our data that BGP loop suppression virtually eliminates inter-AS looping.

6.4 Erroneous routing

A final example of a routing loop occurred during a `connix` ⇒ `ucl` `traceroute`, which also exhibits *erroneous* routing, where the packets clearly took the wrong path:

```
 1  mfd-01.rt.connix.net  8 ms  4 ms  3 ms
 2  sl-dc-5-s2/0-512k.sprintlink.net  39 ms  39 ms  39 ms
 3  sl-dc-6-f0/0.sprintlink.net  39 ms  38 ms  50 ms
 4  psi-mae-east-1.psi.net  48 ms  66 ms *
 5  * * core.net218.psi.net  90 ms
 6  192.91.187.2  1139 ms  1188 ms *
 7  * * *
 8  biu-tau.ac.il  1389 ms * *
 9  tau.man.ac.il  1019 ms * *
10  * * *
```

```
11  * cisco301s1.huji.ac.il  1976 ms *
12  * * *
13  * * *
14  * * cisco101e5.huji.ac.il  1974 ms
15  * * *
16  * cisco103e2.gr.huji.ac.il  1010 ms  1069 ms
17  cisco101e01.cc.huji.ac.il  2132 ms * *
18  cisco102e13.huji.ac.il  888 ms  976 ms  2005 ms
19  cisco103e2.gr.huji.ac.il  1657 ms * *
20  * * cisco101e01.cc.huji.ac.il  1349 ms
21  * * *
etc.
```

Recall that `connix` is sited in Middlefield, Connecticut, and `ucl` in London, England. Yet at hop 6, instead of routing towards London, the route winds up visiting `192.91.187.2` as the next hop—`192.91.187.2` is sited at the Weizmann Institute in Rehovot, Israel! (As can be seen by the long latency to hop 6, a satellite link is involved here.) Not surprisingly, the bewildered Israeli routers do not really know what to make of the London-bound packet: it enters a routing loop between `cisco101e01.cc.huji.ac.il`, `cisco102e13.huji.ac.il`, and `cisco103e2.gr.huji.ac.il` prior to being discarded. The lack of any response to `traceroute` probes beyond hop 20 may be due to the route being terminated further upstream, or because growing congestion on the US–Israel link led to subsequent probes getting dropped.

There is a security lesson to be considered here, too: one really cannot make any safe assumptions about where one's packets might travel on the Internet. If the Israeli routers had an alternate path to London available to them, it is possible that this highly circuitous route would have succeeded (cf. § 6.9).

6.5 Connectivity altered mid-stream

In 10 of the $\mathcal{R}_1$ traces we observed routing connectivity reported earlier in the `traceroute` later lost or altered, indicating we were catching a routing failure as it happened:

```
 1  netlab1-gw.usc.edu  3 ms  3 ms  3 ms
 2  rtr1.usc.edu  3 ms  2 ms  2 ms
 3  isi-usc-gw.ln.net  5 ms  4 ms  5 ms
 4  ucla-isi-gw.ln.net  121 ms  230 ms *
 5  * * *
 6  * * *
 7  * * *
 8  * * *
 9  * rtr1.usc.edu  2 ms !H *
10  * * *
11  rtr1.usc.edu  2 ms !H * *
12  * * *
13  rtr1.usc.edu  2 ms !H *  2 ms !H
```

In this trace from `usc` to `ucol`, by hop 4 the packets have made it from `usc` out to the UCLA/ISI Los Nettos gateway. The large round-trip times reported at hop 4 indicate trouble, however,[6] and after the second hop 4 reply, connectivity is lost for about 70 seconds. When it returns, connectivity is only present to the hop 2 router, which reports that the destination host is unreachable (the "`!H`" flag). Because the recovery only extends to the 2nd hop, we infer that the problem occurred not at the hop 4 router but rather at hop 3, the gateway between USC and ISI.

In the other traces, a connectivity loss was followed by a recovery, as shown in this `traceroute` between `bnl` and `usc`:

```
 1  cerberus.90.bnl.gov  2 ms  2 ms  2 ms
 2  nioh.bnl.gov  3 ms  2 ms  4 ms
 3  192.12.15.224  3 ms  2 ms  2 ms
 4  pppl-bnl.es.net  11 ms  11 ms  14 ms
 5  * * *
 6  * 192.12.15.224  4 ms !H *
 7  * 192.12.15.224  3 ms !H *
 8  * 192.12.15.224  5 ms !H *
 9  * * *
10  * * *
11  * 192.12.15.224  4 ms !H *
12  * 192.12.15.224  84 ms !H *
13  * * *
14  * usc-cit-gw.ln.net  563 ms  257 ms
15  rtr5.usc.edu  283 ms  317 ms  242 ms
16  catarina.usc.edu  282 ms  102 ms  211 ms
17  escondido.usc.edu  199 ms  306 ms  392 ms
```

Router `192.12.15.224` is located at the `bnl` site. At hop 5, it clearly loses its link to `pppl-bnl.es.net`, and the link does not return for two minutes. Once it does, the `traceroute` probes are able to continue all the way to `usc`.

Three additional $\mathcal{R}_1$ traces revealed similar high-delay recoveries, incurring outages ranging from about 1 minute to almost 5 minutes. One striking example is from `wustl` to `ucol`:

```
 1  jcr-166.cs.wustl.edu  5 ms  2 ms  2 ms
 2  ncrc-eng.wustl.edu  3 ms  2 ms  2 ms
 3  128.252.5.120  3 ms  3 ms  2 ms
 4  128.252.1.2  4 ms  4 ms  3 ms
 5  sl-dc-7-s7-t1.sprintlink.net  30 ms  28 ms  28 ms
 6  sl-dc-6-f0/0.sprintlink.net  81 ms  27 ms  33 ms
 7  sl-dc-8-f0/0.sprintlink.net  106 ms  37 ms  30 ms
 8  * * *
 9  * * sl-dc-8-f0/0.sprintlink.net  32 ms !H
10  * * *
11  * * *
12  * * *
13  * * *
```

[6] Between `usc` and `ucol` this hop usually had a latency of 5-10 msec. We did not, however, undertake any rigorous evaluation of hop latencies, because of the potentially large noise associated with these times, as discussed in § 4.2.2, and as illustrated above.

```
14  * * *
15  * * *
16  * * *
17  * * *
18  * * *
19  * * *
20  * * *
21  * * *
22  * * *
23  * * *
24  * * *
25  clark.cs.colorado.edu  128 ms  106 ms  105 ms
```

Here, connectivity was lost for between 15-17 hops. At first it might appear from this `traceroute` that the route upon recovery consisted of 25 hops, but that is instead a measurement artifact: by the time the network had recovered, the `traceroute` hop-count had ratcheted so high that the first successful probes following the outage made it all the way to the `ucol` endpoint. They no doubt would also have done so if they had been transmitted with somewhat lower TTL's.

Two other traces revealed different, quite quick recovery behavior:

```
 1  netlab1-gw.usc.edu  3 ms  3 ms  3 ms
 2  rtr1.usc.edu  4 ms  3 ms  3 ms
 3  cit-usc-gw.ln.net  8 ms  3 ms  4 ms
 4  cerfnet-cit-gw.ln.net  17 ms  23 ms  6 ms
 5  sdsc-cit.cerf.net  84 ms  39 ms  21 ms
 6  mobydick.cerf.net  30 ms  37 ms  35 ms
 7  ucop-sdsc-2.cerf.net  85 ms  43 ms  50 ms
 8  sl-ana-3-s2/6-t1.sprintlink.net  68 ms  86 ms  84 ms
 9  sl-ana-1-f0/0.sprintlink.net  94 ms  72 ms  53 ms
10  sl-fw-6-h2/0-t3.sprintlink.net  100 ms  99 ms  62 ms
11  sl-fw-2-f0.sprintlink.net  120 ms  130 ms  132 ms
12  sl-colorado-1-s0-t1.sprintlink.net  146 ms  151 ms  172 ms
13  * t3-0.cnss56.washington-dc.t3.ans.net  121 ms  140 ms
14  t3-0.enss145.t3.ans.net  132 ms  127 ms  120 ms
15  icm-fix-e-f0.icp.net  155 ms  129 ms  306 ms
16  icm-dc-2b-h3/0-t3.icp.net  370 ms  137 ms  148 ms
17  sl-dc-8-f0/0.sprintlink.net  127 ms  144 ms  145 ms
18  * sl-fw-5-h4/0-t3.sprintlink.net  334 ms  211 ms
19  sl-fw-2-f0.sprintlink.net  156 ms  183 ms  157 ms
20  sl-colorado-1-s0-t1.sprintlink.net  202 ms *  199 ms
21  gw2.boulder.co.coop.net  179 ms  193 ms  189 ms
22  bandicoot.xor.com  237 ms  199 ms  210 ms
```

The path here is from `usc` to `xor`. It looks fairly straight-forward, suffering only three isolated losses, but observe that hop 11 and hop 19 are identical! (As are hops 12 and 20.) The `sl-colorado-1-s0-t1.sprintlink.net` router is only two hops from the destination, `bandicoot.xor.com`, so apparently this `traceroute` was on the verge of reaching its destination at hop 14 (and indeed two of the other `usc` $\Rightarrow$ `xor` `traceroutes` took only 14 hops) when a routing change occurred upstream, forcing the packets to detour all the way to the East coast of the U.S. on their trip from California to Colorado. In contrast to the examples in the previous section, in this case the routing change occurred quite smoothly, with only a single packet loss at hop 13 indicating a 5-second outage during the switch-over.

By inspecting other `usc` routes involving `t3-0.cnss56.washington-dc.t3.ans.net` at hop 13, we conclude that the change occurred at hop 10, where instead of routing from Anaheim, California to Fort Worth, Texas, as shown above, and staying inside Sprintlink, the switch was made to route to Houston, Texas, using ANS.

Another example, a `ucl` ⇒ `wustl` `traceroute`, is even more striking:

```
 1  cisco.cs.ucl.ac.uk  13 ms  5 ms  5 ms
 2  cisco-pb.ucl.ac.uk  14 ms  4 ms  4 ms
 3  cisco-b.ucl.ac.uk  5 ms  4 ms  4 ms
 4  gw.lon.ja.net  48 ms  36 ms  81 ms
 5  eu-gw.ja.net  71 ms  58 ms  72 ms
 6  icm-lon-1.icp.net  56 ms  120 ms  119 ms
 7  icm-dc-1-s3/2-1984k.icp.net  162 ms  137 ms  175 ms
 8  sl-dc-7-f0.sprintlink.net  160 ms  197 ms  189 ms
 9  sl-starnet-1-s0-t1.sprintlink.net  166 ms  122 ms  634 ms
10  ncrc-acn.wustl.edu  457 ms  127 ms  119 ms
11  ncrc-eng.wustl.edu  140 ms  237 ms  174 ms
12  cisco-b.ucl.ac.uk  488 ms !H jcr.ecl.wustl.edu  244 ms  232 ms
13  tango.cs.wustl.edu  228 ms *  151 ms
```

Note that the first hop 12 router, `cisco-b.ucl.ac.uk`, is the same as the hop 3 router! This router also reports “`!H`”, indicating it could not forward the packet, and yet the second and third `traceroute` probe packets for that hop make it all the way to `wustl`. This `traceroute` appears to reflect a 500 msec outage, quickly repaired.

We thus see that the distribution of recovery times from routing problems is at least bimodal—some recoveries occur quite quickly, on the time scale of congestion delays, while others take on the order of a minute to resolve. The latter type of recovery presents significant difficulties to time-sensitive applications that assume outages are short-lived.

Sometimes the presence of a connectivity change is more subtle, such as in this $\mathcal{R}_1$ `traceroute` from `korea` to `ucol`:

```
 1  fpls.postech.ac.kr  2 ms  1 ms  1 ms
 2  fddicc.postech.ac.kr  3 ms  2 ms  2 ms
 3  ktrc-postech.hana.nm.kr  30 ms  30 ms  51 ms
 4  gateway.hana.nm.kr  31 ms  31 ms  31 ms
 5  hana.hana.nm.kr  33 ms  44 ms  32 ms
 6  bloodyrouter.hawaii.net  1152 ms  1275 ms  968 ms
 7  bloodyrouter.hawaii.net  744 ms  336 ms  325 ms
 8  arc1.nsn.nasa.gov  384 ms  491 ms  691 ms
 9  jpl6.nsn.nasa.gov  791 ms  772 ms  1082 ms
10  jpl3.nsn.nasa.gov  876 ms *  1641 ms
11  ncar1.nsn.nasa.gov  1117 ms  1225 ms  848 ms
12  * cu-gw.ucar.edu  1280 ms  805 ms
13  cu-ncar.co.westnet.net  774 ms  884 ms *
14  cs-gw.colorado.edu  1079 ms  897 ms  603 ms
15  lewis.cs.colorado.edu  283 ms  383 ms  899 ms
```

In this example, hop 6 and hop 7 were both to `bloodyrouter.hawaii.net`.[7] The subsequent route shown above is exactly the route taken by every other `korea` ⇒ `ucol` `traceroute`, except each hop is delayed by one (e.g., `jpl6.nsn.nasa.gov` is hop 9 here instead of hop 8 as usual).

Duplicate hops such as this one are most likely due to upstream route changes (§ 4.2.3) which, in this example, added an extra hop upstream to `bloodyrouter.hawaii.net`. The change would have had to occur just between the end of the probes for hop 6 and the beginning of those for hop 7. We considered all such duplicated hops to be midstream route changes.

In contrast with the rarity of connectivity changes in $\mathcal{R}_1$ (10 total), in $\mathcal{R}_2$ we observed 155 instances of a change, a fact we comment upon further in § 6.10.

6.6 Fluttering

We use the term "fluttering" to refer to rapidly-variable routing. On the time scale of a single `traceroute` (seconds to minutes) we would expect the path we are measuring to remain stable, yet surprisingly often our data showed that the packets belonging to a single `traceroute` took multiple paths through the Internet.

6.6.1 A simple example

Route fluttering can be detected from `traceroute` output by the presence of more than one host listed for a single hop, as in this example of a $\mathcal{R}_1$ `traceroute` between `korea` and `austr`.

```
 1  fpls.postech.ac.kr  2 ms  2 ms  2 ms
 2  fddicc.postech.ac.kr  3 ms  2 ms  2 ms
 3  ktrc-postech.hana.nm.kr  57 ms  123 ms  30 ms
 4  gateway.hana.nm.kr  31 ms  31 ms  31 ms
 5  hana.hana.nm.kr  33 ms  140 ms  32 ms
 6  bloodyrouter.hawaii.net  825 ms  722 ms  805 ms
 7  usa-serial.gw.au  960 ms  922 ms  893 ms
 8  national-aix-us.gw.au  1039 ms * *
 9  * rb1.rtr.unimelb.edu.au  903 ms rb2.rtr.unimelb.edu.au  1279 ms
10  itee.rtr.unimelb.edu.au  1067 ms  1097 ms  872 ms
11  * * mulkirri.cs.mu.oz.au  1468 ms
12  mullala.cs.mu.oz.au  1042 ms  1140 ms  1262 ms
```

Here, the 9th hop shows two different hosts (as well as no reply for the first `traceroute` packet), `rb1.rtr.unimelb.edu.au` and `rb2.rtr.unimelb.edu.au`. Thus, it appears that for the second packet `national-aix-us.gw.au` routed the packet to `rb1.rtr.unimelb.edu.au`, and for the third packet to `rb2.rtr.unimelb.edu.au`. (This change occurred most likely for purposes of load-balancing—see § 6.6.2 and § 7.4.)

It is important to keep in mind, though, that the actual route flutter could have occurred *upstream* from `national-aix-us.gw.au`, and that for the hop 9 `traceroute` packets, the 8th hop was actually a different router altogether (§ 4.2.3).

[7] In the example we have shown hostnames rather than IP addresses, as this aids in placing the router's location and service provider. It is possible for two different IP addresses to translate to the same hostname (indeed this is very common for routers). But inspecting the raw `traceroute` reveals the same IP address for both hop 6 and hop 7.

For subsequent hops, we cannot tell which of `rb1.rtr.unimelb.edu.au` or `rb2.rtr.unimelb.edu.au` was used (indeed, it could have been all of one or the other, or a continuation of switching between the two, or still a third router; the path was consistent with others we observed from the two routers).

6.6.2 A more dramatic example

The preceding example is straight-forward and demonstrates only minor fluttering, which presumably has no significant effect on the characteristics of the Internet path between `korea` and `austr`. A more dramatic example comes from a $\mathcal{R}_1$ `traceroute` between `wustl` and `umann`:

```
 1  128.252.166.249  11 ms  29 ms  8 ms
 2  128.252.123.254  3 ms  2 ms  2 ms
 3  128.252.5.120  3 ms  3 ms  14 ms
 4  128.252.1.135  6 ms  3 ms  3 ms
 5  199.217.253.1  19 ms  35 ms 199.217.253.3  64 ms
 6  144.228.73.17  56 ms 144.228.27.5  26 ms  28 ms
 7  144.228.20.101  29 ms  38 ms 144.228.70.2  55 ms
 8  144.228.10.25  69 ms  65 ms 192.157.65.74  57 ms
 9  144.228.8.233  217 ms  117 ms 194.41.0.17  118 ms
10  144.228.10.22  107 ms 193.172.4.8  122 ms  114 ms
11  192.203.230.253  68 ms 193.172.4.12  130 ms 192.203.230.253  70 ms
12  193.174.74.94  194 ms 140.222.8.4  72 ms 193.174.74.94  192 ms
13  193.174.74.29  192 ms  189 ms  192 ms
14  140.222.112.2  108 ms 129.143.6.16  222 ms  216 ms
15  140.222.64.1  128 ms 153.17.62.105  236 ms 140.222.64.1  141 ms
16  129.143.61.2  238 ms  284 ms 140.222.104.2  162 ms
17  134.155.48.125  242 ms 140.222.72.1  164 ms 134.155.48.125  263 ms
```

Here we show the route using untranslated IP addresses, since showing the names of all of the various routers would make for messy reading. However, consider hop 10:

```
10  icm-fix-w-h2/0-t3.icp.net 107 ms amsterdam6.empb.net 122 ms 114 ms
```

The first packet visited FIX-West at NASA AMES Research Center (Moffett Field, San Francisco Bay Area), while the second and third made it to Amsterdam!

The divergence begins at hops 4-5:

```
 4  128.252.1.135  6 ms  3 ms  3 ms
 5  stl1-e0.starnet.net  19 ms  35 ms stl3-e0.starnet.net  64 ms
```

The WUSTL border router (`128.252.1.135`) picks two different STARnet routers for the next hop, each of which presumably has a different notion of the best path to Europe. The confused `traceroute` shown above can be reduced to two separate `traceroutes` at this split. First, the "successful" path—the one that first reaches `umann`:

```
 5  ?
 6  sl-dc-7-s7-t1.sprintlink.net
 7  icm-dc-1-f0/0.icp.net
 8  icm-dante-e0.icp.net
```

```
 9  amsterdam1.dante.net
10  amsterdam6.empb.net
11  duesseldorf2.empb.net
12  ipgate2.win-ip.dfn.de
13  duesseldorf2.win-ip.dfn.de
14  heidelberg1.belwue.de
15  mannheim.belwue.de
16  belwue-gw.uni-mannheim.de
17  eratosthenes.informatik.uni-mannheim.de
```

Geographically, this route traverses: St. Louis, Missouri; Washington, D.C.; Amsterdam, the Netherlands; and Duesseldorf, Heidelberg, and Mannheim, in Germany.[8]

The second route instead criss-crosses the United States:

```
 5  ?
 6  sl-ana-3-s3/1-t1.sprintlink.net
 7  sl-ana-2-f0/0.sprintlink.net
 8  sl-stk-6-h2/0-t3.sprintlink.net
 9  144.228.8.233
10  icm-fix-w-h2/0-t3.icp.net
11  t3-0.enss144.t3.nsf.net
12  t3-3.cnss8.san-francisco.t3.ans.net
13  ?
14  t3-1.cnss112.albuquerque.t3.ans.net
15  t3-0.cnss64.houston.t3.ans.net
16  t3-1.cnss104.atlanta.t3.ans.net
17  t3-0.cnss72.greensboro.t3.ans.net
```

Geographically, this route traverses: St. Louis, Missouri; Anaheim, Stockton, FIX-West, and San Francisco, California; Albuquerque, New Mexico; Houston, Texas; Atlanta, Georgia; and Greensboro, North Carolina.[9] From other `traceroutes` that included `t3-0.cnss72.greensboro.t3.ans.net`, we can determine that eventually this route would also have made it to the destination, albeit with many more hops. For example, from a trace from `sri` to `umann`, we have:

```
12  t3-0.cnss72.greensboro.t3.ans.net
13  t3-0.cnss56.washington-dc.t3.ans.net
14  t3-0.enss145.t3.ans.net
15  umd-rt1.es.net
16  umd2-e-stub.es.net
17  pppl2-umd2.es.net
18  ipgate2.win-ip.dfn.de
```

[8] Hop 5 is marked as "?" because from the trace it is not clear which of the two STARnet routers picks this route (by forwarding to `sl-dc-7-s7-t1.sprintlink.net`), and which picks the longer route.

[9] Hop 13 is missing because, in the raw trace, all three replies to the hop 13 `traceroute` probe were returned by `duesseldorf2.win-ip.dfn.de`, which clearly is not the next hop following `t3-3.cnss8.san-francisco.t3.ans.net`, but rather represents hop 13 from the first route.

By inspecting other `traceroutes` from `wustl` to `umann`, it is evident that hop 13 for the second route is `t3-0.cnss16.los-angeles.t3.ans.net`, so we can add Los Angeles to the list of California cities traversed by the route.

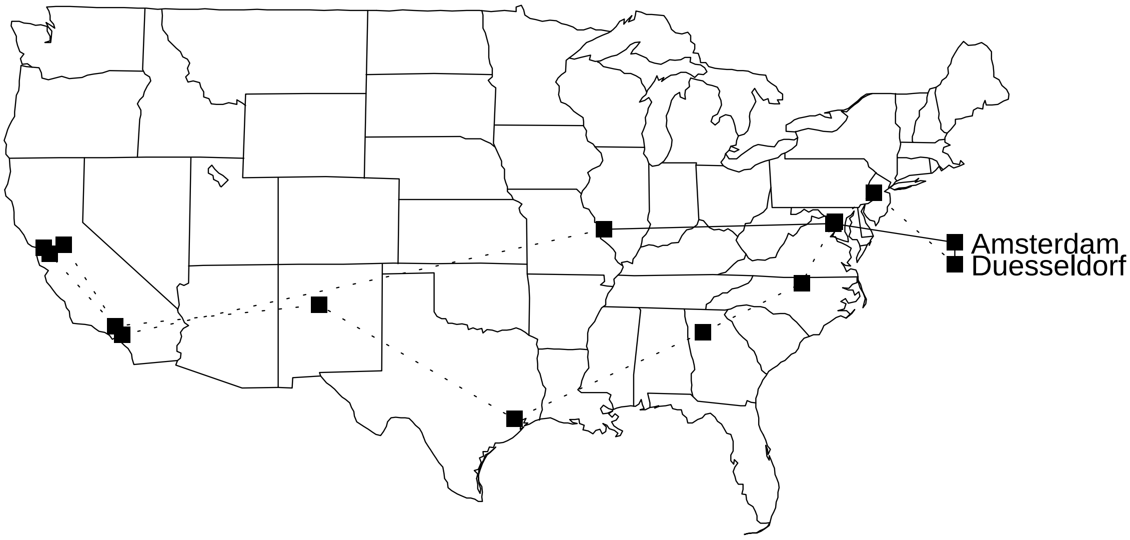

Figure 6.1: Routes taken by alternating packets from `wustl` (St. Louis, Missouri) to `umann` (Mannheim, Germany), due to fluttering

```
19  ipgate2.win-ip.dfn.de
20  duesseldorf2.win-ip.dfn.de
21  heidelberg1.belwue.de
22  mannheim.belwue.de
23  belwue-gw.uni-mannheim.de
24  eratosthenes.informatik.uni-mannheim.de
```

Thus, it appears that the second `wustl` ⇒ `umann` route would also succeed in delivering packets, though using 29 hops instead of 17.

The `wustl` fluttering occurs over very small timescales, essentially the time between successive `traceroute` probes, which are spaced out by the amount of time it takes for each reply to the previous probe packet (§ 4.2.2). One routing mechanism that can lead to such small-scale fluttering occurs when a router alternates between multiple next-hop routers in order to split load among the links to those routers. Such behavior is explicitly allowed in [Ba95, p.79], though that document also cautions that there are situations for which it is inappropriate, and so it should at most be a configurable option for a router. It turns out that the `wustl` fluttering was indeed due to load-splitting: STARnet had two T1 links for its access to Sprintlink, one to Anaheim and the other to Washington, D.C. (as shown above), and would alternate packets "round-robin" between them in order to balance load [My95].

Figure 6.1 shows the two routes that packets can take from `wustl` to `umann`. The dramatic difference in the lengths of the two routes highlights the great impact an early routing discrepancy can make.

Of the 380 `traceroutes` initiated by `wustl`, 255 exhibited fluttering, all but one occurring before 12PM PST, December 13. After this point, the Anaheim link apparently became unavailable, and the routing was no longer split. This change however was not due to a decision to eliminate fluttering, but, apparently, simply due to an outage along the Anaheim link. On December 20 the Anaheim link again became operational, and led to an interesting pathology:

```
 1  128.252.166.249  4 ms  2 ms  3 ms
 2  128.252.123.254  3 ms  2 ms  4 ms
 3  128.252.5.120  3 ms  2 ms  2 ms
 4  128.252.1.2  5 ms  3 ms  3 ms
 5  199.217.253.2  4 ms  3 ms  4 ms
 6  199.217.253.1  4 ms  11 ms 199.217.253.3  6 ms
 7  199.217.253.2  4 ms 144.228.73.17  58 ms  56 ms
 8  144.228.70.1  56 ms 199.217.253.3  4 ms  5 ms
 9  144.228.10.29  85 ms 144.228.73.17  74 ms  63 ms
10  144.228.30.5  102 ms  217 ms  218 ms
11  144.228.10.29  81 ms 144.228.10.17  93 ms  92 ms
12  144.228.20.6  84 ms  131 ms  125 ms
13  192.157.65.227  85 ms 144.228.10.29  80 ms 192.157.65.227  81 ms
14  144.228.20.6  137 ms 144.228.30.5  264 ms 144.228.20.6  165 ms
15  144.228.10.17  70 ms * *
16  144.228.30.5  90 ms * 144.228.20.6  74 ms
17  * 192.157.65.227  105 ms *
18  137.39.128.7  120 ms * *
19  * 192.157.65.227  84 ms *
20  * * *
21  * * *
22  * * 137.39.128.7  202 ms
23  * * *
24  * * *
25  * * *
26  * * *
27  * * *
28  * * *
29  * * *
```

The fluttering begins at hop 6:

```
 5  stl2-e0.starnet.net  4 ms  3 ms  4 ms
 6  stl1-e0.starnet.net  4 ms  11 ms stl3-e0.starnet.net  6 ms
```

Here, packets again alternate between `stl1-e0.starnet.net` and `stl3-e0.starnet.net`. Hop 7, though, shows that the routing is further confused:

```
 7  stl2-e0.starnet.net 4 ms sl-ana-3-s3/1-t1.sprintlink.net 58 ms 56 ms
```

It appears that either `stl1-e0.starnet.net` or `stl3-e0.starnet.net` forwarded the packet *back* to `stl2-e0.starnet.net`, while the other forwarded the packet to `sl-ana-3-s3/1 t1.sprintlink.net` in Anaheim, California. In the next hop:

```
 8  sl-ana-1-f0/0.sprintlink.net  56 ms stl3-e0.starnet.net  4 ms  5 ms
```

one of the packets makes it to the next Anaheim hop, while the other is forwarded (apparently from `stl2-e0.starnet.net`) to `stl3-e0.starnet.net`.

At this point, the packets proceed to `bsdi` but with some making one (or even more!) visits from `stl2-e0.starnet.net` to the non-forwarding STARnet router (it is difficult to determine whether this is `stl1-e0.starnet.net` or `stl3-e0.starnet.net`). Viewed geographically:

```
 9  Fort-Worth-6  85 ms Anaheim  74 ms  63 ms
10  Fort-Worth-5  102 ms  217 ms  218 ms
11  Fort-Worth-6  81 ms Washington-DC-8  93 ms  92 ms
12  Washington-DC-6  84 ms  131 ms  125 ms
13  Boone-VA  85 ms Fort-Worth-6  80 ms Boone-VA  81 ms
14  Washington-DC-6  137 ms Fort-Worth-5  264 ms Washington-DC-6  165 ms
15  Washington-DC-8  70 ms * *
16  Fort-Worth-5  90 ms * Washington-DC-6  74 ms
17  * Boone-VA  105 ms *
18  Dallas  120 ms * *
19  * Boone-VA  84 ms *
```

The `Fort-Worth-5` router at both hop 10 and hop 16 indicates that one of the hop 16 packets made *three* trips to the non-forwarding STARnet router prior to getting forwarded to the working router. Most likely this pathology occurred due to a set of inconsistent routing tables introduced by the reactivation of the Anaheim link.

For reference, a flutter-free route from `wustl` to `bsdi` is:

```
 1  jcr-166.cs.wustl.edu  5 ms  2 ms  2 ms
 2  ncrc-eng.wustl.edu  3 ms  2 ms  2 ms
 3  128.252.5.120  4 ms  3 ms  2 ms
 4  128.252.1.2  6 ms  6 ms  3 ms
 5  sl-dc-7-s7-t1.sprintlink.net  29 ms  28 ms  25 ms
 6  sl-dc-6-f0/0.sprintlink.net  156 ms  26 ms  64 ms
 7  boone1.va.alter.net  30 ms  35 ms  28 ms
 8  dallas1.tx.alter.net  80 ms  67 ms  69 ms
```

where the `128.252.x.y` routers are local to WUSTL (`traceroutes` to `bsdi` stop in Dallas, as explained in § 6.7.4).

The STARnet routing remained split for many more months. Here is a traceroute from `wustl` to `umann`, taken on July 2, 1995:

```
 1  128.252.166.249  3 ms  2 ms  2 ms
 2  128.252.123.254  4 ms  2 ms  2 ms
 3  128.252.5.120  4 ms  2 ms  2 ms
 4  128.252.41.2  4 ms  3 ms  3 ms
 5  199.217.253.1  4 ms  6 ms  11 ms
 6  144.228.73.17  71 ms 144.228.27.5  41 ms 144.228.73.17  166 ms
 7  144.228.20.8  30 ms 144.228.70.1  151 ms  56 ms
 8  144.228.10.29  87 ms 144.228.10.42  61 ms 144.228.10.29  90 ms
 9  144.228.30.5  143 ms  258 ms 192.41.177.252  35 ms
10  144.228.10.17  91 ms 134.55.12.161  81 ms  67 ms
11  192.188.33.10  138 ms  159 ms 144.228.10.42  74 ms
12  192.41.177.252  79 ms  73 ms  74 ms
13  153.17.200.105  198 ms *  220 ms
14  192.188.33.10  202 ms * *
15  193.174.74.141  224 ms 134.155.48.125  245 ms  214 ms
```

Fluttering occurs downstream of the hop 5 router:

```
 5  stl1-e0.starnet.net  4 ms  6 ms  11 ms
 6  Anaheim-3  71 ms Washington-DC-7  41 ms Anaheim-3  166 ms
```

and continues from there. This example is slightly different from the previous ones we looked at, in that the STARnet routers `stl2-e0.starnet.net` and `stl3-e0.starnet.net` no longer appear. Instead, it looks like `stl1-e0.starnet.net` is doing its own load-splitting between `sl-ana-3-s3/1-t1.sprintlink.net` and `sl-dc-7-s7-t1.sprintlink.net`, on opposite sides of the country.

STARnet has since switched to a single connection (via MCI), so this pathology no longer occurs [My95].

In § 13.1.3 we analyze the effects that the split-routing had upon TCP performance. Surprisingly, it was generally quite minor. While `wustl` packets very often arrived out of order, they only very rarely arrived so far out of order as to trigger a spurious fast retransmission, as discussed in § 6.6.5 below.

6.6.3 Fluttering at another site

Putting aside `traceroute` probes initiated at `wustl`, of the remaining 6,079 $\mathcal{R}_1$ probes, 295 (about 5%) exhibited fluttering. None of these sites suffered such extreme fluttering as `wustl`; all of the flutters affected either a single hop or at most two hops. Here is an example of a two-hop flutter, between `ncar` and `ucol`, both sited in Boulder, Colorado:

```
 1  north-gw.scd.ucar.edu  3 ms  2 ms  2 ms
 2  server-gw.ucar.edu  3 ms  2 ms  2 ms
 3  cu-gw.ucar.edu  4 ms  3 ms  3 ms
 4  129.19.248.62  5 ms cu-ncar.co.westnet.net  5 ms 129.19.248.62  6 ms
 5  cs-gw.colorado.edu  6 ms  6 ms  5 ms
 6  lewis.cs.colorado.edu  8 ms  19 ms  9 ms
```

The 4th hop shows a flutter from `129.19.248.62` (at Colorado State University) to `cu-ncar.co.westnet.net` and back again. We note that the problem occurred during a hop to Colorado State University, which suggests that those routers may be prone to fluttering. Indeed, of the 295 remaining flutters, 277 involved `ucol`. For all but 6 of these, the fluttering occurred immediately downstream from either the `cu-gw.colorado.edu` router (for traffic outbound from `ucol`) or the `cu-gw.ucar.edu` (traffic inbound to `ucol`). It appears that these routers were splitting load just as did the STARnet router in the previous section, but both downstream routers they alternated between had the same view of subsequent wide-area routing, so the effect remained localized.

Neither the `ucol` nor the `wustl` fluttering was present in $\mathcal{R}_2$. The only repeated pattern we found was that every route originating at `sdsc` that passed through `nynap-sdsc-atm-ds3.cerf.net` suffered from downstream fluttering. Here is an example, from a `traceroute` to `adv`:

```
 1  tigerfish.sdsc.edu  8 ms  8 ms  8 ms
 2  mobydick.cerf.net  85 ms  246 ms  18 ms
 3  nynap-sdsc-atm-ds3.cerf.net  475 ms  380 ms  71 ms
 4  sprintnap.ans.net 73 ms t3-3.cnss32.new-york.t3.ans.net 75 ms 77 ms
 5  cnss33.new-york.t3.ans.net  76 ms  77 ms  76 ms
 6  enss240.t3.ans.net  80 ms  80 ms  79 ms
 7  enss240.t3.ans.net  173 ms betelgeuse.advanced.org  81 ms  87 ms
```

There were only 7 of these, however, so their overall impact on routing performance in $\mathcal{R}_2$ was insignificant.

6.6.4 Skipping

When analyzing the traces for fluttering, we notice an interesting anomaly in which routers were visited "prematurely." Here is an example, taken from an xor $\Rightarrow$ ucl traceroute:

```
 1 xor-gw.xor.com 0 ms 0 ms 10 ms
 2 gw1.boulder.co.coop.net 0 ms 0 ms 0 ms
 3 sl-fw-2-s9-t1.sprintlink.net 30 ms 30 ms 30 ms
 4 sl-fw-5-f1/0.sprintlink.net 30 ms 20 ms 40 ms
 5 sl-dc-8-h3/0-t3.sprintlink.net 60 ms 60 ms 60 ms
 6 icm-dc-1-f0/0.icp.net 1520 ms
   icm-london-1-s1-1984k.icp.net 160 ms
   icm-dc-1-f0/0.icp.net 60 ms
 7 icm-london-1-s1-1984k.icp.net 150 ms 140 ms 150 ms
 8 smds-gw.ulcc.ja.net 140 ms 150 ms 140 ms
 9 smds-gw.ucl.ja.net 150 ms 150 ms 140 ms
10 cisco-pb.ucl.ac.uk 160 ms 160 ms 160 ms
11 cisco.cs.ucl.ac.uk 150 ms 160 ms 160 ms
12 neptune.cs.ucl.ac.uk 160 ms 160 ms 170 ms
```

At hop 6, we see flutter between icm-dc-1-f0/0.icp.net and icm-london-1-s1-1984k.icp.net. But hop 7 then reveals that icm-london-1-s1-1984k.icp.net is actually the next hop!

All told, 11 traceroutes in $\mathcal{R}_1$ and 22 in $\mathcal{R}_2$ (at a number of different routers) showed this "skipping" effect. Furthermore, very often the packet return time just prior to the skip was unusually high (note in the example above the return time of 1,520 msec, much larger than any other in the traceroute). It appears that the router was under a period of stress during the time of the skip, and (perhaps due to a forwarding bug only exhibited under high load) a packet was erroneously forwarded without decrementing and checking its TTL. The downstream router then decremented the TTL, noted it had expired, and returned an ICMP message. The upstream router subsequently recovered from the error condition and continued to correctly forward packets, as is shown for the third probe of hop 6 above.

If the source of the router load were network traffic, then the response from the downstream router should have been heavily delayed too, but, as shown above, it was not. Another explanation is that the load was instead due to the upstream router processing a routing update. This agrees with the fact that the router recovered quickly from the load condition: all that was needed was a single packet's worth of time (about 160 msec above) for the load to disappear.

That a router might, under stress, forward a packet without decrementing its TTL raises a possibility of network instability. If the router stress was due to a routing loop, packets might circulate around the loop indefinitely because their TTL's would not correctly expire, which might in turn maintain the router stress.

We considered traceroutes exhibiting "skipping" as reflecting a pathology separate from "fluttering," since the underlying mechanisms (load-balancing vs. an apparent packet forwarding error) are quite different.

6.6.5 Significance of fluttering

While fluttering can provide benefits as a way to balance load in a network, it also creates a number of problems for different networking applications:

1. A fluttering network path presents the difficulties that arise from *unstable* network paths, as discussed in § 7.1: difficult-to-predict behavior, potential inconsistencies in state information created in the routers on behalf of connections, and problems with constructing consistent measurements of the network's condition. However, if fluttering occurs only at a larger granularity than individual packets—for example, per connection or per end-to-end "flow"—then these problems are ameliorated.

2. If the fluttering only occurs in one direction (as it does for `wustl`, but not for `ucol`), then the path is necessarily *partially asymmetric*, too, suffering from the problems discussed in § 8.1: difficulties in computing unidirectional latencies for protocols such as NTP, difficulties in using "sender-only" measurement techniques, and inefficiencies in keeping state for bidirectional flows.

3. Constructing reliable estimates of the path characteristics, such as round-trip time and available bandwidth, becomes potentially very difficult, since in fact there may be *two* different sets of values to estimate.

4. When the two routes have different propagation times, such as many of those from the `wustl` site, then packets will often arrive at the destination out-of-order, depending on whether they took the shorter route or the longer route. At a minimum, this can lead to extra processing at the receiver to reassemble the out-of-order data stream.

   It can lead to a more serious problem for TCP connections, however. Whenever a TCP endpoint receives an out-of-order packet, the receipt triggers the sending of a redundant acknowledgement in reply, as a mechanism for informing the sender that the receiver has a hole in its sequence space. If three out-of-order packets arrive in a row, then the receiver will generate three redundant acknowledgements. These are enough in turn to trigger "fast retransmission" by the sender (§ 9.2.7), leading it to needlessly retransmit data. Thus, out-of-order delivery can result in redundant network traffic, both due to the extra acknowledgements, and due to possible data retransmissions. We explore this phenomenon further in § 13.1.3.

These problems all argue for eliminating large-scale fluttering whenever possible, where we define fluttering as large-scale if it leads to significantly different routes (as it does for `wustl`). On the other hand, when the effects of the flutter are confined, as for `ucol`, or invisible at the network layer (such as split-routing used at the link layer, which would not show up at all in our study), then these problems are all ameliorated.

Finally, we note that "deflection" and "dispersion" routing schemes that forward packets along varying or multiple paths have many of the characteristics of fluttering paths [BDG95, GK97]. While these schemes can offer benefits in terms of simplified routing decisions, enhanced throughput, and resilience, they bring with them the difficulties discussed above. From the discussion of dispersion routing in [GK97], it appears that the literature in that area to date has only considered the problem of out-of-order delivery, which is addressed simply by noting that the schemes require a resequencing buffer.

Failure mode	# Failures	Notes
Host down	81 (65 %)	umann, sdsc, and inria accounted for 93%
Stub network outage	31 (25 %)	ustutt accounted for 74% of these
Infrastructure failure	13 (10 %)	no dominant pattern

Table VIII: Failure modes for unreachable hosts in $\mathcal{R}_1$

Failure mode	# Failures	Notes
Host down	277 (45 %)	panix accounted for 61% of these
Stub network outage	170 (27.5 %)	nrao accounted for 57% of these
Infrastructure failure	170 (27.5 %)	no dominant pattern

Table IX: Failure modes for unreachable hosts in $\mathcal{R}_2$

6.7 Unreachability

In addition to traceroute failures due to persistent routing loops and erroneous routing, 125 of the $\mathcal{R}_1$ traceroutes and 617 of the $\mathcal{R}_2$ traceroutes failed to reach the destination host for other reasons. We analyzed these failures to determine the corresponding failure modes, summarized in Tables VIII and IX.

6.7.1 Host down

We concluded that a host was down (first row) if the traceroute to it terminated at one of the routers which in another traceroute proved to be the penultimate hop to that host. In $\mathcal{R}_1$, this occurred 81 times out of a total of 6,459 traceroutes, giving us an unconditional probability that a site participating in our study was down during an experiment of $p \approx 1.25\%$. This probability corresponds to an availability of $\approx 98.75\%$, Similarly, for $\mathcal{R}_2$ we get an availability of $\approx 99.2\%$. These values are a bit higher than the median availability of 97.2% reported in [LMG95], though our "polling" frequency is lower than theirs (a mean of 10 minutes), which could explain the discrepancy. Also, as noted in § 4.4, our sites do *not* plausibly constitute a random sample of Internet hosts (while [LMG95]'s sites are much closer to such), so disagreement between the two figures is not particularly significant. Finally, note that most of the failures were due to just a few of the sites, as indicated in the tables.

6.7.2 Stub network outage

We classified an Unreachability failure as a "stub network outage" (second row) if the final router reached during the traceroute was sited inside the same institute as the endpoint (but not a penultimate hop), or at the border between the institute and the remainder of the Internet.[10]

[10] Such a failure could also occur at the traceroute source's institute. One might think we would never observe this in our traces because, in order to generate a traceroute, the npd_control site had to be able to connect to

The numbers of observations of such failures correspond to availabilities of 99.5% for both $\mathcal{R}_1$ and $\mathcal{R}_2$, though again we cannot draw a general conclusion about connectivity to Internet sites because our collection of participating sites might not be representative. We also need to be wary about generality given the strong dominance of this type of failure by routes to the `ustutt` and `nrao`[11] sites.

On the other hand, the prevalence of network outages to `ustutt` gives us an opportunity to assess how quickly a router learns that the next-hop router has crashed. If a router does not have a route to a packet's destination, the router is required to generate some form of ICMP "Destination Unreachable" message [Ba95]. However, a router *may not know* that it has no route to the packet's destination, because it is unaware that the next-hop router has crashed. These two cases result in different `traceroute` behavior: the first elicits a "`!H`" (or "`!N`") response in the `traceroute` output, while the second will simply show a dropped packet. Consider the following `traceroute` from `ukc` to `ustutt`:

```
 1  rtcomp.ukc.ac.uk  2 ms  2 ms  2 ms
 2  brtcomp.ukc.ac.uk  2 ms  2 ms  2 ms
 3  brtsj.ukc.ac.uk  3 ms  3 ms  3 ms
 4  smds-gw.ulcc.ja.net  7 ms  7 ms  6 ms
 5  eu-gw.ja.net  8 ms  8 ms  6 ms
 6  london4.empb.net  12 ms  11 ms  8 ms
 7  duesseldorf2.empb.net  33 ms  31 ms  38 ms
 8  ipgate2.win-ip.dfn.de  91 ms  52 ms  46 ms
 9  duesseldorf4.win-ip.dfn.de  70 ms  44 ms  32 ms
10  stuttgart4.belwue.de  67 ms  68 ms  56 ms
11  stuttgart1.belwue.de  84 ms  85 ms  74 ms
12  belwue-gw.uni-stuttgart.de  63 ms  57 ms  69 ms
13  * * *
14  * * *
15  * * *
16  * * *
17  * * *
18  * * *
19  * * *
20  * * *
21  * * *
22  * * *
23  * * belwue-gw.uni-stuttgart.de  68 ms !H
24  * * *
25  * * *
26  * * *
27  * * *
28  * * *
29  * * *
30  * * belwue-gw.uni-stuttgart.de  64 ms !H
```

the source in the first place. However, some sources have multiple connections to the Internet, and we did observe several instances where we were able to connect to a source but it was unable to advance packets to any routers outside of its site. We include these instances in the tables as stub network outages.

[11] It turns out that the entire `nrao` site was intentionally disconnected from the Internet from November 28 through December 6, 1995, following a serious break-in by a network cracker.

Hop 12 makes it to belwue-gw.uni-stuttgart.de, ustutt's border router. Normally the next hop would be to cisco1.rus.uni-stuttgart.de, inside the ustutt site, and hop 12 gives no indication of an impending problem here. But the next 36 packets are dropped, reflecting an outage of 3.5 minutes. At hop 23, belwue-gw.uni-stuttgart.de again responds, but this time includes an ICMP unreachable message. Thus, it appears that it took belwue-gw.uni-stuttgart.de at least 3.5 minutes to learn that the next hop had crashed.

What follows, from hops 24-30, remains a puzzle: belwue-gw.uni-stuttgart.de apparently forgets that the next-hop router has crashed and only relearns the fact after another 100 seconds. At this point the traceroute terminates because it has reached the 30-hop limit.

Of the 23 $\mathcal{R}_1$ stub network outages involving ustutt, 19 exhibited this pattern.[12] For those 19, the learning periods range from 0 seconds (the router immediately knew that the next hop was unavailable) to 170 seconds, with a median of 30 seconds and a mean of 50 seconds (distributed roughly exponentially–see § 6.8 for the significance of this). For the other four ustutt outages, the router failed to learn the unavailability of the downstream hop before the traceroute terminated due to the 30-hop limit. These failures spanned between 105 and 225 seconds, so those give lower bounds on the learning time.

Clearly, for belwue-gw.uni-stuttgart.de, the router does not quickly learn about a next-hop crash. If this slow response is typical (we lack enough data to know if it is), then Internet traffic is subject to outages on the order of a minute whenever a router crashes. This finding is consistent with the BGP specification, which recommends that routers wait for 90 seconds' worth of unanswered polls before deciding that a peer is unreachable [RL95]. The higher this figure is, the less prone a network is to routing oscillations; but high delays in detecting unreachable peers also present serious difficulties for real-time protocols that need to quickly adapt to such faults [GR95].

6.7.3 Infrastructure failure

The final type of failure (third row in each table) reflects a problem inside the Internet infrastructure: the terminating router in the traceroute was in the middle of the network, not at the source or destination.[13] In this case, we *can* make a general statement about availability, since the basis for our study is the assumption that the collection of routes between our sites *is* representative of Internet connectivity as a whole (§ 4.4). A total of 13 failures out of 6,459 $\mathcal{R}_1$ observations corresponds to an Internet infrastructure availability of 99.8%, while for $\mathcal{R}_2$ this percentage drops to 99.5%. The difference is significant using the methodology discussed in § 4.5. If we add to these failures the instances of persistent routing loops (§ 6.3.1) and erroneous routing (§ 6.4), then the $\mathcal{R}_1$

[12] All of the ustutt outages occurred between the early morning of Saturday, December 10th and the early morning of Monday, December 12th (Stuttgart time), indicating that the crashed router was down for the weekend.

[13] In some cases, such a termination can still reflect an unreachable host or a stub network outage, if the unreachability information has been propagated into the interior of the network. However, in these cases we would expect that the information is not propagated *deeply* into the network, since the need to "aggregate" routing information means that information pertaining to individual host or stub network outages cannot be propagated beyond the point at which it is aggregated with information for other, reachable hosts or networks.

We inspected the points in the terminating routers for the infrastructure failures and found that in the vast majority of cases, the router was sited far from the unreachable destination. For example, we observed several infrastructure failures for traceroutes going from bnl to European sites, each of which terminated at ames-llnl.es.net in California. Such a termination is much more likely to reflect loss of general connectivity to Europe, than an outage of a single European site being propagated all the way to a router in California.

availability falls to 99.6%, and that for $\mathcal{R}_2$ to 99.35%. We must bear in mind, however, that these numbers will be skewed by the fairly large proportion of our attempted measurements that failed due to an inability to contact the remote `npd` site (§ 5.2); some of these failures could be due to infrastructure problems, making these availability numbers overestimates.

A solid figure for Internet infrastructure availability is important for network service providers wishing to provide a form of *guaranteed service* in which the guarantees carry legal (contractual) obligations [Fe90, PaFe94]. We do not claim that the availabilities given in the preceding paragraph are such solid figures, but they are a step in that direction.[14]

6.7.4 Consistently unreachable hosts

Several hosts in our study were either always or frequently unreachable. Those always unreachable—`bsdi` in $\mathcal{R}_1$, and `oce` and `ucol` in $\mathcal{R}_2$—all reside behind firewalls that drop incoming, unidentified UDP packets (such as used by `traceroute`; § 4.2.3), so `traceroutes` to it always showed connectivity lost after the hop prior to the firewall. We adjusted for this behavior by considering any `traceroutes` that made it to that hop as making it all the way to the host.

The other frequently unreachable host, `lbli`, is connected to the Internet via an ISDN circuit. This circuit disconnects after any idle period during which `lbli` did not use the circuit for a configurable amount of time (typically 10-20 minutes). Thus, many `traceroutes` to `lbli` found the circuit down, and terminated at the Internet side of the ISDN link. As with the firewall hosts, we considered these `traceroutes` as having successfully reach the `lbli` host.

The net effect of these adjustments is to introduce possible underestimation into our assessment of the prevalence of stub network outages and hosts being down. Most likely, this introduced bias is quite slight, given how our stub network outages and downed hosts statistics were dominated by just a few sites anyway.

6.7.5 Unreachable due to too many hops

As noted in § 4.2.1, `traceroute` by default probes up to 30 hops of the route between two hosts. This length sufficed for all of the $\mathcal{R}_1$ measurements, and all but 6 of the $\mathcal{R}_2$ measurements.[15] The fact that it failed occasionally in $\mathcal{R}_2$, however, indicates that the operational diameter of the Internet has grown beyond 30 hops, and argues for using large initial TTL values when a host originates an IP datagram. In informal studies of the link connecting the Lawrence Berkeley National Laboratory to the rest of Internet, we have found that most hosts send IP datagrams with TTL's well above 30, but a non-negligible proportion of the datagrams (10% in one dataset) appear to have been sent with TTL's of around 30.

While routes of more than 30 hops were not correctly measured by `traceroute` in our experiment, they were so rare as to not present any significant source of error.

A final note concerning large hop counts: it is sometimes assumed that the hop count of a route equates to its geographical distance. While from our data this appears roughly the case, we

[14] Naturally, a network service provider will keep detailed statistics on their own network, and not need a figure such as that we have computed. But if they must deal with other providers for portions of the end-to-end route, such a figure as a rule-of-thumb will prove useful.

[15] 5 of the 6 were to or from `inria`. Routing within France (and international routing in general) often has many hops. The other was between `umont` and `umann`, also international in scope.

noticed some remarkable disagreements, both in terms of a few hops corresponding to large distances, and many hops corresponding to little distance. For example, the shortest route we observed from `ncar`, in Colorado, to `sdsc`, in southern California (about 1,500 km distant), was three hops:

```
cs-vbns.ucar.edu
cs-atm0-0-3.sdsc.vbns.net
rintrah.sdsc.edu
```

This route traveled over the VBNS ATM backbone (recall from § 4.2.3 that `traceroute` elicits paths at the *network layer*, and does not measure any "hops" made at the link layer). We also observed in $\mathcal{R}_1$ a 5 hop route from `pubnix` to `bsdi`, about 2,000 km distant.

On the other hand, all of the routes we observed between `mit` and `harv` (in either direction), sited about 3 km apart, were 11 hops, and we observed 14 and 17 hop routes between `sri` and `lbl`, about 50 km apart.

6.8 Temporary outages

The final pathology we studied was temporary network outages. When a sequence of consecutive `traceroute` probes are lost, the most likely cause is either a temporary loss of network connectivity, or very heavy congestion lasting 10's of seconds. For each `traceroute`, we examined its longest period of consecutive probe losses (other than consecutive losses at the end of a `traceroute` when, for example, the endpoint was unreachable).

The resulting distribution of the number of probes lost appears trimodal. In $\mathcal{R}_1$ ($\mathcal{R}_2$), about 55% (43%) of the `traceroutes` had no losses, 44% (55%) had between 1 and 5 losses, and 0.96% (2.2%) had 6 or more losses[16]

Of these latter, after eliminating those to `ukc` in $\mathcal{R}_1$ (because these "outages" are actually unresponsive routers; see § 6.1), the distribution of the number of probes lost in the $\mathcal{R}_1$ data is quite close to geometric. Figure 6.2 plots the outage duration on the x-axis vs. the probability of observing that duration or larger on the y-axis (logarithmically scaled). The outage duration is determined by the number of probe losses multiplied by 5 seconds per lost probe. The line added to the plot corresponds to what would be expected for a geometric distribution with probability $p = 0.92$ that a probe beyond the 5th is dropped. (The line appears straight due to the logarithmic y-axis scale and the fact that the geometric distribution is the discrete counterpart to the exponential distribution.) As can be seen, the fit is fairly good, especially in the tail.

From the above evidence it is reasonable to argue that long outages are well-modeled as persisting for 30 seconds plus an exponentially distributed random variable with mean equal to about 40 seconds. This finding would be convenient, since the exponential distribution often makes for tractable analysis.

If we turn to the $\mathcal{R}_2$ data, however, we find that the geometric tail with $p = 0.92$ is still present, but only for outages more than 75 seconds long, as illustrated in Figure 6.3. For outages between 30 and 70 seconds, the duration still exhibits a geometric distribution, but with $p = 0.62$, suggesting two different recovery mechanisms, one operating on time scales of 30 seconds to a minute or so and the other on significantly longer time scales.

[16] Recall from § 4.2.3 that probe "losses" can also be due to ICMP rate-limiting, which we do not differentiate. We analyze true packet losses in much greater detail in Chapter 15.

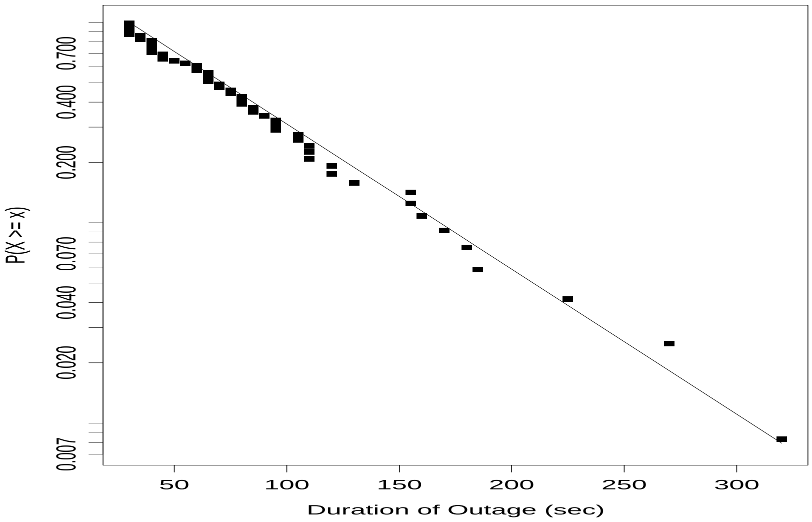

Figure 6.2: Distribution of long $\mathcal{R}_1$ outages

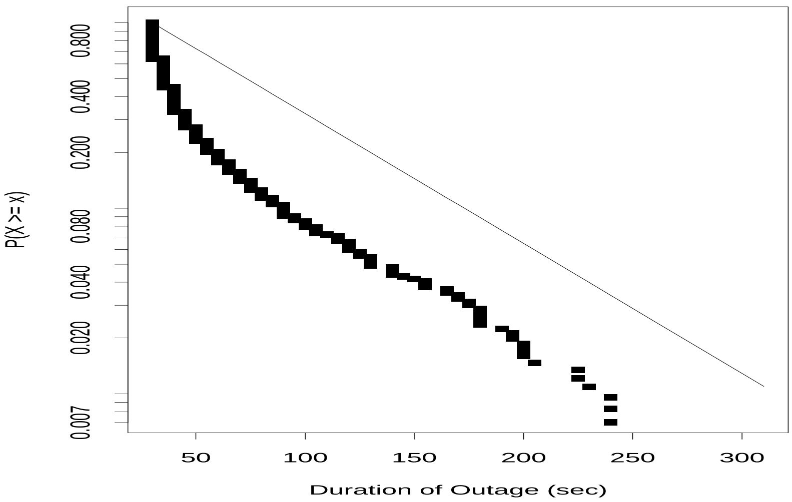

Figure 6.3: Distribution of long $\mathcal{R}_2$ outages

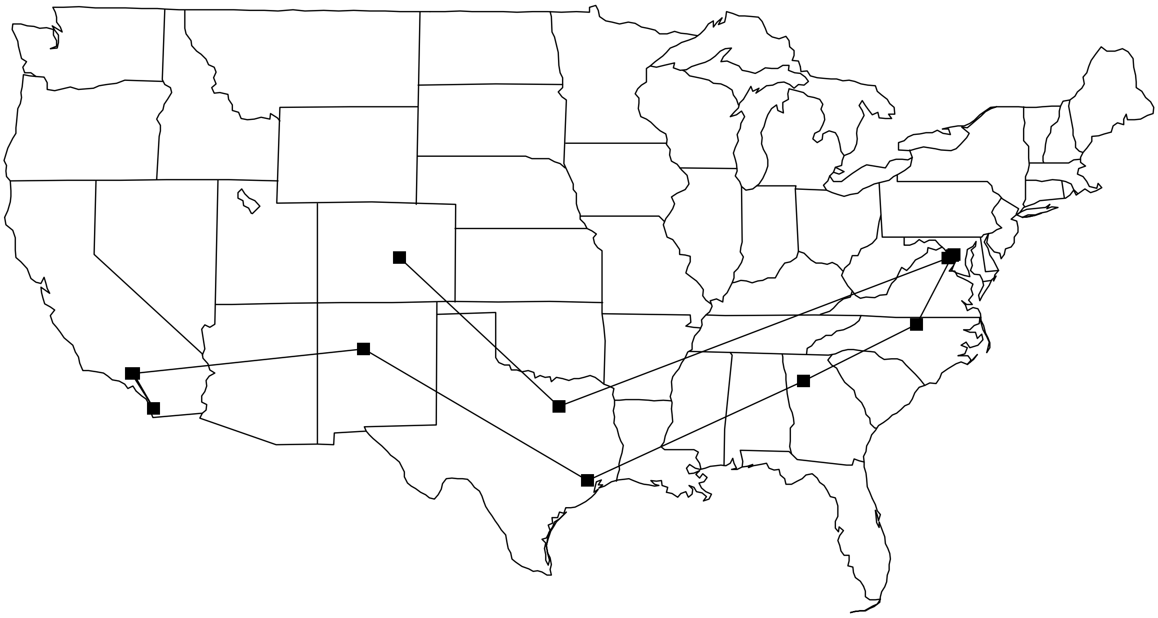

Figure 6.4: Circuitous route from `bsdi` to `usc`

Note that § 6.7.2 provides separate evidence that the time taken for routers to recover from the loss of a next-hop router is exponentially distributed, with a mean of 50 seconds (shorter than the $\mathcal{R}_1$ fit, but in agreement with the $\mathcal{R}_2$ data).

6.9 Circuitous routing

Since the inception of the Internet Protocol, one of its main goals has been resilience in the presence of network failures [Cl88]. In this section we document some of the more circuitous routes the network found in order to maintain connectivity in the presence of failures. These routes do not represent pathologies *per se* but rather triumphs of robust routing, or, sometimes, simply the lack of the necessary infrastructure to take advantage of more direct routes.

Figure 6.4 shows a route used from `bsdi`, in Colorado Springs, Colorado, to `usc`, in Los Angeles, California. The route is perhaps three times longer than the `bsdi` route to `sri` (located in Northern California), which also makes a first hop to Dallas, Texas, but from there travels to San Jose, California, rather than to the East coast.

Figure 6.5 shows one of the routes used from `lbli`, in Berkeley, California, to `ucol`, in Boulder, Colorado. Here the packets travel all the way to the East coast, then back to the West coast, and finally over to Colorado. A more direct path, also present in our data, travels straight from New Mexico to Colorado. Presumably this link was unavailable during the time of the longer route.

Figure 6.6 shows a route from `nrao`, in Charlottesville, Virginia, to `wustl`, in St. Louis, Missouri. This route increases the distance of the more direct route we also observed (via Washington, D.C., and then straight to St. Louis) by roughly a factor of five.

Figure 6.7 shows an even more tortuous route to `wustl`, this time from `lbl`. The packets

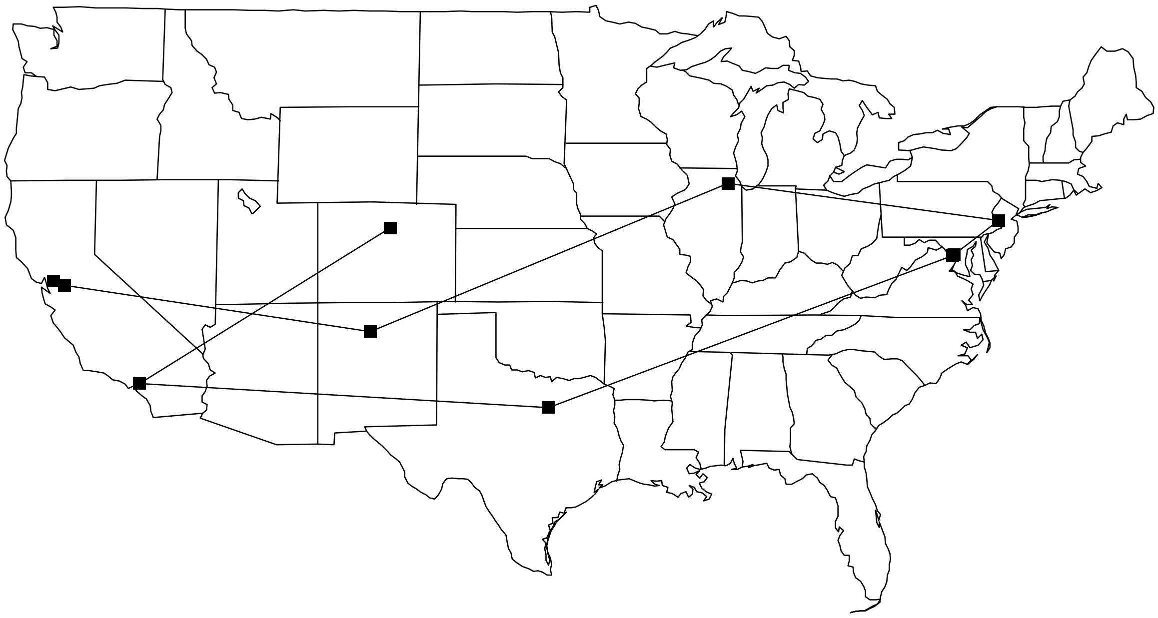

Figure 6.5: Circuitous route from `lbli` to `ucol`

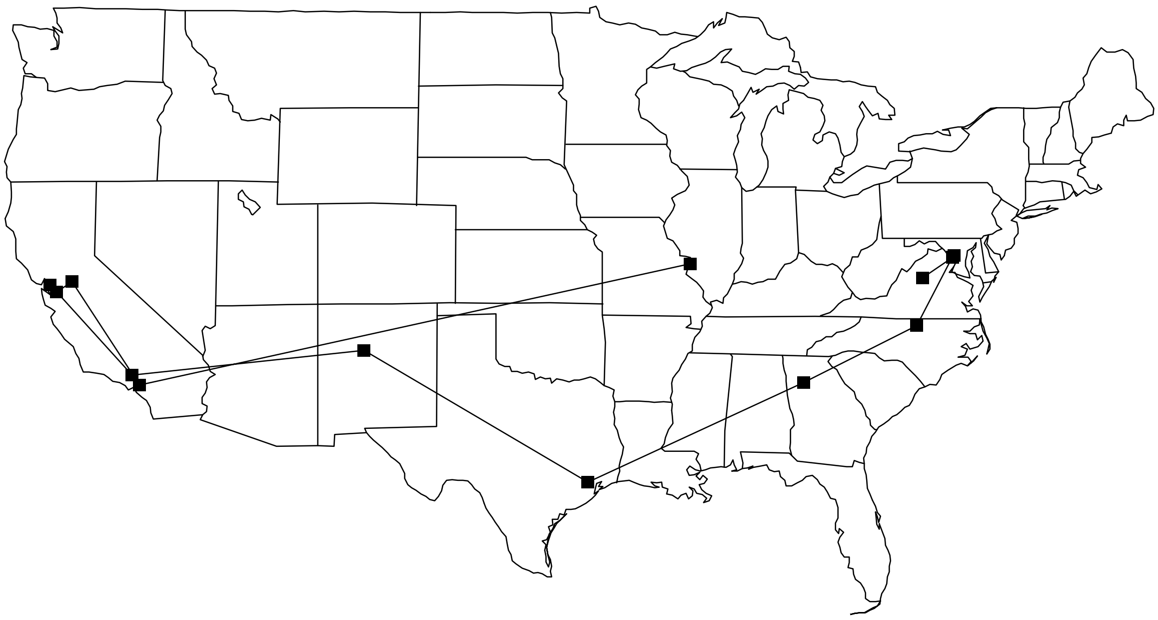

Figure 6.6: Circuitous route from `nrao` to `wustl`

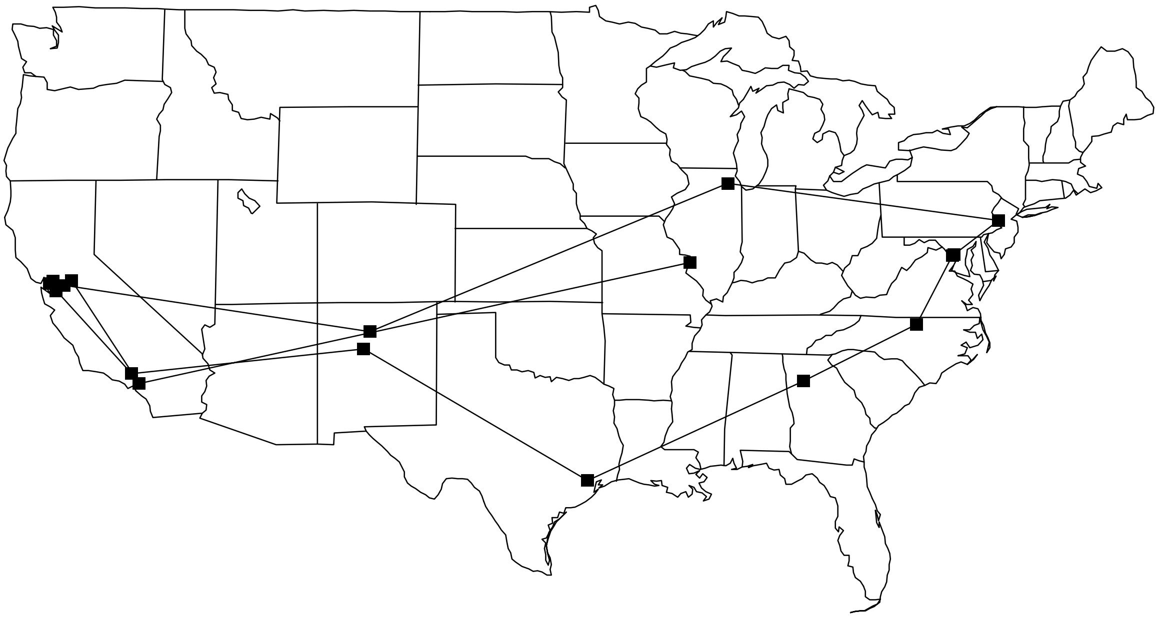

Figure 6.7: Circuitous route from `lbl` to `wustl`

first travel to Livermore, California, and then Los Alamos, New Mexico, via ESNET. They continue up to Illinois and across to Washington, D.C., via Princeton, New Jersey, and College Park, Maryland. They next take a southern route all the way back to northern California (!), back to southern California, and finally across to St. Louis. Figure 6.8 shows the 29 hops making up this path. One might be tempted to conclude that the path must have been the product of some sort of one-time glitch, but it showed up 5 different times in the $\mathcal{R}_1$ data.

In Figure 6.9 we see an illustration of the difficulties sometimes encountered even when going a very short distance. This route was the only one we observed from `ncar` to `xor` (8 observations total). `ncar` is located in Boulder, Colorado, and `xor` in East Boulder, Colorado, a few miles to the east. Yet the route between them visits the Gulf of Mexico and the East coast before crossing those few miles.

Circuitous routing is not limited to the United States. Figure 6.10 shows the route from `inria`, located in Southern France, to `oce`, located in the Netherlands, a few hundred kilometers to the North. The routing takes the packets across the Atlantic ocean to Vienna, Virginia (and nearby Falls Church), before crossing the Atlantic again to Amsterdam. The return path from `oce` to `inria` also follows this path, except in one instance the routing went from Amsterdam to Paris via Vienna, Austria (shown with a dotted line), rather than Vienna, Virginia. We speculated that perhaps the trans-Atlantic routing was due simply to accidental misconfiguration based on the similarities of the names; but we learned from EUnet personnel that much more likely the trans-Atlantic routing was intentional, due to its low-cost and higher available capacity compared to the underprovisioned intra-European links [Bi95].

Persistent circuitous routing might strike us as pathological, and unexpected in a well-run network. Because we do not know the underlying reasons for the routing configurations, we are unable from our data to answer why circuitous routing exists. We speculate, however, that

```
ir6gw.lbl.gov                        (Berkeley, CA)
er1gw.lbl.gov
lbl-lc2-1.es.net
llnl-lbl-t3.es.net                   (Livermore, CA)
lanl-llnl-t3.es.net                  (Los Alamos, NM)
fnal-lanl-t3.es.net                  (Batavia, IL)
pppl-fnal-t3.es.net                  (Princeton, NJ)
pppl-nis.es.net
umd-pppl.es.net                      (College Park, MD)
mf-0.enss145.t3.ans.net
t3-2.cnss56.washington-dc.t3.ans.net  (Washington, DC)
t3-1.cnss72.greensboro.t3.ans.net    (Greensboro, NC)
t3-0.cnss104.atlanta.t3.ans.net      (Atlanta, GA)
t3-2.cnss64.houston.t3.ans.net       (Houston, TX)
t3-0.cnss112.albuquerque.t3.ans.net  (Albuquerque, NM)
t3-1.cnss16.los-angeles.t3.ans.net   (Los Angeles, CA)
t3-2.cnss8.san-francisco.t3.ans.net  (San Francisco, CA)
t3-0.enss144.t3.ans.net              (Moffett Field, CA)
fix-w.icm.net
sl-stk-5-h2/0-t3.sprintlink.net      (Stockton, CA)
sl-stk-6-f0/0.sprintlink.net
sl-ana-2-h4/0-t3.sprintlink.net      (Anaheim, CA)
sl-ana-3-f0/0.sprintlink.net
sl-starnet2-1-s0-t1.sprintlink.net   (St. Louis, MO)
stl2-e0.starnet.net
ncrc-acn.wustl.edu
ncrc-eng.wustl.edu
jcr.ecl.wustl.edu
tango.cs.wustl.edu
```

Figure 6.8: Individual routers comprising circuitous path from `lbl` to `wustl`

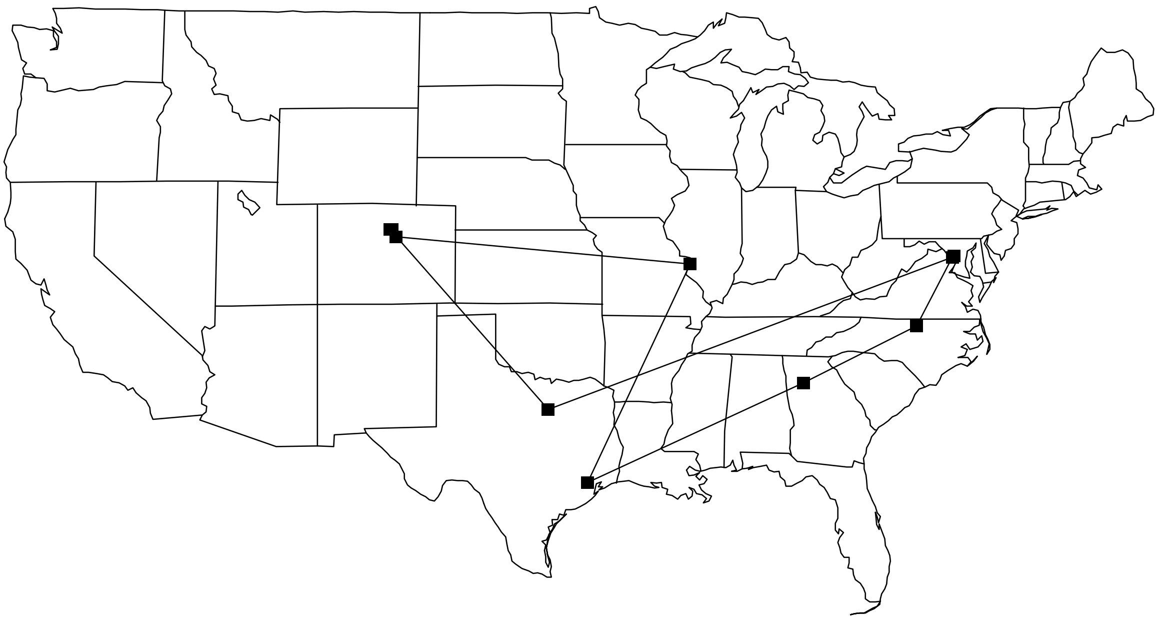

Figure 6.9: Circuitous route from `ncar` to `xor`

Figure 6.10: Circuitous route from `inria` to `oce`

Pathology	Probability	Trend	Notes
Unresponsive routers	0.00–0.53%		Rare enough to not present a measurement problem.
Failure to decrement TTL	0.18% $\perp$ 0.06%	better	Downstream router visited prematurely.
Persistent routing loops	0.13–0.16%		Some lasted for hours.
Temporary routing loops	0.055–0.078%		
Erroneous routing	0.004–0.004%		Packets in $\mathcal{R}_1$ visited Israel! No instances in $\mathcal{R}_2$.
Connectivity altered mid-stream	0.16% $\perp$ 0.44%	worse	Suggests rapidly varying routes.
Infrastructure failure	0.21% $\perp$ 0.48%	worse	No dominant link.
Temporary outage $\geq$ 30 secs	0.96% $\perp$ 2.2%	worse	Outage duration distributed as constant plus exponential. This distribution in $\mathcal{R}_2$ is bimodal.
Total user-visible pathologies	1.5% $\perp$ 3.4%	worse	

Table X: Summary of representative routing pathologies

it may be an inevitable consequence of the structure of today's Internet: the network is so vast and heterogeneous, and so under-instrumented for purposes of diagnosing end-to-end ailments, that errors inexorably arise and persist for long periods of time.

6.10 Summary

Table X summarizes the routing pathologies we studied in this section. The table is confined to those pathologies for which we claim our samples are representative (§ 4.4). (So, for example, we omit the "fluttering" pathology, which was heavily dominated by a pair of sites in our study; and also "host down," and "stub network outage.") The first part of the table reflects pathologies that are *not* in general visible to an end-to-end user of the network; that is, their presence does not significantly impact most network users. The second part of the table summarizes pathologies that *are* user-visible.

The second column gives the probability of observing the pathology, in two forms. When the probability is given as a range, such as for "persistent routing loops," then the proportion of observations of the pathology in $\mathcal{R}_1$ was consistent with the proportion in $\mathcal{R}_2$ (using the methodology in § 4.5). The range reflects the values spanned by the two datasets.

When the table lists two probabilities separated by "$\perp$," then the proportion of $\mathcal{R}_1$ observations was *inconsistent*, with 95% confidence, with the proportion of $\mathcal{R}_2$ observations. The first probability applies to the $\mathcal{R}_1$ measurements, and reflects the state of the Internet at the end of 1994; and the second to the $\mathcal{R}_2$ measurements, reflecting the state at the end of 1995.

For those pathologies with inconsistent probabilities, the third column assesses the trend during the year separating the $\mathcal{R}_1$ and $\mathcal{R}_2$ measurements. A trend of "better" indicates that the situation improved, and "worse" that it degraded. One pathology improved significantly: the likelihood of a router failing to decrement the TTL decreased. This change likely reflects upgraded and more stable router software.

Note though that this pathology is of no interest to end-to-end users of the network—improvements in the pathology do not reflect any significant gains in network service for the user. On the other hand, of the pathologies given in the second part of the table, which *are* of interest to users, *none of them improved!*, and *a number became significantly worse*.

The final row summarizes the total probability of observing a user-visible pathology. We note that: *During 1995, the likelihood of a user encountering a serious end-to-end routing problem more than doubled, to 1 in 30.* The most prevalent of these problems was an outage lasting more than 30 seconds.

This finding raises concerns regarding the long-term stability of the Internet. Clearly, if the trend continues, then network service will degrade to unacceptable levels. Unfortunately, from only two points in time it is impossible to assess the actual likelihood of the trend continuing.

Finally, we note that, for reasons given in § 5.2, our estimates of the prevalence of pathologies are biased towards underestimation; the true figures are most likely somewhat higher.

Chapter 7

End-to-End Routing Stability

One key property we would like to know about an end-to-end Internet route is its *stability*: do routes change often, or are they stable over time? In this section we analyze the routing measurements to address this question. We begin by discussing the impact of routing stability on different aspects of networking, to motivate our study, and summarizing the reasons why routes change. We then present two different notions of routing stability, "prevalence" and "persistence," and show that they can be orthogonal (i.e., a route can be considered "stable" by one definition independently of whether it is stable by the other definition).

It turns out that "prevalence" is quite easy to assess from our measurements, and "persistence" quite difficult. In § 7.5 we characterize the "prevalence" stability of the routes, and then in § 7.6 we tackle the problem of assessing "persistence."

We finish by evaluating a method for *detecting* route changes based on observing changes in hop count (TTL). We find this method makes a decent heuristic, but generates enough "false negatives" that it should not be trusted if accuracy is crucial.

7.1 Importance of routing stability

One of the stated goals of the Internet architecture is that large-scale routing changes (i.e., those involving different autonomous systems) rarely occur [Li89]:

> The Inter-AS Routing scheme must provide stability of routes. It is totally unacceptable for routes to vary on a frequent basis. This requirement is not meant to limit the ability of the routing algorithm to react rapidly to major topological changes, such as the loss of connectivity between two AS's. The need for adaptive routing does not imply any desire for load-based routing.

This point has been argued by others as well [BE90, Tr95b]. Routing instability sets the foremost limit on how use of BGP can scale to a very large internet, because CPU utilization required by BGP routers increases directly in proportion to the frequency of routing changes (but not, otherwise, in proportion to the overall size of the network) [Tr95b]. Hence, the key concern is that routing instability can in turn lead to general network instability (i.e., loss of packet-forwarding function).

There are a number of aspects of networking affected by routing stability:

1. Some of the most important properties of a network—latency, bandwidth, congestion levels, packet losses—are all *route* properties. If the route through the network changes, so might some or all of these properties. Therefore, the degree to which a network's behavior is *predictable* is directly related to the stability of its routes. This is not to say that, even if the route remains stable, these properties will too. Rather, routing stability is *necessary* but not *sufficient* for predictable network behavior.

   One particular example affected by routing stability is the *predictive service* scheme proposed for real-time network traffic [CSZ92]. Predictive service attempts to satisfy the performance requirements of real-time traffic by only admitting new real-time flows if recent traffic measurements suggest the network has sufficient capacity for them. If routes are unstable over short time scales, however, then these predictions become considerably difficult to make.

2. The degree to which endpoints can benefit from *caching* information of previously encountered path conditions is limited by (among other factors) whether the route observed in the past is likely to be the same as the present route.

3. New network protocols supporting "real-time" applications such as audio and visual flows generally require establishing state in routers in order to assure that the flows receive the necessary performance. Real-time flows will often be long-lived, existing for time spans on the order of human interactions (minutes to hours) rather than computer interactions (milliseconds to seconds). If routing changes occur frequently, then these long-lived flows will be prone to losing the state they have established in the routers in the network, and will suffer outages or degraded service while they attempt to find alternate routes with sufficient resources.

   Some protocols use "hard state" in the routers, meaning that, if state information for a given flow is not present in the router, then the router will not forward the flow's packets [DB95, FBZ94]. Other protocols use "soft state" schemes in which, even if a router has no corresponding state information for the flow, it will forward a flow's packets, though with possibly degraded performance [ZDESZ93, BCS94, DEFJLW94]. Hard state and soft state schemes trade off performance guarantees versus flexibility in the face of errors. Part of the question of evaluating the flexibility gain of soft state schemes concerns the degree of route stability. If routes do not tend to change frequently, then the soft state gain in flexibility is minor, but, if routes change frequently, then the gain will be larger.

   For an overview of the difficulties of dealing with routing changes in real-time protocols, see [GR95]. We do not attempt here to evaluate the flexibility gain of soft state versus hard state schemes. Indeed, the question is much more complex than stated above[1]. But we do attempt to characterize the stability constants that could then be used in such an evaluation.

4. Another form of router state arises from schemes for supporting *advance reservations*, in which the network allows resources to be reserved for future use [FGV95]. If the state con-

[1] For example, both types of schemes often use "route pinning," in which the route available when a flow is established remains the route used by that flow for its lifetime. If a route is pinned, then only route changes due to the *failure* of a router used by the flow affect the flow; not those due to the discovery of improved routes (§ 7.2).

Similarly, some hard state schemes have explicit recovery mechanisms for when a flow's route *does* fail ([Ba94, DB95, GR95]), so these schemes do not necessarily stop working in the presence of route changes.

cerning these reservations is stored in the network's routers (a logical choice, to avoid centralized bottlenecks), then frequent route changes may lead to reservations failing because the routers used to establish the reservations are no longer the routers relevant to the real-time path.

5. If routes change frequently, then network measurements face difficult consistency problems. For example, several studies of end-to-end network behavior rely on repeated measurements of a network path made over the course of hours to days [Mi83, CPB93a, Bo93, SAGJ93, Mu94, BCG95]. Whether these measurements all observe the same path significantly affects the accuracy of the studies.

   Similarly, distributed algorithms for analyzing the network's state also face consistency problems if routes change frequently. For example, recent theoretical work has developed "tomography" techniques for inferring end-to-end network traffic intensities using just measurements of aggregate traffic intensities along the network's links [Va95]. The work assumes stable routing (an extension explores Markovian routing). If routes change frequently, then it may prove extremely difficult to capture a consistent global snapshot of any significant portion of the Internet for purposes of operational monitoring.

We now look briefly at why routes change, and then introduce two different notions of routing stability, to encompass the different stability concerns discussed above.

7.2 Why routes change

There are several different reasons why a route might change:

1. If a link or router *fails*, then the network must reroute traffic using that link or router.

2. If a link or router *recovers*, then the network *may* elect to route previously redirected traffic back to using that link or router. If routes are "pinned," however, then they will not be changed due to recoveries.

3. If a link *degrades* or *improves*, where such notions might for example be measured by congestion levels, then the network might *adapt* by changing routes to account for the altered view of the cost of the link. For example, the ARPANET routing algorithms were designed to route around congested areas of the network. As experience with the ARPANET showed, such adaptive routing is tricky to get right: the initial routing scheme reacted "very quickly to good news, and very slowly to bad news" [MFR78], and the first revision of the algorithm [MRR80] also exhibited oscillations under heavy load [KZ89]. Because it is difficult to achieve stable adaptive routing, in which routes are not subject to rapid oscillation in response to transient congestion, adaptive routing is not widely used [Mo95], and a number of researchers argue for caution in its use [ERH92, RG95].

4. A router might cycle between different routes to the same destination in order to *balance load*. We analyzed this sort of route "flutter" in § 6.6, where we found that often its effects are confined to a single hop in an Internet path, but sometimes the split routes fail to rejoin, leading to drastically different path characteristics.

We would hope to observe four different time constants associated with these four reasons, of decreasing durations. Link failures should occur only rarely, hopefully on the time scale of days. Link recoveries should occur significantly quicker (i.e., shortly after the link failure), on the time scale of minutes (if a reboot or restart is all that is required) to hours (if human intervention and repair is required). If adaptive routing is used, then changes should occur on the time scales of congestion epochs (unfortunately not well characterized in the literature), which one presumes is on the order of seconds to minutes; adaptive routing algorithms generally *damp* rapid changes, though, to avoid oscillations, so we would expect this time constant to be more on the order of minutes. Finally, load balancing is generally done on very small time scales (such as every other packet), on the order of milliseconds.

7.3 Two definitions of stability

As suggested in § 7.1, there are two distinct views of routing stability. The first is: "Given that I observed route r at time t, how likely am I to observe r again at time $t + s$?" We refer to this notion as *prevalence*. A route's prevalence directly affects the first two motivations discussed above, namely predictability of service, and our ability to learn from past conditions. In general, the degree of route prevalence will depend on s. For large s, however, we would expect the observation at time $t + s$ to be (nearly) independent of the observation at time t. In this study, for simplicity we focus on the unconditional probability of observing a route, confining our analysis to $s \to \infty$, i.e., the steady-state probability of observing r again at a point far in the future. We leave the interesting question of how prevalence evolves for different intervals s for future work.

A second view of stability is: "Given that I observed route r at time t, how long before that route is likely to have changed?" The likelihood of routes changing in the near future has implications for the latter three motivations, namely hard and soft router state, resource reservations, and network measurement consistency. We refer to this notion as *persistence*.

Intuitively, we might expect these two notions to be coupled. Consider, for example, a sequence of routing observations made every T units of time. If the routes we observe are:

$$R_1, R_1, R_1, R_1, R_1, R_1, R_1, R_1, R_1, R_1, R_1, R_2, R_1, R_1, R_1 \ldots$$

then clearly route R_1 is much more prevalent than route R_2. We might also conclude that route R_1 is persistent, because we observe it so frequently; but this is not at all necessarily the case. For example, suppose T is one day. If the mean duration of R_1 is actually 10 days, and that of R_2 is one day, then this sequence of observations is quite plausible, and we would be correct in concluding that R_1 is *persistent and prevalent*. Furthermore, depending on our concern, we might also deem that R_2 is persistent, since on average it lasts for a full day (if its lifetime were much shorter, then we would have been unlikely to observe it from measurements made only once a day). If we consider a route that last for more than a few hours as persistent, then from the above observations we could argue that R_2 is *persistent but not prevalent.*

But suppose instead that the mean duration of R_1 is 10 seconds and the mean duration of R_2 is 1 second, and that alterations between them occur as a semi-Markov process,[2] where state 1

[2] Such processes consist of a set of states. Each state i has associated with it a distribution of durations, G_i. The distribution depends on the state number i, but not on anything else. Upon entering state i, a duration is drawn independently

of the process corresponds to R_1, state 2 to R_2, and $P_{1,2} = P_{2,1} = 1$ (i.e., whenever a change occurs, it is a change to the other route). Then a well-known result from the theory of stochastic processes states that the proportion of time the system spends in state 1 is equal to the mean duration of state 1 divided by the sum of the mean durations of states 1 and 2 [Ro83]. For our example, we have that the proportion of time spent in state R_1 is $\frac{10}{11}$, reflecting that R_1 is prevalent. Similarly, the proportion of time spent in state R_2 is $\frac{1}{11}$. Given these proportions, the sequence of observations is *again plausible*, even though each observation of R_1 is actually of a separate instance of the route. In this case, R_1 is *prevalent but not persistent*, and R_2 is *neither prevalent nor persistent*. In other words, we very likely are missing instances of R_2 between observations of R_1, and hence R_1 is not persistent.

This example shows that the notions of "prevalent" versus "persistent" stability are orthogonal, in the sense that the presence or absence of one does not necessarily indicate anything about the presence or absence of the other.

7.4 Reducing the data

To begin our analysis, we first need to reduce the more than 40,000 `traceroutes` measurements in $\mathcal{R}_1$ and $\mathcal{R}_2$ to those relevant for assessing stability. Before we had gathered the $\mathcal{R}_2$ measurements, we performed an initial stability analysis of the $\mathcal{R}_1$ data. Doing so, we concluded that the inter-measurement spacing of the $\mathcal{R}_1$ `traceroutes` , on average about one day, was too large to allow any assessment of routing stability in terms of persistence, because of the ambiguities discussed in the previous section. Consequently, we confine our routing stability analysis to $\mathcal{R}_2$, which contains the bulk (85%) of the 40,000 measurements. 60% of these were taken with a 2-hour inter-measurement spacing. As shown in the remainder of this chapter, this granularity is sufficient to resolve the persistence ambiguities.

Of the 35,109 $\mathcal{R}_2$ measurements, we began by excluding those exhibiting the pathologies discussed in Chapter 6, because they reflect connectivity difficulties distinct from routing instabilities.[3] (We did not exclude "circuitous" routes, however, because, as mentioned in § 6.9, these are not true pathologies.) Doing so eliminated 805 `traceroutes`.

We also omitted `traceroutes` for which one or more hops were completely missing (all three of the probe packets unanswered). These measurements are inherently *ambiguous*, because we could not tell if the route was the same as that observed at other instances. This decision eliminated another 2,595 measurements, leaving us with a total of 31,709 measurements.

We next made a preliminary assessment of the patterns of route changes by seeing which changes occurred the most frequently. We found the pattern of changes dominated by a number of

---

from G_i. The process remains in state i until the duration elapses. At this point, a new state j is chosen based on a set of probabilities fixed for state i.

[3] An exception is the pathology of a routing change during a `traceroute`. Including these pathologies, however, can lead to overestimating the frequency of route changes. Suppose we make three route measurements of a particular path, yielding routes A, A/B, and B, where A/B indicates a `traceroute` that included a change from route A to route B. If we included the second, pathological measurement, we would conclude that over the three observations two changes occurred (A to A/B and A/B to B), whereas in reality only one change occurred (A to B).

It is possible that instead the sequence we observe is A, A/B, A, because route B was short-lived; in this case, omitting the pathological `traceroute` underestimates the frequency of changes. But this becomes an issue only if B was quite short-lived, and we account for such routes separately, as discussed in § 7.6.1.

Routers	Notes
asd01.nl.net, amf01.nl.net	These routers are located in different cities, but provide equal bandwidth and latency to their peers [Lin96].
icm-dc-1.icp.net, icm-dc-2b-s4/0-1984k.icp.net	
rgnet-b1-serial2-3.seattle.mci.net, rainnet-inc.seattle.mci.net	
rb1.rtr.unimelb.edu.au, rb2.rtr.unimelb.edu.au	
unit-gw.unit.no, sintef-gw.sintef.no	Both at the University of Trondheim.

Table XI: Tightly-coupled routers

single-hop differences, at which consecutive measurements showed exactly the same path except for a single router. Furthermore, the names of these routers often suggested that the pair were administratively interchangeable.[4] For example, many of the routing changes to the `austr` site only differed in whether the University of Melbourne border router in the route was `rb1.rtr.unimelb.edu.au` or `rb2.rtr.unimelb.edu.au`. Which of these two routers provides the route to the `austr` host depends on the distribution of load within other parts of the University, but the two routers are under the same direct administration and would indeed be one machine if a single router with sufficient capacity had been available at the time of acquisition [El96].

It seems likely that many route changes differing at just a single hop are due to shifting traffic between two tightly coupled machines. For the stability concerns given in § 7.1, such a change is likely to have little consequence, provided the two routers are co-located and capable of sharing state. We decided that, if a single pair of routers with like names were responsible for more than 200 routing transitions, then we would classify them as "tightly coupled," and merge them into a single router for purposes of evaluating stability. Table XI summarizes these routers. After merging those responsible for > 200 changes, the remaining pairs were all responsible for 80 or fewer changes. We left these as separate routers, as changes between them did not dominate the data, and we would like to minimize assumptions about which routers are tightly coupled.

Finally, we reduced the acceptable routes to three different levels of *granularity*. First, we considered each route as a sequence of Internet hostnames. We call this *host* granularity. We then reduced the routes to sequences of *cities*, as outlined in § 5.3. Note that a route change at host granularity might *not* be a route change at *city* granularity, though the converse always holds. The motivation behind the distinction of host granularity vs. city granularity is to introduce a notion of "any change" vs. "major change." A route change at city granularity will likely have considerably more repercussions than a change visible only at host granularity. For example, the latency of the route will often be different. Overall, 57% of the route changes at host granularity were also route changes at city granularity.

[4] Sometimes the routers *were* identical. For example, IP address `192.157.65.130`, which translates to `icm-paris-1-s0-1984k.icp.net`, is actually also an interface on `paris-ebs2.ebone.net`.

The third level of granularity was *AS path*—the sequence of autonomous systems visited by the route (§ 4.4). A change at *AS* granularity reflects a possible change in the intermediate routing algorithms and policies, and as such is another form of major change. Overall, 36% of the route changes at host granularity were also changes at AS path granularity. Note that a change at AS path granularity is not necessarily a change at city granularity, nor vice versa, though overall we found AS path granularity coarser (i.e., comprising fewer changes) than city granularity.

7.5 Routing Prevalence

In this section we look at routing stability from the standpoint of *prevalence*: how likely we are, overall, to observe a particular route (c.f. § 7.3). We can associate with prevalence a parameter π_r, the steady-state probability that a path at an arbitrary point in time uses a particular route r.

We can assess π_r from our data as follows. We suppose that routing changes follow a semi-Markov process. In this model, each route's duration has a fixed distribution (but different routes can have different distributions), and the duration of each instance of a route is independent of all previous route durations. Furthermore, the probability that route r_1 is followed by route r_2 is fixed and independent of past events.

We then use the result that, for a semi-Markov process, the steady-state probability of observing a particular state is equal to the average amount of time spent in that state [Ro83].[5] Furthermore, because of PASTA, our independent exponential sampling gives us an unbiased estimator of this time average (§ 4.3). Suppose we make n observations of a path and k_r of them find state r (i.e., route r). Then we will estimate π_r as $\hat{\pi}_r = k_r/n$.

We proceed as follows. For a particular path p (and for a given granularity), let n_p be the total number of `traceroutes` measuring that path, and d_p the number of distinct routes seen. We will denote the most commonly occurring route as the *dominant* route, and others as *secondary* routes. Thus, there are always $d_p - 1$ secondary routes. Let k_p be the number of times we observe the dominant route. We then confine our analysis to:

$$\hat{\pi}_{\text{dom p}} = k_p/n_p,$$

the prevalence of the dominant route.

Figure 7.1 shows the cumulative distribution of the prevalence of the dominant routes over all of the paths in our study (i.e., all 1,054 source/destination pairs), for the three different granularities. For example, at host granularity, nearly half (49%) of the paths (y-axis) were dominated by a route with a prevalence of at least 80% (x-axis).

There is clearly a wide range, particularly for host granularity. For example, for the path between `pubnix` and `austr`, in 46 measurements we observed 9 distinct routes at host granularity, and the dominant route was observed only 10 times, leading to $\hat{\pi}_{\text{dom}} = 0.217$. On the other hand, at host granularity more than 25% of the paths exhibited only a single route ($\hat{\pi}_{\text{dom}} = 1$). For city and AS path granularities, the spread in $\hat{\pi}_{\text{dom}}$ is more narrow, as would be expected (the figure also

[5] This result requires that the distribution of time spent in each state be *nonlattice*: i.e., not always an integral multiple of some constant, so that the notion of “steady state” can be defined without reference to specifics about exactly when, in the far future, we observe the process. For route durations, this seems like a plausible assumption.

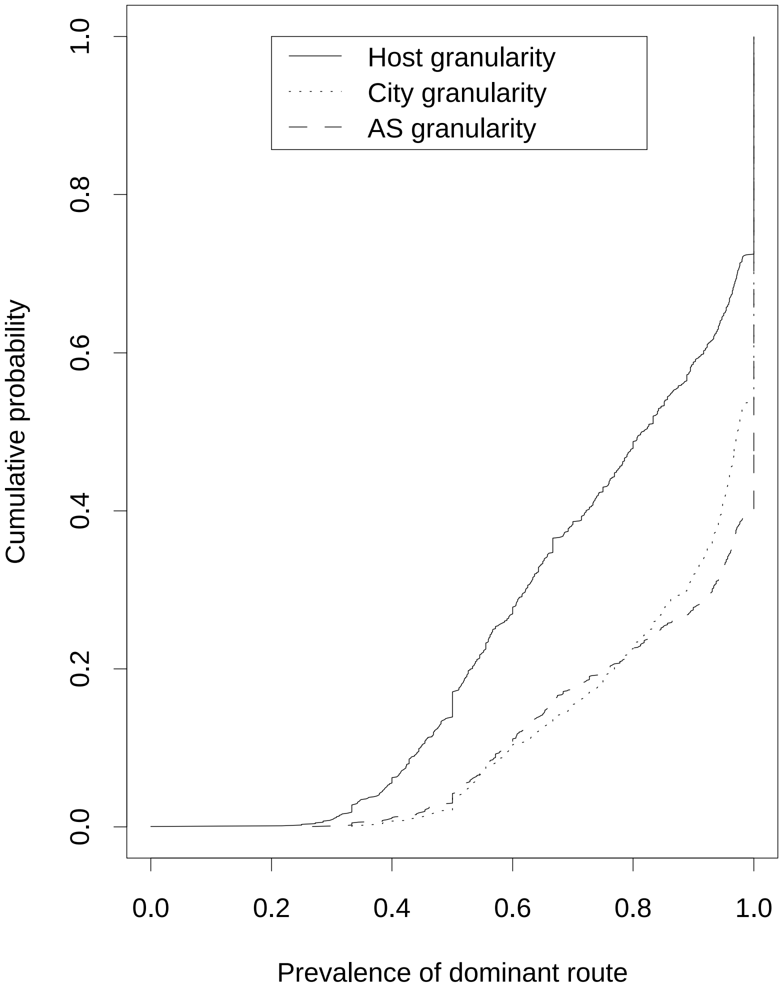

Figure 7.1: Fraction of measurements observing the dominant route, for all paths, at all granularities

shows how route changes at city or AS path granularity do not necessarily imply changes at the other granularity, since neither is strictly below the other).

A key figure to keep in mind from this plot, however, is that, while there is a wide range in the distribution of $\hat{\pi}_{\text{dom}}$ over different paths, its *median* value at host granularity is 82%; 97% at city granularity, and 100% at AS path granularity. The clustering of many paths only ever exhibiting a single route (i.e., prevalence = 100%) reflects the finding we develop below in § 7.6 that many routes are long-lived. (If we had data gathered over periods of time exceeding several weeks, we would doubtless find that the spike at prevalence = 100% would spread out to values in the upper 90%'s.) Thus, we can conclude: *In general, Internet paths are strongly dominated by a single route.*

Our previous work, however, has shown that many characteristics of network traffic exhibit considerable site-to-site variation [Pa94a], and thus it behooves us to assess the differences in $\hat{\pi}_{\text{dom}}$ between the sites in our study. To do so, for each site s (and for each granularity) we computed:

$$\hat{\pi}_{\text{src s}} = \frac{\sum_{\text{src paths } s_i} k_{s_i}}{\sum_{\text{src paths } s_j} n_{s_j}}.$$

where k_{s_i} is the number of times we observed the dominate route when measuring a path from source s to destination i, and n_{s_j} is the total number of times we made a measurement of the path from source s to destination j.

The aggregate estimate $\hat{\pi}_{\text{src s}}$ then indicates the overall prevalence of dominant routes from s to different destinations. We expect variations in this estimate for different sites to reflect differing routing prevalence due to route changes *near* the source. Route changes further downstream from the source occur either deep inside the network (and so will affect many different sites), or near the destination (and thus will not affect any particular *source* site unduly).

Similarly, we can construct $\hat{\pi}_{\text{dst s}}$ for all of the paths with destination s. Studying $\hat{\pi}_{\text{src s}}$ and $\hat{\pi}_{\text{dst s}}$ for different sites and at different granularities reveals considerable site-to-site variation, in agreement with the general findings in [Pa94a]. Figure 7.2 shows the values computed for $\hat{\pi}_{\text{src s}}$ for each of the $\mathcal{R}_2$ sites, at host granularity. We find that the prevalence of the dominant routes originating at the `ucl` source is under 50% (we will see in § 7.6.1 the main cause for this), and for `bnl`, `sintef1`, `sintef2`, and `pubnix` is around 60%; while for `ncar`, `ucol`, and `unij`, it is just under 90%. Even at AS path granularity, the `ucl` source has an average prevalence of 60%, with `ukc` about 70%, and the remainder from 85% to 99%. At city granularity, however, the main outlier is `bnl`, with a prevalence of 75% (c.f. § 7.6.2), because the `ucl` and `ukc` instabilities, while spanning autonomous systems, do not span different cities.

We find similar spreads for $\hat{\pi}_{\text{dst s}}$ for different destination sites s. Figure 7.3 shows the per-site values, computed for host granularity. Sometimes the sites with low overall prevalence are the same as the sites with low prevalence for $\hat{\pi}_{\text{src s}}$ (e.g., `ucl`), and sometimes they are different (e.g., `ukc`); this variation is due to *asymmetric* routing, which we analyze in Chapter 8.

We can thus summarize routing prevalence as follows: *In general, Internet paths are strongly dominated by a single route, but, as with many aspects of Internet behavior, we also find significant site-to-site variation.*

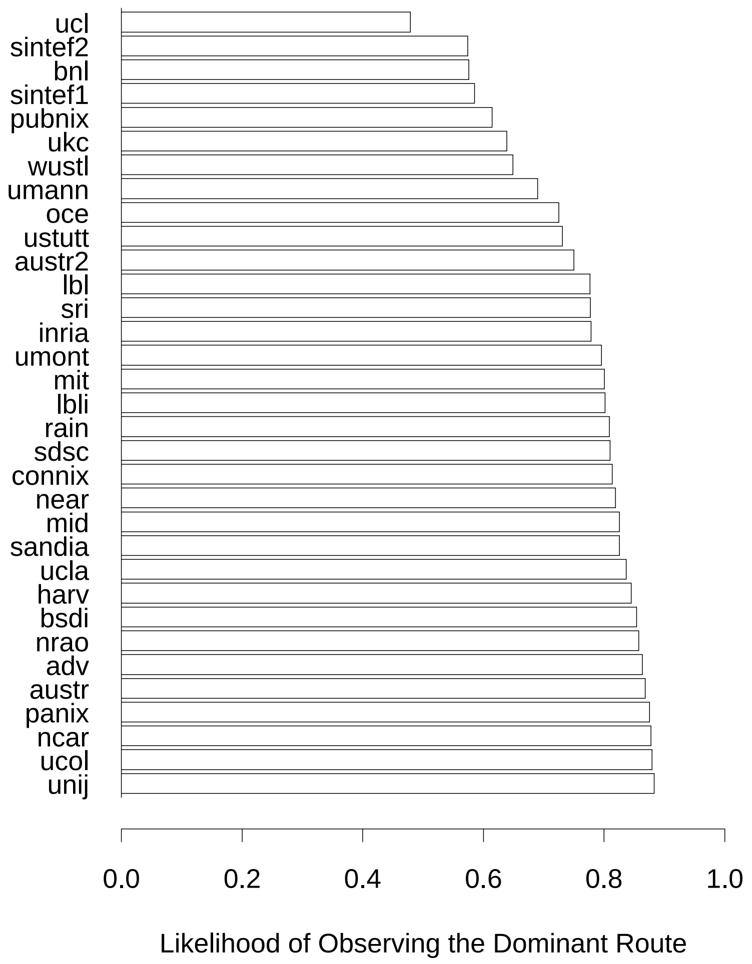

Figure 7.2: Fraction of measurements observing the dominant route, for different source sites, at host granularity

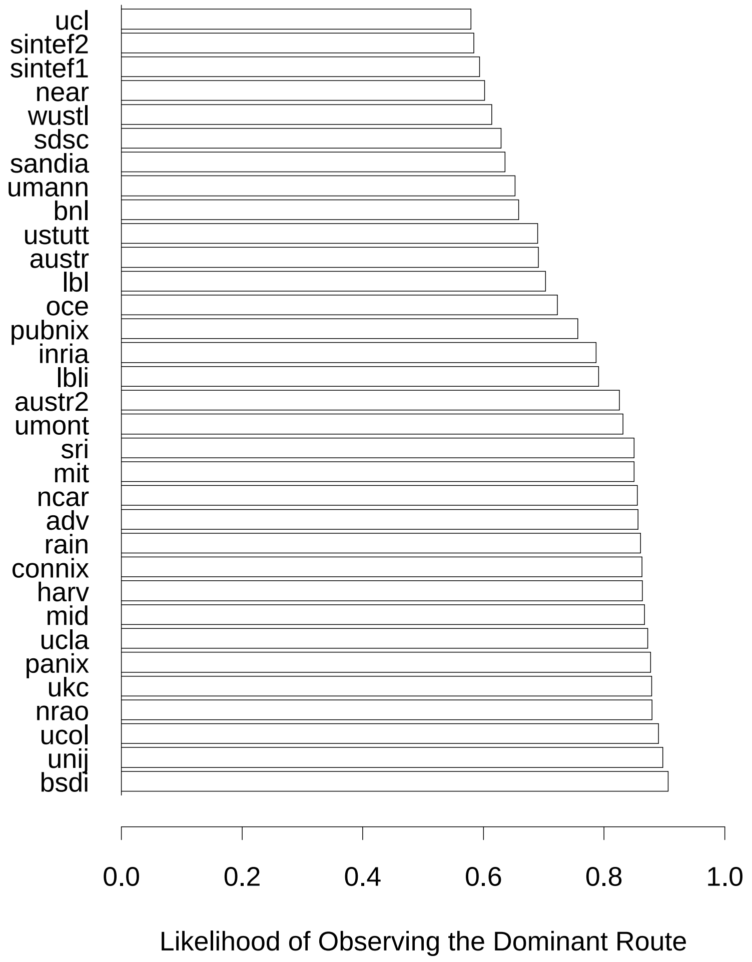

Figure 7.3: Fraction of measurements observing the dominant route, for different destination sites, at host granularity

7.6 Routing Persistence

We now turn to the more difficult task of assessing the *persistence* of routes: How long they are likely to endure before changing. As illustrated in § 7.3, unlike *prevalence*, routing persistence can be difficult to evaluate because a series of measurements at particular points in time do not necessarily indicate a lack of change *and then change back* in between the measurement points. Thus, to accurately assess persistence requires first determining whether routing alternates on short time scales. If not, then we can trust shortly spaced measurements observing the same route as indicating that the route did indeed persist during the interval between the measurements. If shortly spaced measurements can be trusted in this fashion, then they can be used to assess whether routing alternates on medium time scales.

Fortunately, we have measurements made at a number of different intervals: about 60% of the $\mathcal{R}_2$ measurements were exponentially distributed with a mean of 2 hours, and the other 40% with a mean of about 66 hours (with wide variation in the actual intervals, since they were exponentially distributed). While these measurements do not allow us to directly address the problem of assessing persistence—doing so would require a way to unambiguously determine exactly when a route changed, which could be done by tracing BGP routing information exchanges,[6] but not from end-to-end `traceroutes`—our strategy is to analyze the measurements with the shorter spacing to assess the frequency of route alternations, and, in turn, to determine to what degree we can trust the measurements with larger spacing. In this fashion, we aim to "bootstrap" ourselves into a position to be able to make sound characterizations of routing persistence across a number of time scales.

7.6.1 Rapid route alternation

In order to reliably analyze widely-spaced `traceroute` measurements, we must first assess the predominance of rapidly alternating routes. We have already identified two types of rapidly alternating routes, those due to "flutter" and those due to "tightly coupled" routers. We have separately characterized fluttering (§ 6.6) and consequently have not included paths experiencing flutter in this analysis. As mentioned in § 7.4, we merged tightly coupled routers into a single entity, so their presence also does not further affect our analysis of rapidly alternating routes.

We next note that in $\mathcal{R}_2$ we observed 155 instances of a route change during a `traceroute`. The combined amount of time observed by the 35,109 $\mathcal{R}_2$ `traceroutes` was 881,578 seconds. (That is, the mean duration of a $\mathcal{R}_2$ `traceroute` was 25.1 seconds.) Since when observing the network for 881,578 seconds we saw 155 route changes, we can estimate that on average we will see a route change every 5,687 seconds ($\approx$ 1.5 hours). This reflects quite a high rate of route alternation, and bodes ill for relying on measurements made much more than a few hours apart (though see § 7.6.2); but it is not such a high rate that we would expect to completely miss routing changes for sampling intervals significantly less than an hour.

We first looked at those `traceroute` measurements that were made less than 60 seconds apart. There were only 54 of these, but all of them were of the form "R_1, R_1"—i.e., both of the measurements observed the same route. This provides credible, though not definitive, evidence that

[6] As briefly mentioned in § 3.2, recent work by Jahanian, Labovitz and Malan pursues this approach with very interesting results [JLM97]. We became aware of this work too late to discuss it here, but will address it in the version of [Pa96b] that we are presently revising for publication in *IEEE/ACM Transactions on Networking*.

there are no additional widespread, high-frequency routing oscillations, other than those we have already characterized.

We then looked at measurements made less than 10 minutes apart. There were 1,302 of these, and 40 *triple* observations (three observations all within a ten minute interval). The triple observations allow us to double check for the presence of high-frequency oscillations: if we observe the pattern R_1, R_2, R_1 or R_1, R_2, R_3, then we are likely to miss some route changes when using only two measurements 10 minutes apart. If we only observe R_1, R_1, R_1; R_1, R_2, R_2; or R_1, R_1, R_2, then measurements made 10 minutes apart are not missing short-lived routes. Of the 40 triple observations, none were of the form R_1, R_2, R_1 or R_1, R_2, R_3, confirming the finding from the 60 second observations that there are no additional sources of high-frequency oscillation.

The 1,302 ten-minute observations included 25 instances of a route change (R_1, R_2). This suggests that the likelihood of observing a route change over a ten minute interval is not negligible, and requires further investigation before we can look at more widely spaced measurements.

A natural question to ask concerning 10-minute changes is whether they are equally likely to occur along paths between any two sites, or if just a few sites are responsible for most of the 10-minute changes.[7] This is an important consideration: if all paths are equally likely to exhibit a change during a 10-minute interval, then from the figure above of 25 changes observed out of 1,302 ten-minute observations we could conclude that routes change, on average, 25 times per $(1,302 \cdot 10$ min), or about once every eight hours.

We test whether paths to or from particular sites are more prone to change than others as follows. For each site s, let $N^{10}_{\text{src s}}$ be the number of 10-minute pairs of measurements originating at s, and $X^{10}_{\text{src s}}$ be the number of times those pairs reflected a *transition* (i.e., the pair was R_1, R_2). Similarly, define $N^{10}_{\text{dst s}}$ and $X^{10}_{\text{dst s}}$ for those pairs of measurements with destination s. Here we are aggregating, for each site, all of the measurements made using that site as a source (destination), in an attempt to see whether route oscillations are significantly more prevalent near a handful of the sites.

For each site s, we can then define:

$$P^{10}_{\text{src s}} = \frac{X^{10}_{\text{src s}}}{N^{10}_{\text{src s}}},$$

and similarly for $P^{10}_{\text{dst s}}$. These values then give the estimated probability that a pair of ten-minute observations of paths with source (or destination) s will show a routing change. We now check the $P^{10}_{\text{src s}}$ (and $P^{10}_{\text{dst s}}$) estimates for each site to determine which sites appear particularly prone to exhibiting changes during ten minute intervals.

Figure 7.4 shows the sorted $P^{10}_{\text{dst s}}$ estimates. We see, for example, that none of the 10-minute measurements of paths to the destination `adv` observed a route change, but more than 12% of those to `austr` did. From the plot, `austr` appears to be an outlier, and merits further investigation. Before removing it as an outlier, however, we must be careful to first look at its routing oscillations to see what patterns they exhibit.

For the destination `austr`, the 10-minute changes involve a number of source sites: `inria`, `mit`, `near` (twice), and `pubnix`. All of the changes take place at the point-of-entry

[7] Certainly no single path (between the same source/destination pair) is skewing the count of 10-minute changes, since the most frequently observed single path only accounted for 8 of the 1,302 observations.

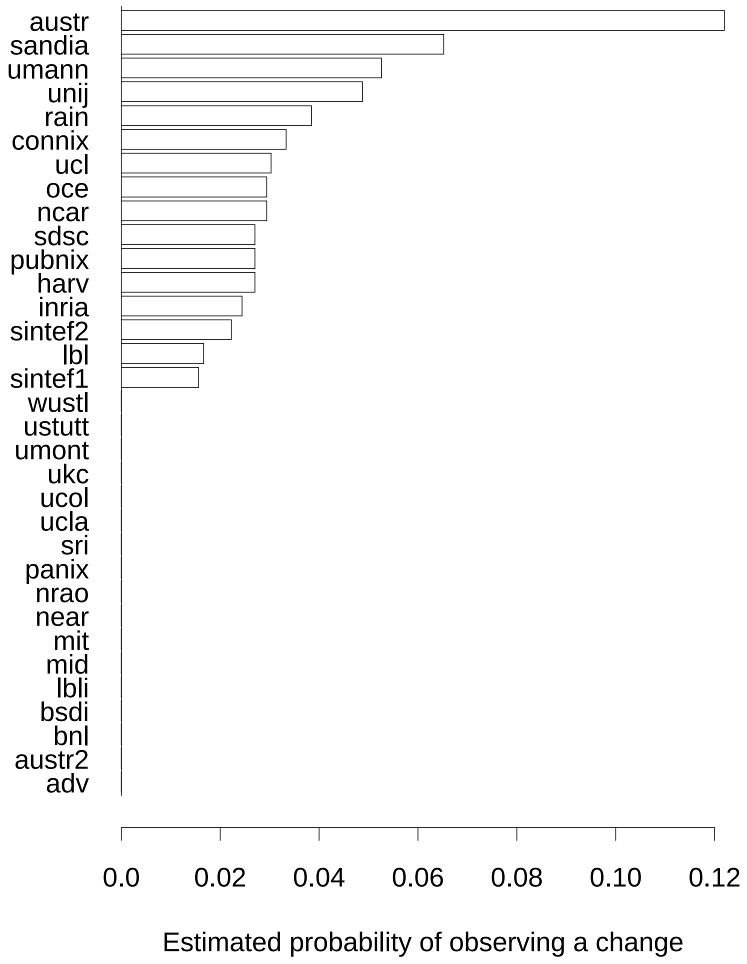

Figure 7.4: Site-to-site variation in $P^{10}_{\text{dst s}}$

into Australia.[8] The changes are either the first Australian hop of `vic.gw.au`, in Melbourne, or `act.gw.au`, in Canberra, or `serial4-6.pad-core2.sydney.telstra.net` in Sydney followed by an additional hop to `nsw.gw.au` (also in Sydney). These are the only points of change: before and after, the routes are unchanged. Thus, the destination `austr` exhibits rapid (time scale of tens of minutes) changes in its incoming routing, and these changes are non-negligible, since they involve different Australian cities. As such, the routing *to* `austr` is not at all persistent.

However, for the next potential outlier, `sandia`, the story is different. Both of its changes occurred along the path originating at `sri`, and reflected the following change at hops 8 and 9:

```
core-fddi-0.sanfrancisco.mci.net
borderx2-fddi0-0.sanfrancisco.mci.
```

versus:

```
core2-fddi-0.sanfrancisco.mci.net
borderx2-fddi-1.sanfrancisco.mci.net
```

These changes are localized to a single city. Furthermore, had this change been more prevalent, we might have decided that the two pairs of routers in question were “tightly coupled” (§ 7.4), except that it turns out that they are responsible for routing changes only between `sri` and `sandia`. Thus, we can deal with this outlier by just eliminating the path `sri` $\Rightarrow$ `sandia`, but keeping the other paths with destination `sandia`.

In addition to the destination `austr`, a similar analysis of $P^{10}_{\mathrm{src\ s}}$ points up `ucl`, `ukc`, `mid`, and `umann` as outliers. Both `ucl` and `ukc` had frequent oscillations in the routers visited between London and Washington, D.C., alternating between the two hops of:

```
icm-lon-1.icp.net
icm-dc-1-s3/2-1984k.icp.net
```

and the four hops of:

```
eu-gw.ja.net
gw.linx.ja.net
us-gw.thouse.ja.net
icm-dc-1-s2/4-1984k.icp.net
```

Note that these different hops also correspond to different AS's, as the latter includes AS 786 (JANET) and the former does not. For `mid` and `umann`, however, the changes did not have a clear pattern, and their prevalence could be due simply to chance.

On the basis of this analysis, we conclude that the sources `ucl` and `ukc`, and the destination `austr`, suffer from significant, high-frequency oscillation, and excluded them from further analysis. After removing any measurements originating from the first two or destined to `austr`, we then revisited the range of values for $P^{10}_{\mathrm{src\ s}}$ and $P^{10}_{\mathrm{dst\ s}}$. Both of these now had a median of 0 observed changes, and a maximum corresponding to about 1 change per hour (this latter rate is computed by dividing the number of route changes observed for the site's paths by total amount of time spanned by the measurements of those paths). On this basis, we believe we are on firm ground treating pairs of measurements between these sites, made less than an hour apart, both observing the same route, as consistent with that route having persisted unchanged between the measurements.

[8] Note that in general the paths to `austr` and `austr2` use two different trans-Pacific links, which is why `austr2` does not exhibit these rapid changes.

7.6.2 Medium-scale route alternation

Given the findings in the previous section that, except for a few sites, route changes do not occur on time scales less than an hour, we now turn to analyzing those measurements made an hour or less apart to determine what they tell us about medium-scale routing persistence. We proceed much as in § 7.6.1.

Let $P^{\mathrm{hr}}_{\mathrm{src\ s}}$ and $P^{\mathrm{hr}}_{\mathrm{dst\ s}}$ be the analogs of $P^{10}_{\mathrm{src\ s}}$ and $P^{10}_{\mathrm{dst\ s}}$, but now for measurements made an hour or less apart. After eliminating the rapidly oscillating paths identified in the previous section, we have 7,287 pairs of measurements to assess.

The data also included 1,517 triple observations spanning an hour or less. Of these, only 10 observed the pattern R_1, R_2, R_1 or R_1, R_2, R_3, indicating that, in general, two observations spaced an hour apart are not likely to miss a routing change.

Plots similar to Figure 7.4 immediately pick out paths originating from `bnl` as exhibiting rapid changes. These changes are almost all from oscillation between `llnl-satm.es.net` and `pppl-satm.es.net`. The first of these is in Livermore, California, while the other is in Princeton, New Jersey, so this change is definitely major. ESNET oscillations also occurred on one-hour time scales in traffic between `lbl` (and `lbli`) and the Cambridge sites, `near`, `harv`, and `mit`.

The other prevalent oscillation we found was between the source `umann` and the destinations `ucl` and `ukc`. Here the alternation was:

```
ch-s1-0.eurocore.bt.net
uk-s1-1.eurocore.bt.net
```

which goes through Switzerland to reach England, versus

```
nl-s1-1.eurocore.bt.net
uk-s1-0.eurocore.bt.net
```

which goes through the Netherlands instead, also a major change.

Eliminating these oscillating paths leaves us with 6,919 measurement pairs. These paths are not statistically identical (i.e., we find among them paths that have significantly different route change rates), but all have low rates of routing changes. For these paths, the median $P^{\mathrm{hr}}_{\mathrm{src\ s}}$ and $P^{\mathrm{hr}}_{\mathrm{dst\ s}}$ correspond to one routing change per 1.5 days, and the maximum to one change per 12 hours.

7.6.3 Large-scale route alternation

Given that, after removing the oscillating paths discussed in § 7.6.1 and § 7.6.2, we expect at most on the order of one route change per 12 hours, we now can analyze measurements less than 6 hours apart of the remaining paths to assess longer-term route changes. There were 15,171 such pairs of measurements. As 6 hours is significantly larger than the mean 2 hour sampling interval (§ 7.6), not surprisingly we find many triple measurements spanning less than 6 hours. But of the 10,660 triple measurements, only 75 included a route change of the form R_1, R_2, R_1 or R_1, R_2, R_3, indicating that, for the paths to which we have now narrowed our focus, we are still not missing many routing changes using measurements spaced up to 6 hours apart.

Employing the same analysis, we first identify `sintef1` and `sintef2` as outliers, both as source and as destination sites. The majority of their route changes turn out to be oscillations between two sets of routers. The first alternates between:

```
trd-gw2.uninett.no
```

in Trondheim, and

```
oslos-gw.uninett.no
trds-gw.uninett.no
```

(or the reverse of this, for paths originating at `sintef1` or `sintef2`), which includes an extra hop to Oslo. The second alternates between:

```
nord-gw.nordu.net
no-gw.nordu.net
```

(or the reverse), the first hop in Stockholm and the second in Trondheim, and

```
syd-gw.nordu.net
no-gw2.nordu.net
oslos-gw.uninett.no
trds-gw.uninett.no
```

which again adds a visit to Oslo (middle two hops).

Two other outliers at this level are traffic to or from `sdsc`, which alternates between two different pairs of CERFNET routers, all sited in San Diego, and traffic originating from `mid`, which alternates between two MIDNET routers, both in St. Louis.

Eliminating these paths leaves 11,174 measurements of the 712 remaining paths. The paths between the sites in these remaining measurements are quite stable, with a maximum transition rate for any site of about one change every two days, and a median rate of one change every four days.

7.6.4 Duration of long-lived routes

We will term the remaining measurements as corresponding to "long-lived" routes. For these, we might hazard to estimate the durations of the different routes as follows. We suppose that we are not completely missing any routing transitions (changes of the form R_1, R_2, R_1, where we only observe the first and last). We base this assumption on the overall low rate of routing changes. Then, for a sequence of measurements all observing the same route, we assume that the route's duration was at least the span of the measurements. So if the last observation was made two weeks after the first observation, we assume the route's duration was at least two weeks. Furthermore, if at time t_1 we observe route R_1, and then the next measurement at time t_2 observes route R_2, we make a "best guess" that route R_1 terminated and route R_2 began half way between these measurements, i.e., at time $\frac{t_1+t_2}{2}$.

For routes observed at the beginning (end) of our measurement period, but not spanning the entire measurement period, we assign a starting (ending) time as follows. If the next (previous) measurement also observed the route, then we estimate that the route persisted for at least that much time into the past (future). If the next (previous) measurement did *not* observe the route, then we take the lone observation of the route as its starting (ending) time. This rule will tend to underestimate routing durations, while the rule in the previous paragraph will tend to overestimate (due to occasionally missing a routing change), so these estimation errors will to some degree tend to cancel.

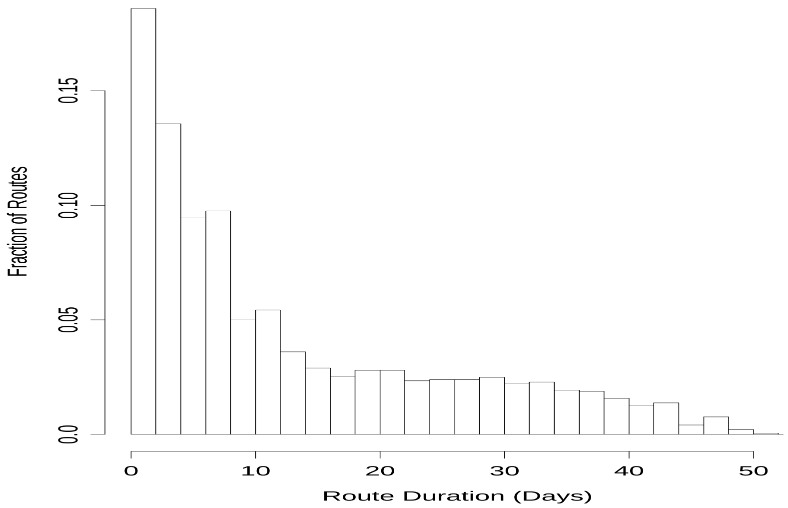

Figure 7.5: Estimated distribution of long-lived route durations

Figure 7.5 shows the distribution of the estimated durations of the “long-lived” routes. Even keeping in mind that our estimates are rough, it is clear that the distribution of long-lived route durations has two distinct regions, with many of the routes persisting for 1-7 days, and another group persisting for several weeks. (Although not evident from the plot, about 4% of the routes had durations under 6 hours, so we might consider the distribution as having three distinct regions.) About half the routes persisted for under a week, but the half of the routes lasting more than a week accounted for 90% of total persistence, meaning the integrated amount of time during which routes remained unchanged. This means that, if we observe a path at an arbitrary point in time, *and we are not observing one of the numerous, more rapidly oscillating paths outlined in the previous sections*, then we have about a 90% chance of observing a route for that path with a duration of at least a week.

7.6.5 Summary of routing persistence

We summarize routing persistence as follows. First, *routing changes occur over a wide range of time scales, ranging from seconds to days.* Table XII lists different time scales over which routes change. The second column gives the percentage of all of our measurement paths (source/destination pairs) that were affected by route changes at the given time scale. (The first two rows show “N/A” in this field because the changes were due to a very small set of routers, so we do not claim any sort of representative fractions.) The third column gives the section where we discuss the changes, and the final column any associated notes. When the note mentions “inside the network” or “intra-network,” we mean that the changes occurred not at the stub networks where the sites themselves connect to the Internet, but instead in what we would deem the Internet infrastructure.

One important point apparent from the table is that routing changes on shorter time scales

Time scale	% Paths Affected	§	Notes
seconds	N/A	§ 6.6	"Flutter" for purposes of load balancing. Treated separately, as a pathology, and not included in the analysis of persistence.
minutes	N/A	§ 7.4	"Tightly-coupled routers." We identified five instances, which we merged into single routers for the remainder of the analysis.
10's of minutes	9%	§ 7.6.1	Frequent route changes inside the network. In some cases involved routing through different cities or AS's.
hours	4%	§ 7.6.2	Usually intra-network changes.
6+ hours	19%	§ 7.6.3	Also intra-network changes.
days	68%	§ 7.6.4	Two regions. 50% of routes persist for under 7 days. The remaining 50% account for 90% of the total route lifetimes.

Table XII: Summary of persistence at different time scales

(fewer than days) happen *inside the network* and not at the stub networks. Thus, *those changes observed in our measurements are likely to be similar to those observed by most Internet sites*.

On the other hand, while the changes occurred inside the network, only those involving `ucl` and `ukc` (§ 7.6.1) involved different sequences of autonomous systems. While this bodes well for the scalability of BGP, we do not claim this finding as having major significance: one could make a much more thorough assessment of the degree of inter-AS route flapping by analyzing the data discussed in [Do95, Me95b].

Finally, two thirds of the Internet paths we studied had quite stable paths, persisting for days or weeks. This finding is in accord with that of Chinoy's, who found that most networks are nearly quiescent (in terms of routing changes) while a few exhibit frequent connectivity transitions [Ch93].

7.7 Detecting route changes

Given our findings that routes change in the Internet on a wide range of time scales, we would like to find mechanisms by which an endpoint can detect that its route to a remote destination has changed. This knowledge has two different applications. The first is that it allows the endpoint to flush any cached information associated with the route, such as round-trip time or available bandwidth. The second application is for network measurement experiments. A number of Internet experiments have been made in which a path through the network is repeatedly sampled [Mi83, CPB93a, Bo93, SAGJ93, Mu94, BCG95]. For such measurements it is important to know whether each time the path is measured, the measurement is observing the same route for that path, or whether the route may have changed (affecting the measurement).

While `traceroute` can be used to elicit the route currently used for a given Internet path, its use is expensive in terms of network resources, and also slow because of the necessity to wait for (possibly dropped) replies to many probe packets.

Granularity	False positives	False negatives	Error rate
host	0%	25%	3%
city	4%	26%	5%
AS path	5%	10%	5%

Table XIII: Summary of TTL method for detecting route changes at different granularities

On the other hand, endpoints can easily determine whether a route's hop count has changed by seeing whether the TTL of packets arriving from the remote destination differ from the previously observed TTL. Because the IP TTL field is in fact a hop count and not a time-to-live (§ 4.2.1), this measurement has no noise, provided the remote destination always sends packets with the same initial TTL. Thus, the endpoint need receive only a single packet from the destination in order to detect that the hop count of the path from the destination to the endpoint has changed. We call this method the "TTL method." To our knowledge, it was first used in [CPB93a].

While the TTL method has an attractive simplicity, it will sometimes result in "false negatives": the underlying route might have changed, perhaps drastically, but if the new route happens to have the same number of hops as the cached one, the TTL method will report it as unchanged. In this section, we explore the degree to which these false negatives affect the practicality of the method.

After removing pathologies and fluttering paths, the data contained 30,145 consecutive `traceroutes` for us to test. Of these, 3,380 were route changes when viewed at host granularity, 1,928 at city granularity, and 1,266 at AS path granularity.

We consider a route to have changed if and only if it did not visit exactly the same hosts (cities; AS's) in the same order. Before determining the host visited at each hop, however, we merged the "tightly-coupled" routers discussed in § 7.4 into a single router.

We deem the method as generating a "false positive" if it erroneously declares that the route changed, and a "false negative" if it fails to detect that the route did indeed change. To make these notions more precise, suppose that, out of N observations, K were genuine route changes at a given granularity, but of these K the method only detects k, and it also erroneously "detects" b bogus route changes. Then the false positive rate is $b/(N - K)$, and the false negative rate is $(K - k)/K$. We can also define an overall "error rate," which is the proportion of time that the method misinforms us one way or the other: $(b + K - k)/N$.

Barring the remote host altering its initial TTL setting, or routers actually decrementing the TTL field for each second they delay a packet, the TTL method will never generate a false positive at host granularity[9]. It can do so at other granularities, however, when the underlying route changes in the number of hops, but the same cities or AS's are still visited. At all three granularities, the TTL method can generate false negatives.

Table XIII summarizes the effectiveness of the TTL method for detecting different granularities of route changes. Its overall error rate is consistently low. This is mostly a reflection of the fact that all-in-all the underlying route does not change very often. Because in the absence of any change whatsoever the TTL method always reports "no change," it is correct whenever the

[9] Provided we exclude from testing pathological routes that visit a given hop more than once, which we did.

underlying route has not changed.

At no granularity, however, is the false negative rate especially good, and at city and AS path granularities the false positive rate is non-negligible, too. Thus, we conclude that the TTL method serves as a handy heuristic, but is definitely not fool-proof. Still, it seems worthwhile to use the TTL method to detect route changes when conducting the network measurement studies mentioned at the beginning of this section, and the generally low false positive rate suggests that flushing cached route information upon observing a TTL change will usually be the correct action. One must not, however, be too complacent in accepting the absence of a TTL change as indicative of an unchanged route.

A final note concerning the TTL method: The TTL value most easily available to an endpoint for caching is that in packets the endpoint receives from the remote host. The TTL's in these packets reflect the hop count for the route *from the remote host to the local host*. If the routes between the two hosts are asymmetrical, however, then this hop count does *not* necessarily reflect the hop count along the route in the other direction (local host to remote host), which is generally the direction of interest. As shown in Chapter 8, routing asymmetry is not uncommon. Because of this, use of the TTL method may require some additional mechanism by which the local host can learn the TTL the remote host observed in packets it received from the local host. We do not attempt here to offer a well thought out mechanism for doing so. We only comment that any such mechanism must take care that, when a route changes, the network is not immediately flooded with messages to that effect. Perhaps a solution can be found using multicasting techniques to minimize the number of messages sent after route changes.

Chapter 8

Routing Symmetry

We now analyze the routes from our measurement study to assess the degree to which routes are *symmetric*. We first motivate the investigation by discussing the impact of routing asymmetry on different network protocols and measurements. We then give an overview of various mechanisms that can introduce asymmetry into Internet routing, including "hot potato" routing (§ 8.2), which could result in a greater proportion of asymmetric routes in the future. We next introduce a definition of routing symmetry, and show that practical considerations require a revision in which we view routes as asymmetric only if they visit different cities or autonomous systems. We then assess our data for these asymmetries and find that, overall, 50% of the time an Internet path includes a major asymmetry in terms of the cities visited in the different directions, and 30% of the time it includes a major asymmetry in terms of autonomous systems visited. We finish with a discussion of the magnitude of the asymmetries, most of which differ at just one "hop," but some at many hops.

8.1 Importance of routing symmetry

Routing symmetry affects a number of aspects of network behavior. When attempting to assess the one-way propagation time between two Internet hosts, the common practice is to assume it is well approximated as half of the round-trip time (RTT) between the hosts [CPB93a]. The Network Time Protocol (NTP) needs to make such an assumption when synchronizing clocks between widely separated hosts [Mi92a]. If routes are asymmetric, however, the assumption might easily lead to error. The NTP design utilizes multiple time server peers and robust algorithms to choose among them for the best time offset to use to account for propagation effects. Thus, routing asymmetry has an impact on NTP only if the paths between two NTP communities are predominantly asymmetric, with similar differences in one-way times. In that case, the two communities will keep consistent time among themselves, but not between each other.[1]

Claffy and colleagues studied variations in one-way latencies between the United States, Europe, and Japan [CPB93a]. They discuss the difficulties of measuring *absolute* differences in propagation times in the absence of separately-synchronized clocks, but for their study they focussed on *variations*, which does not require synchronization of the clocks. They found that the

[1] Recently, however, highly accurate atomic clocks have become much more affordable than in the past (as have Global Positioning System receivers, which also provide reliable time). These provide an independent solution to the problem of keeping widely separated NTP servers synchronized.

two opposing directions of a path do indeed exhibit considerably different latencies, in part due to different congestion levels, and in part due to routing changes, which they detected using the TTL method (§ 7.7).

Along with affecting Internet protocols such as NTP, routing asymmetry can render network measurement considerably more difficult. Often it is easiest to perform measurements at a single endpoint of a network path, but in the face of routing asymmetries, such measurements might be unable to distinguish between considerably different behavior along the forward and reverse directions of the path. We explore this problem at length in Part II (see § 9.1.3 for a general discussion).

Closely related to this measurement problem, routing asymmetry also potentially complicates mechanisms by which connection endpoints can infer network conditions from the pattern of packet arrivals they observe. For example, we develop a technique in Chapter 14 for estimating the "bottleneck bandwidth" of the network path used by a connection. The technique works by examining the timing with which packets arrive at their receiver. If routing is symmetric, then (for most link technologies) the bottleneck bandwidth measured by this technique will be the same as that encountered by packets sent in the other direction. Symmetry could, for example, allow the server for a request/reply application such as the World Wide Web [BCLF+], or, more generally, T/TCP [Br94], to determine the link bandwidth available for sending its reply, based on the bandwidth inferred from the request. If routing is asymmetric, however, then the server runs the risk of inferring an incorrect value for the bandwidth.[2] However, we show in Chapters 14 and 16 that bottleneck bandwidths and delays are often asymmetric along the two directions of a path, and attribute the difference at least in part to routing asymmetries.

Finally, recent work has investigated the characteristics of network traffic *flows* as viewed by a router [CBP95]. That study describes a taxonomy of methodologies that can be used by routers to define and manage flow state. One finding of the study is that a large number of flows are bidirectional, due in part to request/reply transactions such as those used by the Domain Name System (DNS; [MD88]) and the World Wide Web. When a router R sees a flow likely to be bidirectional, for example a DNS request from A to B, one might consider establishing *anticipatory flow state* in the router for the reply coming from B to A, to avoid the overhead of two separate trips through the "slow path" associated with flows for which there is no cached state. With prevalent routing asymmetry, however, while B may very likely send such a message shortly, the reply could well *not* be routed via R, in which case the anticipatory flow state is wasted effort and resources.

Similarly, *accounting* used to charge for carrying network traffic is complicated by the possibility of locally observing only one direction of a traffic flow. For example, a recently developed architecture for Internet traffic flow measurement has a basic assumption that routers observe bidirectional flows [BMR97].

8.2 Sources of routing asymmetries

In this section we discuss several mechanisms that can lead to routing asymmetries. To illustrate, we assume the viewpoint of a router R_0 faced with the decision of how to forward packets originated by host A and destined for host B. In addition to the upstream router from which R_0

[2] Even if routing is symmetric, the server cannot rely on the congestion levels being symmetric. Thus, as with routing stability, routing symmetry is *necessary* but not *sufficient* for predicting network behavior.

receives packets sent by A, R_0 is connected to two potential downstream routers, R_1 and R_2, and the decision it must make is to which of these it forwards packets bound for B. Let us also assume that packets from B headed to A arrive at R_0 via R_1 (but in general R_0 does not itself know this fact), and that these packets first pass through a router R_3, which makes the decision whether to use the route that ultimately delivers the packets to R_0 via R_1, or a different route that results in the packets arriving at R_0 via R_2.

In general, routing algorithms incorporate "link costs" or *metrics* to quantify the desirability of using a particular link for a given route [Pe92, St95]. To assure reliable operation, a router also generally knows of multiple paths available to a remote destination B, so we assume that R_0 has two metrics, μ_1 and μ_2, associated with forwarding packets to B via R_1 or R_2. If $\mu_1 = \mu_2$, then R_0 must somehow arbitrate between them. If it does so deterministically, by picking R_2, then an asymmetry is created.[3]

Another way of introducing asymmetry is via configuration asymmetries or errors. For example, if due to misconfiguration R_0 believes that using the link to R_1 is very expensive, but R_1 does not share this view, then R_0 will artificially inflate the cost of using R_1 to get to B, and instead pick R_2.

Network topology changes can also introduce routing asymmetries, albeit transient ones, due to the non-negligible amount of time required for changes to propagate through the network. For example, suppose R_2 learns of a better route to R_3 than it had before. If knowledge of this new route propagates to R_0 before R_3, then R_0 will switch from R_1 to R_2, and an asymmetry will exist until R_3 learns of the route.

Another transient mechanism for creating routing asymmetries can arise due to *adaptive routing* (§ 7.2), in which a router attempts to shift traffic from a highly loaded link to a less loaded link. For example, R_0 might decide that it is sending too much traffic via the link to R_1 (the bulk of this traffic might not be destined for B), so it increases the metrics associated with R_1 to the point where routing via R_2 becomes the preferred route to B. More generally, if routing metrics include a notion of current congestion levels, then asymmetric congestion in the network can lead to asymmetric routing, as the network alters its routing to avoid the congested region.

A final mechanism introducing asymmetry, and one of possibly growing importance, concerns "hot potato" and "cold potato" routing. In the past, Internet backbones were primarily operated by a single entity. In recent years this has changed, with the growth of competing Internet Service Providers (ISP's) due to the privatization of the Internet infrastructure.

Suppose host A in California uses ISP I_A, and host B in New York uses I_B. Assume that both I_A and I_B provide Internet connectivity across the entire United States. When A sends a packet to B, the routers belonging to I_A must at some point transfer the packet to routers belonging to I_B. Since cross-country links are a scarce resource, both I_A and I_B would prefer that the other convey the packet across the country. If the inter-ISP routing scheme allows the upstream ISP (I_A, in our example) to determine when to transfer the packet to I_B, then, due to the preference of avoiding the cross-country haul, I_A will elect to route the packet via I_B as soon as possible. This form of routing is known as "hot potato." In our example, it leads to I_A transferring the packet to I_B in California. But when B sends traffic to A, I_B gets to make the decision as to when to forward the traffic to I_A, and with hot potato it will choose to do so in New York. Since the paths between California and New York used by I_A and I_B will in general be quite different, hot potato routing thus leads to a

[3] If it alternates between R_1 and R_2, it creates *fluttering*, as discussed in § 6.6.

major routing asymmetry between A and B.

Conversely, if the *downstream* ISP can control where the upstream ISP transfers packets to it, then the result is "cold potato" routing, in which I_B instructs I_A that, to reach B, I_A should forward packets to I_B's New York network access point (NAP). Similarly, I_A advertises to I_B that, to reach A, I_B should forward packets to I_A's California NAP. The result is that packets from A to B travel across the country via I_A's links, while those from B to A travel via I_B's links. The paths are the opposite of those resulting from hot potato routing, but the degree of asymmetry remains the same, and potentially large.

For further discussion of asymmetry issues, see [Che95].

8.3 Definition of routing symmetry

In this section we develop a definition for whether two routes are symmetric. We first try the following:

Definition 1 *For two hosts A and B, let $r_1, \ldots, r_n$ denote the routers visited in sequence by packets sent from A to B, and $r'_1, \ldots, r'_m$ denote those visited in sequence by packets from B to A. Then the two routes are* symmetric *if and only if $n = m$ and:*

$$\forall i, 1 \le i \le n : r_i = r'_{n+1-i}.$$

Definition 1 presents two problems. First, for routes considered asymmetric, the definition fails to provide a notion of the *degree* of asymmetry. For example, if a site has two Internet access points, then we could find that traffic from A to B leaves the site at the first access point for a downstream router R, while traffic from B to A comes to the site also from R, but arriving at the second access point. Such an asymmetry is minor. For example, it will have minimal impact on the accuracy of the NTP protocol (§ 8.1). On the other hand, if the route from A to B visits a different *city* than does the route from B to A, then the two paths might have considerably different properties, and the asymmetry is major.

To illustrate these differences, consider the route we observed in $\mathcal{R}_1$ from `ucol` to `ucl` (where we have annotated the cities visited in parentheses), shown in Figure 8.1. One of the complementary routes we observed from `ucl` to `ucol` is shown in Figure 8.2. This route visits the same cities as the reverse route, though not the same routers; the asymmetry is minor. On the other hand, we also observed a route from `ucl` to `ucol` as shown in Figure 8.3. In this case, the detour via California is skipped, shaving perhaps 2,000 kilometers of travel from the route: a major asymmetry.

A second problem with Definition 1 is determining whether two routers r_i and r'_j are indeed the same router. The difficulty arises because `traceroute` provides an IP address for each hop, but these do not uniquely identify routers. In general, routers have multiple IP addresses, one for each network interface attached to the router. Furthermore, these IP addresses can translate to different hostnames. Thus, for example, it is difficult to determine whether the IP address with hostname `sl-ana-3-s2/4-t1.sprintlink.net` in Figure 8.1 corresponds to the same router as that with hostname `sl-ana-3-f0/0.sprintlink.net` in Figure 8.2.

We address both these difficulties using a revised definition:

```
cs-gw-discovery.cs.colorado.edu  (Boulder, CO)
cu-gw.colorado.edu
sl-ana-3-s2/4-t1.sprintlink.net  (Anaheim, CA)
sl-ana-1-f0/0.sprintlink.net
sl-fw-6-h2/0-t3.sprintlink.net   (Fort Worth, TX)
sl-fw-5-f1/0.sprintlink.net
sl-dc-8-h3/0-t3.sprintlink.net   (Washington, D.C.)
icm-dc-1-f0/0.icp.net
icm-london-1-s1-1984k.icp.net    (London, UK)
smds-gw.ulcc.ja.net
smds-gw.ucl.ja.net
cisco-pb.ucl.ac.uk
cisco.cs.ucl.ac.uk
neptune.cs.ucl.ac.uk
```

Figure 8.1: Route observed from `ucol` to `ucl`

```
cisco.cs.ucl.ac.uk               (London, UK)
cisco-pb.ucl.ac.uk
cisco-b.ucl.ac.uk
gw.lon.ja.net
eu-gw.ja.net
icm-lon-1.icp.net
icm-dc-1-s3/2-1984k.icp.net      (Washington, D.C.)
sl-dc-6-f0/0.sprintlink.net
sl-dc-8-f0/0.sprintlink.net
sl-fw-5-h4/0-t3.sprintlink.net   (Fort Worth, TX)
sl-fw-6-f0/0.sprintlink.net
sl-ana-1-h2/0-t3.sprintlink.net  (Anaheim, CA)
sl-ana-3-f0/0.sprintlink.net
sl-ucb-2-s0-t1.sprintlink.net    (Boulder, CO)
cs-gw.colorado.edu
clark.cs.colorado.edu
```

Figure 8.2: Route observed from `ucl` to `ucol`

```
cisco.cs.ucl.ac.uk                (London, UK)
cisco-pb.ucl.ac.uk
cisco-c.ucl.ac.uk
smds-gw.ulcc.ja.net
icm-lon-1.icp.net
icm-dc-1-s3/2-1984k.icp.net       (Washington, D.C.)
sl-dc-8-f0/0.sprintlink.net
sl-fw-5-h4/0-t3.sprintlink.net    (Fort Worth, TX)
sl-fw-4-f0/0.sprintlink.net
sl-ucb-1-s0-t1.sprintlink.net     (Boulder, CO)
cns-gw-suns.colorado.edu
cs-gw.colorado.edu
lewis.cs.colorado.edu
```

Figure 8.3: Second route observed from `ucl` to `ucol`

Definition 2 *For two hosts A and B, let $c_1, \ldots, c_n$ denote the* cities *visited in sequence by packets sent from A to B, and $c'_1, \ldots, c'_m$ denote those visited in sequence by packets from B to A. Then the two routes are* symmetric *if and only if $n = m$ and:*

$$\forall i, 1 \le i \le n : c_i = c'_{n+1-i}.$$

This definition deals with the first difficulty of the original definition by discarding all minor routing asymmetries—we consider a routing asymmetry interesting only if it is major. It resolves the second difficulty because it is considerably easier to tell whether two IP addresses are located in the same city than whether they refer to the same router, since with a bit of effort it is generally possible to determine the city corresponding to an Internet hostname (cf. § 5.3). For example, we know from the Sprintlink naming convention that both `sl-ana-3-s2/4-t1.sprintlink.net` and `sl-ana-3-f0/0.sprintlink.net` are located in Anaheim, California.

We can make an analogous definition for routes differing in the autonomous systems they visit, rather than the cities.

8.4 Analysis of routing symmetry

In $\mathcal{R}_1$, we did not make simultaneous measurements of the paths $A \Rightarrow B$ and $B \Rightarrow A$, which introduces ambiguity into an analysis of routing symmetry: if a measurement of $A \Rightarrow B$ is asymmetric to a later measurement of $B \Rightarrow A$, is that because the route is the same but asymmetric, or because the route changed?

In $\mathcal{R}_2$, however, the bulk of the measurements were *paired*: we first measured $A \Rightarrow B$ and then immediately afterward measured $B \Rightarrow A$. Barring rapid route oscillations (which we can avoid by eliminating pathological `traceroutes` from our analysis), these measurements allow us to unambiguously determine whether the route between A and B is symmetric.

The $\mathcal{R}_2$ measurements contain 11,339 successful pairs of measurements, in which we were able to conduct `traceroutes` in both directions between sites A and B, neither of the measurements encountering pathologies.

We find that *49% of the measurements observed an asymmetric path that visited at least one different city*.

There is a large range, however, in the prevalence of asymmetric routes among paths to and from the different sites. For example, 86% of the paths involving `umann` were asymmetric, because nearly all outbound traffic from `umann` travel via Heidelberg, but none of the inbound traffic does. At the other end of the spectrum, only 25% of the paths involving `umont` were asymmetric (but this is still a significant amount).

If we consider autonomous systems rather than cities, then we still find asymmetry quite common: about 30% of the paired measurements observed different autonomous systems traversed in the path's two directions. The most common asymmetry was the addition of a single AS in one of the directions. This can reflect a major change, however. For example, the most common of these additions was the presence of SprintLink routers in one direction along the path but not in the other.

Again, we find a wide range in the prevalence of asymmetry among the different sites. Fully 84% of the paths involving the `ucl` site were asymmetric, mostly due to some paths including JANET routers in London and others not (unsurprising, given the rapid oscillation between JANET and non-JANET routers discussed in § 7.6.1). On the other end of the spectrum, only 7.5% of `adv`'s paths were asymmetric at AS granularity.

8.5 Increasing prevalence of asymmetry

We previously analyzed $\mathcal{R}_1$ for routing asymmetry, attempting to adjust for the non-simultaneity of its measurements by only using measurements spaced less than a day apart. The mismatch is likely to overestimate routing asymmetry, since if the route changes between measurements that may be incorrectly regarded as an asymmetry, per our discussion at the beginning of § 8.4. The mismatch can also introduce false symmetries, if the route happens to change to the symmetric counterpart, but this circumstance is probably more rare than introducing false asymmetries.

In the $\mathcal{R}_1$ measurements, we found 30% of the paths contained city-level asymmetries. The large discrepancy between this figure and the 50% figure for the $\mathcal{R}_2$ measurements suggests that over the course of a year routing became significantly more asymmetric. We surmise that the increase of asymmetry is likely due to the "hot potato" effect discussed in § 8.2. If so, then the rise in asymmetry has its roots in commercial factors, and frequent routing asymmetry may continue to be common in the Internet in the future. From a measurement perspective, this would be unfortunate, for the reasons given § 8.1, and further developed in § 9.1.3.

8.6 Size of asymmetries

We finish our study of routing symmetry with a look at the size of the different asymmetries. We can assign a "magnitude" to each asymmetry in terms of the number of cities different in the two directions. We consider each "city hop" at which the two directions of a path differ as contributing a magnitude of 1; if one direction has more "city hops" than the other, each additional city contributes $\frac{1}{2}$. For example, for the paths between `rain` and `bnl`, we observed simultaneous measurements of the following routes:

```
r0.pdx.rain.rg.net                    (Portland)
```

```
sl-stk-13-s2/2-t1.sprintlink.net  (Stockton)
sl-stk-5-f0/0.sprintlink.net
sl-dc-6-h1/0-t3.sprintlink.net    (Washington, D.C)
sl-pen-1-h2/0-t3.sprintlink.net   (Pennsauken)
sl-pen-2-f0/0.sprintlink.net
ny-nyc-2-h1/0-t3.nysernet.net     (New York)
ny-nyc-6-f0/0.nysernet.net
ny-dp-1-h0/0-t3.nysernet.net      (Deer Park)
ny-bnl-2-s0-t1.nysernet.net       (BNL)
cerberus.bnl.gov
frog.rhic.bnl.gov
```

and

```
cerberus.90.bnl.gov               (BNL)
nioh.bnl.gov
192.12.15.224
llnl-satm.es.net                  (Livermore)
ames-llnl.es.net                  (Mountain View)
fix-west-cpe.sanfrancisco.mci.net (San Francisco)
borderx2-hssi2-0.sanfrancisco.mci.net
core2-fddi-1.sanfrancisco.mci.net
core1-hssi-2.sacramento.mci.net   (Sacramento)
core-hssi-3.seattle.mci.net       (Seattle)
border1-fddi-0.seattle.mci.net
rgnet-b1-serial2-3.seattle.mci.net
chia.rain.net                     (Portland)
```

The paths differ at five "city hops," Stockton/Seattle, Washington/Sacramento, Pennsauken/San Francisco, New York/Mountain View, and Deer Park/Livermore, so we assign a magnitude of 5 to this asymmetry.

Figure 8.4 shows the distribution of asymmetry magnitudes. We see that the asymmetries typically include only one different city hop, or, even more commonly, just one additional city. About one third of the asymmetries have magnitude 2 or greater. We should bear in mind, though, that this corresponds to almost 20% of all the paired measurements in our study, and can correspond to a very large asymmetry. For example, a magnitude 2 asymmetry between `ucl` and `umann` differs at the central city hops of Amsterdam and Heidelberg in one direction, and Princeton and College Park in the other!

In general, the presence of such asymmetries highlights the difficulties of providing a consistent topological view in an environment as large and diverse as the Internet.

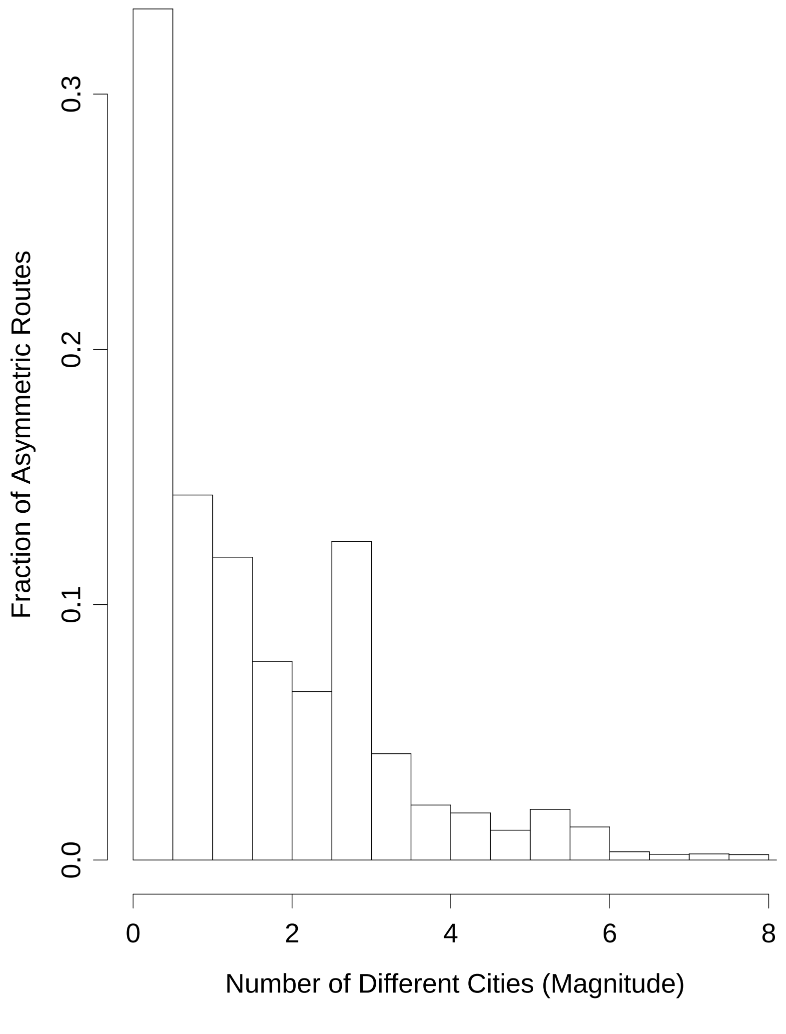

Figure 8.4: Distribution of asymmetry sizes

Part II

End-to-End Internet Packet Dynamics

Chapter 9

Overview of the Packet Dynamics Study

In this part of our study we present our efforts to find convincing answers to questions about end-to-end Internet packet dynamics such as "how often are packets dropped?" As in Part I, we devise a large-scale measurement experiment based on the "Network Probe Daemon" (NPD) measurement framework. Our goal with this part of our study is to develop persuasive characterizations of the dynamics of Internet packet loss and delay. To do so, however, requires a great deal of groundwork in order to assure that the resulting findings are sound.

First, we need to calibrate our basic packet measurements, detecting those that are untrustworthy or inaccurate so that we can discard them to avoid drawing false conclusions. We describe how we do so in Chapter 10. Because we use TCP transfers as our basic "probes" for measuring network paths, our probes have a complicated structure due to the particulars of TCP. In Chapter 11 we discuss our development of an analysis tool, `tcpanaly`, that accounts for the details of the various TCPs in our study, and thus can separate their effects from true networking effects. The development of `tcpanaly` also gives us an opportunity to look at the differences in behavior between the TCP implementations in our study. These turn out to be quite significant, including some sufficiently broken TCPs that, if ubiquitously deployed, would devastate Internet performance due to *congestion collapse*.

Because one of our goals is to characterize one-way packet delays, we must also deal with the problem of calibrating the clocks used in our study. This proves much more difficult than we had originally anticipated. Chapter 12 details our efforts.

In Chapter 13 we turn to examining network "pathologies," meaning unexpected network behavior. These include out-of-order delivery, in which packets arrive at the receiver in a different order than that in which they were sent; packet replication, in which the network delivers multiple copies of a single packet; and packet corruption, in which the data in the packet delivered by the network differs from that in the packet as originally sent.

In order to then soundly evaluate packet delay and loss, we need to first determine each connection's *bottleneck rate*, i.e., the upper bound imposed by the network path on the connection's throughput. This rate plays a crucial role because it determines when closely-spaced packets must necessarily queue behind each other in the network. Network conditions observed by such packets are *correlated* and must be treated separately from uncorrelated observations. In Chapter 14 we discuss shortcomings of the main existing technique for estimating bottleneck bandwidth, "packet pair," and develop a robust algorithm, PBM ("packet bunch modes"), to address these problems.

In addition, we characterize the range of bottleneck rates we observed among the various Internet paths, and assess the stability of a path's bottleneck rate over time.

We then proceed in Chapter 15 to an analysis of patterns of Internet packet loss. We look at many different facets of loss, including the differences between loss rates of data packets and acknowledgements; correlations between loss rates along the two directions of a network path; trends in loss rates; differences in loss rates due to geography; the duration of loss "outages"; the location, with respect to the path's bottleneck element, where packet loss occurs; how well a connection's observed packet loss predicts those of future connections; and how well TCP deals with packet loss, in terms of retransmitting only when necessary.

We finish in Chapter 16 with an analysis of patterns of Internet packet delay. We look at variations and extremes of round-trip times (RTTs) and one-way transit times (OTTs); symmetry in OTT variation along the two directions of a network path; correlations between delay variations and loss; how well a connection's delay variations predict those of future connections; the phenomenon of packet timing "compression"; the time scales on which queueing occurs; and the degree of *available bandwidth* present along Internet paths.

Chapter 17 summarizes the findings of both Part I and Part II, and sketches the main themes of the work.

In the remainder of this chapter, we discuss our experimental methodology (§ 9.1); those aspects of the TCP protocol relevant to our study (§ 9.2); and the raw data produced by the experiment (§ 9.3).

9.1 Methodology

In this section we discuss the methodology underlying the packet dynamics experiment. We address two separate issues: how to make the measurements, and how to analyze them.

9.1.1 Measurement considerations

For our packet dynamics study, our measurement "probes" consisted of TCP transfers of 100 Kbyte files over different Internet paths. We discuss in § 9.1.2 the reasoning behind using TCP for the study. The transfers were *unidirectional*: data only flowed along one direction of the path. Such connections are referred to as *bulk transfers* [DJCME92, Pa94a]. There are other classes of traffic in the Internet (such as request/response, interactive, multicast, and real-time). All of these ultimately boil down to dividing data into packets for delivery by the Internet's packet forwarding infrastructure. Our goal is to characterize what happens to packets once they are in the hands of this infrastructure. For this purpose, bulk transfers serve well, as they provide a fairly steady stream of data packets traveling in one direction, and a corresponding stream of ack packets traveling in the other. We can then analyze the fate and timing of the packets to determine how the two directions of the Internet path performed.

Each transfer was traced using the `tcpdump` utility [JLM89] at both the sender and the receiver, resulting in two trace files. We term the combination of the two trace files a "trace pair." Our findings are all based on analyzing trace files and trace pairs.

For security reasons, the NPD transfers used fixed TCP sending and receiving "ports," so `tcpdump` could immediately filter out traffic not related to the transfer. That we did so has two drawbacks. First, it means that the traces lack some network traffic relevant to the transfer, namely any associated Internet Control Message Protocol (ICMP; [Po81b]) messages. We discuss in § 11.3.3 how we inferred the presence of a particular type of ICMP message, termed "source quench." In addition, using fixed ports resulted in our measurements incurring a *minimum separation* between consecutive measurements of the same pair of hosts, because TCP has rules governing how quickly a pair of ports can be reused for a new connection.[1]

As with the routing dynamics experiment, we used exponentially-spaced sampling intervals in order that our measurements might observe an unbiased sample of conditions along the different Internet paths (§ 4.3). We conducted two experimental runs, $\mathcal{N}_1$ and $\mathcal{N}_2$, detailed in § 9.3. For $\mathcal{N}_1$, source hosts were randomly paired with destination hosts, and we conducted a single measurement for each pairing. The drawback of this approach is that, if we want to study how an Internet path's characteristics change (or "evolve") over time, then random pairing results in widely-spaced measurements of individual pairs. For example, in $\mathcal{N}_1$ the mean sampling interval for a given pair was about two days. Consequently, we cannot analyze much finer time scales of evolution.

We addressed this difficulty in the second run, $\mathcal{N}_2$, by randomly pairing source and destinations into *groups* of measurements. Each measurement group consisted of two subgroups. Within a subgroup, we conducted six measurements, separated by 180 sec plus exponentially-distributed intervals with means 30 sec, 60 sec, 120 sec, 240 sec, and 480 sec.[2] These spacings allow us to analyze evolutions over short time intervals.

The two subgroups were then separated by an exponentially-distributed interval with mean 2 hours, allowing us to characterize evolution over medium time intervals. In addition, source/destination pairs would conduct additional groups of measurements separated from the previous group by another exponential interval with a mean of 12 hours. Finally, the pairs would be revisited on the order of a number of days later. These last two groups of measurements allow us to characterize relatively long time intervals, too.

9.1.2 Using TCP

Most previous end-to-end studies have used ICMP "ping" messages [Mi83, CPB93a] or User Datagram Protocol (UDP; [Po80]) "echo" messages for their network probes [Bo93].[3] Both have the considerable advantage of logistical ease: most Internet hosts readily reply to "ping" messages,[4] and activating the UDP echo service is often a one line configuration tweak.

However, these types of probes also incur disadvantages. The most significant of these is that of the *rate* at which the probes are sent. To probe fine time scales requires sending closely-spaced probe packets. Yet, if this is done blindly, say by deciding to send packets 1 msec apart, then depending on the mismatch between the sending rate and the capacity of the network path, the measurement traffic can grossly overload the path. Consequently, both "production" traffic sharing the path suffers, and the measurements are skewed by the abnormal loading. Unfortunately, there is a very wide range in network path capacity (we develop this claim in detail in Chapter 14 and

[1] Nominally, this minimum time is four minutes, twice the "maximum segment lifetime" of two minutes. In practice, it varies between TCP implementations.

[2] The 180 sec constant interval was required to avoid problems with reusing the fixed source and destination ports, discussed above.

[3] An exception is Mogul's study of TCP packet dynamics [Mo92].

[4] This is changing, with the advent of firewalls.

Chapter 16), so there is no *a priori* correct choice to use for the probe spacing.

Furthermore, capacity *changes* over the course of a series of probes, so we cannot determine a single correct choice for a path even after studying the path a bit. Therefore, ICMP- and UDP-based measurement must make a trade-off between possibly overloading the network path, and probing conservatively but with no possibility of analyzing finer time scales. In general, researchers have prudently chosen the latter.

One could devise a probing strategy based on *adapting* the probe transmission rate to the current network conditions. However, to do this properly, one essentially must implement TCP's congestion control. At this point, it becomes easier to just start with TCP in the first place!

Another drawback with echo-based techniques is that the echo services return a full copy of whatever packet they receive. Consequently, the measurement loads the network path both in the forward and the return direction. If the measurement is conducted using "sender-only" techniques (§ 9.1.3), then the reverse-path loading makes it impossible to determine which direction of the path is responsible for what proportion of the phenomena observed. If the echoes are instead *small*, such as are TCP acknowledgements for data packets, then the connection does not load the reverse path,[5] which lessens the conflation of the two directions.

Both of these considerations, particularly the first, argue favorably for using TCP transfers as network probes, since then, by construction, our probes do not load the network any more than does a routine file transfer. Using TCP has one other major advantage: TCP is very widely used. Consequently, the end-to-end performance observed by TCP transfers is a much closer match to the service Internet users actually obtain from the network than are echo-based techniques. We will also see in Chapter 11 that one result of our using TCP is to uncover a large variation in how different TCP implementations perform, some with major performance and congestion implications.

Using TCP, however, also brings with it some serious drawbacks. The first of these is that the TCP protocol behavior is quite complex. When casually inspecting TCP measurements, it can be difficult to determine which facets of the overall connection behavior were due to the state of the network path, and which were due to the behavior of the TCP implementations at the endpoints. If our goal is to characterize the network path, we *must* be able to separate these two, which entails understanding the nitty-gritty details of how different TCP implementations realize the protocol. To do so, for our study we developed a program, `tcpanaly`, which has knowledge of various TCP implementations and can analyze `tcpdump` traces in order to separate TCP endpoint effects from those due to the network path. Writing `tcpanaly` was a significant undertaking, much harder than we had initially anticipated (because we had not realized the wide range of real-world TCP behaviors). We discuss it in detail in Chapter 11.

The other major drawback with using TCP is that often it sends small groups of data packets at rates exceeding that of the network path's capacity (§ 9.2.5). These packets necessarily queue behind one another at the path's bottleneck. Therefore, for measuring the network's state such a group constitutes a *correlated* set of probes. We address this difficulty at length in Chapter 14.

[5] There is one way in which small packets can contribute to load along the reverse path similarly to large packets. If a congested router manages its buffers for queued packets on a *per-packet* basis, rather than allocating the number of bytes required to queue a packet out of a shared pool, then small packets consume the same amount of resource when queued at the router as do large packets. In this regard, small packets can push the congested router to the point of buffer overflow as fast as large packets do. Once, however, the small packets receive service, by transmission across their outbound link, then their contribution to the router's load immediately diminishes, since they require significantly less transmission time.

Furthermore, the TCP sender *adapts* the rate at which it transmits data packets based on previously observed network conditions (in particular, packet loss, per § 9.2.6). Thus, even when uncorrelated, the data packets do *not* reflect an unbiased measurement process, but rather one that changes its sampling rate in order to try to *minimize* observed packet loss. We discuss this property in Chapter 15.

On the other hand, for a TCP bulk transfer, both of these problems *only occur along the forward path*. The traffic along the reverse path is comprised entirely of small acknowledgement packets. These in general do *not* necessarily queue behind one another at the bottleneck, and, furthermore, their transmission rate is adapted not to conditions along the reverse path, which they observe, but to conditions along the forward path. We show in Chapter 15 that these conditions are generally uncorrelated. Thus, the "ack stream" along the reverse path reflects a much cleaner measurement process.

In summary, by using TCP transfers, we get two basic types of measurements: those that correspond to conditions that TCP data packets encounter (the forward path), and those that tell us about general Internet path properties (the reverse path ack stream). The combination makes for rich analysis.

9.1.3 Tracing at both sender and receiver

End-to-end measurement is often done using what we term "sender-based" or "sender-only" measurement, meaning that probes and their replies are recorded only at the location of the probe sender. Sender-based measurement has the enormous logistical advantage of not requiring access to the remote site in order to instrument the probe arrivals. Such access can be difficult to gain, for administrative and security reasons.

On the other hand, sender-based measurement carries with it the limitation that from it one can say little about how traffic behaves along the path's two different directions. For example, suppose a measurement consists of sending a flight of 20 ICMP "ping" packets from A to B, and timing at A the arrival of their echoes. If only 6 echoes return, we have no way of knowing whether B never sent the 14 others, because their corresponding pings never arrived at B; or if B did send them, but they were lost on their journey from B back to A; or if some combination of loss from A to B and loss from B to A occurred. Consequently, it is difficult to say anything concrete about the nature of the loss event.

This consideration becomes more subtle, but equally important, when applied to analyzing packet delay. A sender-based scheme can only observe round-trip time (RTT) delays. These are perforce the sum of the one-way transit time (OTT) delays in the two directions, plus the (unobserved) delay of the receiver generating its reply. If the goal of the timing measurement is to estimate capacity along the forward path, such as for TCP Vegas [BOP94], then any delay variations incurred on the return path are pure noise, and at best dilute the precision with which the sender can estimate the path capacity.

Because we traced our transfers at both the sender and the receiver, we can fully separate effects due to the forward path, the reverse path, and the processing delays at both the sender and the receiver. Throughout our study we examine issues of path symmetry with an eye to gauging the effectiveness of sender-only measurement. We find, overall, that such measurement is significantly less accurate than receiver-based measurement. Consequently, it behooves us to consider mechanisms for coordinating measurement between sender and receiver.

9.1.4 Analysis strategies

In this section we discuss the principles underlying our analysis of the measurement data. They are all in response to three dominant considerations. The first is that we gathered a very large volume of data: more than 20,000 transfers recorded at both sender and receiver. Each transfer consisted of 100–400 packets, resulting in well over a gigabyte of data. The second consideration is that we lack separate means of *calibrating* the measurements. All we have to work with are the packet traces. It is easy to *assume* that such traces accurately reflect the true number and timing of the packets comprising the traffic we wish to measure, but no large-scale study has been made to test the overall integrity of packet traces, so the validity of this assumption is unproven. The third consideration is that network behavior almost inevitably includes "noise" in a variety of forms and on a variety of scales. We observe "extreme" behavior much more often than we might expect using a traditional statistical framework (such as one based on assumptions of normality and tame correlations).

That we must deal with a large volume of data lies at the heart of our study: the study is interesting precisely because the volume of data is large. By (very careful) analysis of it, we have a hope of capturing a useful description of the immensely diverse behavior of the huge, heterogeneous network that is the Internet. We further argue that future Internet traffic studies must likewise measure on a large scale, otherwise we have little hope of divining from them general results. Thus, a central contribution of our work is the set of approaches we develop to deal with this large, uncalibrated, noisy mass of measurements.

In addition, in the hopes of abetting future studies, we will make our TCP data publicly available via the *Internet Traffic Archive*, sited at:[6]

```
http://www.acm.org/sigcomm/ITA
```

The routing data analyzed in Part I is already available in the Archive, under the name *NPD-Routes*.

Automated analysis

Confronted with 20,000 traces to analyze, it is clear that we cannot hope to individually analyze each trace. We must instead turn to *automated analysis*. That is, we realize part of our analysis in terms of a computer program that has coded into it the different reductions and calculations required by the analysis. We briefly mentioned this program, `tcpanaly`, above. One of its basic tasks is to separate TCP endpoint behavior from network behavior, hence its name. Another is to then characterize the network dynamics reflected in the trace of the connection.

`tcpanaly` undertakes what we might call "micro-analysis." It is limited in its scope to analyzing single connections. The "macro-analysis," namely the sifting through the individual micro-analyses in search of unifying observations and themes (much in the sense of "scientific inference," as discussed in [Cha95]), is then done manually.[7] Both forms of analysis are highly iterative processes, and each gives insight into the other by identifying patterns that merit further investigation.

[6] At the time of this writing, the Archive is moving from its old location to this URL. If the reader has any difficulty accessing the Archive, send email to `vern@ee.lbl.gov`.

[7] We used the *S* statistical environment [BCW88] for the macro-analysis.

Self-consistency checks

To address the second problem—lack of separate calibration—we must turn to "self-calibration" in the form of *self-consistency* checks: testing, to as great a degree as possible, for any ways in which different aspects of the data contradict one another.

Calibration is all about detecting *error*, whether introduced by the measurement process, or by the subsequent analysis. Ideally, all of the effort is for naught; the data and analysis are wholly free of error. Consequently, it can sometimes be tempting to skip calibration or treat it lightly, since it only provides negative results. Doing so, however, undermines the entire validity of the measurement process. Furthermore, our experience in conducting both this study and several other large-scale studies [Pa94a, Pa94b, PF95] is that, when the scale becomes sufficiently large, errors are inevitable, since even rarely observed problems have sufficient opportunity to manifest themselves. Thus, we discuss self-consistency checks throughout our study. (For example, Chapter 12 is almost entirely about developing self-consistency checks for calibrating the timing measurements recorded in our traces.) The degree to which these checks prove persuasive is the degree to which one might accept our findings as well-grounded.

Robust statistics

The final problem we must address with our analysis strategies is that of widespread noise. For example, if we wish to summarize a connection's round trip times (RTTs), we might at first think to express them in terms of their sample *mean* and *variance* (or *standard deviation*, the square root of variance). However, in practice we find that often a connection observes one or two RTTs that are *much* higher than the remainder. These extreme values greatly *skew* the sample mean and variance, so that the resulting summaries do not accurately reflect "typical" behavior.

To address these sorts of problems, statisticians have developed the field of *robust statistics* [HMT83]. These are statistics that remain resilient in the presence of extremes, or "outliers." One example is use of the *median*, or 50th percentile, as a statistic for summarizing a distribution's central location, rather than the mean. Unlike the mean, the median is virtually unaffected by the presence of outliers.

In our study, we make heavy use of medians as robust estimates of central location. To compute a median of n points, $x_i = x_1, \dots, x_n$, we sort the points to obtain $x_{(1)}, \dots, x_{(n)}$, and then use:

$$\text{median}(x_i) = x_{(\frac{n+1}{2})},$$

if n is odd, or:

$$\text{median}(x_i) = \frac{1}{2}(x_{(\frac{n}{2})} + x_{(\frac{n+1}{2})}),$$

if n is even.

A robust statistic for measuring variation is the *interquartile range*, or IQR [Ri95]. The IQR is the difference between a distribution's 75th percentile and its 25th percentile. Thus, it characterizes the distribution's "central variation." It is likewise virtually unaffected by the presence of outliers, since these by definition fall outside of the range of the values used to compute the IQR. We likewise in our study often make use of IQR rather than standard deviation.

One other technique we borrow from robust statistics is that of fitting a line to a series of $\langle x, y\rangle$ points. Techniques such as least-squares can be heavily skewed by trying to minimize the distance between the fitted line and any outliers. The technique we use, taken from [HMT83], is to first estimate the slope of the line as the median of all of the pairwise slopes between the different points, and then estimate the intercept as the median of the offset of the y coordinates from a line with the given slope and zero-intercept.

9.2 An overview of TCP

In this section we give an overview of how the Internet's TCP protocol works. We make numerous references to its operation in subsequent chapters. Our presentation is not exhaustive, but confined to those aspects of TCP relevant to our later discussion.

The main protocol used in the Internet for reliable data delivery is the Transmission Control Protocol, or TCP. TCP is specified in [Po81c], with updates and clarifications given by [Br89], as well as several other documents specifying optional extensions [BJ88, BBJ92, Br94, MMFR96]. Stevens gives an excellent, detailed description of how TCP works [St94], and [WS95] analyzes an entire TCP implementation line-by-line.[8] TCP is implemented on top of the Internet Protocol, or IP, described in [Po81a]. The combination is often referred to as "TCP/IP."

9.2.1 Data delivery goals

TCP is a complex protocol, since it was designed to accomplish a number of objectives:

- *In-order* delivery, meaning that data is presented to the receiving application in the same sequence as transmitted by the sending application.
- A *byte-stream* model, in which the sender and receiver view the data simply as a series of bytes, with no apparent boundary points (such as those introduced by packetization).
- *Reliable* data delivery, meaning that all of the data transmitted ultimately arrives at the receiver with its original contents (i.e., undamaged).

Accomplishing these objectives in an environment where packets can be delayed, dropped, reordered, duplicated, or corrupted is quite challenging. TCP achieves in-order, byte-stream data delivery by assigning each byte of data a *sequence number*, corresponding to its offset from the beginning of the byte stream. It does so efficiently by associating with each data packet a beginning sequence number (i.e., the sequence number of the first byte in the payload) and a length, which then gives the packet's upper sequence number. In subsequent discussion, we will adopt the convention of using upper sequence numbers to distinguish between different data packets. When this identification is not unique, we will also give the time at which the packet was sent or received, to disambiguate.

TCP achieves reliability by having the data receiver return *acknowledgements*, or "acks," to the data sender.[9] Each ack includes an acknowledged sequence number, which indicates *all* of the in-order data that the receiver has successfully received. For example, if data packets with sequence numbers 1, 2, 3, 5, and 6 arrive at the receiver, then it can acknowledge up to sequence

[8] Both books also discuss other Internet protocols in depth.

[9] It also uses a 16-bit *checksum* to verify data integrity, a point we return to in § 11.4.2.

number 3. It cannot acknowledge 5 or 6, since they are not (yet) in-order. When the receiver subsequently receives sequence number 4, then it can acknowledge all the way up to 6. Such acks are termed “cumulative,” since receipt of any ack serves to acknowledge all of the data correctly received so far. [MMFR96] describes a TCP extension for “selective acks” (SACKs), which allow more detailed feedback of exactly which out-of-order packets have arrived at the receiver so far. In Chapters 13 and 15 we study some aspects of the efficacy of this extension, finding that it has considerable merits.

If a TCP sender does not eventually receive an ack for data it has sent, then it concludes that the data packet was lost (“dropped” or “discarded”) during its journey through the network, and it retransmits the data in a new packet. Such a retransmission is termed a “timeout retransmission,” because it occurs when a timer expires indicating that enough time has elapsed that the packet was presumably lost, since an ack should have been received by now. The amount of time to wait before retransmitting is termed the retransmission timeout (RTO). Choosing a good value for RTO is a major problem, which we discuss in more detail below. We discuss another form of retransmission in § 9.2.7.

9.2.2 Achieving high performance

Achieving these objectives would be considerably simpler if TCP did not have another goal, namely *performance*. Without performance considerations, one can achieve in-order, reliable byte-stream delivery by simply sending one packet at a time until the receiver acknowledges it, and then advancing to the next packet (“stop-and-go”). Stop-and-go can be tremendously inefficient in terms of the performance achieved. If packets are b bytes and the round-trip time (RTT; the interval between when a packet is sent and when the corresponding acknowledgement arrives) is ΔT seconds, then even if the network path is completely unloaded and does not suffer from any undue loss or delay, the maximum achieved throughput is:

$$\rho = \frac{b}{\Delta T}. \tag{9.1}$$

A typical value for b is 512 bytes, and a typical cross-country path in the U.S. has $\Delta T \approx 100$ msec, so $\rho = 5,120$ bytes/sec, even though the path might be capable of transferring megabytes per second.

TCP addresses performance issues in several ways. First, it sends packets that are as large as possible. Each Internet path has a Maximum Transmission Unit (MTU), which is the largest IP packet that can be transmitted along the path without incurring potentially expensive “fragmentation” into smaller packets. An end-to-end path's MTU is the minimum of the MTUs of the various links that comprise the path. When a connection is established between two TCP endpoints, they negotiate a Maximum Segment Size (MSS), which is the largest amount of data each TCP is prepared to receive in a single packet transmitted to it by the other TCP. In general, the MSS is less than the MTU, since the MTU must also include the overhead associated with each packet, namely its protocol header information.[10] Given these considerations, TCP implementations strive to transmit “full-sized” data packets, meaning those that carry MSS bytes of user data. They cannot always do so, if the sending application has not provided them with enough data to completely fill

[10] Some TCPs confuse MSS and MTU, as described in § 11.5.4.

a data packet. For our bulk-transfer connections, however, this is generally not a problem, and the TCPs usually sent full-sized data packets.

Using full-sized data packets helps increase b in Eqn 9.1, but never beyond MSS. Generally, MSS values are on the order of 512 bytes or sometimes 1460 bytes or 4 Kbytes, so this increase alone does not suffice for achieving good performance along a high-speed path. The much larger performance gain comes from having *multiple* packets in flight at one time. If a TCP has k packets in flight, then the potential throughput is:

$$\rho = \frac{kb}{\Delta T},$$

which can in principle match any available path speed ("bandwidth") by using a suitably large k.

The problem then becomes how to choose k. There are two separate concerns: how fast the receiver can accept data, and how fast the network can accept data. The first problem is referred to as "flow control," and the second as "congestion control."

TCP addresses flow control by including in the receiver's acks an "offered window" (also referred to as "advertised window", "receiver window," and, in some contexts, as simply the "window"). The offered window specifies how much new data the receiver promises to accept from the sender. It reflects the buffer available at the receiver, which is used to absorb discrepancies between the rate at which new data arrives at the receiver, and the rate at which the receiving application consumes that data from the receiving TCP. When the available buffer changes, the receiving TCP may send "window updates," which are acknowledgements with revised values for the offered window.

The offered window is expressed in terms of a "credit" beyond the packet acknowledged by the ack. For example, suppose the MSS is 512 bytes and the receiver has 4,096 bytes of buffer available. If data packets with sequence numbers 512, 1024, 2048, and 2560 arrive,[11] then the receiver can acknowledge up to 1024, since the first two packets arrived in sequence. It cannot acknowledge 2048 or 2560 because they arrived "above sequence." So far, the receiving application has not consumed any of the data, even though it could read the first 1,024 bytes if it wished. So the receiving TCP needs to hold the data from all four of the packets in its buffer. Because it has 4,096 bytes of buffer, it can accommodate additional data up to sequence 4096, so in the ack it includes an offered window of 4,096$-$1,024 $=$ 3,072 bytes, instructing the sender that it can accommodate 3,072 bytes beyond what is has now acknowledged receiving. Note that it advertises 3,072 bytes even though it has already committed 4 packets worth of buffer, or 2,048 bytes, leaving it with 2,048 bytes of uncommitted buffer. This works because the sender can only use the 3,072 byte credit as a window beyond the acknowledgement point ("ack point"). So it can only transmit 2,048 bytes' worth of data not already buffered, namely those corresponding to sequence numbers 1536, 3072, 3584, and 4096.

Suppose now the receiving application reads the 1,024 bytes that have been acknowledged. (It cannot yet read the data in the later packets, since they are presently "above sequence," so they cannot be read in-sequence yet.) Now the receiving TCP no longer needs to buffer the first two packets, so it can accommodate an additional 1,024 bytes from the sender. It may at this point send another acknowledgement for sequence 1024, but this time with an offered window of 4,096, allowing the sender to transmit all the way up to sequence $1024 + 4096 = 5120$.

[11] Where we are using the conventions that the sequence number refers to the upper sequence number carried in the packet, and that data packets are always full-sized.

When sequence 1536 arrives, then up to sequence 2560 may be ack'd, since the data up to there can now be delivered in-sequence. When the sender receives the ack of 2560, its window, meaning the range of data it can now send, “advances.” As part of this advance, the “upper edge” of the window, meaning the largest sequence number the sender can transmit (equal to the ack point plus the offered window), “slides” to a new maximum. Consequently, transport protocols using this form of flow control are termed “sliding window” protocols.

In this fashion, the receiving TCP can (if it wishes) assure that it is always able to accommodate data arriving from the sender. If, for example, the receiving application ceases to consume data, then eventually the TCP's buffer will fill. When it does, the TCP will advertise a window of 0 bytes, requiring the sender to cease transmission.

9.2.3 Congestion control

Quite separate from flow control is the vital performance issue of *congestion control*. The limitation on how fast the sender should transmit may derive not from limited buffer at the receiver, but limited capacity inside the network. Originally, TCP dealt with congestion control by setting the RTO (retransmission timeout) to a multiple of the estimated mean RTT (round-trip time). When the RTO expired, unacknowledged packets were retransmitted, and the RTO was doubled (“exponential backoff”), so that during periods of high congestion, the connection would progressively lower its sending rate.

In a landmark paper, Jacobson described the shortcomings of this form of congestion control: in particular, its excessive consumption of resources due to retransmitting multiple packets at one time, and the instability that occurs because it does so precisely when the network has been overloaded to the point of packet loss. He also identified inadequacies in the RTO algorithm, which used only the estimated mean RTT, without including an estimated RTT variance [Ja88]. He addressed these problems by introducing a second window, the *congestion window* (or, *cwnd*), and a modified RTO algorithm that includes the estimated RTT variance, both of which have been incorporated into the TCP specification [Br89, St97]. It is no exaggeration to say that the Internet works today only because of these changes. Without them, the network would inevitably devolve into “congestion collapse” (discussed below). Thus, *proper TCP congestion control is* vital *to the Internet's stability*, a point we return to in Chapter 11, where we find that some TCP implementations fail to follow these requirements.

We will focus on the first of Jacobson's changes, the congestion window. *cwnd* is completely separate from the receiver's offered window. At any time, a TCP sender must not send beyond the *minimum* of the two windows. The offered window governs how much in-flight data the receiver's buffer can accommodate, and *cwnd* governs how much the buffers along the network path can accommodate. The networking infrastructure, however, does not provide this latter information explicitly (nor can it, for scalability reasons). Jacobson's insightful observation was that the network path does, however, provide an *implicit* signal that its buffer resources are scarce: namely, it drops packets. Thus, packet loss is interpreted as a sign of congestion (also observed by Jain in [Jai89]). Such losses are termed “congestive losses.” While packet loss can occur for other reasons, the presumption is that most losses occur due to congestion, and so merit a response by the sending TCP: diminishing the rate at which it transmits packets. It does so by reducing *cwnd*.

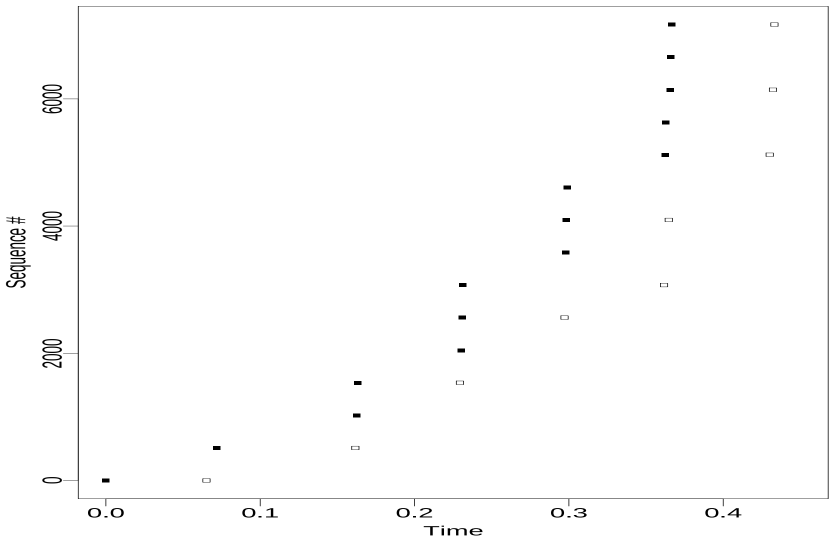

Figure 9.1: Sequence plot of a TCP connection during its "slow start" phase

9.2.4 Slow start

Jacobson discussed two different issues in managing *cwnd*. The first is what value to use for it initially, which we address in this section. The second is how it should be cut upon detecting loss, to adapt to congestion, which we address in § 9.2.6. His scheme addresses the first issue by initializing *cwnd* to one packet (more precisely, to MSS bytes), so connections begin by transmitting just one packet and waiting for an acknowledgement. Each ack that arrives then increases *cwnd* by one packet (again, actually by MSS bytes). Thus, if the receiving TCP sends an ack for every in-sequence packet it receives, then in the absence of loss the congestion window will be 1 packet, 2 packets, 4 packets, 8 packets, and so on, where each increase reflects *cwnd* after the packets in the previous "flight" have been acknowledged. (We use the term "flight" to refer to a set of packets transmitted within a single RTT's worth of time.) Thus, in the absence of loss, *cwnd* increases exponentially quickly. It continues to do so until either it is limited by the receiver's offered window; or the connection suffers a loss, indicating that a network-imposed congestive limit has been reached; or the connection completes before either of these occur.

This form of window increase is called "slow start," since the window starts at a small value, and hence the TCP transmits slowly at first. Figure 9.1 shows a "sequence plot" of the packets sent and received by a TCP sender during its slow-start phase. We will make extensive use of such plots and so describe them here in detail. The x-axis gives time since the connection was established. The y-axis gives sequence numbers: these are either upper sequence numbers for data packets (shown as solid squares), or acknowledged sequence numbers for acks (hollow squares).

Sequence plots are highly informative illustrations of what happens during a connection. Here, the solid square at $T = 0$ sec with sequence number 1 corresponds to the "initial SYN" packet. Each connection begins with the originator transmitting a packet with the "SYN" flag set in the header to request establishment of a connection ("SYN" is short for "synchronize sequence numbers”). If the connection request is accepted, then the responder replies with a SYN acknowledgement (“SYN-ack”) packet. If the sequence numbers in the SYN-ack accord with those that the originator sent, then the sender acknowledges the SYN-ack and the connection has been established. Because establishment entails exchanging three packets—the initial SYN, the SYN-ack, and the final ack of the SYN-ack—it is referred to as a “three-way” handshake.[12] TCP terminates connections in a similar fashion, using an exchange of “FIN” (“finish”) packets and a final ack, for another three-way handshake.

The initial SYN carries a sequence number of 1 because the SYN flag conceptually occupies the first sequence position of the byte stream. At about $T = 0.07$ sec, the plot shows the arrival of an acknowledgement for sequence number 1. This is the SYN-ack packet. Shortly after, the sender begins transmitting. It sends a single packet carrying 512 bytes (and with sequence number 513), because *cwnd* has been set to one packet due to slow start. This data packet also carries the ack for the SYN-ack packet, and hence completes the three-way handshake. When 513 is ack'd at $T = 0.17$ sec, the congestion window opens to two packets, and these are promptly sent (sequence numbers 1025 and 1537). Both of these are acknowledged by a single ack at $T = 0.23$ sec, which opens *cwnd* by an additional packet, and three new packets are sent.[13]

At $T = 0.30$ sec, an ack arrives for the first two of the three packets in this flight. It opens the window to four packets. Since one of these four is already in flight, the TCP only transmits the three new ones. At $T = 0.36$ sec an ack arrives for the third packet of the earlier flight (sequence 3073). This is a *delayed* ack, one that the receiver momentarily refrained from sending in hope that more data would arrive and it could ack two packets at once. (The receiver employs an “ack-every-other” policy for sending its acknowledgements, as do many TCPs.) Very shortly after this ack arrives, so does another one, for sequence 4097, corresponding to the first two packets of the most recent flight. Each of these acks advances *cwnd* by one packet, so after both arrive, *cwnd* is 6 packets. One of these is already in flight (and not yet unacknowledged), so the TCP sends the other five.

Note that we can read the RTT directly from the plot: it is the x-axis distance between the transmission of a packet and its acknowledgement, in this case about 70 msec.

The sending TCP continues to open up *cwnd* until loss occurs. If *cwnd* reaches the point where it exceeds the size of the offered window, then the connection becomes *receiver-window limited*. Figure 9.2 shows a sequence plot of the same connection later in its transmission, when this has occurred. We have added circles to the plot indicating the upper “edge” of the window, that is, the sum of the offered window and the sequence number acknowledged by the ack (the “ack point”). We see that the sending TCP closely tracks the upper edge, sending packets up to that limit every time the edge advances.

9.2.5 Self-clocking

Another effect shown in Figure 9.2 is the important phenomenon of *self-clocking*. In the figure, each flight of data packets elicits in response an ack “echo” that preserves the temporal

[12] TCP uses a three-way handshake for reliability concerns that we will not describe further here; see [St94] for a detailed discussion.

[13] One might expect that *cwnd* would open by two additional packets, since the received ack acknowledges receipt of that many packets. However, the TCP standard governing congestion window management [St97] specifies that, during slow start, each ack for new data increases the window by one packet, regardless of how much new data has been ack'd.

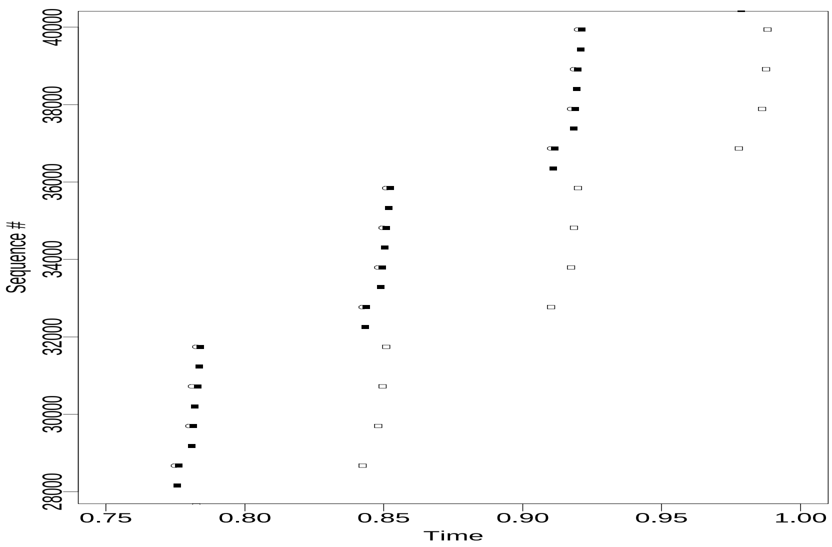

Figure 9.2: Sequence plot of a "window-limited" TCP connection

structure of the flight. When the flight of acks arrives at the sender, the window advances in step with the echo, because the receiving application is consuming the data as fast as it arrives, and hence the offered window remains constant—4,096 bytes, in this case. Because the connection is receiver-window limited, the sending TCP then transmits new data whose temporal structure reflects that of the window advances, and thus ultimately reflects that of the previous flight of data packets, so the cycle continues.

The term "self-clocking" is also due to Jacobson. It comes from the observation that a window-limited TCP connection will over time naturally pace out its data packets to exactly match the bandwidth available along the network path. Figure 9.3, reproduced from [Ja88], illustrates this property. The top "pipe" represents that network path from the sender to the receiver, and the bottom pipe that from the receiver back to the sender. The thickness of each component in a pipe reflects the bandwidth available at that part of the pipe, and horizontal distances correspond to differences in time. Each packet occupies a portion of the pipe, shown as a shaded region. The width of these regions indicates how long it takes the packet to traverse that portion of the pipe, and the height reflects that the packet consumes the region's available bandwidth during the traversal. In the figure, the sender has sent a number of packets back-to-back into the local, high-speed end of the path. These packets travel through the network closely spaced, until they reach the path's *bottleneck* (thin central region), where the available bandwidth sharply diminishes. At this point, it takes much more time to transmit each packet, so they spread out in time.

The key observation underlying self-clocking is that once the packets have been spaced out to a distance P_b by the bottleneck, they *remain* spaced out. That is, P_r in the figure is equal to P_b. There is no mechanism for subsequently recovering their initial spacing.[14] Furthermore, such recovery is not desirable: the distance P_b is in fact the optimal spacing for the connection's

[14] To first order. See § 16.3.

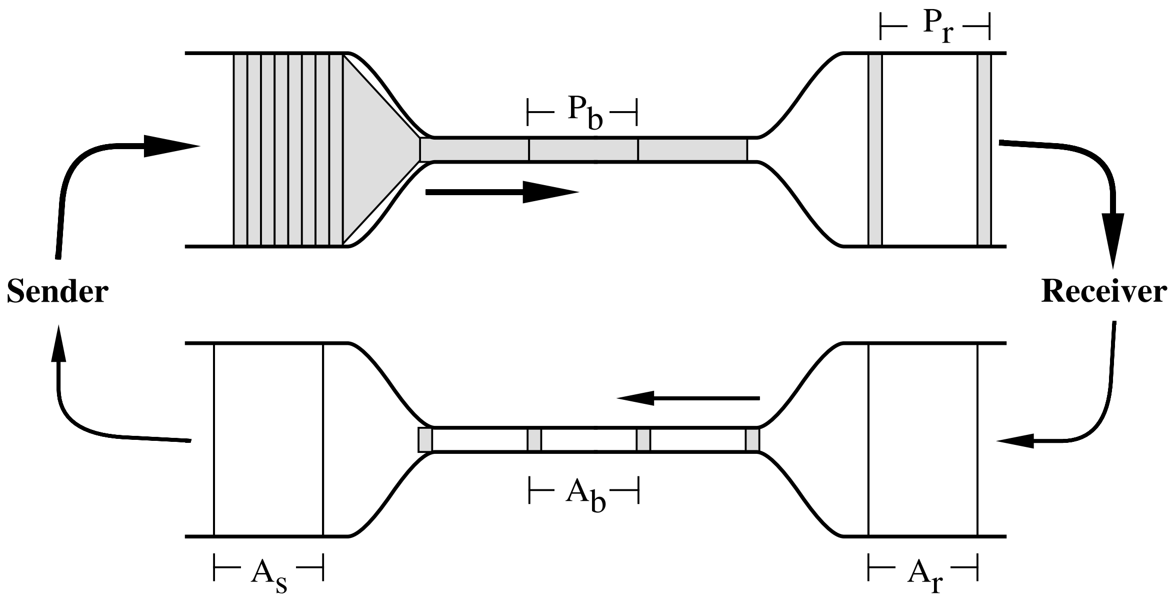

Figure 9.3: TCP "self-clocking." Reprinted with permission from [Ja88], copyright ©1988 Association for Computing Machinery.

packets. If they are transmitted any closer together, they will simply have to *wait* in a queue at the bottleneck anyway, because it cannot accommodate a faster rate. So we *want* packets ideally to be transmitted with a distance P_b between them. Any less and they cause queueing without any gain in performance. Any more and the connection underutilizes the available bandwidth.

The very nice property of window-based flow control is that, as the data packets arrive at the receiver with a spacing of $P_r = P_b$ between them, the receiver generates acks for them and these *also* have a spacing $A_r = P_r = P_b$ between them. Furthermore, the acks are *small* and are *not* spaced out by the bottleneck along the return path, even if it is smaller than that along the forward path. Consequently, the acks arrive at the sender with a spacing A_s between them, and because the timing has not been perturbed, $A_s = P_b$. Finally, these acks then advance the window, and the sender transmits new packets in response to them. The timing of the new packets, however, reflects that of the acks, and hence they have a spacing of P_b, just as we desire. Thus, the connection settles into a state in which it "clocks out" new packets at exactly the proper rate for the available bandwidth.

Receivers that employ ack-every-other policies, such as that shown in Figure 9.2, perturb self-clocking in only a minor fashion. Instead of generating acks with a spacing $A_r = P_b$ between them, they generate acks with a spacing of $A_r = 2P_b$. Consequently, in the absence of extra delays along the return path, they arrive at the sender with that same spacing. Note, however, that these acks advance the window by *two* packets each instead of one packet, so the sender then transmits two packets that are spaced a distance $P_r = 2P_b$ apart from the previous group of two packets. Thus, over intervals of $2P_b$, the connection transmits at exactly the bottleneck rate. On finer time scales, it can transmit faster, but the excess is only one additional packet, so very modest buffer space at the bottleneck can accommodate the burst. In § 11.6.1 we will see other acking policies that involve acking large numbers of packets with single acks. These can lead to highly bursty arrivals at the bottleneck.

Self-clocking is an idealized state. In practice, connections might not self-clock due to delay variations along the network paths in either direction, discussed in depth in Chapter 16. One particular form of delay variation that defeats self-clocking is *timing compression*, which we discuss in § 16.3.

Connections also do not self-clock when in the slow-start phase, since arriving acks do not simply advance the window, but widen it also. Consequently, the average spacing between the packets transmitted by the sender during slow start will be less than P_b. When the network is unloaded, this behavior is not only acceptable but desirable, because our goal is to *continually* send packets with intervals of P_b between them ("filling the pipe"). If we can accomplish this, then the connection sustains transmission at the full available bandwidth, and we have achieved the maximum performance possible along the given path. However, a TCP sender does not know in advance the proper value of P_b (and the proper value might change over the course of a connection). Slow start is a mechanism for *hunting* for the correct spacing, by continually opening up the window until the connection finds itself in the self-clocking regime corresponding to the currently available bandwidth. The difficulty with this hunt is knowing when to stop. TCP currently determines the stopping point when it has driven the network to the point of packet loss (see below). But this point corresponds to having exceeded the available bandwidth by a factor proportional to the available *buffer space*, too, since Internet routers today only drop packets when their buffers are exhausted. Thus, when beginning a connection, using slow start will often *drive the network to the point of loss*, which is excessive, since we instead want them to *drive the network to the limit of available throughput*.

There are proposals for modifying either TCP [WC91, WC92, BOP94] or the drop policy used by routers [RJ90, FJ93] so that connections can find the available bandwidth without unduly stressing the network. We comment on both approaches as we analyze our measurements. The TCP modifications are of particular interest for our study because they rely on accurate packet timing information, which we will find can be elusive.

In Figure 9.2, the connection fails to fill the pipe because it is receiver-window limited. In general, to fill the pipe requires that the window size in bytes, w, exceeds the "bandwidth-delay product," i.e.:

$$w \geq \rho_A \cdot \text{RTT}, \tag{9.2}$$

where ρ_A is the available bandwidth in bytes per second, and RTT is the round-trip time in seconds. We develop this relationship in detail in § 16.1. While estimating RTT is not difficult, estimating ρ_A is, so connections cannot readily use Eqn 9.2 to determine their correct window size. In Chapter 14 we discuss ways of estimating the *bottleneck* bandwidth, ρ_B, which is an upper bound on ρ_A, and in § 16.5 we look at the relationship between the two.

9.2.6 Responding to congestion

The other fundamental component of Jacobson's modifications to TCP concerns how TCP reacts to *congestion*, i.e., periods when some element of the end-to-end chain of routers and links is under stress: a sustained interval during which packets arrive more quickly than the element can service them. During congestion, the unserviced packets are queued at the congested router until they can be serviced. If the congestion lasts long enough, the queues build and build until eventually the queued packets exhaust the router's buffer. At this point, the router must discard incoming packets.

Note that congestion spans a *spectrum* between busy periods during which queues grow large, and periods when no more buffer remains and packets are lost. Thus, packets may or may not be lost during congestion periods, depending on the sizes of the buffers and the duration of the congestion. (We examine the interplay between delay and loss in § 16.2.4.)

Congestion is potentially *lethal* to a network because it can lead to positive feedback that sustains and even magnifies the congestion. In particular, if packet loss leads to retransmissions that are sent at the same rate as the original packets, then the load borne by the network will not diminish, and the congestion sustains itself. Packets from newly-initiated connections add further to the load, leading to even higher levels of congestion.

The positive feedback can thus bring the network to a state of *congestion collapse*, in which the network load stays extremely high but throughput is reduced to close to zero [Na84]. Exactly this happened in the early days of the Internet, and led to Jacobson's work on TCP congestion avoidance [Ja88]. As discussed above, one of the key insights of that work is that the network provides an implicit signal of congestion in the form of packet loss. Barring loss due to causes such as transmission noise on a network link, the network should only discard packets if it no longer has enough buffers to carry them. Consequently, when a TCP sender observes a packet loss, it should infer that the network path is congested, and ease its use of the path by cutting *cwnd* (and hence limiting its transmission rate). It does so as follows.

First, upon retransmitting, *cwnd* is set to one packet, so the connection begins a "slow start" phase in order to hunt for the correct value of the available bandwidth again. Second, the TCP state variable *ssthresh* ("slow start threshold") is set to half of the window in effect at the time of the retransmission (i.e., the smaller of either the offered window, or the value of *cwnd* prior to setting it down to one packet). The intent behind *ssthresh* is to denote the window size beyond which it is likely that there is no more available bandwidth. The sending TCP should only gingerly expand *cwnd* beyond *ssthresh*.

As acks arrive for packets now transmitted by the sender, each increases *cwnd* by one packet, per the usual slow-start increase. Once *cwnd* reaches *ssthresh*, however, then the TCP increases *cwnd* by only one packet per RTT. Thus, the rate at which *cwnd* increases changes from *exponential* during the slow-start phase to *linear* during the "congestion avoidance" phase.

Figure 9.4 illustrates how a TCP timeout retransmission appears on a sequence plot. At $T = 2.3$ sec, data up to 24577 has been acknowledged, and eight more packets are in flight, which equals the offered window. A little later two more acks for 24577 arrive ("duplicates," as discussed in § 9.2.7 below). However, no additional acks are forthcoming. At $T = 3.4$ sec, the RTO expires and the sending TCP retransmits the first unacknowledged packet, 25089. At this point, *cwnd* has been set to 1 packet (which is why only one packet is retransmitted), and *ssthresh* has been set to 4 packets, half of the window in effect at the time of the retransmission. The retransmitted packet elicits an ack for 28673, corresponding to all of the outstanding data. This indicates that only 25089 was dropped by the network—all of the later packets arrived at the receiver, so a retransmission of 25089 was all that was needed to fill the sequence "hole."

The ack for 28673 both advances the window edge and enlarges *cwnd* to 2 packets, so

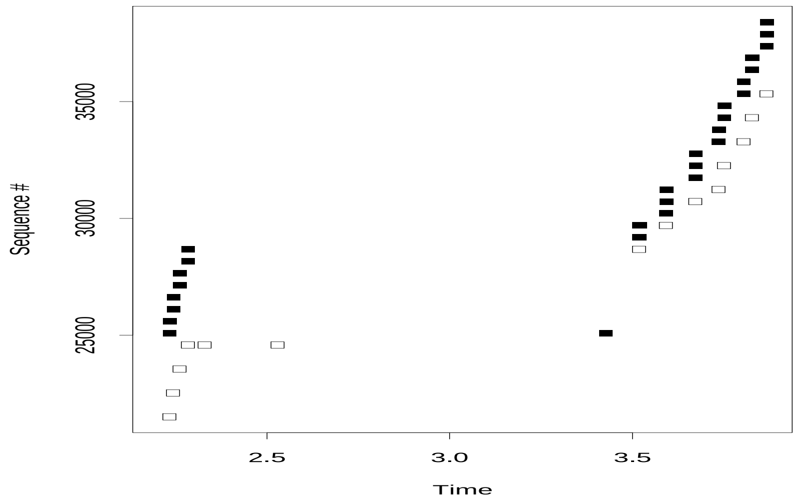

Figure 9.4: Sequence plot showing a TCP timeout retransmission

the sender now transmits two new packets. As acks arrive, the sender continues rapidly increasing *cwnd* in slow-start fashion. However, when the ack for 32257 arrives at $T = 3.75$ sec, *cwnd* does not increase from 5 packets to 6 packets, but remains at 5 packets, because the TCP has now entered congestion avoidance. It is only after the arrival of the ack for 35329, the last in the plot, that *cwnd* increases to six packets.

While the above outlines the congestion avoidance principles, in practice there are many fine points regarding exactly how congestive avoidance is implemented. (For example, why in Figure 9.4 it took more than one RTT during congestion avoidance for *cwnd* to increase by one packet.) We discuss a number of these in Chapter 11.

9.2.7 Fast retransmit and recovery

In addition to timeouts, TCP supports another retransmission mechanism, called "fast retransmit." It is also due to Jacobson. Although not part of the TCP specification, it is widely implemented. Fast retransmission is an attempt to avoid the sometimes lengthy lulls a connection experiences upon a loss, due to the RTO being much larger than the RTT. Figure 9.4 above illustrates the problem. For this connection, the RTT was about 65 msec, but the RTO wait was 1.2 sec.

In general, RTO should be larger than the *maximum* RTT a connection's packets might experience, in order to allow enough time for acks to arrive. Yet, it is difficult to estimate this maximum due to frequent fluctuations in RTT, and, furthermore, it is important to estimate it conservatively, i.e., overestimate it rather than underestimate it, so that packets are not needlessly retransmitted. (We will see the effects of underestimation in § 11.5.10.) Finally, many TCP implementations have access to only coarse-grained clocks, so it is difficult for them to time small RTOs.

To address this problem, Jacobson noted that TCPs receive an additional, implicit signal when a packet has been lost. This signal comes in the form of the arrival of "duplicate acks." When

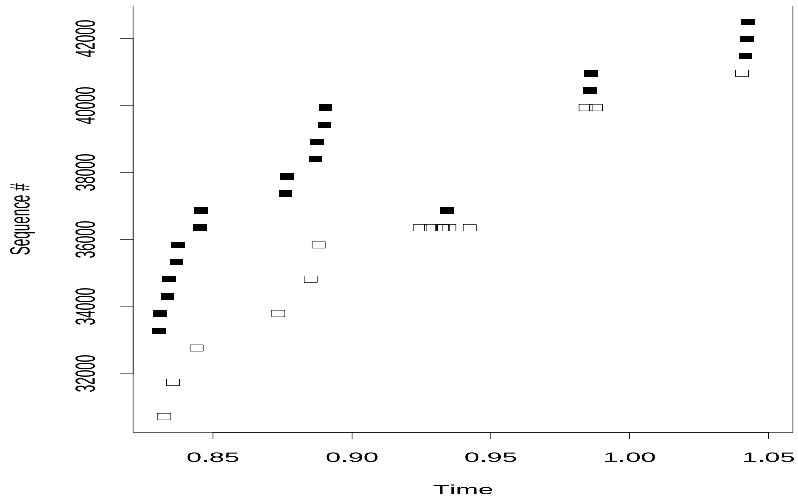

Figure 9.5: Sequence plot showing a TCP "fast retransmission"

above-sequence data arrives at a TCP receiver, the specification states that the TCP should generate a redundant acknowledgement.[15] These are termed "duplicate acks" or "dups." In Figure 9.4 we see two of these arriving at $T = 2.33$ sec and $T = 2.53$ sec. Since all but the first packet beyond the ack point arrived at the receiver, it should have sent 7 dups. From the plot, we cannot tell whether it did so but 5 were lost, or if it failed to do so. (It turns out that, in this case, it failed to do so. The TCP implementation was one that does not include the recommended generation of dups.)

Fast retransmission works by counting duplicate acks, and, if their number reaches a given threshold, N_d, then the sending TCP infers that the packet beyond the ack point was lost, and retransmits it. Current implementations use $N_d = 3$. This value was chosen as a trade-off between not missing fast retransmit opportunities because too few dups arrive, versus not misinterpreting the arrival of dups and retransmitting unnecessarily. The latter can occur when packets arrive out of order. In § 13.1.3 we examine how well $N_d = 3$ performs, and find that it does very well, almost always detecting true loss and not being fooled by reordering; and, further, that $N_d = 2$ would result in TCPs being fooled significantly more often.

Figure 9.5 shows a sequence plot depicting a fast retransmission. Packet 36865, originally transmitted at $T = 0.85$ sec, was lost, but all of its 6 successors arrived successfully. These then elicit six dups, the third of which causes a fast retransmission at $T = 0.93$ sec. At this point, *cwnd* is one packet and *ssthresh* is 4 packets. When the retransmission is ack'd at $T = 0.98$ sec, slow start advances *cwnd* to 2 packets, and then to 3 packets upon receipt of an ack for those two.[16]

Fast retransmit works very well for eliminating the lengthy timeout lull, provided enough above-sequence packets arrive at the receiver to elicit at least 3 dups. (If the receiver's offered

[15] It also does this if below-sequence data arrives, i.e., unnecessarily retransmitted data. We explore the distinction between these two in § 13.1.3.

[16] The apparent duplicate ack for 39937 is in fact a "window update," per § 9.2.2. TCPs are careful to distinguish between window updates and true duplicates, as the former do not indicate the safe arrival of an additional data packet.

window is small, or if *cwnd* is small, then this may be a problem.) Jacobson further refined it with a mechanism termed "fast recovery." The observation underlying fast recovery is that each additional dup beyond the first $N_d = 3$ indicates that another data packet has arrived at the receiver. Thus, it is sound to increase *cwnd* (which was cut to 1 packet upon the fast retransmission) by one packet for each of these, though not to exceed *ssthresh*. Furthermore, it is sound to increase *cwnd* by N_d packets upon a fast retransmission, too, because each of the first N_d dups likewise corresponds to a successfully received packet.

Thus, fast recovery opens *cwnd* more quickly. If this were all that the TCP did, then fast recovery would lead to a large burst when the TCP received an ack for the retransmitted packet ($T = 0.98$ sec in Figure 9.5), because at this point *cwnd* would often be much larger than 1 packet (then increased in Figure 9.5 to 2 packets by the arrival of the ack). To eliminate this burstiness, fast recovery also specifies that, if the TCP receives enough additional dups, it then begins transmitting *new* data, *before* it has received an acknowledgement for the retransmitted data. Thus, the algorithm looks like:

1. Upon receiving 3 dups, set *ssthresh* to half the effective window, set *cwnd* to one packet, and retransmit the first unacknowledged packet.

2. Next, "inflate" *cwnd* using:
$$\textit{cwnd} \leftarrow \textit{ssthresh} + 3,$$
where the constant 3 reflects the three duplicates already received.[17]

3. Whenever another dup arrives, increase *cwnd* by one more packet. If *cwnd* is now large enough to transmit new data, do so.

4. When an ack arrives that advances the ack point to or beyond the last packet that was in flight prior to the fast retransmission, then fast recovery ends. Execute:
$$\textit{cwnd} \leftarrow \textit{ssthresh}$$
to "deflate" the window to its proper post-recovery size, and update *cwnd* from the ack normally.

Figure 9.6 illustrates how fast recovery appears on a sequence plot. A number of dups arrive for 74573, which is retransmitted after the third dup is received (i.e., after four acks for 74573 arrive, the first being the "original" and the others being dups). Prior to the retransmission, the window was 8 packets, so after the retransmission *ssthresh* is 4 packets, and after window inflation, *cwnd* is $4 + 3 = 7$ packets. The next dup advances *cwnd* to 8 packets, but the TCP already has 8 packets' worth of data in flight, so it cannot retransmit at this point. The dup after that, though, arriving at $T = 7.94$ sec, advances *cwnd* to 9 packets, and this is enough to liberate a new data packet, 79361. Two more dups after it advance *cwnd* to 10 and 11 packets, and two more data packets are sent. Then, at $T = 7.96$ sec, all of the data outstanding prior to the retransmission is ack'd (closely followed by a window update, the second ack shown overlapping with the first). At

[17] We have simplified discussion by presenting the algorithms in terms of full-sized *packets*, when in fact they are implemented in terms of *bytes*. Provided all of the packets contain a full MSS' worth of data, these two are equivalent.

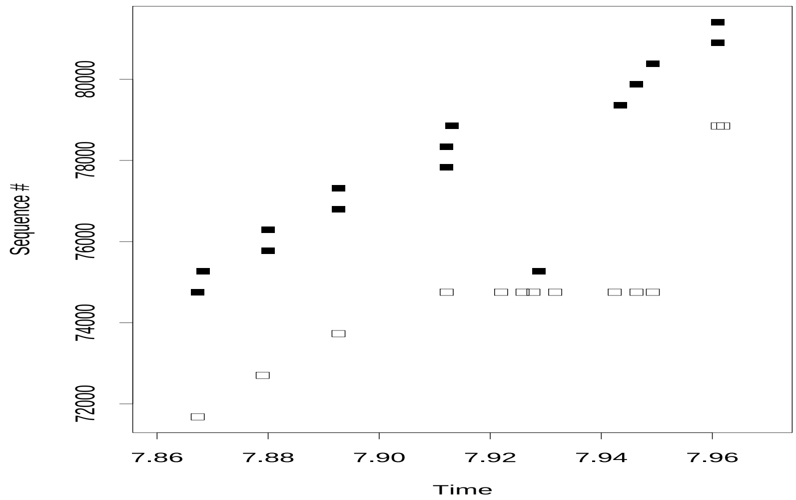

Figure 9.6: Sequence plot showing TCP "fast recovery"

this point, the window deflates back to *ssthresh*, or 4 packets. The ack is then processed, and since this TCP's test for congestion avoidance is

$$cwnd > ssthresh$$

rather than

$$cwnd \geq ssthresh$$

(used by some other TCPs), the connection is deemed still in slow start, so the ack advances *cwnd* to 5 packets. Three of these are already in flight, so the TCP transmits two new packets. Thus, the TCP was able to continue transmitting, and ended the retransmission period with *cwnd* having just entered congestion avoidance, and it did so without generating any unduly large bursts.

We make one final point regarding fast recovery. The window inflation and deflation is subtle (and often confusing). It arises due to conflating the meaning of *cwnd* to be both "how many packets the connection can have in flight" and "how far above the ack point can the connection transmit." During fast recovery, these notions are separate, since some of the packets above the ack point are indeed no longer in flight (because they are what caused the dups). Because these points are subtle, we should not be too surprised to learn in Chapter 11 that TCPs implementing fast recovery suffer from more than one bug in managing the window deflation.

9.3 The Raw Measurements

Table XIV lists the 35 sites that participated in the two experimental runs, $\mathcal{N}_1$ and $\mathcal{N}_2$. Tables I and II in Part I summarize the sites.

We conducted the first run, $\mathcal{N}_1$, during December 1994, coincident with the routing study. Likewise, we conducted the second, $\mathcal{N}_2$, during November–December 1995. As with the routing

Name	# $\mathcal{N}_1$	# $\mathcal{N}_2$	Tracing machine
adv	–	1,244	
austr	207	1,036	BSDI 1.1
austr2	–	1,259	
bnl	307	1,200	
bsdi	166	1,374	
connix	308	1,474	
harv	190	1,061	
inria	172	1,180	
korea	49	–	HP/UX 9.01
lbl	318	1,412	SunOS/BPF
lbli	230	1,134	SunOS/BPF
mid	–	1,295	
mit	308	–	
near	–	1,296	SunOS/BPF
nrao	301	982	
oce	126	838	
panix	–	240	
pubnix	148	1,085	
rain	–	1,289	
sandia	–	1,182	
sdsc	259	964	
sintef1	–	1,469	NetBSD 1.0
sintef2	–	1,524	SunOS/BPF
sri	194	1,306	
ucl	230	1,266	
ucla	–	1,397	SunOS 4.1
ucol	275	1,208	SunOS 4.1 ($\mathcal{N}_1$)
ukc	299	989	SunOS 4.1
umann	222	998	
umont	144	1,469	SunOS 4.1
unij	74	1,412	SunOS 4.1
usc	231	–	SunOS 4.1
ustutt	240	1,165	
wustl	304	1,232	
xor	316	–	
Total	2,805	18,490	

Table XIV: Sites participating in the packet dynamics study

study, differences between $\mathcal{N}_1$ and $\mathcal{N}_2$ give us an opportunity to analyze how Internet packet dynamics changed during the course of 1995.

The second and third columns give the number of connections in which the site participated as either sender or receiver. The final column lists the operating system of the machine used to trace the site's TCP traffic, or empty, if the tracing was conducted on the same machine as ran the TCP. Tracing systems listed as "*XYZ*/BPF" had the Berkeley Packet Filter installed [MJ93], which greatly aids with accurate packet measurement. One site, `ucol`, changed its measurement setup between $\mathcal{N}_1$ and $\mathcal{N}_2$, using a separate machine during $\mathcal{N}_1$ but the same machine during $\mathcal{N}_2$.

As discussed above, each measurement was made by instructing the Network Probe Daemons (NPDs) running at two of the sites to send or receive a 100 Kbyte TCP bulk transfer, and to trace the results using `tcpdump`. An important difference between $\mathcal{N}_1$ and $\mathcal{N}_2$ is that in $\mathcal{N}_2$ we used Unix socket options to assure that the sending and receiving TCPs had sufficiently large buffers that they were never "window limited" (§ 9.2.4), to prevent window limitations from throttling the transfer's throughput. This change has a downside, which is that it sometimes clouds apparent trends between the $\mathcal{N}_1$ measurements and the $\mathcal{N}_2$ measurements with questions concerning whether the trends are simply artifacts of using bigger windows in $\mathcal{N}_2$. Nevertheless, the change was worth making, since the bigger windows enabled the $\mathcal{N}_2$ connections to push considerably harder on the network path, with more opportunities to observe the amount of resources the path had available as a result.

Finally, we limited measurements to a total of 10 minutes, as a mechanism to prevent measurement attempts from indefinitely consuming resources at the NPD sites. This limit leads to *under-representation* of those times during which network conditions were poor enough to make it difficult to complete a 100 Kbyte transfer in that much time. Thus, our measurements are *biased* towards more favorable network conditions. In § 15.1 we show that the bias was negligible for North American sites, but noticeable for European sites.

Chapter 10

Calibrating Packet Filters

The data for our entire packet dynamics study are traces of packets sent through the network recorded by the `tcpdump` utility, written by Van Jacobson, Craig Leres, and Steve McCanne [JLM89]. `tcpdump` uses a host computer's *packet filter* to measure when packets appear on the local network. Packet filters are operating system services for recording network packets. In this chapter we discuss the general problem of how to test the soundness of a trace measured by a packet filter, and the specific issues that arise from the different packet filters used in our study.

We begin by introducing the notion of "wire time" (§ 10.1), and then describe how packet filters work (§ 10.2). One of the goals of a packet filter is to record wire times as accurately as possible. In § 10.3 we give an overview of the sorts of measurement errors packet filters can exhibit. For each error, we discuss how `tcpanaly` attempts to detect its presence when analyzing a `tcpdump` trace. While not a measurement error, a packet filter's "vantage point"—where in the network it is located—can also complicate the analysis of a `tcpdump` trace, which we discuss in § 10.4. Finally, it is not quite as simple as it might at first appear to pair up instances of the same packets in two `tcpdump` traces, one recorded at the TCP sender and one at the receiver. § 10.5 covers the details of doing so.

10.1 The notion of "wire time"

If we wish to accurately describe how packets travel through a network, then we need to carefully specify exactly what we mean when we associate a time with a packet's appearance on a link in the network. To do so, we introduce the notion of "wire time." Wire time is defined in terms of a particular measurement location M on a particular network link L. For a given packet p, the wire time of p on L is the time t at which p appears at M on L. Note that this definition is vague in some fundamental ways. Sometimes what we (ideally) want to know is when p *first* appears at M, which one might define as p's "wire arrival time," corresponding to the first moment at which any bit of p is viewed at M on L. Other times we want to know when p *finishes* appearing at M, its "wire completion time," corresponding to the first moment at which all the bits of p have been seen at M on L. These two times can be quite different if L has a low bandwidth, and so it takes a long time for all of p's bits to pass M.

Depending on the particular link, wire times can vary considerably with different measurement points on the link, such as the two ends of a satellite link; or very little, such as the various measurement points on an Ethernet.

For each packet recorded by a packet filter, the filter generates a *timestamp* corresponding to the time at which the filter captured the packet (discussed further in § 10.2). One goal of a well-designed packet filter is to ensure that this timestamp is as close as possible to the packet's wire time with respect to the packet filter's measurement location, M.

However, the filter's location M will often differ from the location E where the connection endpoint whose traffic we wish to measure resides. (This difference affects the packet filter's *vantage point*, an issue we discuss in detail in § 10.4.) Consequently, it may be difficult to accurately estimate from packet filter timestamps recorded at M the wire times as seen at E. In our study, however, the packet filters always monitored the same local-area network (LAN) as was used by one of the endpoints in our study, or ran on the endpoint itself. Since the LAN's had small propagation times, this means that the packet filter timestamps were (potentially) quite close to wire times as seen at E.

10.2 How packet filters work

The goal of a *packet filter* supplied with an operating system is to selectively record network traffic. This operation is referred to as packet "capture." The captured packets might be only to or from the computer running the packet filter, or might be ancillary traffic that has nothing to do with the local computer. In the latter case, the filter still needs some way to "see" the traffic in order to measure it. This is done by means of passively monitoring broadcast media such as Ethernet or FDDI networks, a mode of operation referred to as "promiscuous." With non-broadcast media such as point-to-point links or Ethernet "hubs," passive operation is sometimes not possible (depending on the design of the networking elements) unless considerable pains are taken to split the physical signal so that the passive monitoring machine receives its own copy. For our study, measurement was always done either in the context of a broadcast medium, or on the endpoint host itself.

The position of a packet filter with respect to the TCP endpoints, or its "vantage point," can complicate analysis of cause-and-effect among the streams of packets between the sender and the receiver. We discuss this issue further in § 10.4. We note here that vantage point complexities are often more significant for passive monitoring because the monitoring machine is further removed from the TCP endpoint. Apart from this issue, which can be important, we in general prefer passive monitoring because it minimizes measurement error. A passively-monitoring packet filter often can yield more accurate estimates of "wire time" because the computer doing the measurement is not also busy processing the network traffic itself.

Packet capture usually takes place inside the operating system's kernel, since dealing with hardware devices such as network interfaces generally falls within the kernel's domain. It is presumably at this point that the packet's *timestamp* is generated, reflecting the time at which the packet was captured. Hopefully this occurs as early in the process as possible, so that the timestamp is as close to the packet's wire time (§ 10.1) as possible.[1]

Depending on what one wishes to measure, often most of the network traffic seen by the filter is irrelevant and needs to be discarded. Doing so is termed packet "filtering," and provides the genesis for the name "packet filter."

[1] The timestamps generally are closer to "wire completion" times than "wire arrival" times, since usually the timestamp is generated after the entire packet has been received from the network interface.

Operating systems greatly differ on the amount of filtering provided by their kernels. Some provide only very simple filtering, while others allow quite sophisticated pattern-matching. The difference can be very important for network measurement, because, if a kernel supports only crude filtering, then additional filtering must be performed by the application program accessing the packet filter. This filtering is done at the user-level, which entails copying the potentially very high volume of network traffic from the kernel up to the user-level, merely so almost all of it can be discarded. This copy operation can take considerable processing, and thus can greatly aggravate the problem of packet filter *drops* (§ 10.3.1). For this reason, one generally prefers what is termed a *kernel packet filter*, meaning a packet filter that implements sophisticated filtering at the kernel level, since these can much more rapidly winnow down the packet stream to just those of interest to the application.

We used the `tcpdump` utility for generating our packet traces. `tcpdump` is written in terms of `libpcap`, a library that knows about a great number of packet filters provided by different operating systems [MLJ94]. `libpcap` provides packet filtering using the BSD Packet Filter (BPF; [MJ93]). For operating systems that fail to provide much in terms of kernel-level packet filtering, `libpcap` hauls up all the packets received by the filter and uses the BPF matcher at user-level to filter. For systems that provide BPF-equivalent kernel filtering, `libpcap` knows how to download a filter from the application program (`tcpdump`, in our case) to the kernel, to obtain the benefits of kernel-level filtering.

Of the sites participating in our study, `libpcap` was able to use kernel-level filtering on those systems running the following operating systems: BSDI (`bsdi`, `connix`, `pubnix`, `rain`; `austr`'s separate tracing machine), NetBSD (`panix`; `sintef1`'s separate tracing machine), and Digital OSF/1 (`harv`, `mit`, `ucol` in $\mathcal{N}_2$, `umann`). In addition, some systems had BPF manually added to their kernels (`lbl`, `lbli`, `near`; `sintef2`'s tracing machine). For the remainder, `libpcap` performed packet filtering at the user-level.

In all cases, the filtering used in our study was for packets with the IP addresses for the NPD source and destination hosts in their IP header, and also with both a source port of 7,505 and a destination port of 7,505, as these were the ports used by all of the NPD probe traffic. Note that we did *not* additionally capture ICMP traffic directed to either host, the lack of which subsequently complicated our TCP analysis, since one form of ICMP message ("source quench," cf. § 11.3.3) alters the TCP behavior of a host receiving it.

A final measurement consideration concerning packet filters is the use of a "snapshot length" or *snaplen* to control how much of each packet the filter records. Often, for network analysis all that is required is to record the packet *headers*. Doing so and omitting the packet *contents* can save large amounts of both copying (minimizing processing time and thus decreasing the chance of measurement drops) and storage space. Consequently, for our study we only recorded packet headers. Doing so limited certain types of analysis that require packet contents for full accuracy, such as assessing the prevalence of data corruption. We discuss how we worked around this limitation in § 11.4.2.

10.3 Packet filter errors

It is crucial in any study based on packet filter measurement to consider the forms of measurement errors that packet filters can exhibit. In this section we discuss five types of errors:

drops; additions; resequencing; timing; and misfiltering. For each, we look at the impact of the error on subsequent analysis, and how `tcpanaly` attempts to diagnose the presence of the error.

10.3.1 Drops

The most widely recognized (and often most common) form of packet filter error is the presence of *drops*, in which the trace produced by the filter fails to include all of the packets appearing on the network link that matched the filter pattern. The missing packets are said to have been "dropped." The usual reason that drops occur is that the measuring computer lacks sufficient processing power to keep up with the rate at which packets arrive on the monitored network link. This is particularly a problem for machines requiring "user-level" filtering (§ 10.2), because for them considerable processing can be spent simply moving the stream of monitored packets up to the user level from the kernel level.

Packet filter drops can present serious problems for analyzing network traffic. For example, any analysis of network packet loss rates must be certain not to confuse filter drops with true network drops. Furthermore, filter drops generally occur during periods of peak network load. These are often precisely the times of greatest interest for studying traffic dynamics. If the peaks are "clipped," one can easily underestimate the maximum load the network experiences [FL91].

In general, packets can be dropped at two different places. The *network interface card* that connects the monitoring computer to the network link can run out of buffer memory for storing packets awaiting recording, because the kernel is too busy doing other things to read them quickly enough from the card; or the kernel itself can exhaust its buffer for storing packets awaiting consumption by the user-level tracing utility. Once a packet is successfully transferred to the tracing utility, it is usually immune from further drops (unless it fails to match the filter, naturally), but the time required to subsequently transfer it to permanent storage can result in the user-level utility failing to consume new packets at the same rate that the kernel makes them available, eventually exhausting the kernel's buffer memory.

As discussed in § 10.2, kernel-level packet filters are generally much less susceptible to drops because they pare down the measured packet stream much more rapidly than do user-level packet filters, and hence require much less processing time.

10.3.2 Packet drop reports

The operating system's packet filter interface usually includes a mechanism to query how many packets the kernel dropped, taking care of the second place where packets can be dropped. Network interface cards, on the other hand, often supply only crude signals that packets were dropped (such as a boolean flag indicating simply whether or not any drops have occurred), making it more difficult to evaluate drops occurring due to the kernel being unable to keep up with rate of packet arrivals.

Unfortunately, some operating systems do not report drops (`harv`, `ucol` in $\mathcal{N}_2$, `korea`, `sandia`; most of the Solaris sites). Others report drops when in fact the trace includes all of the connection's packets. This can occur with user-level filtering, because the drop count tallies the number of packets the kernel was unable to deliver to the user level, and it can be the case that none of these belong to the connection of interest. Worse, some report no drops when in fact there were drops. This occurred numerous times for the NetBSD 1.0 machine used to trace `sintef1`'s traffic, and also for some of the Solaris machines that nominally reported drop counts (xor, austr2, nrao in $\mathcal{N}_2$). None of these systems ever reported a drop count other than zero, indicating that the accounting machinery is absent.

Finally, we note that packet drops were quite rare for the systems with kernel-level filtering, though they did sometimes occur.

10.3.3 Inferring filter drops

Because we cannot trust the different packet filters to reliably report drops, tcpanaly employs a number of self-consistency checks to infer their presence. The key in doing so is to be certain not to mistake a genuine network drop for a filter drop, while still detecting filter drops as reliably as possible.

Fortunately, for TCP traffic it is usually possible to discern between a network drop and a filter drop, because TCP is *reliable*. This means that a (correct) TCP implementation will diligently work to repair genuine network drops, while taking no action in response to filter drops (since, in fact, it successfully transmitted the packets).[2] This observation leads to a number of self-consistency checks employed by tcpanaly:

1. Since TCP implementations send data in sequence order, except during retransmission, a "skip ahead" in which new data is sent that does not follow the highest sequence sent so far indicates that the packet filter dropped some earlier-sent data (namely, the data that was indeed in-sequence).

   When applying this check, one must be careful to allow for the possibility of a network "interface drop." That is, the implementation may appear to have skipped ahead because the earlier-sent packet, while successfully transferred from the sending computer to its network interface card, never made it out from the card onto the local network. Interface drops are actually a special case of the "vantage point" problem discussed in § 10.4 below.

   tcpanaly distinguishes between a likely measurement drop and an interface drop by checking to see whether the TCP later retransmits the skipped packet. If so, it most likely did so because the packet did indeed fail to arrive at the receiver, and it was an interface drop. If not, then the packet must have arrived at the receiver (since TCP is reliable), so it was a measurement drop.

2. Even during retransmission, TCPs have a particular order in which they will retransmit data. While this varies between implementations, for those implementations tcpanaly knows about, it can detect whether the TCP deviates from the order, which generally indicates that the packet filter either dropped an incoming ack that altered the retransmission order, or an outgoing data packet that maintained the integrity of the retransmission order.

3. Since a TCP implementation should never send data beyond the upper edge of the *congestion window* (§ 9.2.2), or the inflated congestion window in the case of fast recovery (§ 9.2.7), the presence of such in a trace is much more likely to be due to the packet filter having dropped an ack.

[2] An exception is if a packet is dropped by both the packet filter *and*, later, by the network.

Detecting this inconsistency is difficult because it requires understanding exactly how the particular TCP implementation manages its congestion window. `tcpanaly` does have this knowledge (Chapter 11), however, so it can make this consistency check. This is fortunate, because if the receiver is offering a spacious window, as was the design in $\mathcal{N}_2$ (§ 9.3), then offered window violations (see below) will be very rare, even in the presence of filter drops of acks; but congestion window violations will still flag most instances in which the filter drops an ack.

4. For TCP implementations free of retransmit timer problems (cf. § 11.5.8 and § 11.5.10), the presence of an uncalled-for retransmission usually indicates that the packet filter has dropped one or more acknowledgements that triggered a "fast retransmit" (§ 9.2.7) sequence.

5. A *failure* of a TCP to send data when it apparently was allowed to do so can likewise signal a packet filter drop—the data was actually sent, but the filter failed to record this fact. However, there are many reasons why a TCP might not send data when it appears it can, including not having data available from the sending application; attempting to avoid the "silly window syndrome" ([Cl82]); or the host processor being busy doing something else. Because it can be difficult to determine if one of these is the reason the TCP failed to send, `tcpanaly` does not consider a failure to send as indicative of a measurement drop.

6. A properly functioning TCP will never acknowledge data that has not arrived, nor will it acknowledge data above a sequence "hole" (some earlier data has still not arrived), since TCP acknowledgements are cumulative. Presence of such acknowledgements are thus much more likely to be due to the packet filter having dropped some incoming data packets.

7. Since a TCP implementation should never send data beyond the upper edge of the *offered window* (cf. § 9.2.2), the presence of such in a trace is almost certainly due to the packet filter having dropped an ack (or having resequenced an ack; see § 10.3.6 below).

8. If data is sent before the connection is fully established (§ 9.2.4), this usually indicates that some of the packets in the establishment sequence were dropped by the filter.[3]

Most of these checks can only be conducted from vantage points (§ 10.4) that are local to the point where the bogus traffic is sent (or fails to be sent). If the vantage point is some distance away (in particular, if it is at the opposite end of the connection) then one cannot always distinguish measurement drops from network drops. Consequently, the first five of the checks can only be reliably assessed from traces gathered at the data sender, the sixth can only be reliably assessed at the receiver, and the last two can be reliably assessed at either end, since they should never occur regardless of earlier packets dropped by the network.

For trace pairs, `tcpanaly` makes one further check: if a packet arrives at the receiver that was never sent according to the sender trace, then almost always this indicates a measurement drop at the sender. (Note that this check is complementary to those above, and does not serve to replace them, since it only detects measurement drops at the packet sender.) For further discussion, including why it does not *always* indicate such, see § 10.5 and § 13.2.

[3]The T/TCP extension to TCP allows data to be sent prior to full establishment [Br94, St96]. None of the TCPs in our study used T/TCP, however.

10.3.4 Trace truncation

Related to packet filter drops but slightly different is the problem of trace *truncation*. Truncation occurs when the filter misses the packets belonging to either the beginning of a connection or the end. Both cases are easy to detect because TCP connections are delimited by an exchange of special connection management packets (§ 9.2.4). If this exchange is missing, then the trace has been truncated.

Trace truncation occurs due to a *race* between when the measurement process begins and finishes executing and when the connection itself begins and finishes. `npd_control` attempts to avoid this race by waiting five seconds between requesting that the remote `npd`'s start their measurement processes, and requesting that they proceed with the connection. Similarly, it waits five seconds after the transfer source indicates it has finished before requesting that the remote `npd`'s terminate their measurement processes.

These delays do not always avoid the race, however, particularly because `npd_control`'s trace requests may themselves be held up in the network due to transmission delays, so the transfer request can wind up arriving right on the heels of the measurement request. In addition, the sending application can consider itself as done transmitting its data well before its TCP actually completes the transfer, due to retransmissions that occur after the application has scheduled all of the data for transfer. This mismatch further contributes to the potential for races. A better design would be to use explicit handshaking between the measurement and transfer processes to ensure that the measurements always fully bracket the transfer.

If the beginning of a trace is missing, then `tcpanaly` gives up on trying to analyze it, because it is too difficult to then work out what the congestion window is, and hence to apply the powerful self-consistency check of looking for packets that are sent in violation of the congestion window. If, however, only the end of a trace is missing, then `tcpanaly` can readily analyze the remainder of the trace. When pairing such a truncated trace with the complementary trace made at the other endpoint, `tcpanaly` truncates the trace pair at the last packet appearing in both traces. This occurred in about 6% of the $\mathcal{N}_1$ trace pairs, and 3% of the $\mathcal{N}_2$ pairs. Truncation typically involved only the final few packets of the trace.

A final note: sometimes a trace begins with what is actually leftover traffic from a previous measurement between the same pair of hosts, because at the network level the previous connection's final connection handshake has not yet completed. In principle, this should never happen, because the TCP implementation should not allow the same connection port to be reused while it still maintains state for the earlier instantiation of the connection. In practice, however, we have observed it in several of our traces, sometimes in the traces at both ends of the new connection, indicating it is not simply stale packets left unread from the earlier use of the packet filter but indeed the last wisps of the previous connection. Providing the packets have connection termination flags set (FIN or RST), `tcpanaly` simply ignores them.

10.3.5 Additions

While it is easy to see how packet filters can sometimes fail to record network packets, we might not expect that they can also record *extra* packets! Yet, this does indeed happen, with the IRIX 5.2 and 5.3 packet filters. Figure 10.1 shows part of a sequence plot exhibiting this problem. Here, the ack just before time $T = 11.175$ has liberated five packets.

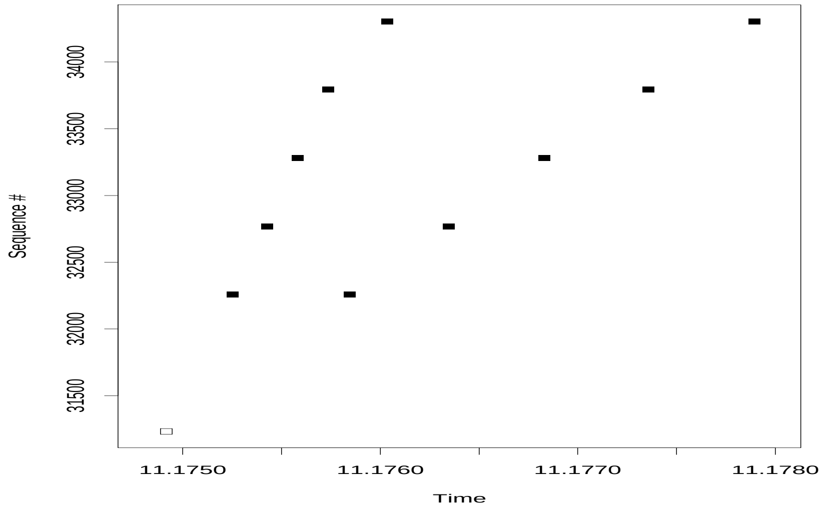

Figure 10.1: Packet filter replication

Each outgoing data packet appears twice. The slope (i.e., data rate, per § 9.2.4) of the two sets of packets is telling. The first corresponds to a data rate of over 2.5 MB/sec, while the second is almost exactly 1 MB/sec. This latter agrees closely with the data rate of an Ethernet, and indeed the host generating the traffic is connected to an Ethernet. Thus, surprisingly, the first set of packets appear to have bogus timing while the second set appears to be accurate! Furthermore, the two sets are indeed intertwined, that is, the second occurrence of sequence number 32,257 appears in the trace before the first occurrence of 34,305.

This puzzling picture all makes sense given the following explanation. This trace was made running the packet filter on the same machine as that generating the network traffic, and the operating system is copying outgoing packets to the filter *twice*, the first time when the packets are scheduled to be sent out onto the local Ethernet, and the second time when they actually depart onto the Ethernet. The 2.5 MB/sec corresponds to how fast the operating system is sourcing the traffic, while the 1 MB/sec reflects the local rate limit of the Ethernet link speed.

About 2,000 of the traces in our study have duplications of this sort. Clearly such duplicates can complicate or skew our analysis. For example the computation of packet loss rates had better not conclude that when the sender's filter reports 400 packets sent but only 200 arrive that the loss rate was 50%! On the other hand, we would rather not discard all these traces for our subsequent analysis, so `tcpanaly` needs to cope with the duplication. Yet, we cannot blithely discard the second copy of each packet, because we might in the process discard a packet truly replicated by the network, an event that would be very interesting to detect (this does indeed happen, see § 13.2).

For our measurement purposes, the second copy is actually preferred to the first, since it is closer to the true wire time (§ 10.1). Unfortunately, while in many traces every single packet sent by the host (data packets, if the IRIX host was the sender, acks if the receiver) appeared twice, in some of the traces a second copy was occasionally missing. (We know the omission was not due to an interface drop, per § 10.3.3, because it was never retransmitted.) Furthermore, in some traces

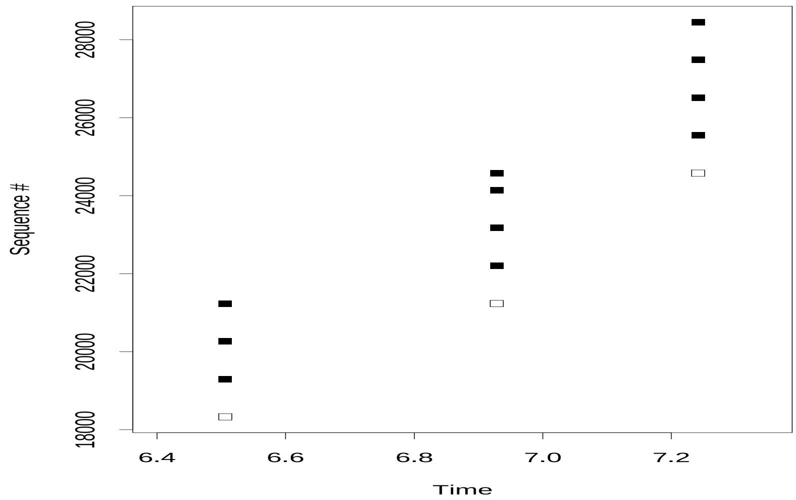

Figure 10.2: Packet filter resequencing

the duplication starts midway through the trace, rather than permeating the entire trace. For these reasons, `tcpanaly` copes with measurement duplicates by discarding the later copy.

It discriminates between a measurement duplicate and a true retransmission as follows. First, it checks whether the “id” field in the packets' IP headers match. This field is used by IP for fragmentation purposes, which we need not delve into here. However, one salient property of the field is that in general it is incremented for each new IP packet that a host sends. Consequently, different TCP packets will usually have different IP “id” fields in their IP headers. If the “id” fields agree, it then checks whether the sequence number fields match, and, for data packets, also whether the second copy was sent less than one quarter of the minimum observed round-trip time (RTT) after the first copy. If the endpoint TCP is known to reuse the IP “id” field when retransmitting a data packet (of the TCPs in our study, only Linux 1.0 does this), then data packets are never considered candidates for measurement duplication, since it is too easy to confuse a true retransmission with a measurement duplicate (especially since Linux 1.0 retransmits too early, per § 11.5.8, and hence would pass the RTT test). Fortunately, the packet filter used to trace the sole Linux host (`korea`) does not appear to suffer from measurement duplications, so we do not lose any calibration by doing so.

10.3.6 Resequencing

Another form of packet filter error is what we term “resequencing,” in which the packet filter alters the ordering of the packets so that it no longer reflects events as they actually occurred in the network. Figure 10.2 shows a portion of a sender trace in which this occurred. At first glance the plot appears normal: acks are occasionally arriving and as they do, the window slides several packets' worth and newly liberated packets depart shortly afterwards.

Figure 10.3, however, shows a blow-up of the central tower from the previous figure.

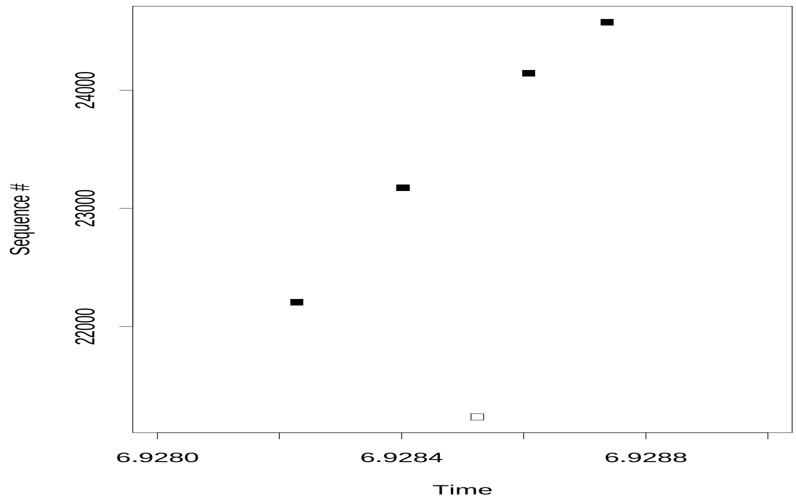

Figure 10.3: Enlargement of resequencing event in previous figure

We see that the packet filter has recorded timestamps for the packets such that the first two data packets are sequenced as having departed *before* the acknowledgement arrived. Since the congestion window would not have permitted their earlier departure, and there was a lengthy lull as shown in Figure 10.2 before their departure but only 100's of microseconds between their alleged departure and the arrival of their liberating ack, it is clear that the filter has misrepresented the true sequence of events. The problem here is not a clock adjustment (§ 12.6), since the packets appear in the shown chronological order in the trace file (and also because this problem is much more common than we find clock adjustments to be). This problem occurs quite frequently for the Solaris 2.3 and 2.4 packet filters, plaguing about 20% of the traces they record. It almost never occurs for any of the other packet filters.

Most likely the resequencing occurs because the packets are being recorded by a packet filter running on the same host as is generating the traffic. We speculate that the Solaris packet filter has two code paths by which packets are copied to the packet filter for recording, one corresponding to incoming packets and one corresponding to outbound ones. If the outbound path is appreciably faster than the inbound one; if copies of packets can queue separately in both paths waiting for the filter to record them; and if packets are only timestamped when the filter processes them, then the resequencing makes sense.

Unfortunately, resequencing presents a considerable analysis headache, as it destroys any ready assessment of cause-and-effect. It also means that the packet timestamps have large margins of error, with a bias towards overestimating how long it takes acks to arrive compared to how quickly data packets are sent out. Thus, `tcpanaly` needs to detect this problem so that it knows not to trust the sequence of events reported by the packet filter. It cannot really correct the problem since we do not know when the ack truly arrived, so we do not have a sound timestamp to assign to it. Instead, it flags the trace as lacking accurate timing and causality information.

To detect resequencing for traces recorded at the data sender, `tcpanaly` keeps track of *stall* packets. These are data packets that are not timeouts (i.e., not retransmissions of the lowest unacknowledged sequence number) and that have been sent after a lengthy lull in network activity. `tcpanaly` considers a lull to have occurred if at least 25 msec has elapsed since the previous data packet was sent, or, if an ack arrived after the last data packet was sent, then at least 50 msec has elapsed since the ack arrived.

If an ack follows a stall packet by less than $\max(1 \text{ msec}, R_s)$, where R_s is the clock resolution of the packet filter's timestamps (§ 12.1), and if the ack acknowledged a sequence number below that of the stall packet (so, transmitting the stall packet after seeing the ack would have made sense), then `tcpanaly` flags the ack as reflecting a resequencing event.

As mentioned above, the stall packet technique only works for traces recorded at the data sender. `tcpanaly` uses a similar technique for receiver traces, namely looking for acknowledgements for data as-yet-unreceived but arriving shortly after.

Note that there is some overlap between detecting measurement drops and resequencing events. For example, an observation of data sent beyond the congestion window could be due to the corresponding ack having been dropped, or due to resequencing, with the ack arriving shortly after the window violation. `tcpanaly` may occasionally mistake one for the other, based on the timing of the packets arriving after the event. For our purposes, this potential misattributing of the exact type of packet filter error is unimportant. The key requirement is simply that `tcpanaly` recognize the trace as untrustworthy.

Finally, the Solaris filters are particularly apt to resequence an ack for a FIN packet terminating the connection, presumably because the associated code paths are particularly asymmetric in terms of processing time. Since for our analysis this reordering is essentially benign, because it comes at the very end of the connection, `tcpanaly` does not consider traces that *only* exhibit resequencing for a FIN packet as untrustworthy. The statistic above of 20% of the Solaris traces having resequencing problems does not include those with only resequenced FIN packets.

10.3.7 Timing

Another type of packet filter error concerns the accuracy of the timestamp recorded for each packet: how close is the timestamp to the true wire time? In Chapter 12 we look at the issue of calibrating these timestamps in detail. Most of the consistency checks we develop in that Chapter rely on comparing *pairs* of packet timings, those corresponding to when the sender's packet filter recorded the packet's departure, and those of when the receiver's packet filter recorded the packet's arrival. These tests prove quite powerful at detecting different clock problems, but require extensive analysis. In this section we confine ourselves to a simple test `tcpanaly` performs to check the validity of a single trace's timestamps, namely ensuring that they never decrease.

We refer to a decrease in the timestamp values as "time travel." One might think that time travel would never occur, and checking for it would be a waste of effort, but, surprisingly, it does happen! We recorded four instances in $\mathcal{N}_1$, all involving `connix`'s clock, and 538 instances (!) in $\mathcal{N}_2$, 498 involving `sintef1`'s clock (that is, the clock of `sintef1`'s NetBSD 1.0 tracing machine) and 40 involving `panix`'s clock (also a NetBSD 1.0 machine).

Figure 10.4 gives an example of how a sequence plot exhibiting time travel appears. If we add lines to the plot showing the order of the packets as they appear in the trace file (Figure 10.5), then we see a sharp backward jump from time $T = 3.6$ sec to $T = 3.05$ sec.

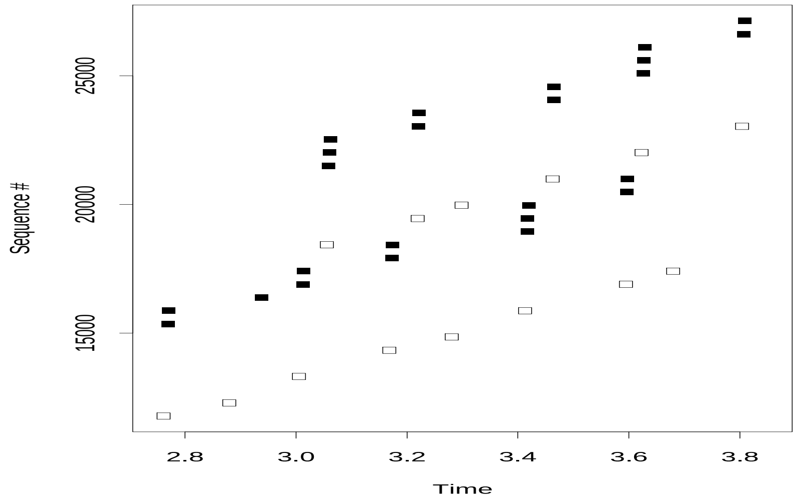

Figure 10.4: Example of “time travel”

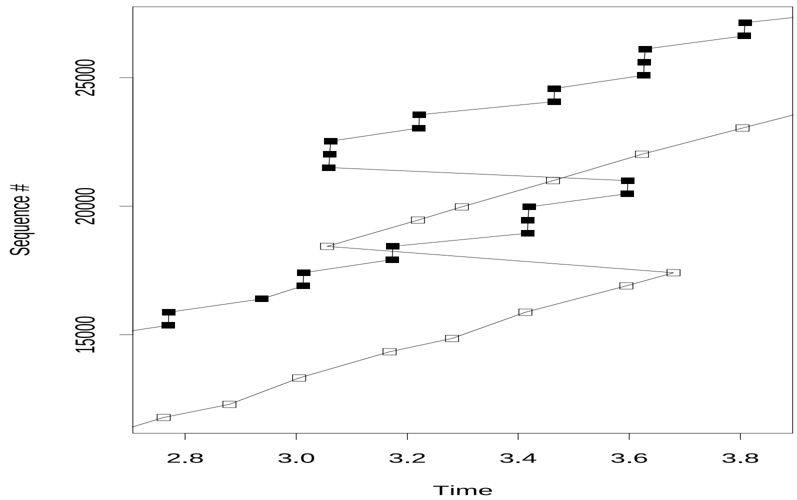

Figure 10.5: Same plot, with lines showing the ordering of the packets in the trace file

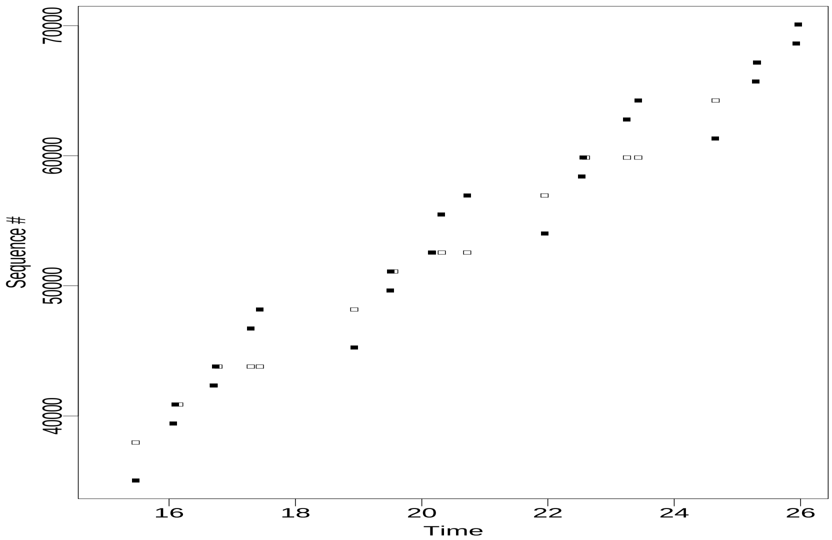

Figure 10.6: Receiver sequence plot showing a forward clock adjustment, undetectable to the eye

Time travel has a simple explanation: it reflects the local clock being set backwards. It can occur frequently, as with `sintef1`, if the clock is periodically synchronized with an external source by setting it to the source's reading, and if the clock tends to run fast. Another form of time travel, considerably more difficult to detect, are *forward* adjustments. Figure 10.6 shows a receiver sequence plot spanning an 11 second period during which the receiver's clock was artificially advanced by an additional 400 msec. To the eye, however, this adjustment is completely hidden.

We look at detecting clock adjustments in greater detail in § 12.6.

10.3.8 Misfiltering

The last type of packet filter error we look at is "misfiltering," meaning that the filter incorrectly executes its pattern matching and either rejects packets it should accept, or accepts packets it should reject. The first of these is similar to a measurement drop, though systematic in nature. The second can in principle be detected by checking the accepted packets to make sure they do indeed match the desired filter. To do this check properly requires a separate implementation of the filtering mechanism than that used by `libpcap` , since otherwise one would expect the same erroneous match to occur again.

`tcpanaly` does not include a full, separate matching mechanism, but it does perform two consistency checks in this regard. First, it checks to make sure that the IP header of each packet indicates a TCP packet. This check never failed. Second, it partitions all the TCP packets it inspects into individual TCP connections based on their host and port numbers, and analyzes each resulting connection separately. In no case did it find more than one connection in a trace, though it occasionally found remnants of an earlier incarnation of the same connection, as discussed in § 10.3.4.

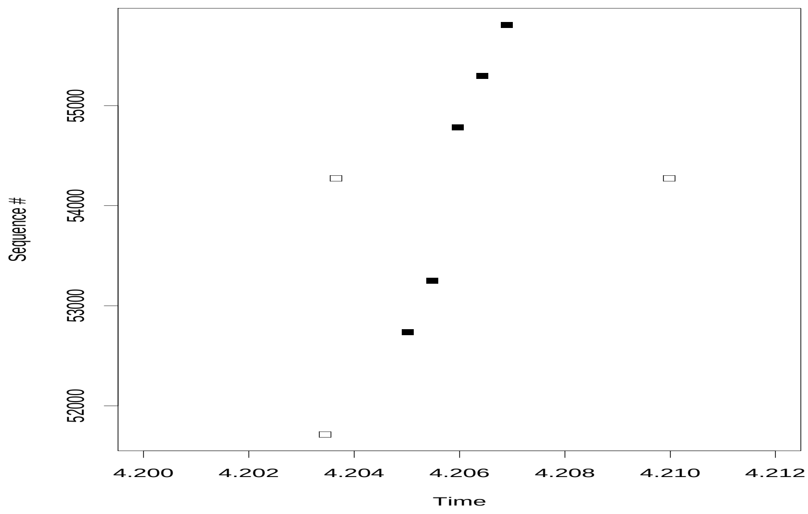

Figure 10.7: Example of an ambiguity caused by the packet filter's vantage point

10.4 Packet filter "vantage point"

While not a measurement error per se, another difficulty in calibrating packet filter measurements arises from complications due to the packet filter's location in the network. We term this its "vantage point." For example, if the packet filter records data packets as they arrive at the receiver, ambiguities arise in trying to determine whether any arrival anomalies observed are due to the network perturbing the packets, or because they were sent by the source in an unusual fashion. Suppose two packets arrive out of sequence order; it is not always apparent whether the network reordered them, or if the packet with the lower sequence number was dropped by the network and the sender has already retransmitted it.

Vantage point effects can be significantly more subtle than in this example, however. They are most insidious when the filter *appears* as if it were located directly at one of the TCP endpoints, and only *occasionally* does its separate location alter the traffic perspective it records.

Figure 10.7 gives an example. The sequence plot is from a packet filter recording traffic at the sending endpoint. A little after time $T = 4.203$, an ack arrives for a sequence number a bit below 52,000. Very shortly afterwards, at time $T = 4.204$, an ack arrives for a sequence number above 54,000. Then at time $T = 4.205$, the sender transmits two packets with sequence numbers below 54,000. If the sequence plot truly reflected the traffic as seen by the TCP endpoint, then the TCP never should have sent these packets, since it had already received an acknowledgement for the corresponding data! As can be seen from the plot, shortly after sending these two packets the endpoint then *does* process the second ack, and sends new, unacknowledged data.

The key point here is that neither the packet filter nor the endpoint TCP are behaving erroneously. The problem is simply that the packet filter's vantage point is not exactly the same as that of the endpoint TCPs, and the problem is exacerbated by the vantage point being very *close* to that of the TCPs, as this then encourages assumptions that the two are indeed the same.

Vantage-point problems can be reduced by running the packet filter on the same machine as the TCP endpoint, although this introduces other measurement problems due to competition for the machine's processing power. This step does not, however, eliminate the problem, because cause and effect can still be obscured if the TCP takes a long time to react to any particular input. For example, when new data arrives, many TCP receivers only acknowledge it after the receiving application process has consumed at least two packets' worth of data, which can take considerable time after the network arrival of the data.

In order to correctly analyze TCP traffic, `tcpanaly` must be able to cope with vantage-point problems. This means that in general it is insufficient for analysis purposes to only remember the most recently received packet. Dealing with vantage-point problems considerably complicates `tcpanaly`'s design, but the result is much more robust analysis. We discuss how we do so in § 11.3.1.

10.5 Pairing packet departures and arrivals

The last packet filter issue we look at is how to take two trace files, $\mathcal{T}_s$ recorded at TCP endpoint s, and $\mathcal{T}_r$ recorded at endpoint r, and from them synthesize a *trace pair* that matches packet departures from s and r with their corresponding arrivals at r and s.

The basic approach we use for trace pairing comes from the observation that each packet has two "fairly" unique fields in its header, its sequence number (or the sequence number it is acknowledging, if an ack) and its IP "id" field (§ 10.3.5). If these fields were indeed unique, then trace pairing would be easy, since the fields would allow unambiguous determination of which packets in $\mathcal{T}_s$ correspond to which in $\mathcal{T}_r$. Those without a corresponding packet were either dropped by the network (if present only in the trace local to their sender), or by the packet filter (if present only in the trace local to their receiver).

The pairing problem lies in the fact that the sequence number and IP id fields are not actually unique. Sequence numbers can reappear in different packets due to retransmissions or duplicate acks (§ 9.2.7). Most TCPs only reuse the IP id field when its 16 bit counter wraps around, but one system in our study (Linux 1.0) reuses the IP id field as well as the sequence number when retransmitting.[4]

`tcpanaly` deals with these problems as follows. Suppose we wish to pair packets sent by s with their arrivals at r (everything works the same when pairing in the other direction). `tcpanaly` first goes through $\mathcal{T}_s$ and for each packet p sent by s it computes a key, K_p, comprised of the triple of the packet's IP id field and its data and acknowledgement sequence numbers.

Using K_p as an index into a table P_s, we check to see whether we have already seen a packet with the same key. If not, the packet is added to P_s and `tcpanaly` proceeds to the next packet. If another packet with the same key has been seen, then we check whether the packets are *identical*, meaning they have the same TCP header flags, data and acknowledgement sequence numbers, length, and offered window.[5] If any of these differ, then `tcpanaly` flags that a serious

[4] This is a reasonable performance decision, and explicitly allowed in § 4.2.2.15 of [Br89]. If the sending TCP keeps its unacknowledged data in the form of fully assembled packets, then for retransmission all it needs to do is copy the packet out to the network interface. The reuse of the IP id field does not present an integrity problem since what is being retransmitted is a verbatim copy of what was sent earlier.

[5] In principle, for data packets we should also check whether the data contents agree. Since, however, the traces in our

analysis error has occurred, since the assumption that the key suffices as a unique identifier has proven incorrect. For all of the traces in $\mathcal{N}_1$ and $\mathcal{N}_2$, this never occurred. We next check whether the packet filter in use is known to create spurious measurement duplications (§ 10.3.5). If so, then `tcpanaly` discards the later copy of p as a measurement artifact. Otherwise, if the sending TCP is known to reuse IP id fields (Linux 1.0, for our study), then the additional packet is entered as a second instance of K_p in P_s. If none of these considerations hold, then `tcpanaly` flags that a packet has apparently been replicated at the sender (these are analyzed further in § 13.2), and does not construct a trace pair for $\mathcal{T}_s$ and $\mathcal{T}_r$ because it cannot do so reliably.

`tcpanaly` next goes through each packet p arriving from s in $\mathcal{T}_r$, again computing its key K_p. If K_p does not appear in P_s (the table of packets sent by s, indexed by their keys), then either p's transmission was dropped by the packet filter at the sender; or $\mathcal{T}_s$ was truncated (§ 10.3.4); or the network garbled p in transmission so that its sequence number or IP id field has changed (analyzed further in § 13.3). If K_p appears in P_s, then p is checked against the $\mathcal{T}_s$ version of the packet to see if they are identical. If not, `tcpanaly` flags that the packet was corrupted by the network (again analyzed in § 13.3). If the two copies agree, then we proceed as follows:

1. If K_p appears exactly once in P_s, and has not yet been paired with an arrival in $\mathcal{T}_r$, then it is paired with p in $\mathcal{T}_r$.

2. If K_p appears exactly once in P_s but has already been paired in $\mathcal{T}_r$ with an arrival p', then p is flagged as a *replication* of p'. Replications are further analyzed in § 13.2.

3. If K_p appears m times in P_s for $m > 1$, then we term the pairing as *ambiguous*. To resolve ambiguous pairings, `tcpanaly` first computes n, how many times the same key occurs in $\mathcal{T}_r$. If $n = m$, then `tcpanaly` assumes that each packet arrived in order and pairs them in order of occurrence. If $n > m$, then we presume a measurement drop occurred in $\mathcal{T}_s$ (it could also have been a packet replication, but that is much less likely). If $n < m$, then some of the original instances of the packet were dropped by the network. In this case, we attempt to pair each departure with the arrival that has the smallest difference in timestamps, provided this difference is no smaller than the smallest such difference for all of the unambiguous pairings. If this pairing results in a single packet departure matching two different packet arrivals, then we abandon the attempt to construct a trace pair, since we cannot construct a plausible set of pairings.

If `tcpanaly` was not able to unambiguously pair the packets in the traces, or if the traces included corrupted packets (which may be erroneously paired), then `tcpanaly` does not construct a "trace pair" and skips any subsequent analysis that requires a trace pair. The latter problem (corrupted packets) was extremely rare, but the former problem is more common: ambiguities due to Linux 1.0 TCP reusing IP id fields rendered 65% (15 out of 23) of the traces with a Linux 1.0 sender unpairable. Consequently, we were unable to perform sound analysis of the trans-Pacific path from Korea to the other sites, especially because the Linux 1.0 traces that did *not* suffer ambiguities were those with especially low levels of retransmission, so analyzing just them would result in a biased assessment of the levels of retransmission and loss along the path.

---

study only include packet *headers*, and not data contents, we could not perform this test.

Finally, if `tcpanaly` removes relative skew from the receiver's clock (§ 12.7.9), it then recomputes the packet pairings, in case any of the ambiguous matches are changed by the altered receiver timestamps.

Chapter 11

Analyzing TCP Behavior

We discussed earlier how one of the main drawbacks to using TCP traffic for our network "probes" is the often quite complex behavior of the TCP endpoints (§ 9.1.2). We argued that the resulting fine-resolution probing outweighs this disadvantage, because the disadvantage can be overcome by careful analysis of the packet arrivals and departures in order to remove those aspects of the traffic behavior due to the TCP endpoints themselves. In this chapter we discuss how `tcpanaly` performs this analysis. In addition, the process of removing the TCP effects reveals a wealth of interesting detail about how different TCP implementations behave. We find a tremendous range both in their performance and in their congestion-avoidance behavior, the latter playing a critical role in the Internet's global stability.

In addition, a solid understanding of each TCP endpoint's exact behavior enables us to distinguish between packet filter errors and bona fide network anomalies. For example, if multiple copies of a single data packet arrive at the TCP receiving endpoint, we can look to see whether the receiver generates an ack for each one. If it does, then the extra copies are bona fide and not measurement duplications (§ 10.3.5). If not, then *if the TCP endpoint is known to correctly generate acks when it receives redundant packets*, we can conclude that a measurement error occurred, and the packets did not really exist. If the TCP is known to not generate acks in this situation, then we cannot tell, and look for a separate indication of whether the packets were indeed real (for example, whether they have different TTL's). Thus, thoroughly understanding TCP behavior provides an invaluable self-consistency check on the soundness of our measurement (§ 9.1.4).

11.1 Analysis strategy

As its name suggests, we began writing `tcpanaly` with the goal of analyzing TCP behavior. Only later did we realize that, in the process of doing so, it develops many of the data structures also needed to analyze network dynamics.

Our original goal was for the program to work in *one pass* over the packet trace by recognizing *generic* TCP actions. The goal of executing only one pass stemmed from hoping `tcpanaly` might later evolve into a tool that could watch an Internet link in real-time and detect misbehaving TCP sessions on the link. Designing the program in terms of generic TCP actions such as "timeout" and "fast retransmission" would then enable it to work for any TCP implementation without needing to know details of the implementation.

After considerable effort, we were forced to abandon both of these goals. One-pass analysis immediately proved difficult due to *vantage point* issues (§ 10.4), in which it was often hard to tell whether a TCP's actions were due to the most recently received packet, or one received in the more distant past. Attempts to surmount this problem by using k-packet look-ahead for small k proved clumsy, and finally foundered when we realized that one basic property `tcpanaly` needs to determine concerning a TCP implementation is only truly apparent upon inspecting an entire connection, namely whether the implementation has a "sender window" (§ 11.3.2). Since sender windows are common, in order to infer them soundly we decided to allow `tcpanaly` to inspect the entire packet trace before making decisions as to how the TCP behaved. Doing so immediately simplified other types of analysis, too.

We abandoned the goal of recognizing generic TCP actions as the wide variation in TCP behavior became apparent. For example, as related below, the Solaris and Linux TCP implementations in our study often retransmit data packets much too early, before the original packet had a chance to arrive at the destination and be acknowledged, and the Linux implementation furthermore retransmits entire flights of packets rather than just one packet at a time. Neither of these behaviors fit a generic TCP action (except "broken retransmission"!), and they are very easily confused with legitimate retransmissions due to "fast retransmission" (§ 9.2.7). Similarly, the fashions in which different implementations open the congestion window differ in subtle ways, with the result that sometimes it can be extremely difficult to tell why a TCP failed to send new data when an ack arrives: is it because its window has not opened another full packet, or because the TCP is simply running slow and has not had time to do so? Both occur quite frequently.

Thus, we are left with a much less flexible but more robust design for `tcpanaly`: it makes two passes over the packet trace, it uses k-packet look-ahead and look-behind to resolve ambiguities, and, instead of characterizing the TCP behavior in terms of generic actions, we must settle for it having coded into it intimate knowledge of the idiosyncrasies of 17 different TCP implementations. Furthermore, when confronted with a trace generated by a new implementation not already coded into it, it can only fruitfully analyze the trace if the new implementation behaves identically to one of the 17 it already knows about, or if the extra effort is made to add knowledge of the new implementation to the program. To ameliorate this shortcoming, the program is capable of automatically running all known implementations against a given trace to determine those with which the trace appears in full accord.

All told, `tcpanaly` is about 14,000 lines of C++ code. Of these, about 7,500 analyze TCP behavior (1,400 concerning individual implementation behavior), 5,000 analyze network behavior, and the remainder perform utility functions. The use of C++ is particularly beneficial for expressing the behavior of one TCP implementation in terms of its differences from that of another implementation. In particular, `tcpanaly` includes a "Reno" implementation that captures the main features of the BSD Reno TCP release, from which most of the TCPs in our study were derived. This allows these derivatives to be expressed succinctly, in terms of just how they differ from "generic" Reno. A widespread Reno variant known as Net/3 is discussed in detail in [WS95].

Table XV summarizes the different TCP implementations known to `tcpanaly`. The first column gives the name of the implementation and the version numbers present among the implementations in our study. The second column lists the sites running each version, separated by ';'s. Sites listed with a subscript of $_1$ or $_2$ participated in both $\mathcal{N}_1$ and $\mathcal{N}_2$, but only used the given implementation during the first or second, respectively.

Implementation	Sites	Notes
BSDI 1.1; 2.0; 2.1α	$\texttt{bsdi}_1$, `connix`, $\texttt{pubnix}_1$, $\texttt{austr}_2$; $\texttt{pubnix}_2$, `rain`; $\texttt{bsdi}_2$	Reno-derived. BSDI 2.1α not a public release.
Digital OSF/1	`harv`, `mit`, $\texttt{ucol}_2$, `umann`	Reno-derived. No differences observed between versions 1.3a, 2.0, 3.0, 3.2.
HP/UX 9.05; 10.00	`sintef2`; `sintef1`	Reno-derived.
IRIX 4.0.1; 4.0.5f; 5.1; 5.2; 5.3; 6.2α	`oce`; `sandia`; $\texttt{bnl}_1$; $\texttt{sdsc}_1$; `adv`, $\texttt{bnl}_2$; $\texttt{sdsc}_2$	Reno-derived. No differences observed between 4.0.1 and 4.0.5f, nor between 5.3 and 6.2α. 6.2α not a public release.
Linux 1.0	`korea`	Implemented independently from BSD.
NetBSD 1.0	`panix`	Reno-derived.
Solaris 2.3; 2.4	$\texttt{inria}_1$, `sri`, $\texttt{ucl}_1$, `ustutt`, `wustl`, `xor`; `austr2`, $\texttt{inria}_2$, `mid`, $\texttt{nrao}_2$, $\texttt{ucl}_2$	Implemented independently from BSD. Very minor differences between 2.3 and 2.4.
SunOS 4.1	$\texttt{austr}_1$, $\texttt{lbl}_1$, `near`, $\texttt{nrao}_1$, `ncar`, `ucla`, $\texttt{ucol}_1$, `ukc`, `umont`, `unij`, `usc`	Tahoe-derived. 4.1.3 and 4.1.4 appear identical.
VJ_1; VJ_2	$\texttt{lbl}_2$; `lbli`	Experimental Reno-variants developed by Van Jacobson. Neither a public release.

Table XV: TCP Implementations known to `tcpanaly`

All but Linux 1.0, Solaris 2.3 and 2.4, and SunOS 4.1 are some variant of "Reno." SunOS 4.1 is a variant of "Tahoe," a Reno predecessor, while the Linux and Solaris implementations were written independently of Reno and of each other.

11.2 Checking packet and measurement integrity

One often assumes that a trace produced by a packet filter sited at a TCP endpoint does indeed reflect the packets sent and received by the endpoint. In Chapter 10 we discussed some ways in which this assumption can be violated. Here we look at additional consistency checks `tcpanaly` uses to avoid misassumptions.

Among all the traces in the study, we never observed any of the following:

1. Options present in the IP header.
2. A packet sent with more data than the MSS (§ 9.2.2).
3. A TCP connection-establishment option present in a non-establishment (non-SYN) packet.
4. An establishment (SYN) packet appearing after completion of the connection establishment handshake.
5. Illegal or unknown TCP header options.
6. SYN packets with other flags set. (We have seen this in other Internet traffic traces.)
7. IP fragments with the "Don't Fragment" bit set.
8. Non-TCP traffic (§ 10.3.8).
9. Illegal IP header lengths.
10. TCP "simultaneous open" [St94].

We did, however, occasionally observe the following:

1. Time travel (§ 10.3.7).
2. IP header checksum errors. `tcpanaly` verifies that the computed checksum for the IP header matches that in the header. This test never failed in $\mathcal{N}_1$, but failed 17 times in $\mathcal{N}_2$. All 17 occurrences were between the same pair of hosts (`connix` and `nrao`), and all of the IP headers flagged with errors suffered from corrupted (too large) length fields. These circumstances strongly suggest a faulty link somewhere in the middle of the path between `connix` and `nrao`, presumably the final hop in the path because otherwise an intermediary router should have discarded the packets. The corrupted length fields are consistent with CSLIP errors, as discussed in § 13.3.
3. TCP checksum errors. Packet traces generated by `tcpdump` have a *snaplen* that limits the amount of data recorded for each packet to the first n bytes (§ 10.2). The *snaplen* can greatly reduce the volume of data the packet filter must copy and record. But it means that, for TCP

data packets longer than the *snaplen*, `tcpanaly` cannot compute the corresponding checksum and compare it to the value in the TCP header. `tcpanaly` can, however, checksum pure ack packets, since they completely fit within the *snaplen* used in our experiment. It does so unless the header checksum is exactly $2^{16} - 1$, because we observed that some IRIX packet filters frequently record outgoing packets with that value in the checksum field rather than the true checksum. We suspect that this occurs because the packet has been copied to the packet filter for recording prior to the checksum computation, because the computation is done later by the network interface hardware.

Checksum errors in pure acks detected by this means are quite rare: 1 instance in $\mathcal{N}_1$ and 26 instances in $\mathcal{N}_2$. All but one of these latter involved `lbli`, which, as discussed in § 13.3, suffered from an atypically strong predilection for checksum errors. We discuss how to infer checksum errors in data packets below in § 11.4.2, and in § 13.3 we find that these are much more common than errors in pure acks.

An interesting question is whether `tcpanaly` ever falsely identified TCP checksum errors because a packet filter recorded a corrupted copy of a packet (while the receiving TCP received an uncorrupted copy). However, with corrupted packets removed from the analysis, `tcpanaly` still found that the receiving TCP behaved as expected, indicating that the packets were indeed corrupted and ignored by the TCP.

4. Truncated packets. These are packets that, according to the IP header, have a length of n bytes, but in fact, as delivered by the local link, had a length of only $k < n$ bytes. There were 4 instances in $\mathcal{N}_1$, 348 in $\mathcal{N}_2$. The latter involved 8 different receiving hosts.

5. Illegal TCP header length. This is a TCP header length field that indicates a length less than the allowed minimum of 20 bytes. It indicates a corrupted packet. We observed only two instances, both in $\mathcal{N}_2$.

6. IP fragments. We observed 5 instances in $\mathcal{N}_2$ (none in $\mathcal{N}_1$) of a packet arriving with an IP header indicating it was the beginning of a fragment. (The packet filter pattern we used precluded capture of any fragment portions other than the initial fragment.) Upon inspection, however, all of these were not actually bona fide IP fragments, but instead a repeated pattern of packet corruption: the packet was enlarged in flight from carrying to 512 bytes of data to purportedly carrying either 980 bytes or 1460 bytes. Both of the latter are popular MTU values (§ 9.2.2). Their presence suggests a SLIP compression error, as discussed in more detail in § 13.3.

It is important for `tcpanaly` to detect corrupted packets, because they are discarded by the receiving TCP rather than processed by it. If `tcpanaly` misses such a corruption, then it can erroneously infer that the TCP failed to act when it should have. Thus, we believe the effort entailed in detecting the sometimes quite rare errors reported above is well worth while, especially because *a priori* we have no solid reason for assuming they are indeed rare.

11.3 Sender analysis

In this section we discuss how `tcpanaly` analyzes a TCP implementation's *sender* behavior: that is, the details of how the TCP reliably transmits data to the other endpoint. The sender behavior includes the TCP's *congestion* behavior, too: how the TCP responds to signals of network stress. Proper congestion behavior (§ 9.2.6) is crucial to assure the network's stability. The next main section (§ 11.4) then discusses how `tcpanaly` analyzes *receiver* behavior: when and how a TCP implementation chooses to acknowledge the data it receives. In general a TCP both sends and receives data. `tcpanaly`, however, only accurately analyzes unidirectional TCP transfers. Extending it to cope with bidirectional transfers would not be a major undertaking, but was not needed for our study and so was left for future work.

11.3.1 Data liberations

To accurately deduce the sender behavior of a TCP from a record of its traffic requires a packet trace captured from a vantage point (§ 10.4) at or near the TCP. If the vantage point is distant from the sender (especially at the receiver), `tcpanaly` has no reliable means of distinguishing between measurement drops, anomalous TCP behavior, and true network drops. It also cannot distinguish lengthy latencies from the vantage point to the sending TCP's location, and a TCP that is simply slow to respond to the acknowledgements it receives.

As discussed in § 10.4, even a vantage point quite close to the sender still can result in timing ambiguities. We accommodate this difficulty by introducing the notion of data *liberations*. Whenever an acknowledgement arrives, `tcpanaly` determines how it updates the offered window and the congestion window (§ 9.2.2). If the new window values permit the TCP to send another packet(s), `tcpanaly` then notes which packets it should send. We term each such newly-allowed data packet a "liberation."

By noting the time at which new acks created liberations, `tcpanaly` can keep a list of all pending liberations and, when the TCP finally does send more data packets, determine their corresponding liberations. The difference in time between when the data packet was sent and when it was liberated then defines the *response time* of the TCP for that ack. Unusually large response times often indicate that `tcpanaly` has an incomplete understanding of the TCP's behavior, and that the delay was really because the purported "liberating" ack did not in fact liberate the data finally sent. It flags such instances so they can be inspected manually to determine the origins of the apparently imperfect behavior.

Sometimes `tcpanaly` will observe a packet being sent that has no corresponding liberation. We term this a "window violation," because it indicates that the TCP exceeded either the congestion window or the offered window. In principle, `tcpanaly` should never observe a window violation if it correctly understands the operation of the sending TCP. Violations can still occur, however, if the trace suffers from measurement drops, or if the understanding of the TCP is incomplete or inaccurate.

`tcpanaly` can use statistics of response times (minimum value, mean value) to compare how closely different candidate TCP implementations match a particular trace. If a candidate implementation is indeed correct, then its response times will usually be quite small. If the candidate is incorrect, then the liberations `tcpanaly` computes for the implementation will not correspond to the times at which packets were actually liberated. The difference leads to either increased response times or window violations. Thus, depending on the relative response times and presence or lack of window violations, `tcpanaly` sorts candidate implementations into those that are close fits, those that are imperfect fits, and those that are clearly incorrect fits (for example, if it observes window violations). These last can also occur due to measurement drops, though, in that case, `tcpanaly` usually rejects all of the candidate implementations.

The process of coding into `tcpanaly` a new TCP implementation likewise relies on minimizing response time statistics and eliminating window violations. For example, we might begin by deriving a C++ class to encapsulate the new implementation I in terms of differences from the generic Reno class. We then run `tcpanaly` against a trace of I's sender behavior. If `tcpanaly` flags a window violation, we manually inspect the trace at the location of the violation (usually using a sequence plot; § 9.2.4) and attempt to determine a rule for how I differs from Reno at that point. Once all window violations have been eliminated, we then turn to the response time statistics. If the maximum response time is quite large, it usually indicates a congestion window that has opened up more slowly than expected, or a failure to take advantage of fast retransmit. Again, a sequence plot greatly aids in diagnosing the behavior. After identifying and codifying I's behavior, we test to assure that this has indeed lowered the response time. If so, we proceed to the next instance of a large response time, or the next trace of I's behavior. If the new TCP is close to one of the existing ones, this is a fairly quick process.

In addition to summarizing the amount of data newly allowed and when it became liberated, liberations include a set of zero or more attributes that describe how `tcpanaly` should interpret a failure of the TCP to promptly use the liberation:

Blameless due to SWS (Silly Window Syndrome) avoidance TCPs are supposed to implement the SWS avoidance algorithm described in [Cl82, St94], which in some cases prevents them from sending data that they otherwise could.

This attribute indicates that the TCP should not be blamed for failing to utilize the liberation, since the TCP's state after receiving the ack that created the liberation corresponds to one in which it should not send due to SWS avoidance.

Blameless due to PSH When a TCP is sending data and has temporarily exhausted the available data, then the TCP marks the last packet it sends with the PSH ("push") flag, informing the receiving TCP that it should not wait for any further data since none will be forthcoming for a while. Any ack received after a PSH packet was sent is marked as blameless-due-to-PSH, since the TCP might still not have any fresh data to send, and hence could reasonably ignore the opportunity created by the ack to send additional data.

Blameless due to no more data `tcpanaly` has looked ahead and the sender will never have any more data to send, so the liberation can be safely ignored. This attribute is separate from the one above because TCPs do not always set PSH when all of the data for a connection has been sent.

Should not be missed If true, then `tcpanaly` should specifically complain if the TCP fails to respond to the ack. An example is for the third duplicate ack that, for many TCP implementations, triggers a "fast retransmission" sequence (§ 9.2.7). For those implementations, the fast retransmission should *always* occur.

These attributes guide `tcpanaly` in correctly assessing the sending TCP's response times. For "blameless" liberations, if the TCP's apparent response time is excessive, it is ignored.

There are many additional, minor details to `tcpanaly`'s accurate management of liberations. We omit further discussion here in the interest of brevity. They are documented in the C++ code.

11.3.2 Inferring sender windows

`tcpanaly` sometimes lacks critical information that affects the sending TCP's behavior. In this and the next two sections we discuss how it infers such information based on testing the directly-available information for self-consistency.

In § 11.1 above we discussed the problem of determining whether the sending TCP has an unstated "sender window," that is, a fixed limit on how many packets it can have in flight separate from its congestion window and the offered window (§ 9.2.2). In practice all TCPs have a sender window, namely the amount of buffer space they can commit for holding previously sent data until it is acknowledged. The key question, though, is whether this limit is ever smaller than the congestion window and the offered window. If so, then it is reasonable for the TCP to not send data even though from recent liberations it looks like it could. However, there is no obvious sign in a packet trace what the TCP's actual sender window is.

`tcpanaly` infers whether a sender window was in effect by calculating the maximum amount of data the connection ever had in flight. Then, during its second pass over the trace, if at some point the TCP's congestion window and the offered window would have allowed it to have sent a full segment (§ 9.2.2) more than this amount, but the TCP failed to do so, then the failure to send additional data was either due to a sender window, or to insufficient understanding of the TCP.[1] One clue sometimes present that the limitation was indeed a sender window is that often the sender window is the same as the offered window advertised by the *sending* TCP in the data packets it transmits to the receiver. `tcpanaly` can still make mistakes, however, particularly when it fails to realize that the reason the TCP did not transmit more data is not because of a sender window, but because of the arrival of a source quench (§ 11.3.3).

11.3.3 Inferring source quenches

Unfortunately, the filter pattern we used to collect the traces in our study was limited to exactly the TCP packets used for each TCP transfer. This limit was imposed for security reasons, to guarantee that the packet filter making the trace could not be used (either accidentally, or maliciously, by a cracker) to spy on other network traffic using the same link. Usually, the TCP packets fully suffice for understanding the resulting TCP behavior. One exception, however, is if some element of the Internet infrastructure sends an Internet Control Message Protocol (ICMP; [Po81b]) message to the sending TCP instructing it to slow down. This message is called a "source quench," and its packet format does not match the filter pattern used for our measurement, so our traces do not include any source quench ICMP messages.

TCP implementations vary on how they respond to source quench messages. In general, the TCP is supposed to diminish its sending rate. BSD-derived TCPs do so by entering a "slow start" phase (§ 9.2.4). Figure 11.1 shows an example of this happening. At time $T = 11.2$ the congestion window is five packets, so the ack at $T = 11.25$, which advanced the window by two packets, should have led to two additional packets being sent. None were, however. About 200 msec later another ack arrives and advances the window another two packets, yet only one packet is sent, as though the window were now only three packets. This would indeed be the case if a source quench had arrived between $T = 11.2$ and $T = 11.25$, setting the window to 1 packet. Due to slow start, the first ack ($T = 11.25$) would then have advanced the window to 2 packets, not enough to send

[1] A particularly easy error to make is to overlook the possibility that the TCP failed to send due to SWS avoidance.

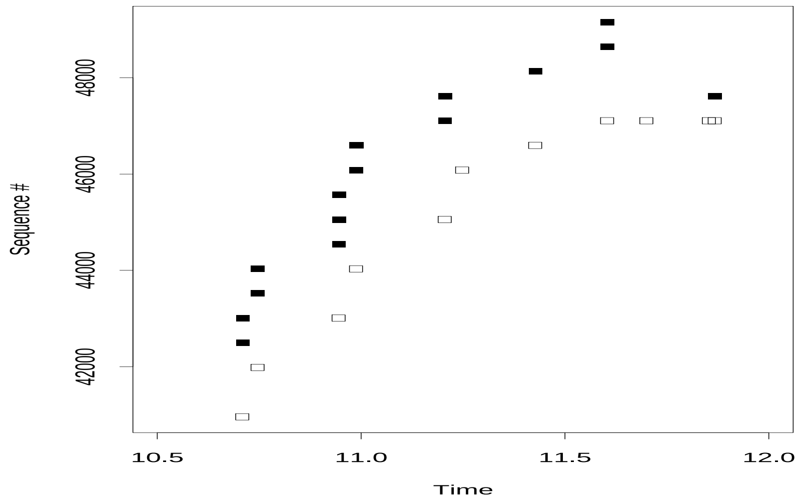

Figure 11.1: Sequence plot showing effects of unobserved source quench

any new data, and the second ack would have advanced it to three packets. Similarly, the ack around $T = 11.6$ advances the window to 4 packets, as can be seen in the plot.

Solaris also enters slow start, but in addition it cuts *ssthresh* by a factor of two. Linux 1.0 diminishes the congestion window by one full segment (MSS).

`tcpanaly` infers the presence of a source quench as follows. Any time it detects a large lull between when a liberation is created and when the resulting packet was actually sent, it looks at the series of packets between the ack creating the liberation and the data packet ostensibly corresponding to the liberation, as well as the packets shortly after. If the whole series is consistent with slow start having begun (for no discernable reason) sometime between the ack and the data packet, then the trace is consistent with an unseen source quench. (This analysis does not work for Linux 1.0, since it does not enter slow start. Consequently, `tcpanaly` fails to infer source quenches for Linux 1.0.)

Source quenches are quite rare—they have been deprecated (§ 4.3.3.3 of [Ba95]), since generating extra network traffic during a time of heavy load violates fundamental stability principles—but they do happen. In $\mathcal{N}_1$, `tcpanaly` inferred a total of 26 source quenches in 20 different traces. Almost all of these included `bnl` as sender (one time as receiver), suggesting that a router near it still generates source quenches when stressed. Likewise, `tcpanaly` inferred 65 source quenches in 64 different $\mathcal{N}_2$ traces, almost all of which involved `connix` or `austr2`. The `connix` source quenches are quite striking in their regularity: the time they arrived after the beginning of the connection was always between 500 msec and 1 sec, with a median and mean of 750 msec. The connections further exhibit a strikingly regular pattern of the `connix` TCP opening its congestion window to about 2^{15} bytes just before the source quench is sent, suggesting that it is single-handedly stressing a particular nearby router.

We note that often the source quenches inferred by `tcpanaly` are almost immediately followed by retransmissions, indicating that the router sending them is indeed almost overwhelmed.

We can see this phenomenon at the end of Figure 11.1. We also note that `tcpanaly`'s analysis of possible source quenches is only heuristic. In particular, if a source quench is followed by a retransmission timeout or a second source quench, then `tcpanaly` will not find an exact match to a slow-start sequence following the first source quench, and does not infer that a source quench occurred.

11.3.4 Inferring initial *ssthresh*

The final inference done by `tcpanaly` is determining whether the sending TCP has an initial limit on *ssthresh*. Recall from § 9.2.6 that the TCP state variable *ssthresh* determines when the TCP should switch from "slow start," in which the congestion window begins at only 1 packet but rapidly expands, to "congestion avoidance," in which the window increases less quickly.

Usually, when a new TCP connection begins, its *ssthresh* variable is initialized to the equivalent of "infinity," allowing it to rapidly probe for the presence of arbitrarily high available bandwidth. (Exceptions are Solaris, which initializes *ssthresh* to 8 packets, and Linux, which sets it to a single packet.) Sometimes, however, the TCP implementation first inspects its *route cache* for information about previous connections to the same remote host. These implementations then initialize *ssthresh* based on the congestion conditions previously encountered.

`tcpanaly` needs to be able to detect when the initial *ssthresh* is lower than normal, because otherwise it will erroneously conclude that the sending TCP is very slow in responding to the acks that would normally—due to slow start—have opened up the congestion window beyond the hidden initial *ssthresh* limit. It does so in a fashion similar to inferring source quenches (§ 11.3.3). Any time the TCP appears to take too long to respond to a liberation, if the TCP has not already undergone a retransmission (which would have altered *ssthresh* anyway) then `tcpanaly` looks ahead to see whether the series of packets beyond the point of the apparent lull is consistent with congestion avoidance rather than slow start. If so, it infers that the connection had an atypical initial value for *ssthresh*.

It turns out that only the experimental $\mathrm{VJ}_{1,2}$ TCPs exhibit non-default initial *ssthresh* values.[2] Other TCPs may in the future exhibit different initial *ssthresh*'s, too, as a recent proposal for improving TCP's start-up behavior includes setting the initial *ssthresh* based on measurements of the connection's first few packets [Ho96].

11.4 Receiver analysis

In this section we discuss how `tcpanaly` analyzes a TCP implementation's *receiver* behavior, namely when and how the implementation chooses to acknowledge the data it receives.

11.4.1 Ack obligations

Similar to the notion of data liberations (§ 11.3.1), when analyzing receiver behavior `tcpanaly` addresses vantage point problems (§ 10.4) by keeping track of a list of pending ack

[2] The HP/UX implementations appeared to, also, but so rarely that we cannot determine whether a different, not yet determined mechanism is leading to the early onset of congestion avoidance.

obligations. Whenever a TCP receives data, it incurs some sort of obligation to generate an acknowledgement in response to that data. The obligation may be *optional* or *mandatory*, as discussed below.

`tcpanaly` has a default set of rules for the sorts of obligations created by different types of packets. It then includes additional rules for specific implementations that do not follow the default set, as discussed in § 11.6. In our discussion of different types of ack obligations below, we also detail `tcpanaly`'s corresponding default rules.

Optional ack obligations

An *optional* ack obligation refers to data that the TCP may choose to acknowledge but can also wait before acknowledging. This occurs when new data arrives that is in sequence. The TCP standard states that a TCP may refrain from acknowledging such data in the hopes that additional data may arrive and the acknowledgements combined, but for no longer than 500 msec (§ 4.2.3.2 of [Br89]). Furthermore, a correct TCP implementation should always generate at least one acknowledgement for every two packet's worth of new data received.[3] Acknowledgement strategy is further discussed in [Cl82].

`tcpanaly` considers the arrival of any new, in-sequence data as creating an optional ack obligation, even if more than one such packet has arrived and not yet been ack'd. When an acknowledgement is finally generated for the new data, we then inspect the number of packets acknowledged to see whether the TCP has heeded the suggested limit of one ack for every two packets. `tcpanaly` reports instances in which the limit is violated, but considers this different than a failure to meet a *mandatory* ack obligation, discussed in the next section.

Mandatory ack obligations

A *mandatory* ack obligation occurs when a packet arrives to which the TCP standard requires the receiving TCP to respond with an acknowledgement. In the original TCP specification, the receipt of a packet containing already-acknowledged data mandated that a new acknowledgement be sent, since the unnecessary retransmission indicates that the sender may be confused as to what data the receiver has successfully received. This was clarified in § 4.2.2.21 of [Br89] to also optionally include the receipt of packets whose data cannot yet be acknowledged due to a sequence "hole" below the packet's sequence, in order to facilitate "fast retransmission" (§ 9.2.7).

Consequently, `tcpanaly` considers the arrival of any out-of-sequence data as creating a mandatory ack obligation. (The mandatory obligation is not to ack the out-of-sequence data, but instead to generate a cumulative ack for all in-sequence data received, since TCP acknowledgements always reflect the extent of cumulative, in-sequence data received, per § 9.2.1.) `tcpanaly` keeps track of statistics concerning how often and how quickly an implementation responds to mandatory obligations separately from those for optional obligations.

Gratuitous acks

If `tcpanaly` observes an ack being sent for which there was no obligation, and which does not change the offered window or terminate the connection, then it flags the ack as *gratuitous*.

[3] § 4.2.3.2 of [Br89] expresses this as "SHOULD," while § 4.2.5 notes it as "MUST."

Observing gratuitous acks plays a role analogous to observing window violations when analyzing a sender's behavior: they can indicate confusion regarding `tcpanaly`'s interpretation of the TCP's behavior, or measurement errors in the packet trace.

11.4.2 Inferring checksum errors

As noted in § 11.2, `tcpanaly` often cannot verify a packet's TCP checksum because the packet filter only records the beginning of the packet and not its entire contents. Nevertheless, checksum failures do indeed occur, and when they do `tcpanaly` needs to deduce their presence to avoid misattributing the receiving TCP's behavior to something else.

There are several situations in which `tcpanaly` infers the possibility that a packet received earlier had a checksum error (and thus the subsequent ack obligations derived from the trace do not correctly reflect the situation as perceived by the receiving TCP):

1. If a retransmission is received for data already apparently received by the TCP, and which should have previously been ack'd by the TCP but was not, and if all sequentially earlier data has been ack'd;

2. if instead of acking increasing sequence numbers in response to a series of optional ack obligations, the TCP generates duplicate acks as each new packet arrives, until the retransmission called for by the duplicate acks arrives; or,

3. if an apparently unnecessary retransmitted packet actually results in an advance of the acknowledged sequence number, indicating that the retransmission did indeed fill a sequence hole. (This item is slightly different from the first item, because here we are considering data that originally arrived above-sequence, and so could not be acknowledged directly at that time.)

More precisely, what `tcpanaly` *really* infers is that the TCP acted as though it ignored an arriving packet. We then assume that the packet was ignored because it failed its checksum test. We return to this point in more detail later.

`tcpanaly` does *not* attempt to infer checksum errors in traces recorded by packet filters that it has determined either dropped (§ 10.3.1) or resequenced (§ 10.3.6) packets, since it is to difficult with these traces to disambiguate between a genuine checksum failure and seemingly confusing TCP behavior because the trace is inaccurate.

Figure 11.2 shows a sequence plot reflecting two checksum errors. The plot comes from a trace recorded at the receiving end of a connection. Consequently, most of the points showing acknowledgements lie directly on top of the data packets being acknowledged and thus do not show up visually. (This is fine for the purposes of this example.) Up through time $T = 20.0$ the data all arrives in sequence, but starting at time $T = 19.5$ the receiving TCP generates duplicate acks for sequence 74,241 rather than advancing the acknowledgements. This continues until data packet 74,241 is retransmitted at $T = 20.2$. The retransmission leads to the TCP immediately acking all of the outstanding data, fully consistent with a single checksum error occurring at the 74,241 data packet. Note that, after the retransmission, the pattern repeats at time $T = 20.5$. Duplicate acks for sequence 78,849 indicate that the 79,361 packet was likewise discarded due to a checksum error.

Figure 11.3 shows a sequence plot of a considerably different instance of checksum errors. Instead of as in Figure 11.2, where two isolated packets were corrupted, here an entire burst of

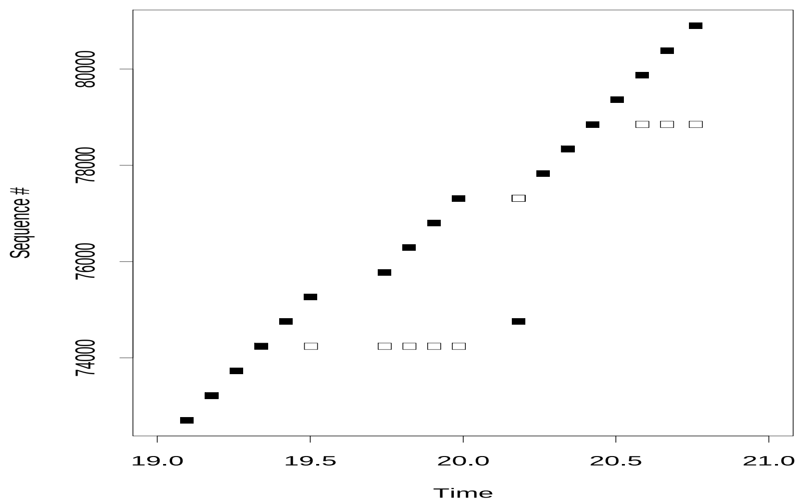

Figure 11.2: Receiver sequence plot showing two data checksum errors

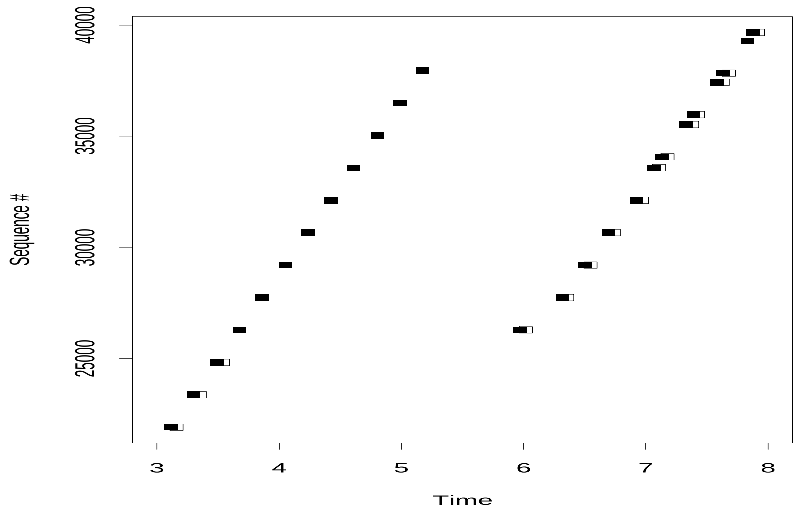

Figure 11.3: Sequence plot showing a burst of checksum errors

9 packets were all discarded by the receiving TCP. We can tell that the TCP did not accept the nine packets from 26,281 to 37,961 at $T = 3.7$ to $T = 5.2$ sec, because as the data is retransmitted the TCP only acknowledges the newly retransmitted packets—they are not shown filling any sequence "holes" as would be the case if some of the 9 packets had been successfully received.

We further discuss checksum bursts such as this one, as well as detailing the prevalence of checksum errors in our datasets, in § 13.3. As noted above, what `tcpanaly` really infers are *packets ignored by the receiver*, which we then *assume* were ignored due to checksum errors. It is possible that the packets were ignored for a different reason, such as the kernel lacking sufficient buffers to keep them until the receiving TCP could process them. In § 13.3 we investigate this possibility and find that almost all of the errors appear indeed due to packet corruption.

11.5 Sender behavior of different TCP implementations

In this section we look at the variations in how the different TCP implementations listed in Table XV act when sending data. Our findings in this section and the next are almost all based on the modifications we had to make to `tcpanaly` in order for it to successfully match the traces of the TCP's behavior. A few other behaviors were discovered by examining source code for the implementations, which we had for Linux 1.0, Solaris 2.5, VJ_1 and VJ_2, as well as the invaluable source code analysis of Net/3 in [WS95]. In addition, in § 11.7 we present brief findings of behavior observed for three other implementations; these were determined by manually studying sequence plots, as `tcpanaly` does not have the behavior of these implementations coded into it.

TCP behavior is very complex, and we do not attempt to exhaustively examine it. Our main interest is in *performance* and *congestion behavior*: does the TCP implementation use the network as effectively as it can, and does it correctly adapt to congestion by decreasing its transmission rate, as is required for global Internet stability? There is a natural tension between these two goals, and a great deal of research has gone into tuning TCP so it balances high performance with stable behavior in the presence of congestion. One of the basic questions we would like to answer in this section is how successfully this research has in fact been incorporated into TCP implementations deployed in the Internet. The answer turns out to be "quite mixed."

We proceed as follows. First, we give an overview of previous work in analyzing the behavior of TCP implementations. The work focuses almost entirely on sender behavior. Next, we present the *sender* behavior of the implementations in our study, beginning with two "generic" implementations, "Tahoe" and "Reno," from which almost all of the implementations derive their behavior. We then discuss each of the different implementations in Table XV. After analyzing sending behavior, we turn in § 11.6 to *receiver* behavior, namely the policy by which the TCP sends acks. Finally, we look in § 11.7 at the behavior of some additional TCP implementations: Windows 95, NT, and Trumpet/Winsock. This last investigation was motivated by our finding that the independently written TCP implementations in our study (Linux and Solaris) suffered from serious congestion and performance problems. We were interested to see whether other non-Reno-derived TCP implementations likewise have these sorts of problems. The answer turns out to be: yes!

11.5.1 Previous studies of TCP implementations

Several researchers have previously studied and characterized the behavior of TCP implementations, using different techniques from ours.

Comer and Lin

Comer and Lin studied TCP behavior using a technique termed *active probing* [CL94]. Active probing consists of treating a TCP implementation as a black box and observing how it reacts to external stimuli, such as a loss of connectivity to the other endpoint, or a failure by the other endpoint to consume data sent by the TCP under study. They examined five implementations, IRIX 5.1.1, HP-UX 9.0, SunOS 4.0.3, SunOS 4.1.4, and Solaris 2.1, to determine their initial retransmission timeout values, "keep-alive" strategies, and zero-window probing techniques. The authors' emphasis was on correctness in terms of the TCP standards, and they found several implementation flaws.

Brakmo and Peterson

Brakmo and Peterson analyzed performance problems they found in TCP Lite, a widely-used successor to TCP Reno (and the basis for some of the implementations in our study) [BP95b]. TCP Lite is also known as "Net/3," which is the term we will use for consistency with other studies we discuss.

Their approach was to simulate Net/3's behavior using a simulator based on the x-kernel [HP91]. The x-kernel is highly configurable, so that the simulations actually directly executed the Net/3 code, an important consideration for assuring accuracy. They found:

1. An error in the "header prediction" code. Net/3 uses this code to make an early decision whether an incoming packet is what would have normally been expected: either an in-sequence, non-retransmitted data packet, or an ack for new data that does not change the size of the offered window [CJRS89]. If the packet matches the expectation, then it can be processed succinctly; for example, without all the computations necessary to update the congestion window.

   The error they found was that the code considered an incoming acknowledgement as expected even if the congestion window had been inflated due to "fast recovery" (§ 9.2.7). Thus, if after fast recovery the acknowledgements all passed the header prediction test, then the window was never deflated.

   Fixing this problem is a one-line addition to the prediction code.

2. Inaccuracies computing the retransmission timeout (RTO) due to details in some of the integer arithmetic used to approximate the true real-numbered calculations. The authors proposed altering the scaling used in the integer arithmetic to remedy the inaccuracy.

3. Confusion between whether the "maximum segment size" variable used to decide when to send new acknowledgements and how to update the congestion window should include the size of TCP header options or not.

4. Very bursty behavior when the offered window advances a large amount (an incoming ack for a large amount of new data). When this occurs, Net/3 (and, in our experience, all other TCPs) immediately sends as many packets as the new window allows. The authors include a small coding addition that would reduce such bursts to 2 or 3 packets at a time.

5. A "fencepost" error in determining whether the congestion window was inflated due to fast recovery, and later needs deflating. The fix is replacing a $>$ test with a $\geq$ test.

Of these problems, we found that a number of the implementations in our study exhibited all of them, except we did not examine the RTO's used by the implementations and thus did not have an opportunity to observe the second problem.

Stevens

In [St96], Stevens devotes a chapter to an analysis of the behavior of a large number of TCP connections made to a World Wide Web server running Net/3 TCP. The analysis was based on a 24 hour `tcpdump` packet trace of 147,103 attempts by remote sites to connect to the Web server. He characterized the range of options offered by the remote TCPs, finding tremendous variation (including many obviously incorrect values); the rate at which connection attempts and re-attempts arrived; the variation in round trip time between the server and the remote clients; and the pending-connection load on the server. In addition, he analyzed three Net/3 implementation bugs, one in which two different TCP connection states become confused ("SYN received" and "performing keep-alive probe"), one in which the TCP fails to time out zero window probes (and thus over time devotes more and more resources to zero window probes for connections that have permanently lost connectivity), and one in which the TCP can skip the first cycle of "slow start" if it happens to have data ready to send upon connection establishment.

He further found that almost 10% of all SYN packets were retransmitted; some remote TCPs sent "storms" of up to 30 SYNs/sec, all requesting the same connection; and some remote TCPs did not correctly back off their connection-establishment retry timer, or reset it after 4 attempts.

Dawson, Jahanian and Mitton

In recent work, Dawson, Jahanian and Mitton studied six TCP implementations using a "software fault injection" tool they developed [DJM97]. The implementations were: SunOS 4.1.3, AIX 3.2.3, NeXT (Mach 2.5), OS/2, Windows 95, and Solaris 2.3. The first and last were also present in our study; the remainder were not.

Their basic approach is a refinement of Comer and Lin's "active probing" (§ 11.5.1). They use the x-kernel to interpose a general purpose packet manipulation program between the TCP implementation and the actual network, so they can arbitrarily alter, delay, reorder, replicate, or discard any packets the TCP sends or receives.

The main focus was on timer management. They found that retransmission sequences vary a great deal; that some TCPs do not correctly terminate the connection with a RST packet if the maximum retransmission count is reached; and that Solaris 2.3 uses a much lower bound for its initial RTO, around 300 msec, than the other implementations, and also takes much longer to adapt the RTO to higher, measured RTTs. We further discuss both of these latter problems in § 11.5.10.

They also studied keep-alive behavior. "Keep-alives" are an optional TCP mechanism for probing idle connections to ensure that the network path still provides connectivity between the two endpoints. The TCP standard specifies that, if a TCP supports keep-alives, then, by default, the idle interval must be at least two hours before the TCP begins probing the path. However, the authors of [DJM97] found that OS/2 begins keep-alive after only 800 sec. In addition, Windows 95 only makes four keep-alive probes, all sent one second apart. If none of these elicit replies, then it abandons the connection. This latter behavior will make Windows 95 connections quite brittle in the face of mid-sized connectivity outages.

Finally, they found that Solaris 2.5.1 (not otherwise part of their study) incorrectly implements Karn's algorithm, which is used to disambiguate round-trip time measurements [KP87].

11.5.2 Generic Tahoe behavior

The goal of our TCP behavior analysis is to delve considerably deeper into the performance and congestion behavior of the different TCPs in our study than done previously. We begin by discussing the generic TCP "Tahoe" implementation that `tcpanaly` uses as a building block for describing the behavior of all of the TCP implementations except Linux 1.0.

Our Tahoe implementation reflects the behavior of the Tahoe version of BSD TCP, released in 1988 [St96, p.27]. It includes *slow start* (§ 9.2.4), *congestion avoidance* (§ 9.2.6), and *fast retransmission* (§ 9.2.7), but not *fast recovery* (§ 9.2.7). It updates the congestion window upon the receipt of any ack for new data. It sets *ssthresh* to half the effective window upon a retransmission, but for fast-retransmit it rounds the result down to a multiple of the Maximum Segment Size (MSS; § 9.2.2), while for a timeout it does not. No doubt this inconsistency is due to the fast retransmit code having been added later than the original timeout code. In both cases, *ssthresh* is never set lower than 2·MSS.

Tahoe updates the congestion window *cwnd* using congestion avoidance if *cwnd* is strictly larger than *ssthresh*. The increase is:

$$\Delta W = \left\lfloor \frac{\mathrm{MSS}^2}{cwnd} \right\rfloor, \tag{11.1}$$

without any additional constant term (Eqn 11.2 below).

11.5.3 Generic Reno behavior

The "Reno" version of BSD TCP was released in 1990. Our generic Reno implementation does not attempt to precisely describe that release, but instead to provide a common base from which we can express as variants the numerous Reno-derived implementations in our study. Reno differs from Tahoe as follows:

1. It implements *fast recovery* (§ 9.2.7), in which following a fast retransmit it inflates the congestion window *cwnd* and will send additional packets if enough additional duplicate acks arrive.

2. It consequently suffers from the "header prediction" and "fencepost" errors when deflating the window, as previously described in [BP95b] (§ 11.5.1).

3. It rounds *ssthresh* down to a multiple of MSS for timeout retransmissions as well as fast-retransmits.

4. It includes an *additive constant* when increasing the window during congestion avoidance. That is, instead of using Tahoe's increase as given in Eqn 11.1, it uses:

$$\Delta W = \left\lfloor \frac{\text{MSS}^2}{cwnd} \right\rfloor + \left\lfloor \frac{\text{MSS}}{8} \right\rfloor. \tag{11.2}$$

   The extra term MSS/8 leads to a super-linear increase of the congestion window during congestion avoidance. Subsequent to its addition to Reno, this extra term has come to be viewed as too aggressive ([BP95b], credited to S. Floyd in footnote 6), but its presence is widespread.

11.5.4 BSDI TCP

We had several BSDI 1.1 and 2.0 sites in our study, as well as one site running an alpha release of 2.1, which we term 2.1α.

BSDI 1.1 appears identical to our generic Reno implementation. We observed two changes with BSDI 2.0. The first is that it omits the extra congestion avoidance increment (i.e., it uses Eqn 11.1 rather than Eqn 11.2). The second is that it computes the MSS governing how much data it should send in each TCP/IP packet in a slightly complicated fashion, as follows.

When initiating a connection, BSDI 2.0 includes the "window scaling" and "timestamp" options in its initial SYN packet. If the remote peer agrees to these options in its SYN-ack, then each subsequent packet sent by BSDI 2.0 includes an accompanying timestamp in its header. With padding, this option requires an additional 12 bytes of space in the header. If, for example, the MSS is 512 bytes, as is often the case, then the TCP should send 512 bytes of data in each packet along with 52 bytes of header, the usual 40 bytes of TCP/IP header plus the timestamp option. Instead, it uses an MSS of 500 bytes. The fundamental problem[4] is that the implementation is overloading the notion of "MSS," trying to make it serve as both the maximum amount of *data* to send to the receiver in one packet, and also as the largest total TCP/IP packet size that can be sent along the Internet path without incurring fragmentation. Yet, the presence of options means the relationship between these two is more complex than simply adding in a constant header size.

To further complicate matters, BSDI 2.0 uses the unadjusted MSS (i.e., its value before deducting 12 bytes for options) in its congestion window computations.

None of these MSS fine points has much impact at all on BSDI 2.0's performance or congestion behavior. But they do subtly alter the conditions under which the TCP will send packets, and thus solid analysis of the TCP's behavior must take them into account.

BSDI 2.1α behaves the same as BSDI 2.0 except for two differences. The first is that it uses the adjusted MSS for its congestion window computations (the MSS still has 12 bytes deducted for the header options). The second is that, if the remote TCP does not include an MSS option in its SYN-ack reply to the BSDI TCP's initial SYN packet, then the congestion window and *ssthresh* are initialized to a huge value[5] instead of MSS bytes. This bug occurs because of an assumption in the Net/3 code that SYN-acks will always include MSS options and that therefore receiving a SYN-ack is the proper time to initialize *cwnd* and *ssthresh*.

[4] Pointed out to me by Matt Mathis.

[5] Specifically: 2^{30} - 2^{14}. See [WS95, p.835].

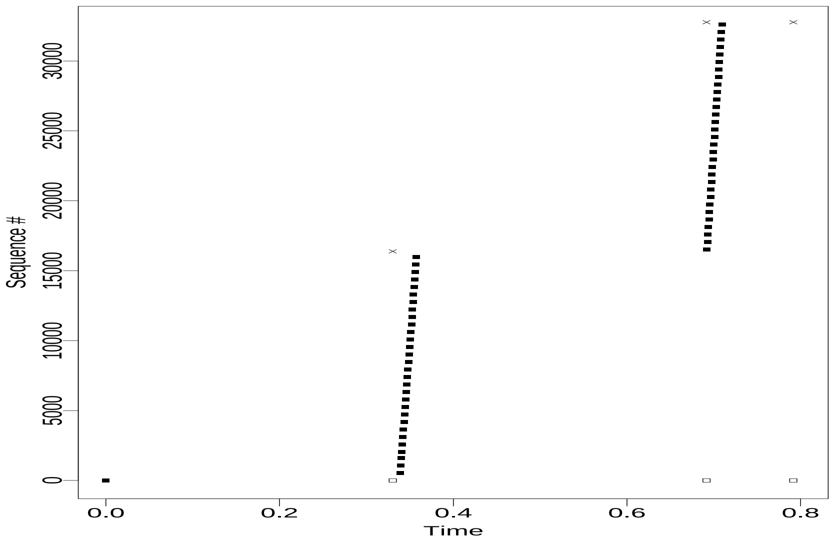

Figure 11.4: Sequence plot showing the Net/3 uninitialized-*cwnd* bug

Figure 11.4 dramatically illustrates the potential burstiness created by this bug. Here, when the initial ack arrives offering a window of 16,384 bytes (and with no MSS option), the BSDI TCP instantly sends all the full-sized (536 bytes, in this case) packets that fit within the window, a total of 30 packets. The next ack (which was sent because it updates the advertised window) offers a larger window (cf. § 9.3), and again the TCP floods the network with packets, taking advantage of the increased window. A third ack arrives but does not advance the window, so nothing further is sent.

Ironically, even the first packet of the storm was lost (as was its retransmission), as can be seen by the lack of progress in the acknowledgements. All told, 14 of the 61 packets sent in the first two spikes were lost (any other connections sharing the path between the two TCPs also surely suffered).

Fortunately, it is relatively rare that this bug manifests itself so dramatically. It requires interaction between the BSDI TCP and a remote TCP that both does not send MSS options in its SYN-ack, and offers a large window. TCPs that do not offer MSS options tend to be of quite old vintage, and these tend to offer small receiver windows.

The bug does not always manifest itself under the conditions given above. We suspect that the times it does not are when the BSDI TCP finds initial *cwnd* and *ssthresh* values in its route cache, and thus begins the new connection with tamer values.

This bug nicely illustrates the fundamental tension between TCP performance and congestion behavior. Fixing it lessens the TCP's performance (blasting out 30 packets at a time can work extremely well in making sure one utilizes all available bandwidth), but also makes the TCP much more “congestion friendly.”

Finally, we note that the IRIX 5.2 TCP implementation also exhibits this bug, as does Net/3. Most likely both BSDI 2.1α and IRIX 5.2 “inherited” the bug as they incorporated enhancements and changes from Net/3.

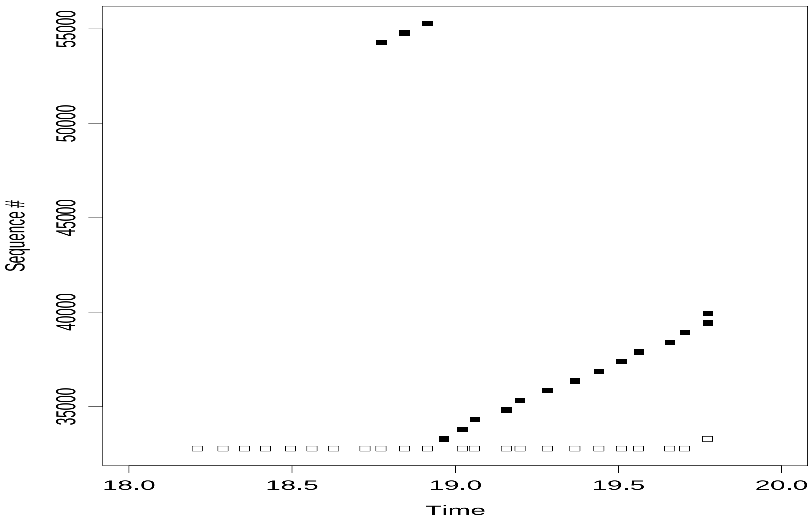

Figure 11.5: Sequence plot showing the HP/UX congestion window advance with duplicate acks

11.5.5 Digital OSF/1 TCP

Digital's OSF/1 TCP implementation appears virtually identical to our generic Reno implementation. The only difference we observed was that it does not always manifest the "header prediction" bug (§ 11.5.1). We could not find a pattern to when it would and when it would not. For analyzing a given trace, `tcpanaly` accommodates its inability to know whether the TCP will exhibit the bug by looking ahead to determine whether in fact the TCP deflated the congestion window.

We did not observe any differences between Digital OSF/1 versions 1.3a, 2.0, 3.0, and 3.2.

11.5.6 HP/UX TCP

HP/UX 9.05 TCP is very similar to our generic Reno implementation. The only differences we observed were two behaviors that rarely have an opportunity to manifest themselves. First, HP/UX 9.05 does not clear its "dup-ack" counter (§ 9.2.7) when a timeout occurs, so if it receives additional duplicate acknowledgements after a timeout, these can lead to fast retransmit or the sending of additional fast recovery packets. Second, such duplicate acks also advance the congestion window, providing that the timeout was for a segment previously retransmitted using fast retransmission.

We illustrate this latter behavior in Figure 11.5, since it is somewhat unusual. The stream of acks along the bottom of the figure are all duplicates. The packet they call for has already been retransmitted, but was dropped. The data packets sent around $T = 18.8$ with sequence numbers near 55,000 are fast recovery packets, sent out by inflating *cwnd*. Just before $T = 19.0$, the previously-retransmitted packet times out and is retransmitted again. As more dups arrive (from an earlier huge flight of packets), each liberates another retransmission via fast recovery. This is not ideal behavior: the packets being retransmitted may all have already arrived at the receiver. The TCP should instead either send additional *new* data, as it was doing prior to the timeout (and which is the intent behind fast recovery, thwarted by the timeout having reset *cwnd*), or simply wait one RTT to see what data the peer has now received.

HP/UX 10.00 behaves identically to HP/UX 9.05 except it advances the congestion window (per Figure 11.5) for dup acks received after any timeout, not just one of a packet previously transmitted using fast-retransmission; and it uses the original MSS it offered to its peer when computing congestion window updates, rather than the final MSS negotiated during the connection setup.

11.5.7 IRIX TCP

IRIX 4.0 appears identical to our generic Reno implementation except it does not manifest the header prediction bug (§ 11.5.1). IRIX 5.1 does, though not always, the same as Digital OSF/1 TCP (§ 11.5.5). IRIX 5.2 is identical to IRIX 5.1 except it also exhibits the uninitialized-*cwnd* bug shown in Figure 11.4. IRIX 5.3 is identical to IRIX 5.2 except that, if the remote peer does not include an MSS option in its SYN-ack, then IRIX 5.3 initializes the congestion window to the value it offered, even if this is larger than the final MSS it used.[6]

11.5.8 Linux TCP

The Linux 1.0 TCP implementation was written independently from any other. Consequently, it is not surprising that it differs in many ways from the others in our study, including some ways that are particularly significant.

The most significant is its *broken retransmission behavior*. First, often when it decides to retransmit, it re-sends every unacknowledged packet in a single burst. Second, it decides to retransmit much too early, leading it to retransmit packets for which acks are already heading back, or, even worse, which are themselves still in flight towards the receiver.[7] Jacobson terms this sort of behavior "the network equivalent of pouring gasoline on a fire" [Ja88], because it unnecessarily consumes network resources precisely when they are scarce. Consequently, it can lead to *congestion collapse*, in which the network load stays extremely high but throughput is reduced to close to zero [Na84].

Figure 11.6 illustrates Linux 1.0's behavior. At about $T = 85$ an acknowledgement arrives advancing the window by three packets, which the TCP immediately sends. At $T = 86$, however, two duplicate acks arrive, the first of which spurs the TCP to retransmit every packet it has in flight. Shortly after, an ack arrives for sequence 77,825; this correctly liberates only new data, as does this ack for 78,849 that follows momentarily. None of the new data arrives successfully—the network is already clogged with the unnecessary retransmissions.

At $T = 87.8$, sequence 79,361 times out and is retransmitted. This happens again at $T = 90.6$ (the timeout is not fully doubling as it backs off, though in other cases it does). At $T = 92$ dup acks for 78,849 arrive. These were sent within 400 msec of the ack received at $T = 86.4$ but took more than 5 seconds to arrive, indicating huge delays in the network. The TCP appears to

[6]The offered MSS can differ from the final MSS used because, if the remote peer does not include an MSS option, then the TCP must use an MSS of no more than 536 bytes (§ 4.2.2.6 of [Br89]).

[7]These retransmissions usually occur shortly after receiving an ack, suggesting that they are not timeout retransmissions per se, but are stimulated instead by the arrival of the ack.

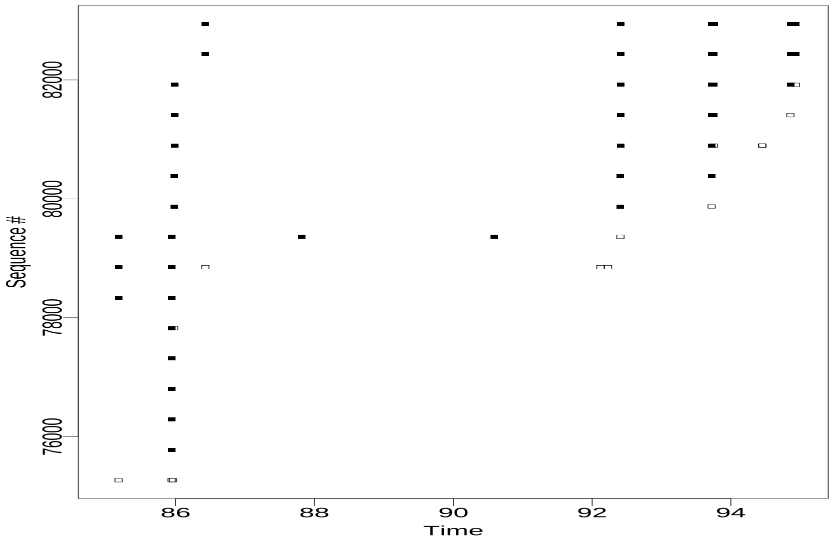

Figure 11.6: Sequence plot showing broken Linux 1.0 retransmission behavior

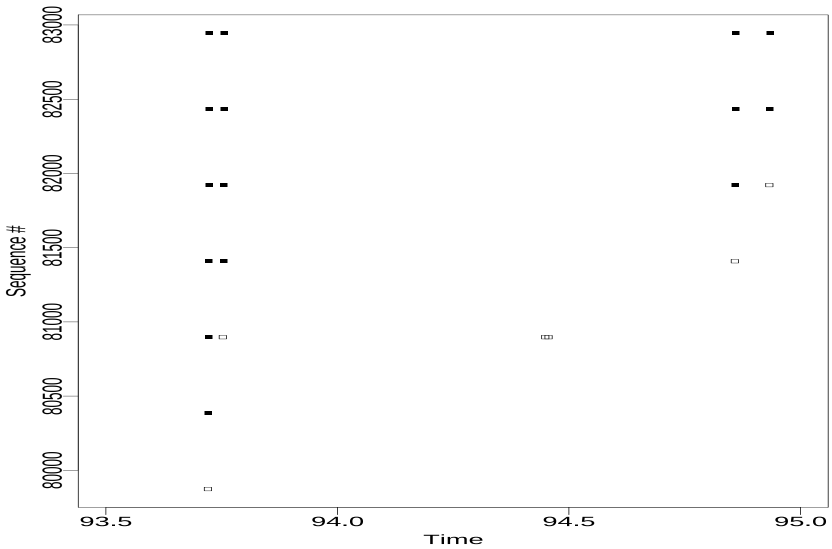

Figure 11.7: Enlargement of righthand side of previous figure

ignore their arrival, however (so would a Reno TCP), but when the twice-retransmitted data packet is ack'd a little while later, again all data in flight is retransmitted, and again 1.3 sec later, and again 1.1 sec later. Worse, not only is all of this data being retransmitted at about 1 sec intervals: if we blow up the activity (Figure 11.7), we see the packets are *also* being retransmitted on much finer time scales![8]

All told, this connection sent 317 packets, 117 of them retransmissions. 20% of the packets were dropped by the network. Of the retransmitted packets that reached the other end, 60% were superfluous, since the data had already arrived safely in an earlier packet. How hard this connection hammered others sharing the network path, we can only guess! But it is clear that, if Linux 1.0 were ubiquitous, its retransmission behavior would bring the Internet to its knees.

The excessive retransmissions clearly follow shortly after the TCP receives an ack, so `tcpanaly` models them as a type of "fast retransmission." We have been unable to determine exactly which incoming acks will trigger these retransmissions, though they appear to occur only for duplicate acks or acks received during a retransmission sequence. Consequently, `tcpanaly` simply allows that either of these might potentially liberate the entire window for retransmission.

The Linux TCP maintainers are aware of this problem and report that it has since been fixed.

Linux 1.0 differs from the other implementations in our study in several other ways:

1. It does not implement fast retransmission or fast recovery.
2. It initializes *ssthresh* to a single packet (MSS), which makes it slow to initially open its window. This behavior is beneficial from the perspective of network stability, as it means that Linux 1.0 TCP connections begin in a fundamentally conservative fashion.
3. The Linux 1.0 code has logic in it to prevent more than 2,048 bytes from ever being in flight, quite conservative behavior. However, a typo[9] renders it ineffective.
4. It does not round *ssthresh* down to a multiple of MSS for any form of retransmission.
5. Its test for slow-start is $cwnd < ssthresh$ rather than $cwnd \leq ssthresh$.
6. In congestion avoidance, it counts the number of acks received, and, when they exceed *cwnd* divided by MSS, then *cwnd* is advanced by MSS and the counter reset to zero.
7. It has no minimum value on how far it can cut *ssthresh*.
8. It acks every packet received (§ 11.6).

11.5.9 NetBSD TCP

As far as we could determine, NetBSD 1.0 TCP is identical to our generic Reno implementation.

[8] We have observed Linux 1.0 retransmitting a packet it sent less than 2 msec before. The first transmission was due to a newly arrived ack advancing the window, and the second was part of a retransmission burst apparently triggered by the receipt of the ack.

[9] The limit is specified as `2048` when what is being tested against it is the number of *packets* in flight.

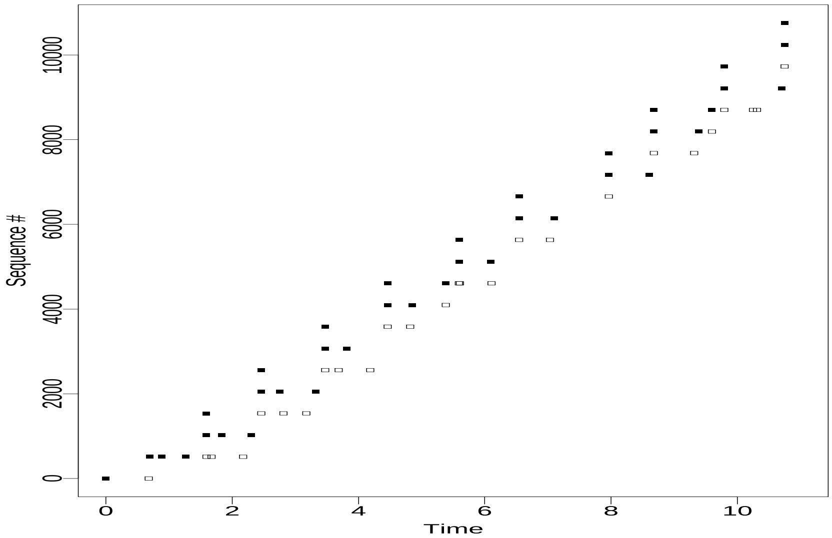

Figure 11.8: Sequence plot showing broken Solaris 2.3/2.4 retransmission behavior, RTT = 680 msec

11.5.10 Solaris TCP

Along with Linux, Solaris TCP is the other independent TCP implementation in our study. `tcpanaly` knows about two versions, 2.3 and 2.4, which differ only in minor ways.

Like Linux, the most striking feature of Solaris 2.3 and 2.4 TCP is its *broken retransmission behavior*. Dawson et al. identified that Solaris uses an atypically low initial value of about 300 msec for its retransmission timeout (RTO). This value, plus difficulties the timer has with adapting to higher RTTs, leads to the broken retransmission behavior. For a connection with a longer RTT, the TCP is guaranteed to retransmit its first packet, whether needed or not. Such an unnecessary retransmission would be only a minor problem if the timer then adapted to the RTT and raised the RTO, but it fails to do so, leading to connections riddled with premature, unnecessary retransmissions.

Figure 11.8 shows an example of the retransmission problem in action. The sender is `sri`, in California, and the receiver is `oce`, in the Netherlands. The round-trip time is about 680 msec, above that of 200 msec for the initial Solaris retransmit timer (but not pathologically large). The Solaris TCP sends almost as many retransmissions as new packets, yet *no* data packets whatsoever were dropped! Each retransmission was completely unnecessary. Furthermore, so many retransmissions are generated that it is difficult to find unambiguous RTT timings, in order to adapt the timer. While the RTO does indeed double on multiple timeouts, it is restored to its erroneously small value immediately upon an acknowledgement for a retransmitted packet, so it never has much opportunity to adapt.

As the path's RTT increases, the problem only gets worse. Figure 11.9 shows a plot for an $\mathcal{N}_2$ connection from `wustl` to `oce`. The smallest RTT in the trace is about 2.6 sec, and it got as high as 9.9 sec. The beginning of the connection is simply disastrous, with the first data packet

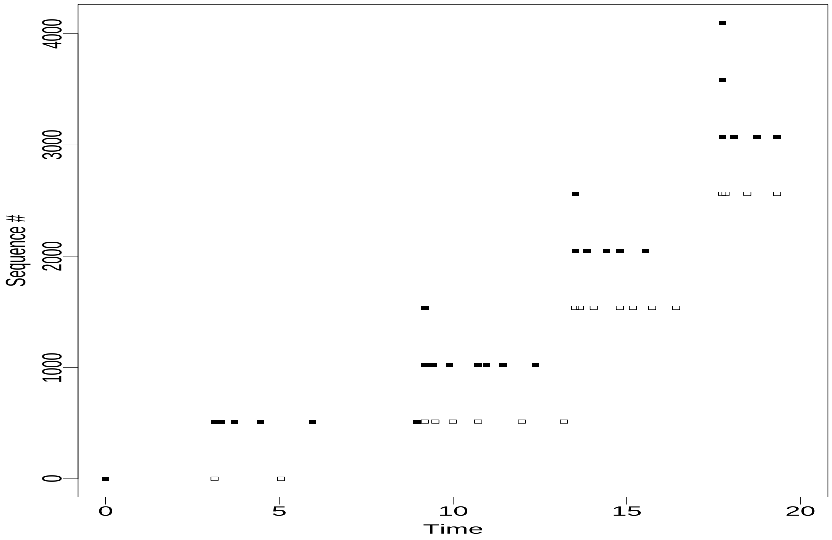

Figure 11.9: Sequence plot showing broken Solaris 2.3/2.4 retransmission behavior, RTT = 2.6 sec

being retransmitted 5 times (the first retransmission occurs closely enough to the original packet that it is hard to distinguish in the plot), the second data packet is retransmitted 6 times, the third 4 times, the fourth 4 times (not all shown), and so on. *None* of the packets or their retransmissions were dropped! All of the retransmissions were needless. Worse yet, because they were needless, they elicited dup acks from the receiver, which eventually reached the level sufficient to trigger fast retransmission (§ 9.2.7), generating *further* needless retransmissions!

The connection eventually ran smoother, as the timer managed to adapt, but was still plagued with needless retransmissions as the RTT grew larger and the timer sometimes failed to track it quickly enough.

Thus, Solaris TCP can effectively increase the load it presents to any high-latency Internet path by a factor of two or even quite a bit more. Unfortunately, many of the most heavily loaded Internet paths—those linking different continents via trans-oceanic or satellite links—have exactly this property. It would be interesting to learn what proportion of the traffic on a very heavily utilized link (such as the U.K.–U.S. trans-Atlantic cable) is due to completely unnecessary retransmissions.

The Solaris TCP maintainers are aware of this problem and have issued a patch to fix it.

Solaris TCP differs from the other implementations in our study in a number of additional ways:

1. It initializes *ssthresh* to 8·MSS. From the perspective of network stability, this is nicely conservative, but from the perspective of performance, it impedes fast transfers unless they are quite lengthy.

2. Sometimes when it receives an ack, it retransmits the packet just after the ack rather than the packet newly liberated by the advance of the window. These retransmissions do not affect the congestion window, nor do they alter the notion of what new data should be sent next time the window advances. Figure 11.10 shows an example. At $T = 10.3$, the Solaris TCP retransmits

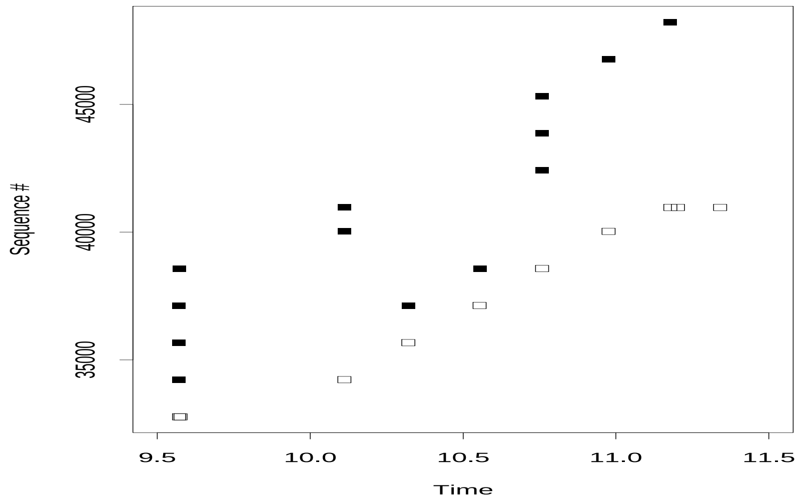

Figure 11.10: Solaris 2.4 retransmitting without cutting *cwnd*

sequence 37,125, and then just after $T = 10.5$ it retransmits 38,577. Yet, when an ack arrives for (the original transmission of) 38,577, we see that the congestion window was not reduced by the retransmissions, but remains at 5 packets.

3. Its duplicate-ack counter survives timeouts, which can lead to a recently retransmitted-via-timeout packet being retransmitted again via fast retransmission.

4. Although there is code in the implementation for fast recovery, it is only exercised under rare circumstances. The problem is that the Solaris implementation is careful to advance the congestion window only upon receiving an ack for new data (see next item). This means that the dup acks that are supposed to keep inflating the window in order to liberate additional packets do not actually increase the window, since they do not acknowledge any new data. The rare circumstance in which the TCP can send a single fast recovery packet is if it has already accumulated during congestion avoidance more “excess” bytes than are required to advance *cwnd* given its current value.

5. During congestion avoidance, the TCP keeps track of exactly how many bytes of data have been acknowledged since the last advance in *cwnd*. Whenever this value exceeds *cwnd*, *cwnd* is increased by the MSS. (Like the Linux congestion avoidance increment strategy, this is closer in spirit to the scheme outlined in [Ja88] than the Tahoe approach given by Eqn 11.1.)

6. Its test for whether it is in a slow-start phase is *cwnd* $<$ *ssthresh* rather than *cwnd* $\leq$ *ssthresh*.

7. Upon receiving an ICMP Source Quench (§ 11.3.3), it sets *ssthresh* to *cwnd*/2 prior to entering slow start.

8. When cutting *ssthresh*, it does not round it down to a multiple of MSS.

The only differences between Solaris 2.3 and 2.4 that we observed are in their acking policies. See § 11.6 for discussion.

11.5.11 SunOS TCP

We had many SunOS 4.1.3 and 4.1.4 sites in our study. We did not observe any differences between the two releases.

SunOS 4.1 appears to have been derived from BSD Tahoe, with the following differences:

1. If the MSS offered by the remote TCP peer is larger than that offered by the SunOS TCP, then it uses the larger value to initialize *cwnd*, though it still uses its own offered value for all subsequent *cwnd* calculations.

2. If it receives a series of acknowledgements for the same sequence number, if any of the acks is a *window recision* (that is, advertises a smaller window than did the previously-received acks), it simply ignores the ack. Other TCPs consider the window-recision ack as resetting the duplicate ack counter, delaying the possible onset of fast retransmission.

   We note that in our study the only window recisions we observed were due to packet reordering. No TCP ever originated an ack that rescinded a previously-offered window.

3. It will only enter fast retransmission for a packet that was not previously retransmitted using fast retransmission (circumstances under which this behavior manifests itself are rare).

4. Upon retransmission, when cutting *ssthresh* it does not round it down to a multiple of MSS, regardless of the type of retransmission.

11.5.12 VJ TCP

Two sites in our study ran experimental TCP implementations developed by Van Jacobson. `lbl` during $\mathcal{N}_2$ ran a version we term VJ_1 (in $\mathcal{N}_1$ it ran SunOS), and in both $\mathcal{N}_1$ and $\mathcal{N}_2$ `lbli` ran a version we term VJ_2. Though it differs from the numbering, VJ_2 is the earlier of the two versions. It behaves the same as our generic Reno implementation except:

1. it uses an additive constant of 4 bytes when updating *cwnd* during congestion avoidance, as opposed to MSS/8 (Eqn 11.2);

2. it does not exhibit the "fencepost" error when deflating the window (§ 11.5.1);

3. it does not cut *ssthresh* if a timeout retransmission occurs during a fast retransmission sequence;

4. it has a bug that leads to it always cutting *ssthresh* down to two segments upon any other timeout.

VJ_1 behaves like VJ_2 except it does not exhibit the header-prediction bug (§ 11.5.1) and it uses Eqn 11.1 to update the congestion window during congestion avoidance (no additive increment).

11.6 Receiver behavior of different TCP implementations

In this section we examine variations in how the different implementations behave as receivers of data: the policies used to acknowledge newly arrived data and the effects of these on performance and congestion. We begin with a discussion of how different implementations acknowledge in-sequence data, the "normal" case of a connection proceeding smoothly (§ 11.6.1). We find a number of different "policies" for choosing exactly when to generate acknowledgements. Some of these have surprisingly negative performance problems. We then look at how implementations acknowledge out-of-sequence data: packets coming above or below a sequence hole (§ 11.6.2). Finally, after characterizing the generation of gratuitous acks (§ 11.6.3), we finish with an analysis of *response delays*, namely, how long it takes a TCP receiver to generate its acknowledgements (§ 11.6.4). Variations in response times can introduce a significant *noise term* for senders that attempt to measure round-trip times (RTTs) to high resolution. One of our goals is to assess the viability of sender-only timing schemes.

11.6.1 Acking in-sequence data

When a TCP receives in-sequence data, it needs to eventually generate an acknowledgement for the data, so the sender knows it has been successfully received and can release the resources allocated for retaining the data in case it required retransmission. There is a basic tension between acknowledging data quickly versus waiting to see if more in-sequence data arrives so that a single ack can take care of acknowledging multiple data packets.[10] The more acks the receiver generates, the more network resources its feedback stream consumes; but also the more likely in the face of packet loss that enough acks will reach the sender that it will not retransmit unnecessarily, and the smoother the resulting stream of transmitted packets, since the window moves in numerous, small increments rather than rare, large increments.

TCPs need to assure that they acknowledge data quickly enough that the sender does not erroneously conclude a packet was lost and retransmit it. The TCP standard requires that acknowledgements be delayed no more than 500 msec, and either recommends or requires (§ 4.2.3.2 and § 4.2.5 of [Br89]) that a TCP acknowledge upon receiving the equivalent of two full-sized packets, that is, 2·MSS bytes (§ 9.2.2).

As discussed in § 11.4, `tcpanaly` associates the acks generated by a TCP with the data packet that prompted the TCP to send the ack, allowing determination of the acknowledgement delay. It also classifies acks into three categories, those for less than two full-sized packets ("delayed acks"), those for two full-sized packets ("normal acks"), and those for more than two full-sized packets ("stretch acks"). We expect: delayed acks to incur considerable delay as the TCP waits hoping for more data to acknowledge; normal acks to be commonplace in any connection with significant data flow, and to take little time to generate; and stretch acks to be rare. We now treat each in turn.

Delayed acks

In both $\mathcal{N}_1$ and $\mathcal{N}_2$, it was exceedingly rare to observe a delayed ack that took longer than 500 msec, on the order of one trace in 1,000.

[10] Or to see if the ack can piggyback on a data packet or window update being sent back to the sender.

All of the BSD- (i.e., Tahoe- and Reno-) derived implementations in Table XV use a delayed-ack timer of 200 msec, meaning that, except for truly unusual conditions (presumably when the host was very busy doing something else), they generate delayed acks within 200 msec of receiving the corresponding packet. These delays are furthermore evenly distributed over the range 0 msec to 200 msec, a consequence of the implementations using a 200 msec "heartbeat" timer. Every time the timer expires, they check to see whether new, unacknowledged data has arrived. If so, they generate an ack. The fact that the new data may have arrived at any point since the last heartbeat leads to the even distribution of the delays.

Linux 1.0 always immediately acknowledges newly arrived in-sequence data, so, by `tcpanaly`'s definition, *all* of its acks are delayed acks. It usually generates the ack within 1 msec.

Solaris TCP differs from the others in that it uses a 50 msec *interval* timer, scheduled upon the arrival of each packet, instead of a 200 msec *heartbeat* timer. As a result, the delay is generally very close to 50 msec (slightly lower, perhaps because the timer is scheduled before the packet filter timestamps the arriving data packet; cf. § 10.3.6), though it is a configurable parameter. One might think that a shorter delay would lead to better performance because the sender waits less before receiving the ack. We note, however, that, for certain link speeds, a low value such as 50 msec guarantees that every ack for in-sequence data will be a delayed ack, which is instead counter-productive because the sender winds up waiting *longer* for acks in terms of the delay required to acknowledge two packets. Suppose the delay timer is set for t seconds, the maximum data transfer rate the Internet path can support is ρ bytes/sec and the data packets have size b bytes. Then whenever:

$$t < b/\rho,$$

it is impossible that two full-sized data packets will arrive before the delay timer expires.[11] Consequently, the sender will wait an extra t seconds for the acknowledgements of every two packets. If $t = 50$ msec and $b = 512$ bytes, then if $\rho < 10$ KB/sec, the delay will be sub-optimal, leading to acking of every packet even if they arrive as fast as possible. This range includes the still-quite-common rates of 56 Kbit/sec and 64 Kbit/sec. If, however, $t = 200$ msec, then only for $\rho < 2.5$ KB/sec is the delay sub-optimal. This rate includes some of today's modems, but no other commonly used link technologies.

Finally, we temper this discussion by noting that the deficiency is fairly minor. Yes, a low delay timer results in extra ack traffic, and somewhat elevated RTTs. However, acks are small, so the additional traffic load is likewise small, and the additional latency is bounded by the small timer setting to an often-imperceptible value.

Normal acks

We term an ack "normal" if it is for two full-sized packets. Since our study concerns unidirectional bulk transfer, we expect that most of the time the receiving TCP will have plenty of opportunity to generate normal acks.

BSD-derived TCPs do *not* simply generate acknowledgements every time they receive two in-sequence, full-sized packets. Instead, they generate the acknowledgements when the receiving *application process* has *consumed* that much data, even if the data it consumed was actually delivered in earlier packets. This means that normal acks are not always promptly generated. We

[11] Well, almost impossible. See § 16.3.2.

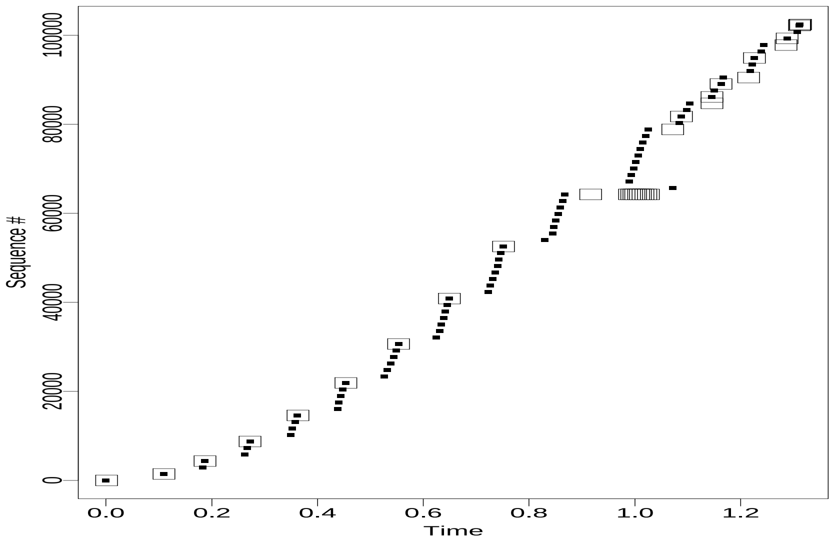

Figure 11.11: Sequence plot showing Solaris 2.4 acknowledgments (large squares) during initial slow-start

analyze the timing of their generation below in § 11.6.4. Here we simply note that quite frequently the delay in generation is very small, presumably because it takes little time for the application process to consume the new data.

Since Linux 1.0 TCP acks every packet, it does not generate normal acks, by `tcpanaly`'s definition of "normal." Solaris TCP generates normal acks after an initial slow-start sequence, but not before (see next section).

Stretch acks

Every implementation in our study except Linux 1.0 sometimes generates "stretch" acks, that is, acknowledgements for more than two full-sized packets, contrary to [Br89] (though they all came less than 500 msec after the last packet they were acknowledging). We suspect most of these occur because of delays in the application process consuming the newly arrived data (discussed above). For most implementations and sites, stretch acks usually were for no more than three full-sized packets.

Some implementations and sites, however, were especially prone to large stretch acks, particularly some of the IRIX sites. These instances, however, were intermittent (except for Solaris—see below): quite often, the site would not generate a significant number of stretch acks, other times it would. Most likely this intermittence reflects periods of heavy versus light load. The IRIX sites might be particularly prone because of some peculiarity of how the IRIX scheduler deals with heavy processor contention: if it delays competing processes for lengthy periods of time, this could easily translate into stretch acks. For example, we noticed that `adv` often generated stretch acks separated by almost exactly a multiple of 30 msec, and posit that 30 msec reflects the host's scheduling quantum.

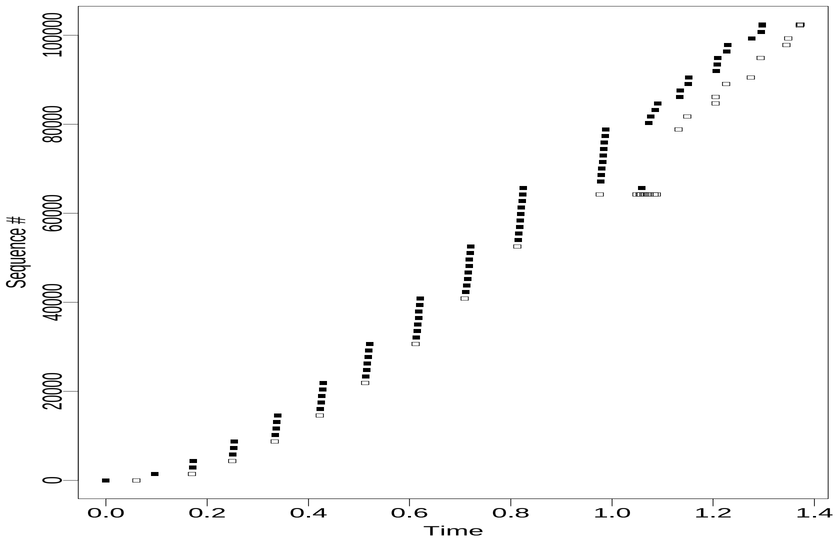

Figure 11.12: Corresponding burstiness at sender

Solaris TCP, however, generates stretch acks in quite a different manner. It apparently has been tuned so that, during the initial slow-start, it generates only one ack for each increasingly-large "flight" of packets. Figure 11.11 shows how this works, using a trace recorded at a Solaris receiver. Here, the acks are shown with large squares, since they lie directly on top of the end of each initial slow-start flight. The delay between the final packet of a flight and the corresponding ack is only 100's of μsec—much too small for timer-driven acking. Since the TCP appears to "know" exactly when each flight ends without waiting any appreciable time for additional packets, we conclude that it does indeed know: it predicts that each flight will be one packet larger than the previous flight (which is exactly the case during slow-start, if each flight elicits only one ack in reply), and counts exactly that many packets before acknowledging.

At around $T = 0.9$ a data packet was lost, and thus the prediction that 10 packets would arrive in that flight failed. The ack for the 9 packets that did arrive is sent when the delayed-ack timer expires, 49 msec after the last packet in the flight arrived. The packets liberated by this ack then arrive above the sequence hole and the TCP generates a series of duplicate acks in response, and the sending TCP retransmits the missing packet. Note that, after this point, the Solaris TCP gives up on trying to ack just once for each flight, and falls back on acking every three full-sized packets (in violation of [Br89]), or fewer if the delayed-ack timer expires before three arrive. This behavior also fits with our hypothesis that the TCP is predicting flights by counting slow-start cycles: once the connection is no longer in slow-start, the TCP cannot easily determine the size of the next flight, so it falls back on a less sparse acking policy.

It seems very likely that this acking behavior was developed in order to maximize throughput for local-area networks. We are led to speculate that this is the case, because the acking policy has four major drawbacks for wide-area network use, worth discussing in detail because at first blush one might find such a frugal ack policy attractive as apparently efficient and streamlined:

1. Because each ack advances the window by increasingly large amounts, the acking behavior

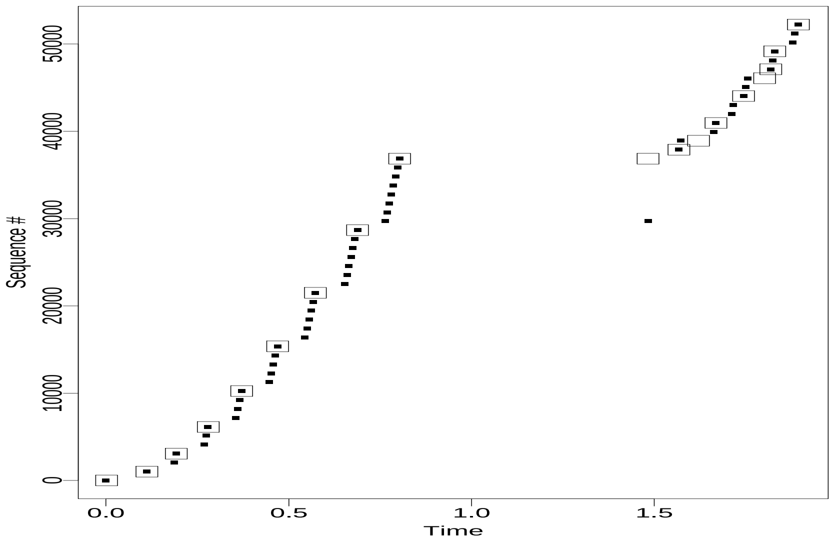

Figure 11.13: Sequence plot showing retransmission timeout due to loss of single Solaris 2.4 ack

leads to progressively burstier transmissions by the sender. Figure 11.12 shows the same trace as in Figure 11.11 except recorded at the sending TCP. We see increasingly taller "towers" of packets, sent at rates up to 1.15 Mbyte/sec, completely saturating the local Ethernet. While a local area network might be able to accommodate such burstiness, it can be very hard on a wide-area network, because it leads to rapid queue growth if the bottleneck bandwidth is significantly lower (in this connection, `tcpanaly` calculated it to be about 350 Kbyte/sec (evidently two T1 circuits), using the methodology discussed in Chapter 14). This queue variation then potentially perturbs all the other connections currently sharing the bottleneck link, by delaying their packets and perhaps causing *their* packets to be dropped.

Much better is for the packets to be spaced out more evenly, approaching the bottleneck bandwidth, which will happen naturally due to "self-clocking" (§ 9.2.5) if the receiving TCP generates acks at a quicker rate. See [BP95b] for a discussion of TCP sender modifications to achieve smoother spacing in the face of large ack advances.

2. Because only one ack is sent per round-trip time, the connection loses the usual benefit of exponential window-increase during slow-start. On the kth slow-start flight, the Solaris acking policy will lead to exactly k packets in flight. A policy of ack-every-packet, on the other hand, leads to 2^{k-1} packets in flight, an enormous difference when trying to fully utilize a network path with a large bandwidth-delay product.

3. Because only one ack is sent per round-trip time, the resulting connections are *brittle* in the face of packet loss, which is much more prevalent in wide-area networks than local-area networks. Since each flight of data elicits only one ack in response, if the ack is lost, then the data/ack "pipeline" *must* shut down with an expensive (in terms of performance) retransmission timeout, because the sender will not receive *any* more information about the data it sent. Figure 11.13 shows a trace recorded at a Solaris receiver in which this occurred.

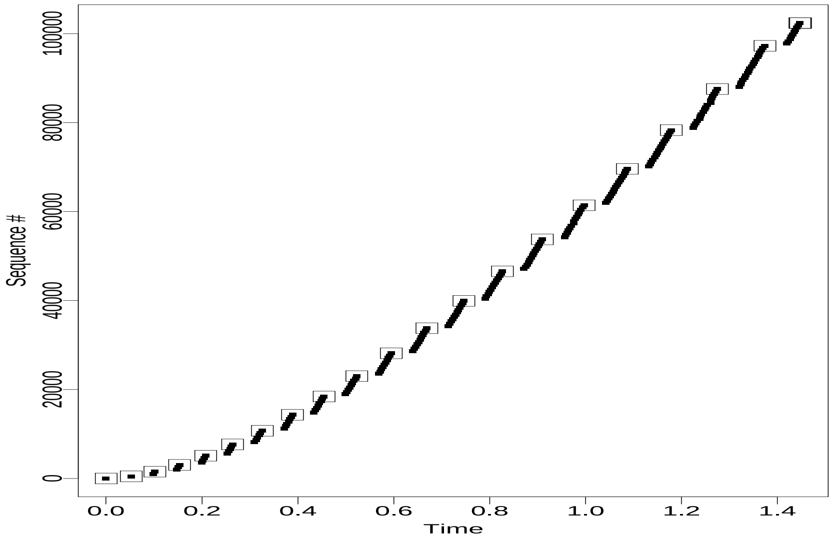

Figure 11.14: Receiver sequence plot showing lulls due to Solaris 2.3 acking policy

The slow-start progresses normally until about $T = 0.8$, at which point the lone ack for the 8th slow-start flight is lost. Even though none of the data packets were lost, the entire connection must shut down until a timeout about 700 msec later restarts the flow of data, and then proceeds on from this point at an unnecessarily reduced transmission rate, due to TCP congestion avoidance. With a more prolific acking policy, loss of the ack would have had no effect on the data flow whatsoever, since more data would have arrived shortly (liberated by acks for packets earlier in the flight) and elicited more acks in response, keeping the flow alive.

4. Finally, the Solaris acking policy is *provably sub-optimal* in the following sense. One of the goals of a solid implementation of a byte-stream transport protocol such as TCP should be that, in the absence of any competing network traffic, a transport connection should quickly reach a state in which it delivers packets to the receiving end continuously and at the available bandwidth. Yet, the Solaris acking policy cannot achieve this goal, even if we allow its linear slow-start window increase discussed above to qualify as "quickly."

   The fundamental problem is that, regardless of how large the slow-start flight grows, it always eventually comes to an end, at which point the Solaris TCP sends the sole ack for that flight. While that ack is traversing the network back to the sender, the sender is perforce doing *nothing*, because it has already sent its entire flight and cannot send any more data until an ack arrives to advance the window. Thus, the Solaris acking policy guarantees that a *lull* equal to the round-trip time will accommodate each flight of data. As long as the sender remains in slow-start, the receiver will *never* see a continuous stream of packets arriving at the available bandwidth!

   Figure 11.14 illustrates this problem. This connection has a RTT of about 44 msec, and a T1 bandwidth limit of about 170 Kbyte/sec. Thus, the connection's bandwidth-delay product

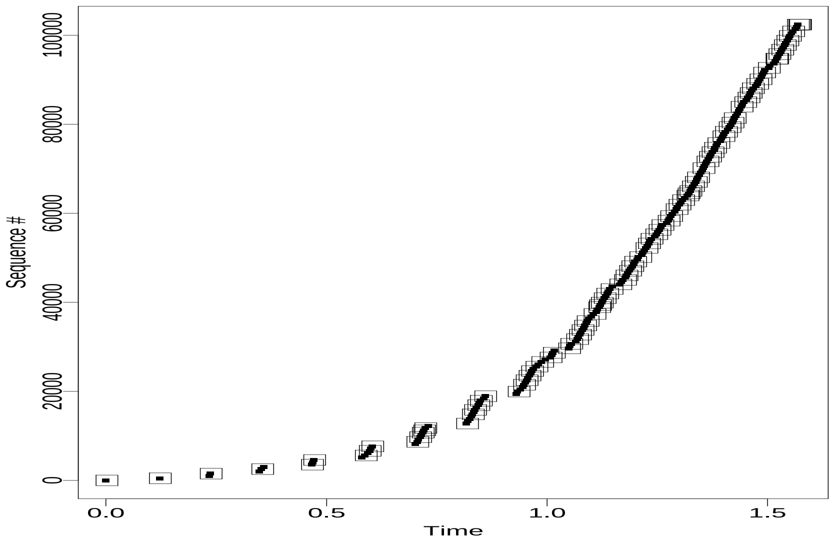

Figure 11.15: Sequence plot showing more frequent acking leading to "filling the pipe"

is about 8 Kbyte, so if the sending TCP has this much data in flight at one time, ordinarily that would suffice to "fill the pipe" and completely utilize the available bandwidth. Near the end of the connection, it has more than 8 Kbyte in flight, and yet *still* does not achieve full utilization, due to the 44 msec delays incurred at the end of each flight.

The only Solaris TCP in our study that did not exhibit this problem was `austr2`, because its bottleneck bandwidth of about 13 Kbyte/sec was so small that the delay ack timer (50 msec in Solaris) would often expire before the full flight could arrive.

Other acking policies avoid this problem because, by acking more often, they can ensure (for a large enough window) that the sender will have additional data already in flight by the time the current flight ends. As the window grows sufficiently large, the packets from this next flight will arrive closer and closer to the end of the first flight, until eventually the distinction between flights blurs and the connection settles into a continuous stream of arriving data packets. Figure 11.15 shows such a connection, with the same sender as in Figure 11.14. Note that this connection had a longer RTT than that shown in Figure 11.14, which explains why it happened to achieve only the same overall throughput, instead of higher throughput, which would have been the case for equal RTTs and a greater degree of "filling the pipe."

11.6.2 Acking out-of-sequence data

When a TCP receives a packet with out-of-sequence data, it either *must* generate an acknowledgement, if the data corresponds to data already acknowledged, which we term "below sequence"; or *should* generate an acknowledgement, if the data is for a sequence number beyond what has been previously acknowledged, which we term "above sequence" [Br89]. (These situations are also discussed above in § 11.4.1.) For example, suppose a TCP has received contiguous data up to sequence 10,000. If it now receives data with a sequence number below 10,000, then it *must* generate another acknowledgement for sequence 10,000. If, instead, it receives data starting at sequence number 11,000, then it *should* generate another acknowledgement for sequence 10,000.

In both cases, the acknowledgement generated is for the highest in-sequence data received. The reason for generating acks in the first case is that the sender has retransmitted unnecessarily and thus appears confused as to how much data the receiver has in fact received, so the receiver needs to inform the sender again of what it has received. The reason for generating acks in the second case is to enable "fast retransmit," discussed in § 9.2.7.

Of the TCPs in Table XV, only SunOS 4.1 exhibited unusual behavior when receiving out-of-sequence data. While it generally will immediately acknowledge below-sequence packets, it does not always do so, and it never immediately acknowledges above-sequence packets. Instead, it apparently checks upon each expiration of the 200 msec delayed-ack heartbeat timer whether any above-sequence (or, sometimes, below-sequence) data has arrived. If so, it generates a single duplicate acknowledgement reflecting its current upper-sequence limit.

One other form of "mandatory" ack not generated by SunOS 4.1 concerns the initial SYN packet used to begin establishing a TCP connection. SunOS 4.1 TCP appears to ignore retransmissions of the initial SYN once it has sent a SYN-ack, and instead continues retransmitting (upon timeout) the SYN-ack until it is acknowledged. This behavior has only minor implications concerning a possible delay in establishing connections when the first SYN-ack is lost.

Other than SunOS, all the implementations in our study tend to generate mandatory acknowledgements promptly (though we have observed more than 1 minute delays for a Solaris implementation while it waited for a sequence hole to be filled!). The few times `tcpanaly` detected a failure to send a mandatory ack were generally due to either vantage-point problems (§ 10.4), packet-filter resequencing errors (§ 10.3.6), or confusion caused by checksum errors.

The only other failure we observed with respect to generating mandatory acks is with Solaris 2.3 TCP. If it receives a packet containing only a FIN option (no data), and arriving above-sequence, then it simply ignores the packet. If the packet contains data, then it elicits a duplicate ack like any other above-sequence arrival, but the presence of the FIN bit is forgotten (so if the sequence hole is filled, the TCP will acknowledge all of the data but not the FIN). This behavior is fixed in Solaris 2.4, and is the only difference in behavior we observed between the two implementations.

11.6.3 Gratuitous acks

`tcpanaly` includes in its analysis checking for "gratuitous acks," meaning acknowledgements that as far as it could determine simply did not need to have been sent. These are quite rare. For example, only about 0.5% of the $\mathcal{N}_2$ receiver traces exhibited a gratuitous ack. SunOS 4.1 TCP is particularly apt to generate them; Figure 11.16 shows a typical gratuitous ack produced by this implementation. The acknowledgement at $T = 0.4$ is sent on the delayed-ack timer, because the TCP has received above-sequence data that it cannot directly acknowledge.[12] (As noted in § 11.4.1, SunOS 4.1 does not acknowledge each above-sequence packet.) The second ack, at time $T = 0.6$, appears completely unneeded. It was sent almost exactly 200 msec after the first ack in the plot, so almost certainly due to the delayed-ack timer. While the last data packet arrived shortly before the $T = 0.4$ ack was sent, we suspect is had not yet been *processed*, and its processing led the TCP to generate another ack the next time the delayed-ack timer expired. (So this example is really a

[12] This ack includes the same offered window as its predecessor; it was *not* sent in order to update the window.

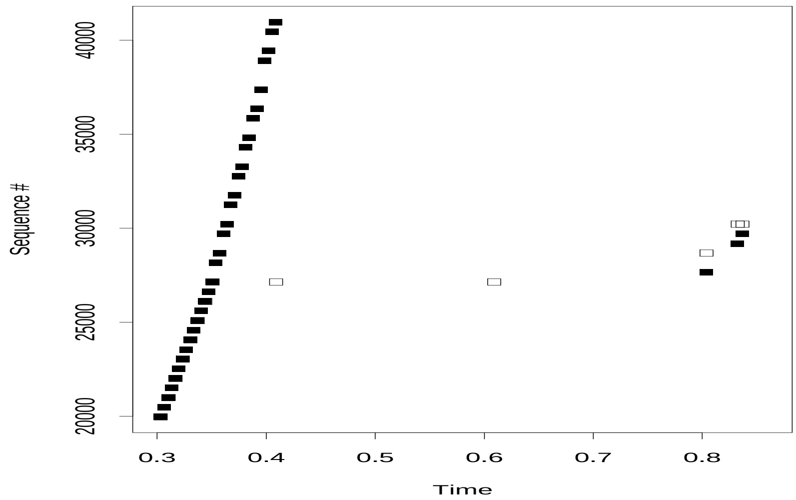

Figure 11.16: Sequence plot showing gratuitous acknowledgement

vantage-point problem, per § 10.4.)

`tcpanaly` can also become confused and falsely conclude a gratuitous ack was sent if the TCP takes a particularly long time to generate an ack, or if a checksum error confuses `tcpanaly`'s analysis of cause and effect. Figure 11.17 shows an example of the former, in which `tcpanaly` views the lower ack sent at $T = 1.28$ as gratuitous, even though it was really a response to an out-of-order packet 21,745 received shortly before the packet preceding it in sequence, around $T = 1.26$. Thus, it took the TCP in this example (HP/UX 9.05) more than 20 msec to generate the mandatory ack required by receiving an out-of-sequence packet, which in the presence of the earlier (likewise tardy) ack for the same sequence number at $T = 1.26$ sufficed to confuse `tcpanaly` as to why the second ack was sent.

One other form of gratuitous ack we observed occurs with Linux 1.0. It will generate an ack if 30 seconds have elapsed without any newly arriving packets. Presumably, this ack is intended to resynchronize the sender with the receiver in the face of a lull induced by the loss of previous acks.

11.6.4 Response delays

As discussed in § 9.1.3, there are a number of advantages to network measurement schemes that rely only on the ability to record packet timings at one of the two connection end-points. One of the main advantages is that it is logistically much easier to secure single-endpoint measurements than dual-endpoint. For example, TCP Vegas has as one if its central congestion control mechanisms an analysis of round-trip times measured by the TCP sender [BOP94]. The goal of these measurements is to infer how the sender's window changes are affecting the queueing delays in the network, i.e., how the sender's behavior is utilizing networking resources. As developed in [BOP94], the RTT timings central to the congestion control policy are made solely by the sender.

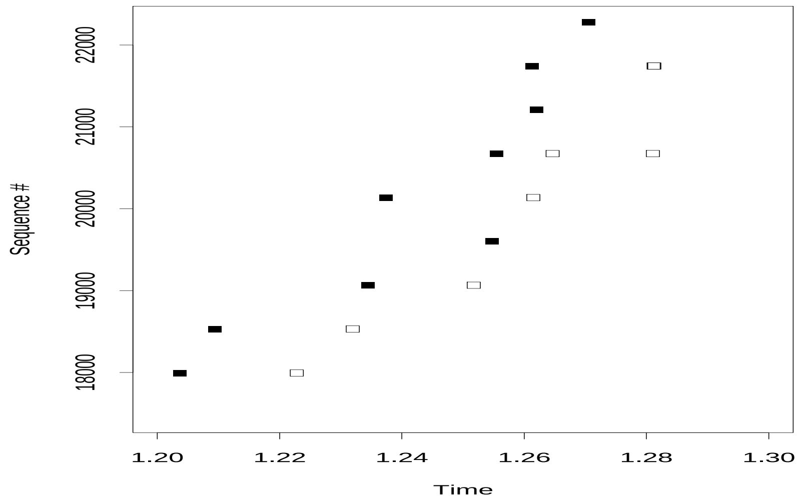

Figure 11.17: Sequence plot showing false gratuitous acknowledgement

Not needing to rely on cooperation by the receiver in making these measurements is a great boon, but it carries with it the risk of having to make control decisions based on considerably less precise measurements than could be obtained if the receiver cooperated.

In this section we look at the variation among a TCP's *response delays*, by which we mean how much time the TCP takes to generate an acknowledgement for new data it has received. We are interested in the *variation* because it *directly affects* the precision with which a sending TCP can measure round-trip time delays. If the receiving TCP exhibits large variations in the time it takes to generate acknowledgements, and if the sender has no way of factoring out these delays, then the sender must contend with considerable *noise* in its RTT measurements, perhaps enough to render impractical the accurate assessment of the network's state based on sender-only measurement.

As we argue elsewhere (Chapter 16), often what is of greatest interest is *variations* in networking delays rather than the absolute magnitude of the delays. Thus, we do not concern ourselves in this section with the *mean* time a TCP takes to generate an acknowledgement, as this contributes nothing to errors in measuring delay variation. Instead, we focus on the *variation* of the time taken to generate an acknowledgement.

Furthermore, we assume that the sender can eliminate one of the common sources of delay variation, namely delayed acks. These are easy to spot, because any time an ack is received that advances the window by less than two full-sized packets, the ack was potentially delayed, so RTTs derived from its arrival should not be trusted beyond the 200 msec of variation known to frequently attend delayed acks.

We also assume that acks generated for exceptional conditions such as out-of-sequence data are not of much interest, since they generally indicate that the sending TCP is about to enter an exceptional state (retransmission) anyway. Thus, we confine ourselves to the time taken by different TCPs to generate acks for two or more full-sized, in-sequence packets.

The maximum time taken by a TCP to generate a “normal” ack (§ 11.6.1) is almost always less than 200 msec and often less than 50 msec, no doubt reflecting the BSD and Solaris delayed-ack timer intervals. We have, however, observed values as high as 1.6 sec. (The mean time taken is less than 1 msec in about two thirds of our traces, and less than 10 msec in about 95% of our traces.)

One final important point is that to assess response time we compute the standard deviation (σ) of the response time, rather than using a more robust statistic (§ 9.1.4). We do so because we argue that a real-time sender-based measurement scheme will only be able to make fairly cheap assessments of delay variations, rather than employing robust statistics. Furthermore, even if the sender can afford to compute robust statistics on the packet timing measurements it gathers, it will still have the serious problem of discerning between "outliers" due to receiver delays versus those due to genuine networking effects. Thus, we argue it is reasonable to assess delay variations in terms of standard deviation, even though we know this estimator can be seriously misleading in the presence of occasionally quite large, exceptional values.

In assessing both $\mathcal{N}_1$ and $\mathcal{N}_2$, we find that about two thirds of the time σ calculated for the response time is below 1 msec. These cases are good news for sender-based measurement, since often clock resolutions are not appreciably more accurate than 1 msec anyway (§ 12.4.2). However, the mean value for σ was about 5 msec, and for the one-third of the traces with $\sigma > 1$ msec, the mean climbs to 15 msec.

There is a great amount of site-to-site variation among the average values of σ, no doubt reflecting large variations in average site-to-site load. For example, `adv`, an IRIX system, has an average value of σ just under 1 msec, while `bnl`, another IRIX system, has an average value of over 5 msec.

We conclude that, for high-precision, sender-only RTT measurement, the ack response delays will often not prove an impediment; but sometimes they will, meaning that the intrinsic measurement errors will be large enough to possibly swamp any true network effects we wish to quantify. Here, "often not" is roughly 2/3's of the time, "sometimes they will" is 1/3 of the time, and "large enough" is on the order of 15 msec. Naturally, the point at which the noise impairs measurement and control depends on the particular time constants associated with the connection, and with what information the TCP wishes to derive from its measurements.

11.7 Behavior of additional TCP implementations

Our analysis of TCP behavior above revealed two implementations with particularly significant problems: Linux 1.0 and Solaris (2.3 and 2.4). These implementations were both written independently of any of the others. Furthermore, of the *15* other implementations we studied, none of which exhibited problems of the same magnitude as these two, *all* were derived from a common implementation, the BSD Tahoe/Reno releases. Thus, we find a striking dichotomy between those TCP implementations exhibiting serious problems, and those that do not: the former were written independently, the latter built upon the Tahoe/Reno code base.

We interpret this difference as highlighting the fact that *implementing TCP correctly is extremely difficult.* The Tahoe/Reno implementations benefited from extensive development and testing by a host of TCP experts. Furthermore, they were the code base used by Jacobson to implement the algorithms in his seminal paper on TCP congestion behavior [Ja88].

However, to test our hypothesis that implementing TCP independently is difficult and fraught with error, we need to examine other independent implementations. To do so, we gathered

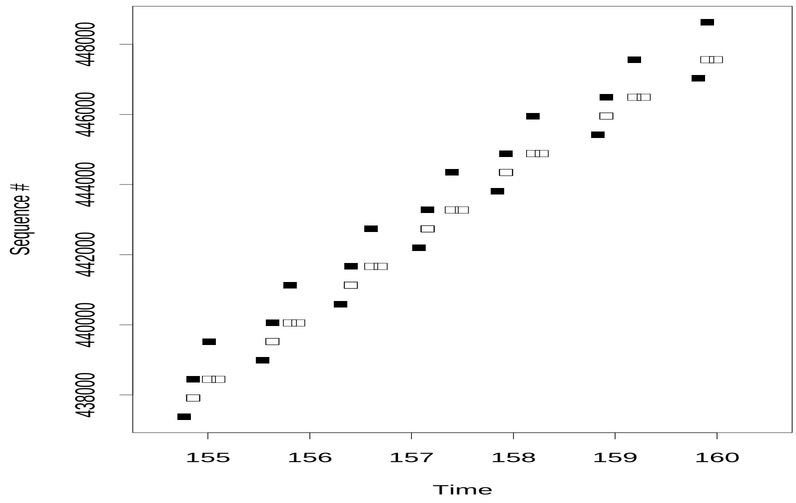

Figure 11.18: Sequence plot showing Windows 95 TCP transmit problem

`tcpdump` traces[13] of three additional TCPs: Windows NT, Windows 95, and Trumpet/Winsock, all implementations for personal computers.[14]

We analyzed these traces by studying sequence plots of their behavior. We did not integrate them into `tcpanaly` because we had only a handful of traces to study. These sufficed, however, to find some interesting behavior.

11.7.1 Windows NT TCP

We inspected four traces of Windows NT TCP, two of it sending data and two of it receiving data. We found no serious problems. It does not do fast retransmit, but this only impedes its own performance; it does not affect network stability (if anything, it abets stability). The only unusual aspect of its behavior we found is that its congestion window during its initial slow-start begins at 2 packets instead of 1. This could be a calculated decision to improve initial performance, or a bug due to treating the ack that completes the three-way SYN handshake establishing the connection as opening the congestion window.

11.7.2 Windows 95 TCP

We obtained only two traces of Windows 95 TCP, one of it sending data and one of it receiving. The sending trace exhibited a striking performance problem: often when it could send out two packets, only the second appeared to have been sent, and the first would subsequently be

[13] Many thanks to Kevin Fall for undertaking the measurement of these.

[14] We have subsequently been informed that the Windows NT and Windows 95 TCPs are in fact the same implementation. We observed different, but not inconsistent, behaviors between them, as noted below. In particular, the Windows 95 behavior that we did not observe in Windows NT may be due to the particular software/hardware combination used when obtaining the Windows 95 traces, which differed from that used to obtain the Windows NT traces.

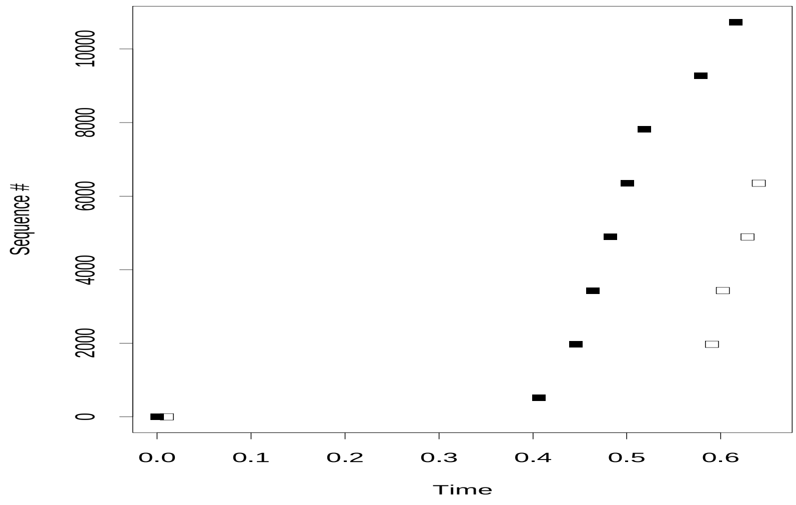

Figure 11.19: Sequence plot showing Trumpet/Winsock TCP skipping initial slow start

sent via timeout "retransmission." Figure 11.18 shows this problem. A pattern of one-ack, two-acks, one-ack, two-acks repeats. The first ack (such as the one a bit before $T = 155$) reflects a timeout retransmission filling a sequence hole. The congestion window is evidently one packet at this point. The TCP sends a single packet and this is acknowledged about 150 msec later. It then apparently sends not the next in-sequence packet, but the one after that. Receiving this out-of-sequence packet elicits a dup ack from the remote TCP, but only one, after which no more acks arrive. The sending TCP thus times out and retransmits the packet it should have sent in the first place, and the cycle repeats. Eventually it breaks out of the cycle, by sending two back-to-back packets when called for by a newly-received ack.

We suspect the problem is that the TCP *is* sending both packets, but the first is frequently being dropped by the network interface card, perhaps because the second arrives too closely on its heels. This would explain why the problem is sporadic, and also why it may have gone unnoticed during development of the TCP. Certainly, if this problem is widespread, then Windows 95 TCP users suffer from very poor performance. Since the retransmission problem lies wholly within the sending host, however, it does not threaten network stability in any way.

11.7.3 Trumpet/Winsock TCP

The last independently implemented TCP we studied was Trumpet/Winsock. We obtained 13 traces of its behavior. Some were made with version 2.0b and some with version 3.0c. We did not detect any difference in behavior between the two, even though the release notes of 3.0c indicate it fixed a retransmission problem with version 2.

The first problem Trumpet/Winsock TCP exhibits it that it *skips the initial slow start*. Figure 11.19 illustrates this behavior. The connection is established just after $T = 0$. The TCP waits 400 msec and then dumps 6 packets of 1460 bytes (except the first, which is 512 bytes)

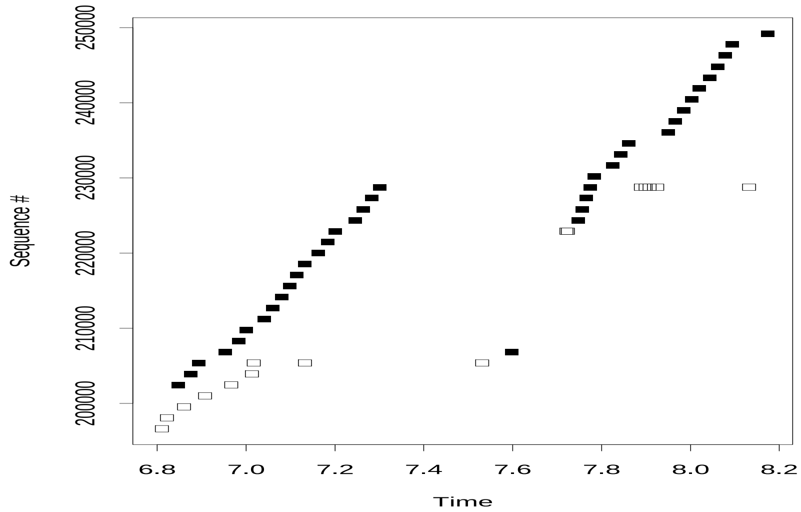

Figure 11.20: Sequence plot showing Trumpet/Winsock TCP skipping slow start after timeout

without waiting for any acknowledgements. When the first ack arrives, the window simply slides and more packets go out. Over time the window opened to 9 packets.

It further *skips slow start after timeout retransmission.* Figure 11.20 illustrates this behavior. At $T = 7.6$, a packet times out and is retransmitted. When an acknowledgement for it and a number of other successfully received packets is received, the TCP sends another 8 packets, and when an ack for the first four of these arrives (along with dups), another 9 are sent! (We observed similar behavior even if the ack for the retransmitted packet only acknowledged a few packets beyond it.) We did also observe some apparent slow-start sequences after retransmission timeouts (though duplicate acks received during this sequences advanced the congestion window), indicating that the *notion* of entering slow start after timeout is present in the implementation, but incorrectly implemented. These sequences had one other unusual aspect, which is that they began with the transmission of a packet followed 10 msec later by a retransmission of that same packet.

We are, unfortunately, not yet finished with cataloging Trumpet/Winsock TCP's implementation flaws. Figure 11.21 shows the TCP's acking policy. The trace was recorded at a Trumpet/Winsock receiver of a bulk transfer. The only acks it sent are those shown distinctly in the plot—none were sent shortly after a data packet arrived. The acking is clearly entirely timer-driven, incurring similar performance implications as for Solaris (§ 11.6.1), except it always acks in this fashion, rather than just during the initial slow-start, and it is acking off of a timer rather than when it knows no more data is in flight.

The final implementation flaw we found in Trumpet/Winsock TCP is that it *discards any above-sequence data it receives*. Figure 11.22 shows this surprising deficiency. Again, the trace was captured at the Trumpet/Winsock side of a connection in which the TCP was receiving a bulk transfer. Shortly after $T = 18.5$, a sequence hole forms due to a packet having been dropped by the network. 13 more packets follow, all arriving safely, yet the TCP does not generate any duplicate acks indicating their reception. Furthermore, when the lost packet is finally retransmitted

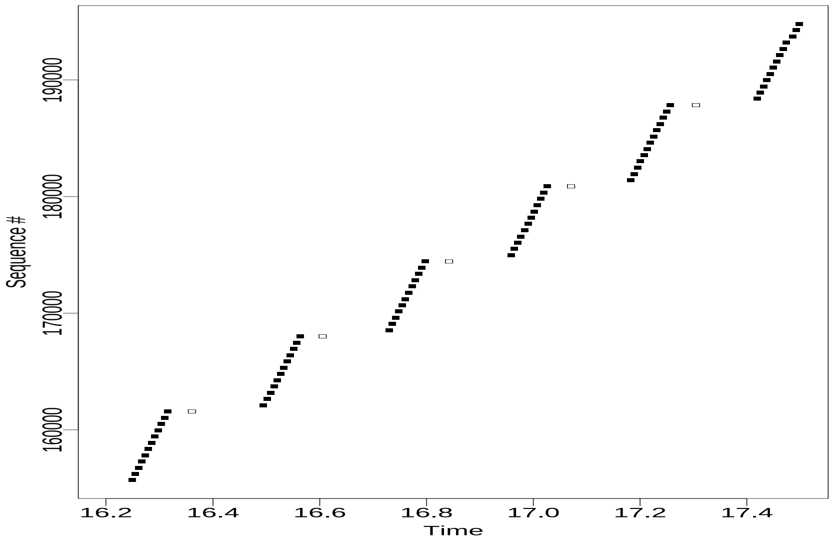

Figure 11.21: Sequence plot showing Trumpet/Winsock timer-driven acking

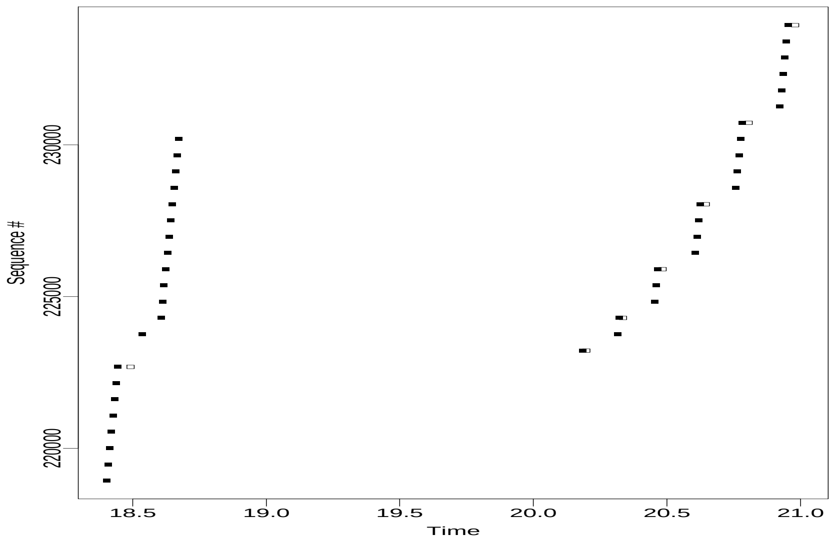

Figure 11.22: Sequence plot showing Trumpet/Winsock failure to retain above-sequence data

due to a timeout, we find it does *not* fill the hole previously created, which would lead to the TCP acknowledging both it and the 13 previously received packets. Instead, only it is acknowledged, and, as additional packets (already safely received) are retransmitted, they too form the limit of the acknowledged data.

Thus, the TCP has *thrown away* all of the additional packets it received above the sequence hole. As noted in § 13.3, this pattern of behavior is possible when a CSLIP link generates a "burst" of checksum failures. When we first observed this behavior, we presumed that was what had happened. However, we[15] subsequently gathered full packet traces (no *snaplen* limitation on the amount recorded for each packet; cf. § 10.2) and enabled `tcpanaly`'s checksum testing (§ 11.2) to determine whether the data packets were received uncorrupted. They were, indicating that the TCP could have kept them but instead discarded them. Furthermore, we *never* observed the TCP generating a duplicate ack upon receiving a packet above a sequence hole, nor acting as though a retransmission had filled a sequence hole.

All of these behaviors have strong, adverse impacts on network stability. Skipping slow start initially and after loss means that Trumpet/Winsock data transfers can present heavy bursts of traffic to the network when it lacks the resources to accept them. It violates [Br89]. Acking only when a timer expires can lead to excessive, unnecessary retransmissions when a single ack for many packets is dropped by the network. This also violates [Br89]. Finally, discarding successfully-received above-sequence data wastes network resources as the other TCP must resend all of the data again. This behavior, while strongly discouraged by [Br89, § 4.2.2.20], is not strictly forbidden, presumably to avoid indefinitely tying up resources in the receiving TCP in cases where connectivity is lost with the sender.

[15] Thanks again to Kevin Fall.

Chapter 12

Calibrating Pairs of Clocks

In this chapter we tackle the difficult problem of calibrating the accuracy of packet filter timestamps. “Wire times,” as defined in § 10.1, lie at the heart of much of our study, and the packet filter timestamps are the only means we have for estimating wire times. Yet, we have no independent means of verifying that the timestamps reported by the packet filters are indeed accurate. We must instead develop self-consistency techniques for calibrating the timestamps against themselves. For the most part, we are successful in doing so.

Undetected clock errors can result in serious systematic errors in our analysis of network dynamics, since superficially a clock error is indistinguishable from variations in packet transit times. These latter variations occur all the time due to queueing in the network, and we are interested in accurately analyzing them.

We begin by defining in § 12.1 basic terminology for describing the different clock attributes of “resolution,” “offset,” “accuracy,” and “skew.” We next discuss in § 12.2 why we did not require the clocks in our study to be synchronized, and how, if we had, use of the popular Network Time Protocol (NTP) would not necessarily have eliminated clock problems. Since the clocks at the connection endpoints lacked synchronization, we introduce in § 12.3 “relative” counterparts of “offset,” “accuracy” and “skew,” for discussing potential disagreements between two network clocks.

We then turn to methods for assessing clock resolution and relative clock accuracy (§ 12.4, § 12.5); detecting clock adjustments (§ 12.6), in which a clock quickly jumps or skews forward or backward because it is being set to a new absolute time; and detecting relative clock skew (§ 12.7). Clock adjustments and skew can introduce large, artificial network “dynamics,” so it is particularly important to detect and remove these effects.

We finish in § 12.9 with a look at how well a clock's synchronization correlates with stable clock behavior (lack of adjustments and of skew). We show that, unfortunately, a high degree of synchronization between two clocks does not necessarily mean that the clocks are free of relative errors.

12.1 Basic clock terminology

In this section we define basic terminology for discussing the characteristics of the clocks used in our study. The Network Time Protocol (NTP; [Mi92a]) defines a nomenclature for discussing clock characteristics, which we will use as appropriate. It is important to note, however, that the main goal of NTP is to provide accurate timekeeping over fairly long time scales, such as minutes to days, while for our purposes we are concerned with much shorter-term accuracy, namely between the beginning of a network transfer and its end. This difference in goals sometimes leads to different definitions of terminology, as discussed below.

12.1.1 Resolution

A clock's *resolution* is the smallest unit by which the clock's time is updated. It gives a lower bound on the clock's uncertainty. (Note that clocks can have very fine resolutions and yet be wildly inaccurate.) *It is crucial that this uncertainty be propagated when deriving estimates of network properties from timestamps produced by the clock.*

Note that we define resolution relative to the clock's reported time and not to true time, so for example a resolution of 10 msec only means that the clock updates its notion of time in 0.01 second increments, not that this is the true amount of time between updates.

12.1.2 Offset

We define a clock's *offset* at a particular moment as the difference between the time reported by the clock and the "true" time as defined by national standards. If the clock reports a time T_c and the true time is T_t, then the clock's offset is $T_c - T_t$.

12.1.3 Accuracy

We will refer to a clock as *accurate* at a particular moment if the clock's offset is zero, and more generally a clock's *accuracy* is how close the absolute value of the offset is to zero. For NTP, accuracy also includes a notion of the frequency of the clock; for our purposes, we split out this notion into that of *skew*, because we define accuracy in terms of a single moment in time rather than over an interval of time.

12.1.4 Skew and drift

A clock's *skew* at a particular moment is the frequency difference (first derivative of its offset with respect to true time) between the clock and national standards.

As noted in [Mi92a], real clocks exhibit some variation in skew. That is, the second derivative of the clock's offset with respect to true time is generally non-zero. [Mi92a] defines this quantity as the clock's *drift*. We in general will only talk about this notion in terms of clock *adjustments*, during which the clock's time is rapidly altered, because during the small time scales of interest for our study, only large drift values have discernable effects.[1]

[1] We will see in § 12.7 that, for the time scale of a single TCP connection in our study, relative clock skew is nearly always very close to linear, indicating near-zero relative drift over small time scales.

12.2 Lack of synchronized clocks

When designing the Network Probe Daemon (NPD) experiment, we made an early decision not to require synchronization between the clocks at the participating NPD sites. There were two reasons for this decision. First, one of the most important requirements of the experiment was to enlist as many participating sites as possible, in the quest for obtaining plausibly representative results. It was felt that requiring sites to install clock synchronization as well as bring up the measurement daemon would significantly add to the burden of participating in the study.

Furthermore, it is not clear that requiring clock synchronization would help in the measurement analysis. The main reason why it might not is because the most common form of clock synchronization used by Internet hosts is the Network Time Protocol (NTP). Use of NTP for the NPD experiment has two important shortcomings. First, NTP's accuracy depends in part on the properties (particularly delay) of the Internet paths used by the NTP peers, and these are exactly the properties that we wish to measure, so it would be less than completely sound to use NTP to calibrate our measurements. Second, NTP focuses on clock *accuracy*, which can come at the expense of short-term clock skew and drift. For example, when a host's clock is indirectly synchronized via NTP to a time source, if the synchronization intervals occur infrequently, then the host will sometimes be faced with the problem of how to adjust its current, incorrect time, T_i, with a considerably different, more accurate time it has just learned, T_a. Two general ways in which this is done are to either immediately set the current time to T_a, or to adjust the local clock's update frequency (hence, its skew) so that at some point in the future the local time T_i' will agree with the more accurate time T_a'. (We will see examples of both of these in § 12.7.)

A key point is that, for the NPD experiment, we are much more interested in correctly estimating *differences* between two timestamps than with the correctness of individual timestamps. That is, we care much more about clock skew than clock accuracy, because it is the differences that measure network delays. So, given a choice, we would prefer to buy very low clock skew at the expense of diminished clock accuracy, but NTP makes the opposite trade-off. In this respect, we prefer to synchronize the clocks *a posteriori* as we do here, after having completed the measurements.

In the future, it may be possible to obtain highly accurate clock synchronization via a mechanism separate from using the network itself; for example, GPS (Global Positioning System) receivers. That would allow us to have both accuracy and very low skew, which would be ideal for network measurement. Unfortunately, obtaining such separate synchronization today remains rare, so it behooves us to see how much use we can make of unsynchronized or NTP-synchronized clocks.

Finally, one might hope that a highly accurate clock will have very low skew, because if it had high skew it would not tend to be highly accurate. In § 12.9 we briefly investigate the degree to which this held for the closely-synchronized hosts, and find that it is only somewhat true. We also briefly argue in that section that, even with separate synchronization such as GPS receivers, sound measurement still calls for calibrating the timestamps.

12.3 Terminology for comparing clocks

A fundamental part of our experimental design was to arrange to record packet departures and arrivals at *both* ends of the end-to-end TCP connections between the NPD hosts. Doing so is crucial for discriminating between network conditions on the forward path, in which the data packets flowed, and the reverse path, over which only the receiver's acks flowed (since the TCP transfers were unidirectional). While recording packets at only one of the connection's endpoints is logistically much easier, analyzing network effects then becomes much more difficult, because the forward and reverse path become deeply intertwined.

Tracing packets at both ends, however, immediately raises questions about how to compare the timestamps produced by the packet filters at the two endpoints. In this section, we develop terminology for discussing differences between the two clocks producing the timestamps. The definitions are, for the most part, analogous to those in § 12.1, except that, instead of comparing a single clock against "true" time, we are comparing one clock against another.

We first introduce the meta-notation of a subscript "s" denoting time measured at the TCP *sender*, and "r" denoting time at the TCP *receiver*. Because our transfers are unidirectional, data flows only from the sender to the receiver, and acks flow from the receiver to the sender. Let C_s and C_r refer to the clocks at the sender and receiver, and R_s and R_r their respective resolutions.

We define C_r's offset relative to C_s at a particular true time T as $T_r - T_s$, that is, the instantaneous difference between the readings of C_r and C_s at time T. For convenience we will sometimes refer to this as C_r's relative offset at time T, with C_s implicitly being the clock to which C_r is compared.

Similarly, C_r's relative skew is the first derivative of C_r's relative offset with respect to true time. Since we lack an independent means of measuring true time, we can only estimate C_r's relative skew in terms of time as measured by either C_s or C_r. See § 12.7 for further discussion.

If C_r is accurate relative to C_s (their relative offset is zero), then we will refer to the pair of clocks as "synchronized." Note that clocks can be highly synchronized yet arbitrarily inaccurate in terms of how well they tell true time. This point is important because, for the analysis of our measurements, synchronization between C_s and C_r is more useful than the absolute accuracy of the clocks. The same is somewhat true of skew, too: as long as the absolute skew is not too great (§ 12.7.9), then minimal relative skew is more important, as it can induce systematic trends in packet transit times measured by comparing timestamps produced by the two clocks. In addition, since we lack an independent time standard in our study, we have no general way of assessing absolute skew, only relative skew.

These distinctions arise because what is often most important for our measurements are *differences* in time as computed by comparing the timestamps from the two clocks. The process of computing the difference removes any error due to clock inaccuracies with respect to true time; but it is crucial that the differences themselves reflect good approximations to differences in true time.

For *resolution*, what we care about is not "relative resolution" but *joint resolution*, which we define as $R_{s,r} = R_s + R_r$. This definition reflects the fact that, when comparing timestamps from C_s with those from C_r, the corresponding uncertainties must be *added* to properly propagate the resulting total uncertainty.

While the presence of generally-unsynchronized clocks in our study presents a number of measurement headaches, it also provides an opportunity for detecting certain types of clock errors—namely adjustments and skew—that sometimes cannot be determined at all when analyzing timestamps produced by a single clock. We delve into methods for detecting such errors in detail in the subsequent sections.

12.4 Assessing clock resolution

All of the computers participating in our study ran some variant of the Unix operating system. Unix defines a data structure for recording timestamps that has two fields, one for how many seconds have elapsed since a particular epoch, and one for how many microseconds have elapsed since the beginning of the current second. Thus, timestamp resolution is never better than 1 μsec. It can be much worse.

The basic idea behind estimating the resolution of the packet filter timestamps produced by the clocks in our study is to examine consecutive timestamps to determine the smallest difference between them. Unfortunately, Unix systems differ on how they report the time on subsequent calls during which the (digital) clock has not advanced. Some systems simply return the same unchanged time as given for previous calls. These are easy to detect, by disregarding timestamp differences of zero when determining clock resolution.

Others Unix systems add a small increment to the reported time to maintain monotone-increasing timestamps. We will refer to these adjustments as *monotonicity increments*. For such systems, we do *not* want to consider monotonicity increments when evaluating the clock's resolution, since they are artifacts of a more coarse resolution. Such systems generally increase the clock by 1 μsec to maintain monotonicity, but we cannot simply disregard timestamp differences of exactly 1 μsec, because it is possible that other processes running on the same machine (or even the packet filter, when discarding unwanted traffic) have queried the clock multiple times, making the increase n μsec. We proceed by hoping that occasionally n is small (in particular, $n < 5$), so that, if we observe a very small, positive timestamp difference, then we can infer that the system uses monotonicity increments.

12.4.1 Method for assessing resolution

Taking these considerations into account, we use the following method for estimating the clock resolution $\widehat{R}$:

1. Let $T_i, 0 \leq i \leq n$ be the ith packet filter timestamp, given $n+1$ successive timestamps.
2. Let $\Delta T_i = T_i - T_{i-1}, 1 \leq i \leq n$, the differences between successive timestamps.
3. If any ΔT_i is less than zero then the timestamps exhibit *time travel*, and the timing is untrustworthy (§ 10.3.7).
4. If any ΔT_i is greater than zero but less than 5 μsec, then set $\widehat{R}'$ to the smallest ΔT_i greater than 100 μsec.
5. Otherwise, set $\widehat{R}'$ to the smallest ΔT_i greater than zero.

This method either produces $\widehat{R}'$, an initial bound on the clock resolution, or the determination that the timestamps are polluted by time travel. If the former, we then form our estimate $\widehat{R}$ as $\widehat{R}'$ rounded to two decimal digits.[2] The rounding is primarily to introduce a reminder that $\widehat{R}$ is only a rough

[2] The exact algorithm used by `tcpanaly` is slightly more complicated. It executes the above algorithm "on the fly," for historical reasons. To minimize computation, `tcpanaly` only decreases $\widehat{R}'$ if a new value is at least 2.5% smaller than the best value so far.

estimate, and not to be taken too exactly. It is also useful for ensuring that a resolution like 10 msec is expressed as such, rather than 9.999 msec, as can happen if two timestamps differ by slightly less than 10 msec because of a monotonicity increment.

Note that this computation of $\widehat{R}$ produces at best an upper bound on R, the clock's true resolution, because it may happen that the packet filter never receives back-to-back packets as little as R seconds apart. For our purposes, this inaccuracy is acceptable, because the extra error introduced is conservative in the sense that it only widens the uncertainties we associate with our timing analysis.

12.4.2 Results of assessing resolution

`tcpanaly` uses the method outlined in the previous section to estimate the timestamp resolution of each trace it analyzes. We would hope to always observe roughly the same value for each particular packet filter, since a computer clock's resolution changes only very rarely (due to a hardware or perhaps operating system upgrade). This is indeed the case. Here we summarize the resolutions of the timestamps returned by the different packet filters.[3]

Three of the systems, `oce`, `ucol` (during $\mathcal{N}_1$), and `xor`, always had an estimated resolution of 10 msec. Their operating systems were IRIX 4.0, SunOS 4.1.3, and Solaris 2.3. A number of other sites running these operating systems also participated in the study, all with finer resolutions, so the limitations must be due to either hardware constraints or user configuration, rather than being fixed by the operating systems. We did not further investigate the hardware differences, as our primary interest is in accurately estimating a packet filter's timestamp resolution, and not the details of why the resolution is what it is.

The coarse 10 msec resolution proves problematic during our later analysis, because it makes it difficult to resolve, for example, bottleneck bandwidths with any sort of precision. We address this difficulty in § 14.7.

One system, `sandia`, also running IRIX 4.0, always had an estimated resolution of either 1 msec or 990 μsec.

All of the Digital Unix OSF/1 systems (`harv`, `mit`, `umann`, `ucol` in $\mathcal{N}_2$) always had a resolution of 980 μsec or 970 μsec, which matches a clock advance of $2^{10} = 1,024$ ticks/sec.

Some of the SunOS (`nrao`, `umont`, `unij`) and BSDI (`austr`, `rain`) always had resolutions $\geq$ 200 μsec, while other SunOS and BSDI systems had finer resolutions, again suggesting hardware differences or user configuration.

Of the remainder, all exhibited resolutions finer than 200 μsec, though not in every trace. The median resolutions over all of the traces were almost always in the 10-300 μsec range. This turns out to be ample for our purposes.

Finally, we note that estimates based on packet traces from a given host H *receiving* a unidirectional data transfer tend to be slightly larger (more coarse) than those from traces of H *sending* the data. The difference is on the order of 3–25%. It can be understood in terms of the overestimation effect discussed in the previous section, namely that, if the packet filter never sees back-to-back packets with a spacing equal to the clock resolution, then `tcpanaly` has no opportunity to accurately estimate the resolution. A TCP sender will often send two packets back-to-back as

[3] Recall that some NPD sites used a separate computer for monitoring the NPD traffic (Table XIV). All of the analysis in this chapter concerns the clock of the host used in *tracing* the traffic, as that is the only clock relevant to our subsequent analysis.

the window slides or the congestion window opens (§ 9.2.2), and these then provide an opportunity to observe minimally-spaced timestamps. TCP receivers, on the other hand, receive these packets spaced out by the bottleneck bandwidth (Chapter 14), generally well above the clock resolution. Furthermore, most implementations will wait to send an ack until the receiving application has read at least two packets' worth of data (§ 11.6.1), which will entail extra delay, perhaps more than the clock's true resolution.

12.5 Assessing relative clock offset

In this section we discuss how to estimate the relative offset between two network clocks. The closer the offset is to zero, the greater the relative clock accuracy (degree of synchronization). For our purposes, estimating relative offset is not crucial to our subsequent analysis of network dynamics. We only need to do so in order to construct legible plots of the two-way flow of packets and acks, and to qualitatively investigate the relationship between large relative offset and other clock problems such as relative skew. Accordingly, we are satisfied with the method developed in this section even though it is not highly accurate.

12.5.1 Method for assessing relative offset

Let ΔT_{p_s} be the time required to send a packet p_s from host s to host r. In general, we refer to this time as the "one-way transit time" or "OTT." Suppose p_s is sent from s with a timestamp T_s from s's clock, and it is received at r with at local timestamp T_r. If the clock C_r were perfectly synchronized with C_s, then we would have $\Delta T_{p_s} = T_r - T_s$ (providing C_r and C_s have no skew with respect to true time).

More generally, if the relative offset between C_r and C_s is $\Delta C_{r,s}$, then we have:

$$\Delta T_{p_s} \quad = \quad T_r - T_s - \Delta C_{r,s},$$

and hence:

$$\Delta C_{r,s} \quad = \quad T_r - T_s - \Delta T_{p_s}. \tag{12.1}$$

Unfortunately, we do not know ΔT_{p_s}, so we cannot use this equation to determine $\Delta C_{r,s}$. But we can *estimate* ΔT_{p_s} and then use that estimate to estimate $\Delta C_{r,s}$ as follows. First, define:

$$\Delta \widetilde{T}_{p_s} = T_r - T_s, \tag{12.2}$$

that is, the "raw" difference in the timestamps for packet p_s's trip through the network. Thus, $\Delta \widetilde{T}_{p_s}$ differs from ΔT_{p_s} by only a constant; in particular, the constant we wish to estimate. We can then rewrite Eqn 12.1 as:

$$\Delta C_{r,s} = \Delta \widetilde{T}_{p_s} - \Delta T_{p_s}. \tag{12.3}$$

In general, ΔT_{p_s}, and hence $\Delta \widetilde{T}_{p_s}$, depends on both network conditions and the size of packet p_s. We have little control over the size of p_s, because for a unidirectional transfer it is almost always large for packets from the sender to the receiver (the exception being the SYN and FIN handshake packets that delimit the connection, and the occasional very small data packet sent due to buffer boundary mismatches), and always small for the acks sent in the reverse direction. We can, however, attempt to control for network conditions, by selecting the *minimal* observed $\Delta\widetilde{T}_{p_s}$. (Here we are applying the assumption that minima occur during times when the network is unloaded.) Selecting the minimal value works because (most) network-induced noise is *additive* and *positive* (§ 12.6.2). Term the minimal value $\delta\widetilde{T}_{p_s}$.

Similarly, we compute $\delta\widetilde{T}_{p_r}$ for the acks sent in the opposite direction. Since $\Delta C_{r,s} = -\Delta C_{s,r}$, we expect to find $\delta\widetilde{T}_{p_s} \approx -\delta\widetilde{T}_{p_r}$. They will not be exactly the same due to differences in the sizes of the packets used to compute each, imprecisions due to limited clock resolutions, the possibility that one or both of the network paths were *never* unloaded during the transfer, differences in skew between C_r and C_s, and asymmetries in the routes in the two directions, which we know from Chapter 8 are quite common. While keeping these uncertainties in mind, we can manipulate Eqn 12.3 as follows. Combining:

$$\begin{aligned} \Delta C_{r,s} &= \Delta\widetilde{T}_{p_s} - \Delta T_{p_s} \\ \Delta C_{s,r} &= \Delta\widetilde{T}_{p_r} - \Delta T_{p_r}. \end{aligned}$$

with:

$$\Delta C_{r,s} = -\Delta C_{s,r},$$

we have:

$$\begin{aligned} 2\Delta C_{r,s} &= \Delta\widetilde{T}_{p_s} - \Delta T_{p_s} - (\Delta\widetilde{T}_{p_r} - \Delta T_{p_r}) \\ &= \Delta\widetilde{T}_{p_s} - \Delta\widetilde{T}_{p_r} + (\Delta T_{p_r} - \Delta T_{p_s}). \end{aligned} \tag{12.4}$$

We then combine Eqn 12.4 with two approximations, the first being that the most accurate instances of $\Delta\widetilde{T}_{p_s}$ and $\Delta\widetilde{T}_{p_r}$ are $\delta\widetilde{T}_{p_s}$ and $\delta\widetilde{T}_{p_r}$, and the second that:

$$\Delta T_{p_r} = \Delta T_{p_s}. \tag{12.5}$$

Eqn 12.5 corresponds to an assumption that the OTTs in the two directions are the same. We know that this is not in general true, for the reasons given above, but are otherwise at a loss at how to rectify the clock readings. It is the inaccuracy of Eqn 12.5 that requires us to make only casual use of the estimate for $C_{r,s}$, as discussed at the beginning of the section. We note that the Network Time Protocol must make this same assumption when attempting to synchronize clocks over the Internet. See Claffy et al. for further discussion [CPB93a].

With this assumption, we then have:

$$\Delta C_{r,s} \approx \frac{\delta\widetilde{T}_{p_s} - \delta\widetilde{T}_{p_r}}{2}. \tag{12.6}$$

We note that, when performing the same calculation, we can also determine min-RTT$_{sr}$, the minimal round trip time between s and r, as:

$$\begin{aligned} \text{min-RTT}_{sr} &= \min \Delta T_{p_s} + \min \Delta T_{p_r} \\ &\approx \delta\widetilde{T}_{p_s} + \delta\widetilde{T}_{p_r}. \end{aligned} \tag{12.7}$$

Eqn 12.7 offers an immediate self-consistency check: it should always be positive due to the underlying "network physics." Surprisingly, this test fails for 57 $\mathcal{N}_1$ trace pairs and 30 $\mathcal{N}_2$ pairs. We discuss these failures in more detail in § 12.8.1 below.

12.5.2 Relative offset for full-sized sender packets

As discussed above, the bulk transfer sender s sometimes will send full, Maximum Segment Size (MSS; § 9.2) packets, and other times shorter packets, including some with no data whatsoever. If the path from s to r is slow (low bandwidth), then the shorter packets might arrive appreciably more quickly than the full-sized packets. Sometimes it is more convenient to discuss the relative clock offset and minimal RTT as computed when considering only the full-sized packets sent by s (and continuing to consider all of the packets sent by r, which tend to be acks of uniform size). To do so, we introduce the terms $\Delta C_{r,s}^{\mathrm{MSS}}$ and min-RTT$_{sr}^{\mathrm{MSS}}$.

12.5.3 Results of assessing relative offset

Using the methodology developed in § 12.5.1, we evaluated the relative clock offsets in $\mathcal{N}_1$ and $\mathcal{N}_2$ to see what sort of variation they exhibited. A single computation of $\Delta C_{r,s}$ does not tell anything about the absolute accuracy of either C_r or C_s, but we would expect that many computations of different $\Delta C_{r_i,s_j}$'s will reveal clusterings among the truly accurate clocks, and a large spread among the inaccurate clocks.

Maximum relative offset

In $\mathcal{N}_1$, the largest observed offset was 207,982 seconds (2.4 days!). Overall, 42 times we observed an offset greater in magnitude than 1,000 seconds, almost all greater than 10,000 seconds. All of the host pairs with these large offsets included `austr`, and the problem clearly lay with its clock. We will see the reason for this in § 12.7.7 below.

In $\mathcal{N}_2$, the largest offset was 824 seconds (13+ minutes). We observed an offset larger than 6 minutes 782 times, always with `oce` as one of the hosts. We will likewise see in § 12.7.8 that `oce`'s clock and network paths have puzzling properties. These two outliers are thus suggestive that, upon observing a very large relative clock offset, we should consider the possibility of other clock errors.

Median relative offset

We next look at clustering host clocks based on the magnitude of their median relative clock offset for all the traces in which they participated. We use the median offset in order to isolate hosts that consistently had large relative offsets, instead of those that only occasionally had large offsets, since the latter could be skewed by unfortunately-frequent pairing of a host with an accurate clock together with a host with a poor clock. We use the median of the absolute value of the offset rather than the median of the offset itself as a way of detecting hosts that often "swing" from being too slow to too fast. For each host, we analyze the relative offsets for those traces in which it was the source; these are quite similar (though opposite in sign) to the offsets when it was the receiver, and limiting our analysis to just when the host was the source simplifies the presentation.

Figures 12.1 and 12.2 shows the median magnitudes of each host's relative clock offset. In both, `oce` is a clear outlier, being typically 5–15 minutes different from the other clock. Note that, for $\mathcal{N}_1$, `austr` is *not* a particularly striking outlier, even though in the previous section we identified it as having the largest *maximum* clock offset magnitudes. The reason it is not an outlier in Figure 12.1 is that its clock ran *accurately* for most of $\mathcal{N}_1$, and only degraded late during the

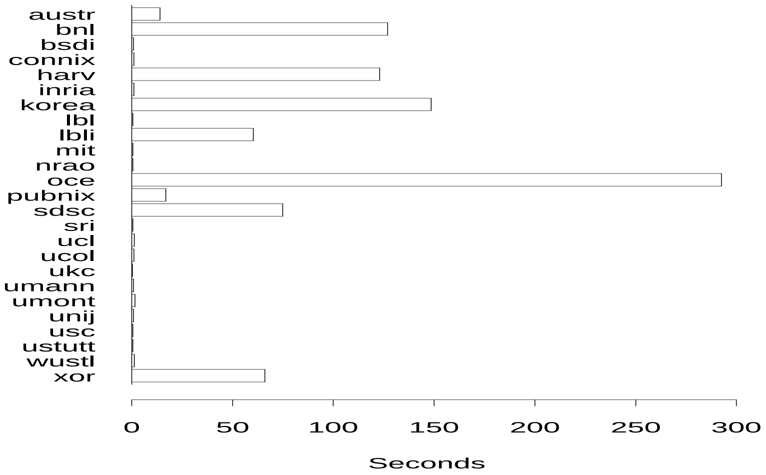

Figure 12.1: Median magnitude of clock offset, $\mathcal{N}_1$ tracing hosts

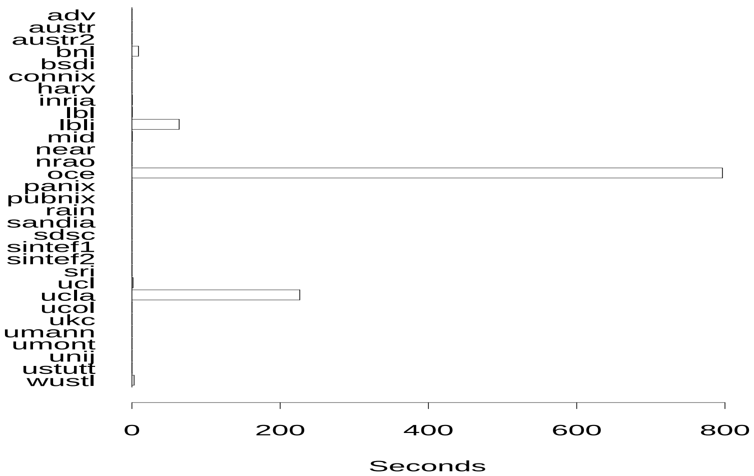

Figure 12.2: Median magnitude of clock offset, $\mathcal{N}_2$ tracing hosts

experimental run (see below). Hence its *median* relative offset over *all* of the transfers it participated in is quite small.

Both figures show other apparent outliers in addition to `oce`. We need to be careful before removing them, though, as there is a possibility that some of them have unusually high proportions of their connections to the other outliers, and hence are outliers only by "association." Thus we remove the connections involving the largest outlier and recompute the plot, then remove those involving what is now the largest remaining outlier and recompute the plot, and so on, similar to the approach developed in § 7.6.1 for assessing the "persistence" of Internet routes. For $\mathcal{N}_1$, this process removes `oce`, `korea`, `bnl`, `harv`, `sdsc`, `xor`, `lbli`, and `pubnix` as being outliers. Note that, during the iterative process, `austr` ceased to be an outlier, even though in Figure 12.1 it looks like it has almost as large a median offset as `pubnix`: this is because it was an outlier only by association with larger outliers. After eliminating these hosts, the remainder all have median offsets < 1.25 sec. We consider this group of 17 hosts as *closely synchronized*. We can, if we wish, continue the process to find a core group of *highly synchronized* hosts: they are `austr` (!), `bsdi`, `mit`, `nrao`, and `ukc`, all with median offsets < 10 msec between one another.

For $\mathcal{N}_2$, outlier removal eliminates the six largest spikes in Figure 12.2, namely, `oce`, `ucla`, `lbli`, `bnl`, `wustl`, and `ucl`, these last two having relatively small median offsets of 3 and 1.5 sec, respectively. We consider the remaining group of 25 hosts as closely synchronized. They all have median offsets < 600 msec, and, if `lbl` is removed from the group, they are all below 250 msec. Eliminating six more of the hosts with the largest median offset leaves a group of 18 synchronized hosts, with median offsets below 50 msec. We can further winnow the group down to a final set of highly synchronized hosts, `adv`, `connix`, `harv`, `near`, `nrao`, `pubnix`, `sdsc`, `sintef2` (but not `sintef1`), `ucol`, and `unij`, all of which have median offsets between each other of less than 10 msec. Note that this group includes hosts on both coasts of North America as well as two in Europe, indicating synchronization well below that of the propagation time between the hosts–very good, and around the accuracy limit for NTP reported in [Mi92b], even though we are performing a cruder estimate of accuracy (and of relative accuracy rather than absolute accuracy).

We will make use of these different groups of closely-synchronized and highly-synchronized hosts in § 12.9 when we test whether high clock accuracy (which we assume can be inferred from close synchronization, although this is not necessarily the case) tends to correlate with low relative clock skew.

Evolution of relative offset

We finish with a look at how a host's relative offset evolves over the course of an experimental run. The evolution is interesting because it provides a large-scale look at how clock accuracy changes. Our interest here is phenomenological—to develop an appreciation for clock inaccuracies and an awareness of how they occur.

To assess offset evolution, for each host we constructed a plot with the relative offsets (in seconds) computed for those connections for which it served as the data source, using the methodology given in § 12.5, on the y-axis; versus the time of the connection (days since the beginning of the experiment) on the x-axis. Since the plots are for the host as the data source, the offsets reflect the receiver's clock minus the host's clock. Hence, positive values indicate the host's clock was running behind the receiver's clock. Note that we include the sign of the offset in the plot—there is no need to use only the magnitude, as we did above.

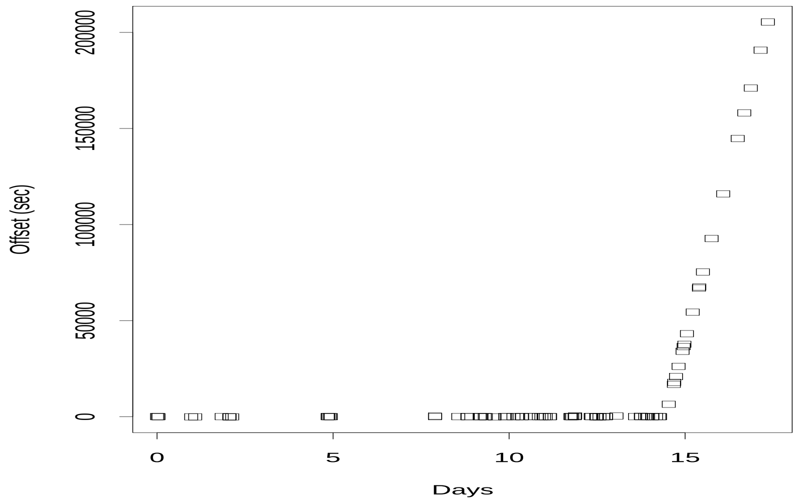

Figure 12.3: Evolution of austr's relative clock offset over the course of $\mathcal{N}_1$

Figure 12.3 shows such a plot for the austr tracing host's clock over the course of the $\mathcal{N}_1$ experimental run. This is the site that we identified above as sometimes having very large relative clock offsets, on the order of days, yet also, surprisingly, found not to be an outlier in terms of its *median* relative offset. From the figure, it is immediately clear how to reconcile the findings: up until the 14th day of austr's participation in $\mathcal{N}_1$, it kept good time, but after that point its clock came unglued and ran very slowly, such that the clocks of the other hosts to which it transferred data ran further and further ahead of it (hence, higher and higher offsets). We look at this phenomenon further in § 12.7.7.

Figure 12.4 shows the evolution of $\mathcal{N}_1$'s greatest median offset outlier, oce, after eliminating its connections with austr. The central points in the plot reflect connections for which oce was paired with sites that had a clock closely synchronized to true time (or at least, so we presume, because of the preponderance of such clocks in the plot).[4] "Noise" values distant from the central points reflect pairings with other sites that had poorly-synchronized clocks.

We see that the 5 minute median offset actually grew increasingly negative over the course of $\mathcal{N}_1$. A robust linear fit (shown in the plot) to the points yields an overall offset decrease of about 1.5 sec/day. This is quite small compared to the magnitude of the offsets themselves.

Figure 12.5 shows the evolution of bnl's relative clock offset, with connections to oce removed. The central line appears to show an increasing trend, but a somewhat complicated one. To look at it in greater detail, Figure 12.6 examines just the region of the line. We observe what appear to be three separate regions of clearly upward trend, one spanning 0–5 days, one spanning 8–14 days, and one spanning 15–16 days. Each increase corresponds to about 0.7 sec/day. What is puzzling are the offset shifts between the regions. These appear to be too small to have been

[4] As discussed in § 12.2, and revisited below in § 12.9, we did not require NTP synchronization of the clocks of the sites in our study. In addition, we assume that when we discover highly synchronized clocks, that the synchronization was achieved using NTP. Regrettably, we did not ask the participating sites specifics regarding the site's clock synchronization.

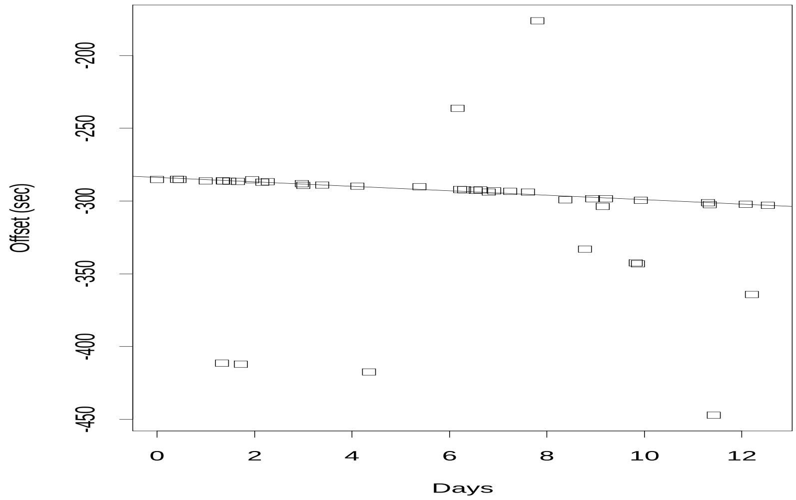

Figure 12.4: Evolution of `oce`'s relative clock offset over the course of $\mathcal{N}_1$

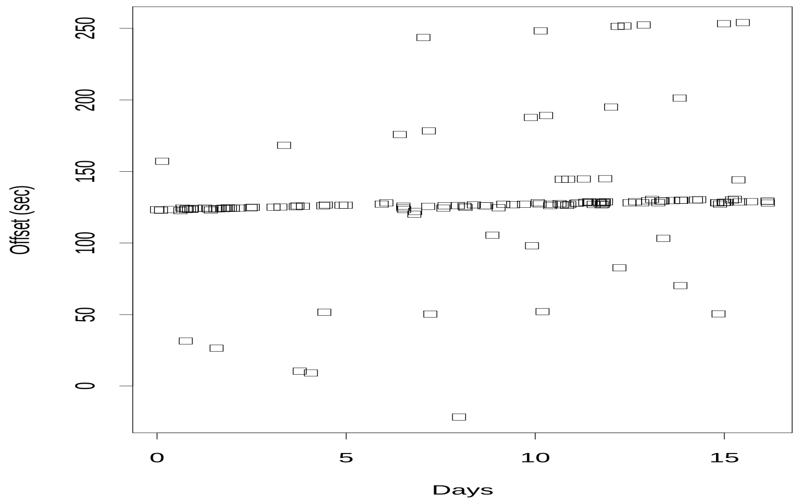

Figure 12.5: Evolution of `bnl`'s relative clock offset over the course of $\mathcal{N}_1$

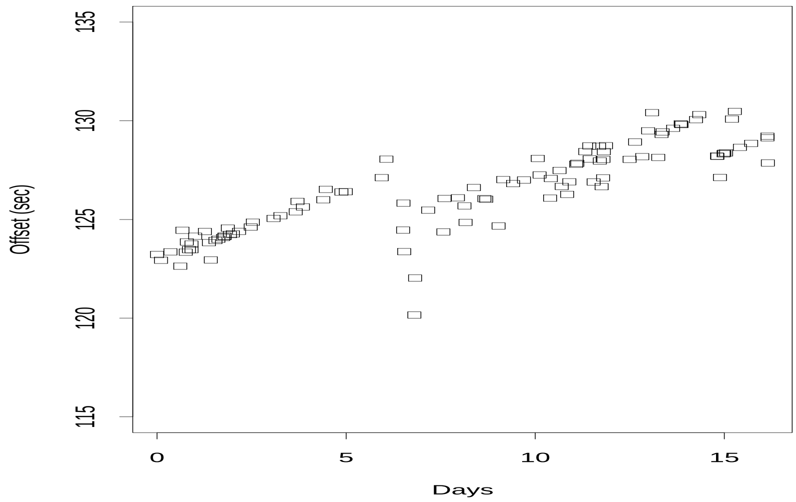

Figure 12.6: Expanded view of the central line in the previous figure

caused by someone adjusting `bnl`'s clock by hand, and too far from true to have been induced by NTP synchronization. Perhaps the changes came from temporary changes in machine-room temperatures, which are known to alter clock skew [Mi92b].

Figure 12.7 shows the evolution of `xor`'s clock during $\mathcal{N}_1$, after removing connections to `austr` and `oce`. It shows not only a steadily increasing relative offset, but a 2-minute adjustment around day 6. We look at clock adjustments in more detail in § 12.6 below.

Figure 12.8 shows the evolution of `oce`'s relative offset over the course of $\mathcal{N}_2$ (as opposed to $\mathcal{N}_1$ in Figure 12.4). The sustained decreasing offset is striking; the fit corresponds to -1.4 sec/day. Figure 12.9 shows the evolution of `lbli`'s clock during $\mathcal{N}_2$. While overall the clock has a clear persistent skew, the skew is reversed around day 8, perhaps in an effort to correct the clock's inaccuracy. But the effort ends a few days later and the original skew returns. However, around day 27 the clock's relative offset jumps by over a minute, reflecting a different sort of correction.

Figure 12.10 shows how `sandia`'s clock evolved during $\mathcal{N}_2$. For most of the experimental run the clock performs very smoothly, but around day 20 it began a slow increase over the next week, eventually reaching 3 seconds. During this week it initiated transfers to a number of different other sites, so this effect is definitely due to its own clock variation rather than those of its NPD peers.

Figure 12.11 presents our last example of interesting clock offset evolution, that for `umont`'s clock during $\mathcal{N}_2$. What is striking here are the presence of offset "towers" that, over the course of hours, slowly elevate the relative offset from nearly zero to several hundred milliseconds. Apparently what is happening is that `umont`'s clock has a fairly hearty intrinsic skew, but NTP synchronization is detecting this and periodically resetting the clock as it strays too far. We will see more regarding this behavior of `umont`'s clock below in § 12.6.5.

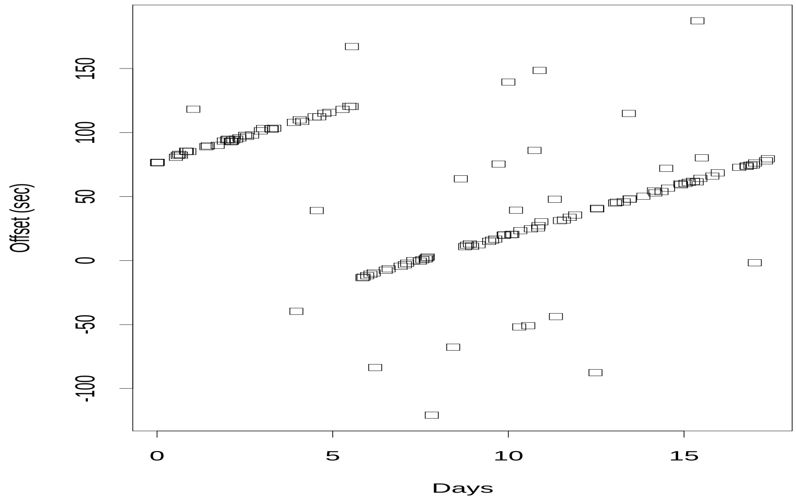

Figure 12.7: Evolution of `xor`'s relative clock offset over the course of $\mathcal{N}_1$

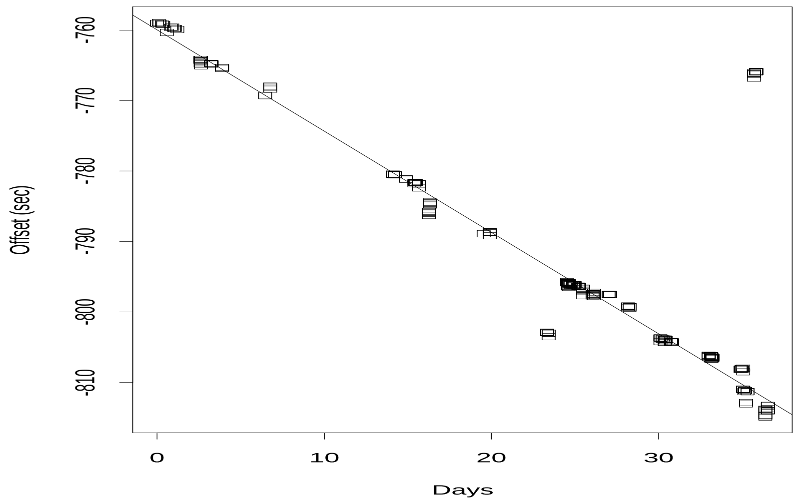

Figure 12.8: Evolution of `oce`'s relative clock offset over the course of $\mathcal{N}_2$

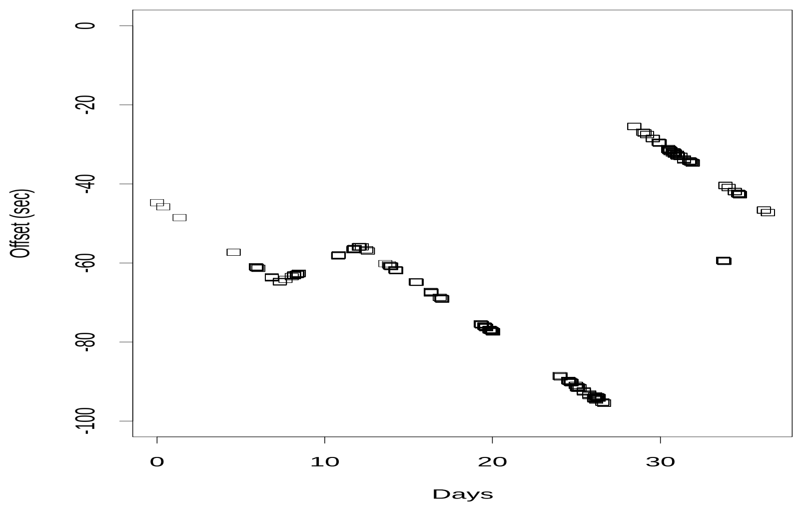

Figure 12.9: Evolution of `lbli`'s relative clock offset over the course of $\mathcal{N}_2$

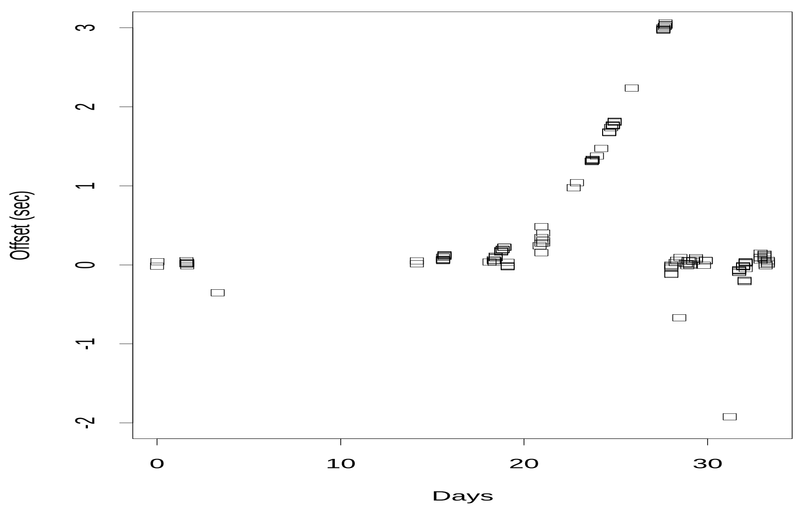

Figure 12.10: Evolution of `sandia`'s relative clock offset over the course of $\mathcal{N}_2$

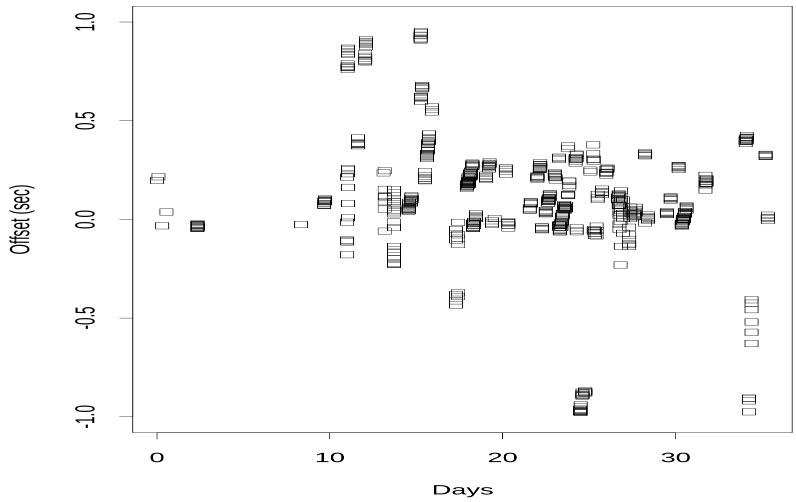

Figure 12.11: Evolution of `umont`'s relative clock offset over the course of $\mathcal{N}_2$

12.6 Detecting clock adjustments

As shown quite strikingly in Figures 12.7 and 12.9, computer clocks are sometimes subject to abrupt adjustments in which the clock's notion of the current time is changed, either gradually or instantaneously (§ 12.2). Gradual change is produced by artificially altering the clock's skew, so that it slowly alters its offset towards the target. Instantaneous change is produced by simply loading a new value into the clock register.

In order to characterize Internet packet dynamics, we will make heavy use in later chapters of variation in one-way trip times (OTTs). A clock adjustment will result in a systematic shift in OTTs between those computed prior to the adjustment and those computed after (illustrated below). If undetected, such a shift can lead to completely erroneous findings of periods of sustained high delay. Since we are very interested in the possibility that network dynamics truly have this property anyway, it is vital that we reliably detect clock adjustments so as not to be fooled by them into drawing such a conclusion.

Backward clock adjustments, in which a clock is set to a value it already registered in the past, can sometimes be easily detected *if the adjustment is large*, by the presence of a pair of timestamps T_1 and T_2 for which $T_2 < T_1$ even though T_2 was recorded after T_1. We refer to this sort of adjustment as “time travel,” and already analyzed it in § 10.3.7. In this section we tackle the harder problem of clock adjustments (both forward and backward) that are *not* apparent by trivial inspection of the timestamp sequences.

12.6.1 A graphical technique for detecting adjustments

Suppose we have a trace pair between s and r. One simple way to detect whether a clock adjustment occurred during the trace is to plot both the OTTs for the packets from s to r and those in

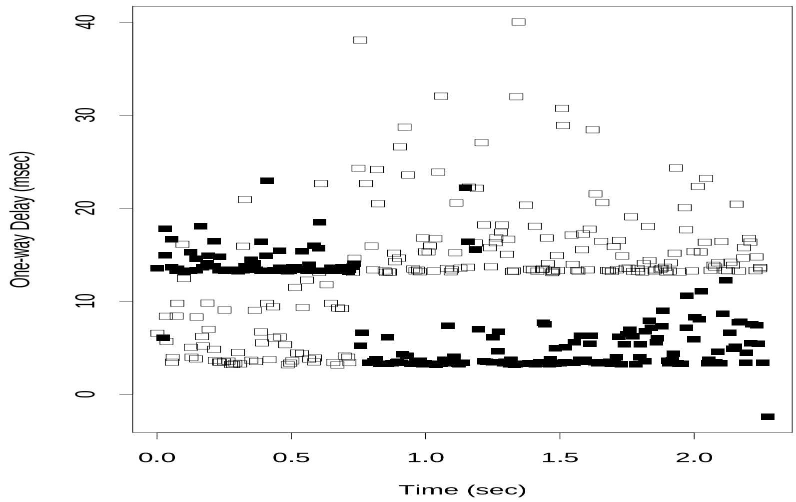

Figure 12.12: OTT-pair plot illustrating a clock adjustment (sender packets are filled, receiver packets are hollow)

the reverse direction. (Packets that are dropped have no OTT associated with them and are omitted from the plot.)

Figure 12.12 shows such a plot made for a connection from `sdsc` to `usc` in $\mathcal{N}_1$. The solid black squares indicate the OTT for packets sent from the sender to the receiver, and the hollow squares reflect the OTTs of the acks sent from the receiver to the sender. The OTTs have been adjusted using Eqn 12.6 to approximately synchronize the two clocks. (In this case, the approximation does not work particularly well, since there is more than one clock offset to estimate!)

The figure shows a striking level-shift occurring for the sender's OTTs around time $T = 0.7$ seconds, a fall of about 10 msec. Furthermore, the OTTs in the opposite direction show an equal and *opposite* change. This equal and opposite change is a crucial aspect of the plot, as it is the signature of a clock adjustment. If the shift were due to a change in network path properties (for example, a route change), then in general we would expect that (1) either it would occur in only one direction, or (2) if it occurred in both directions due to a coupled effect, it would have the same sign.

For a networking change to result in an equal-but-opposite level shift, some resource needs to have been shifted between the two directions of the network path, and furthermore the resource needs to affect the transit times of the small acks equally with those of the large data packets. It is difficult to see what sort of networking change could do this (but see § 12.7.8). The change, however, makes perfect sense if, at around time $T = 0.7$ seconds, `sdsc`'s clock was set ahead 10 msec, or `usc`'s clock was set back 10 msec. In either of these cases, the difference in the timestamps for packets sent from `sdsc` to `usc`, i.e., the quantity $\Delta\widetilde{T}_{p_s}$ defined in Eqn 12.2, will decrease by 10 msec, and similarly $\Delta\widetilde{T}_{p_r}$ will *increase* by 10 msec. This is exactly the behavior shown in the plot.

12.6.2 Removing noise from OTT measurements

Two other points concerning Figure 12.12 merit attention. The first is the presence of a few unusually small sender packet OTTs, one of about 7 msec around $T = 0$, and the other of around -3 msec around $T = 2.3$ (it is negative because for the plots the clocks were rectified using $\Delta C^{\mathrm{MSS}}_{r,s}$, as discussed in § 12.5.2). Both of these reflect sender packets that did not carry any data (the SYN and FIN connection management packets). These travel through the network more quickly than full-sized data packets. Often in OTT plots we will include such packets (as they are a useful reminder of one source of OTT variation), but we need to be careful when developing techniques for analyzing OTT behavior to remember that these packets have unusually low OTTs due to their size. Hence our techniques need to be careful to not weigh their OTT values the same as those for full-sized packets.

The second important point shown in the plot is the large *variation* in OTTs, both for the full-sized sender packets and the receiver packets. For example, note that the OTTs of both some of the acks before the adjustment, and some the data packets after the adjustment, are larger than many of the OTTs on the other side of the adjustment. This variation is the first suggestion that we will require robust algorithms in order to not be fooled by noise when analyzing OTT data. The eye quite readily picks out the twin level shifts in this plot, but doing so algorithmically requires care to screen out noise such as these large OTT values.

OTTs often exhibit considerable network-induced noise in terms of deviation of a given OTT from the value expected if the network were unloaded. The noise, however, has one crucial property that often makes it tractable: barring a significant change in the network path (such as a route change), the noise always takes the form of an additive, positive increase. This means that, given a set of OTT measurements, we can often hope to find those with very little network-induced noise by looking at the smallest values in the set.

We used this property of OTT noise in § 12.5.1 above when we picked $\delta\widetilde{T}_{p_s}$ and $\delta\widetilde{T}_{p_r}$ as the measured raw offsets to use when attempting to estimate the relative clock offset. We will use it again when developing methods to detect clock adjustments and skew. For these latter, what is interesting are *trends* in how the OTT values (with noise removed) change over the course of the connection. Thus, we cannot simply de-noise the OTT values by selecting the global minimum, or we will obliterate the trend. Instead we divide the series of OTT values up into intervals and de-noise each interval by selecting the minimum value observed during the interval. The question then becomes which intervals to use.

One natural way of devising intervals is to allocate them so that each has the same number of packets. Another is to choose them so that they each span the same amount of time. For assessing trends in OTT values over time, the latter seems to be the natural choice. But using fixed-time intervals has a fundamental problem. Sometimes a connection's activity primarily occurs during only a small portion of the connection's total duration, with the rest of the time mostly inactive due to lengthy retransmission timeout lulls.

To address this difficulty, we combine the two approaches by choosing both a packet-count interval, I_p, and a duration interval, I_t. We then advance through the OTT timings and group timings into a single interval whenever we have either encountered I_p packets, or we have reached a point I_t from the beginning of the interval. At this point, if we have any packets at all, we take their minimum as the de-noised OTT value for the interval, and we begin a new interval by resetting the packet count and setting the start of the interval to coincide with the next OTT measurement.

One detail we must attend to is the final partial interval at the end of the connection. It in general will not span I_t nor have a full I_p's worth of packets in it. We adopted the rule that, if the interval had more than $I_p/2$ packets, we included it, otherwise we skipped it.

The final issue is how to pick I_p and I_t. For a set of n OTT measurements spanning an interval ΔT, we used:

$$\begin{aligned} I_p &= \lfloor \sqrt{n} \rfloor, \\ I_t &= \Delta T / \sqrt{n}. \end{aligned}$$

Using these choices means that the number of de-noised OTT values scales as the square-root of the total number of values. This struck us as a good compromise between preserving sufficient detail without using too fine a resolution (which could mean we do not effectively remove noise). Furthermore, we anticipate subsequently applying a number of robust algorithms to the de-noised values, some of which have running times of $O(n^2)$ or higher. For these, if we present them with only $O(\sqrt{n})$ values, then the total running time will remain $O(n)$ or only slightly higher, which is important for performing fast automatic analysis.

We will refer to a measured series of OTT values as x_t. Here, x_t can reflect either a series of data packet OTTs, or ack OTTs. To detect adjustments ultimately requires comparing properties of the data packet OTTs with those of the ack OTTs, but prior to developing the tests on these properties, our discussion will apply to any generic series of OTT values.

We denote the de-noised series derived from x_t as $\check{x}_t$. Note that for each $\check{x}_t$, the index t corresponds to the same index as where in the interval we found the (first) minimal value of x_t. This is an important point—if we instead adjusted the index to reflect, say, the middle of the interval, then we might introduce inaccuracies in the trends. The key idea is that the "best" (least noisy) value of x_t during the interval occurred at a particular t, and we want to take that point and discard all the others in the interval.

Figure 12.13 shows the results of applying this de-noising method to the measurements plotted in Figure 12.12.

12.6.3 An algorithm for detecting adjustments

We now turn to attempting to detect adjustments algorithmically, since it is infeasible to manually inspect all 20,000 of our trace pairs to look for adjustments (§ 9.1.4). The central notion we will use is that of the *signature* of the OTTs in the two directions showing equal but opposite level shifts.

Identifying pivots

The foundation of our approach lies in identifying *pivots*: points in time before which the OTTs all lie predominantly above or below all the OTTs after the given point in time. In Figure 12.12, the pivot we aim to identify occurs around $T = 0.7$ sec.

In this subsection we develop a heuristic for identifying pivots in the series of OTTs for packets sent in a single direction (from s to r or vice versa). In the next subsection we then analyze the pivots identified in both directions to test for a clock adjustment.

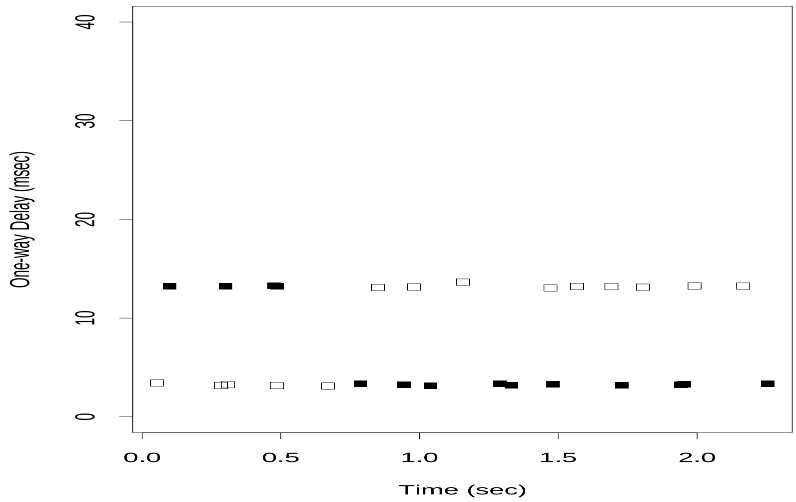

Figure 12.13: Same measurements after de-noising pair-plot

Let $\breve{x}_t$ be a series of de-noised OTT values occurring at times t, ordered by the time index t. Let $\breve{x}_{t_i}$ be the same series numbered from $i = 1 \ldots n$, where t_i is the ith measurement time.

We define a *pivot partition* of $\breve{x}_t$ as a partition of $\breve{x}_t$ into two disjoint sets, $\breve{x}'_t$ and $\breve{x}''_t$, for which the maximum of one set is less than the minimum of the other. Without loss of generality, let $\breve{x}'_t$ be the "larger" of the two sets, i.e., its minimum is larger than the maximum of $\breve{x}''_t$.

We further require that the time intervals spanned by $\breve{x}'_t$ and $\breve{x}''_t$ are disjoint, namely either the largest i in $\breve{x}'_{t_i}$ is less than the smallest j in $\breve{x}'_{t_j}$, or vice versa.

We term the pivot partition *positive* if the measurements $\breve{x}'_t$ occurred *after* those in $\breve{x}''_t$, and *negative* otherwise.

Geometrically, this definition corresponds to being able to draw horizontal and vertical lines on a plot like that in Figure 12.13 such that either all of the points lie in the first and third quadrants formed by the lines (if positive), or they all lie in the second and fourth quadrants (negative).

It is important to note that a given series $\breve{x}_t$ may have more than one pivot partition. For example, if $\breve{x}_t$ is strictly decreasing, then every value of t gives rise to a pivot partition. In addition, any time the largest or smallest value of $\breve{x}_t$ occurs at the lowest value of t, i.e., $\breve{x}_{t_1}$, then there is a pivot partition that isolates that one value versus placing all the other values in the other partition set. Generally, this is not a pivot partition of interest.

We proceed as follows. First, we determine whether to search for a positive or negative pivot by inspecting whether $\breve{x}_{t_1}$ is less than or greater than $\breve{x}_{t_n}$. From here on, we assume without loss of generality that we wish to detect a positive pivot, such as the one exhibited by the receiver packets (hollow squares) in Figure 12.12. We indicate in brackets, like [this], the changes we make to the algorithm when testing instead for a negative pivot.

We search through the measurements to find the point k where

$\min(\check{x}_{t_{k+1}}, \check{x}_{t_{k+2}}) - \max(\check{x}_{t_{k-1}}, \check{x}_{t_k})$ [respectively, for detecting a negative pivot, $\min(\check{x}_{t_{k-1}}, \check{x}_{t_k}) - \max(\check{x}_{t_{k+1}}, \check{x}_{t_{k+2}})$] is largest. Conceptually, we are looking for the intervals that have the greatest difference between them in the same direction as the pivot; we spread the differencing over the additional intervals on either side to combat the problem of the intervals right at the pivot misleading us due to noise. Note that this spreading operation also means that we cannot detect a pivot that occurs right at the beginning or end of a connection (§ 12.6.5).

k is now the candidate pivot (actually, the potential pivot occurs at a point in time between measurement k and measurement $k+1$). We then inspect the points $\leq k$ to find χ_k, the largest [respectively, the smallest] point before the candidate pivot, and likewise those $> k$ to find χ_{k+1}, the smallest [largest] after the candidate. If χ_k is less [greater] than χ_{k+1}, then we conclude that $[k, k+1]$ does indeed straddle a pivot; otherwise, we conclude they do not.

If we find a pivot partition, then we define its magnitude M as the absolute value of the difference between the median of the points after the pivot with the median of those before. We also associate a pivot width, $W = t_{k+1} - t_k$.

Identifying adjustment signatures

We now turn to identifying the signature of a clock adjustment for the clocks of two hosts, s and r. The method we developed is not entirely satisfying, as it uses some heuristics in order to accommodate residual noise in the OTT measurements, while attempting to not mistake genuine networking effects for a clock adjustment. However, the method appears to work well in practice. We note, though, that the method assumes that clock adjustments are relatively rare events: rare enough that our traces are likely to exhibit at most one adjustment, and that the likelihood of *both* of the clocks we are comparing exhibiting an adjustment during the trace is negligible.[5]

Suppose we have two sets of de-noised OTT measurements, $\check{s}_t$ and $\check{r}_t$, corresponding to full-sized packets from the data sender to the receiver, and acks in the other direction, respectively. If either of $\check{s}_t$ or $\check{r}_t$ does *not* exhibit a pivot, or if the pivots are both positive or negative, then we conclude there was not any clock adjustment.

Let M_s, W_s, M_r, and W_r be the magnitudes and widths of the corresponding pivots. We next check whether the pivots *overlap*. Let s_1 and s_2 denote the packets bracketing $\check{s}_t$'s pivot region, and likewise for r_1 and r_2. Let s_1^s denote the time at which s_1 was sent from s (according to s's clock), and s_1^r the time at which it arrived at r (according to r's clock). With analogous definitions for the other packets, we then conclude that the pivots overlap if either of the following holds:

$$\begin{aligned} s_1^r &< r_2^r + \delta t \quad \text{and} \\ s_2^r + \delta t &> r_1^r, \end{aligned}$$

or

$$\begin{aligned} r_1^s &< s_2^s + \delta t \quad \text{and} \\ r_2^s + \delta t &> s_1^s, \end{aligned}$$

[5]This assumption might be violated if NTP updates among widely separated clocks sometimes happen in synchronization. To our knowledge, the possibility of this occurring for NTP has not been studied. Given the findings of synchronized routing messages reported in [FJ94], it does not seem completely implausible.

where δt is the allowed measurement "slop", which we set to:

$$\delta t = \frac{\max(W_s, W_r)}{2}.$$

The idea behind the slop is to allow for other-than-instantaneous adjustments (illustrated below).

If the pivots do not overlap, then we conclude there was no adjustment. If they do, we then next look at the magnitudes of the pivots. If either magnitude is less than the larger of twice the joint clock resolution $R_{s,r}$ (§ 12.3), or 2 msec (an arbitrary value to weed out fairly insignificant adjustments), then we declare the pivot "insignificant" and ignore it.

Finally, we look to see whether M_s and M_r are within a factor of two of each other. If not, then we term the pivot a "disparity pivot," meaning that it may be due to unusual networking dynamics (§ 12.6.5). If the two agree within a factor of two (which experience has shown is a good cut-off point), then we conclude that the trace pair exhibits a clock adjustment with a magnitude of about $\frac{M_s+M_r}{2}$.

12.6.4 Results of checking for adjustments

`tcpanaly` uses the method given in § 12.6.3 to check each trace pair it analyzes for clock adjustments. Doing so, we found 36 trace pairs in $\mathcal{N}_1$ out of 2,335 (1.5%) that exhibited clock adjustments, and 128 out of 15,492 in $\mathcal{N}_2$ (0.8%). While these proportions are fairly low (and not representative, since the behavior of the individual hosts in our study is not necessarily representative), they are high enough to argue that a large-scale measurement study for which accurate timestamps are important needs to take into account the possibility of clock adjustments. Furthermore, *the adjustments are only detectable due to the use of a pair of clocks*. If a study uses timestamps from only one measurement endpoint, then checking the timestamps for clock adjustments becomes much more difficult. The median adjustments were on the order of 10–20 msec, the mean around 100 msec, and the maxima close to 1 sec. These magnitudes are unfortunately small enough to sometimes not be glaringly obvious, but large enough to be comparable to wide-area packet transit times, so they can introduce quite large analysis errors if undetected.

While clock adjustments are usually abrupt, this is not always the case. The adjustment-detection method found some clock adjustments that occurred due to a short period of altered clock frequency (i.e., temporary skew). Figure 12.14 shows a striking example.[6] Here, around time $T = 40$ sec the sender's clock began running more quickly than the receiver's, leading to lower sender OTTs and higher receiver OTTs. Less than 20 seconds later, the frequencies were again equal, but the relative offsets between the clocks shifted by nearly 1 sec in the process.

12.6.5 Problems with detection method

The method given in § 12.6.3 works well in practice, but it does sometimes fail to detect clock adjustments. In this section we look at some cases where we identified this happening.

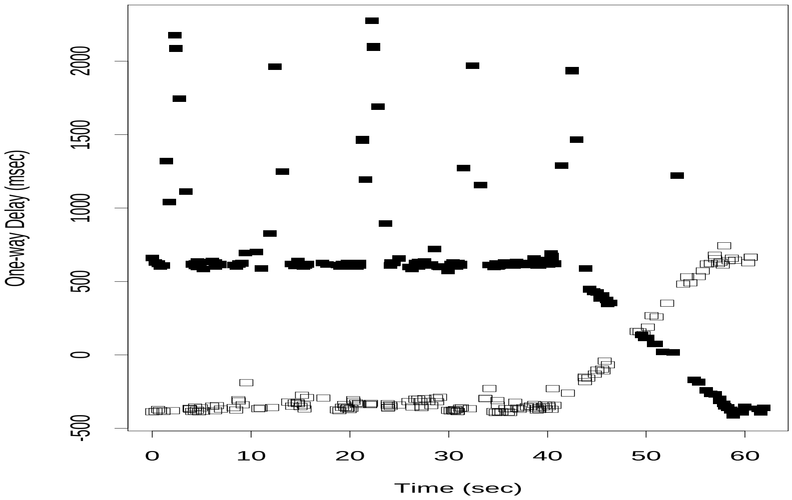

Figure 12.14: Clock adjustment via temporary skew

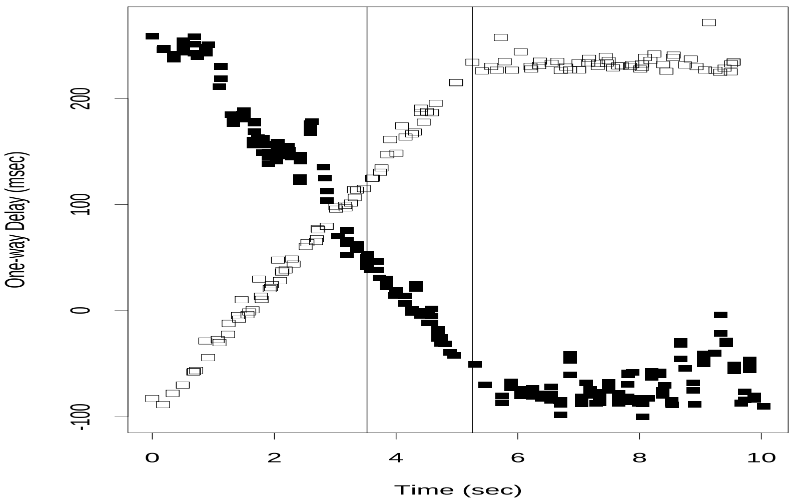

Figure 12.15: Temporary skew leading to separate pivots

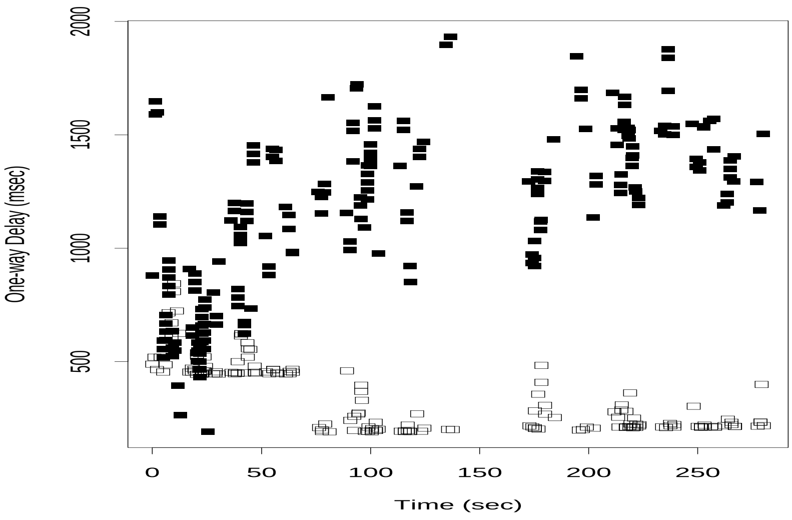

Figure 12.16: Clock adjustment masked by excessive network delays

Failure to detect adjustment via skew

In Figure 12.14 we illustrated how sometimes a clock adjustment can occur due to temporary skew. Figure 12.15, however, shows such a case that the method fails to detect. The problem here is that, due to noise in the forward direction, the two pivots located by the method do not overlap, so the possibility of an adjustment is rejected. The lefthand vertical line marks the pivot the method found for the data packets (solid), and the righthand vertical line marks the pivot for the acks (hollow). In general, this sort of failure will only occur with adjustments using temporary skew; abrupt adjustments have sharply defined pivots. This example *does*, however, exhibit a negative estimate for min-RTT$_{sr}$ (§ 12.5.1), so `tcpanaly` still flags it as having a clock problem.

Excessive network-induced delay

Figure 12.16 shows a case where the reverse path exhibits a clear level shift around $T = 70$ sec, with a magnitude of about 250 msec, but the corresponding shift on the forward path is less clear because it is accompanied by an increase in networking delays, too. In that direction, `tcpanaly` assesses the magnitude of the shift as about 730 msec. Since this is more than twice the magnitude in the other direction, `tcpanaly` rejects the possibility of a clock adjustment.

`tcpanaly` flags a trace pair like this as having a "disparity pivot," namely common pivots that have too great a difference in their magnitudes to be considered a clock adjustment. Disparity pivots are quite rare (only 61 in $\mathcal{N}_2$). We inspected each one and found that only the one shown above was a likely clock adjustment. The rest appear simply due to unfortuitous patterns of noise, often in truncated traces (§ 10.3.4) with few OTT timings.

[6]Note that the OTTs in the plot have not been "de-noised" (discussed in § 12.6.2). Likewise, subsequent OTT plots do not show de-noised OTTs unless so stated.

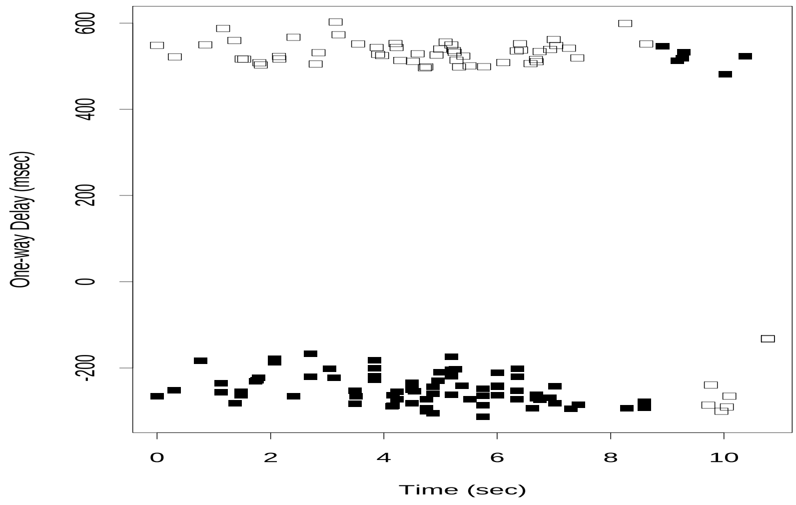

Figure 12.17: Clock adjustment missed because too close to end of connection

Adjustment too close to connection edge

Since our method for identifying pivots (§ 12.6.3) will not accept a pivot right at the beginning or at the end of a connection, `tcpanaly` naturally will miss this sort of adjustment should it occur. Figure 12.17 shows an example. This one, like the one above, is still detected by `tcpanaly` due to a negative estimate for min-RTT$_{sr}$.

Multiple adjustments

The development of the clock adjustment detection algorithm presumes that there is a single clock adjustment to be detected. Sometimes a trace pair suffers from more than one adjustment, and the algorithm either only detects one of them (which suffices, if the policy is to discard trace pairs with any adjustments in them), or fails to detect any of them. The latter is particularly likely if there are two adjustments in opposite directions. Figure 12.18 shows a striking example of a trace pair with two adjustments, both effected using temporary skew. The algorithm fails to detect these adjustments, but `tcpanaly` flags the trace pair due to a negative estimate for min-RTT$_{sr}$, as well as due to strong negative correlation between the two directions (§ 12.6.6 below).

Clock "hiccups"

Related to the multiple adjustments discussed in the previous subsection are clock "hiccups," in which one of the clocks in a trace pair momentarily either ceases to advance or advances very quickly. Figure 12.19 shows an example, occurring at time $T = 6$ sec. It is possible that this example is actually due to surprising network dynamics, as the 4 acks with lowered OTTs come right after the only packet reordering event in the trace. While a clock glitch can change the value of

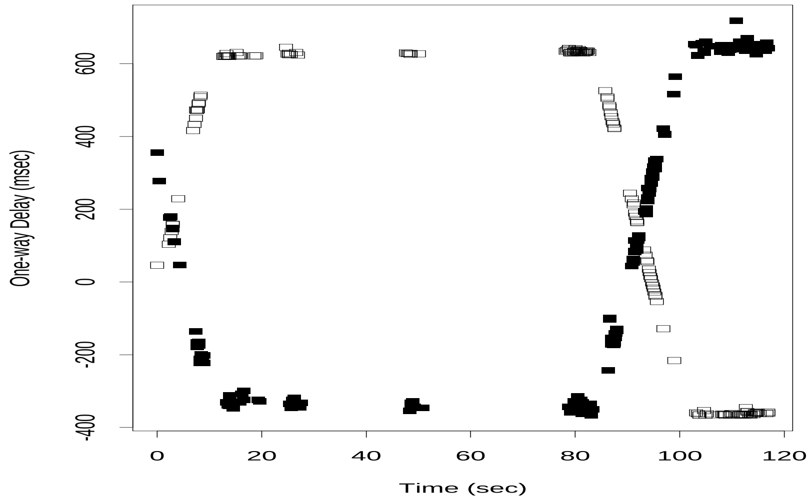

Figure 12.18: Double clock adjustment (both using temporary skew)

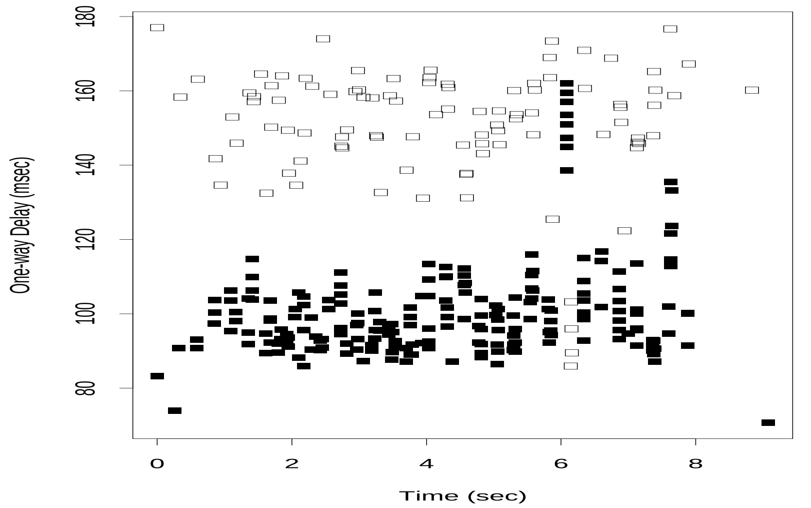

Figure 12.19: Clock adjustment “hiccup”

OTTs, it *cannot* reorder packets on the wire! But it is difficult to see what networking mechanism could lead to the data packets in the opposite direction simultaneously experiencing increased delay.

This hiccup is undetected by `tcpanaly`.

12.6.6 Detecting adjustments via correlation

When we examine a smoothed OTT pair plot such as that in Figure 12.13, a different approach for detecting adjustments suggests itself: look for strong negative correlation between the forward OTTs and the reverse OTTs. In general, this approach suffers from two problems.

First, it is highly susceptible to error due to large noise elements. Periods of inflated OTT values (such as due to an increase in queueing) tend to dominate the computation of the coefficient of correlation. We attempted to address this difficulty by devising a "robust coefficient of correlation" based on the direction of deviations from the median, but this proved no better: we were unable to eliminate the dominant effects of noise.

The second problem is that strong negative correlation is also a signature for relative clock skew, as discussed in the next section. So, by itself, it does not suffice for detecting clock adjustments.

There is still a role for correlation testing, though. In particular, if we only consider correlation significant when it is extremely strong, then the noise effects of momentary congestion periods diminish, and the approach holds promise for detecting cases of large adjustments and relative skews. In particular, *very strong correlations can detect multiple adjustments and adjustments via skew*, and this property motivated us to pursue it further.

The method we devised is based on examining the intervals produced when looking for pivots. For each interval i, we compute the median of the OTT of the packets sent by the sender (either full-sized data, or acks, depending on the direction). Call this s_{m_i}. Similarly, for the packets *received* by the sender from the receiver during the interval, we compute their OTT median, r_{m_i}. (We require that at least three packets were sent and another three received, otherwise we skip the interval in our analysis.) We then compute $\theta_{s,r}$, the coefficient of correlation between the s_{m_i}'s and their corresponding r_{m_i}'s. Similarly, we compute $\theta_{r,s}$ in the opposite direction. That is, we construct similar intervals based on packet departures and arrivals at r instead of at s.

If `tcpanaly` finds that both $\theta_{s,r} < -0.9$ and $\theta_{r,s} < -0.9$, then it flags the trace pair as exhibiting strong negative correlation. We then inspect the trace pair by hand (i.e., using an OTT pair plot) to determine the source of the correlations.

We found that connections only very rarely have the property of strong negative correlation. (If, however, we lower the threshold from -0.9 to -0.8, quite a few more connections are flagged, but upon inspection they do not appear to exhibit any clock anomalies.) In $\mathcal{N}_1$, only two trace pairs were flagged. One of these was the double-adjustment shown in Figure 12.18. In $\mathcal{N}_2$, six connections were flagged. Five of these, however, involved `oce`, which we show below (§ 12.7.8) to have highly unusual behavior in general. The sixth is an "edge" clock adjustment similar to that shown in Figure 12.17.

The second $\mathcal{N}_1$ trace pair with strong negative correlation is quite interesting, however. Figure 12.20 shows the corresponding OTT pair plot. It is clear that the correlation stems from the tendency for the reverse-path OTTs to climb sharply, by 100–200 msec, followed shortly by the forward-path OTTs falling by roughly the same amount. Another striking feature of the plot is the sustained elevated level for the forward OTTs after about time $T = 3$ sec.

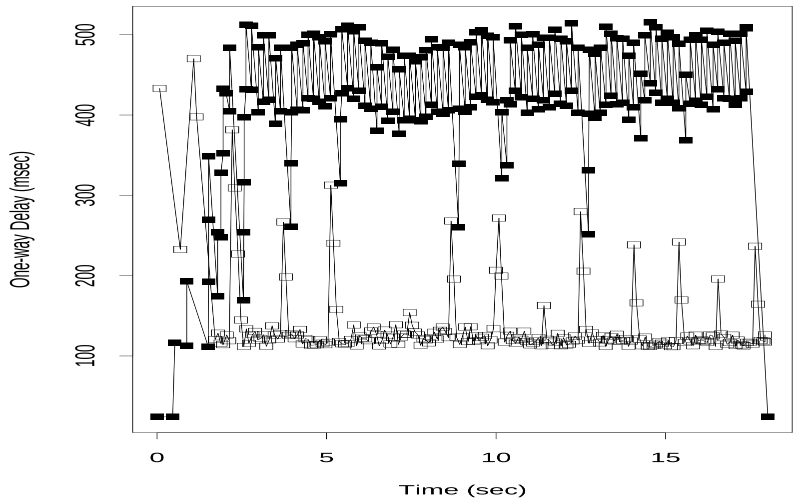

Figure 12.20: An OTT pair plot showing strong negative correlation

These two features are fundamentally related. The link connecting the sender of this connection to the rest of the Internet had a capacity of 56 Kbit/sec, or under 7 Kbyte/sec after link-level overhead is deducted. Thus, it was not difficult for the sender to open its window sufficiently to build up a queue at this link's router. The size of the OTT increase reflects the size of this queue. Occasionally, the acknowledgements sent by the receiver are being *compressed*; that is, several of them all arrive at a queue, and there they have their spacing compressed because they are placed in the queue closely together. (See § 16.3.1 for a more detailed discussion.) The signature of "ack compression" on an OTT plot is a quick build-up in OTT (reflecting having to wait in the queue) followed by a likewise-quick decrease in OTT (as the back-to-back acks all leave the queue closely spaced together).

By inspecting sequence plots corresponding to this connection, we see that what is happening is that the ack compression leads to a delay at the sender as it waits for the lead ack of the compressed group to arrive. During this delay, the queue at the 56 Kbit/s link connecting the sender to the Internet *drains*, so once the acks finally arrive and the sender sends out a bunch of packets, the first packet encounters very little queueing delay at the Internet link. This low delay is reflected in the plot by the dip in the sender OTTs. It then immediately climbs back up as the remaining packets in the bunch queue behind the lead sender packet.

This effect occurs quite often in connections for which there is a low-speed bottleneck link. The example shown above, though, was the only one in which the effect was so strong as to be detected by the negative correlation test.

12.7 Assessing relative clock skew

Many of the clock errors discussed in § 12.5.3—often skews on the order of perhaps a second a day—might seem trivial and perhaps not worth the effort of characterizing. For purposes

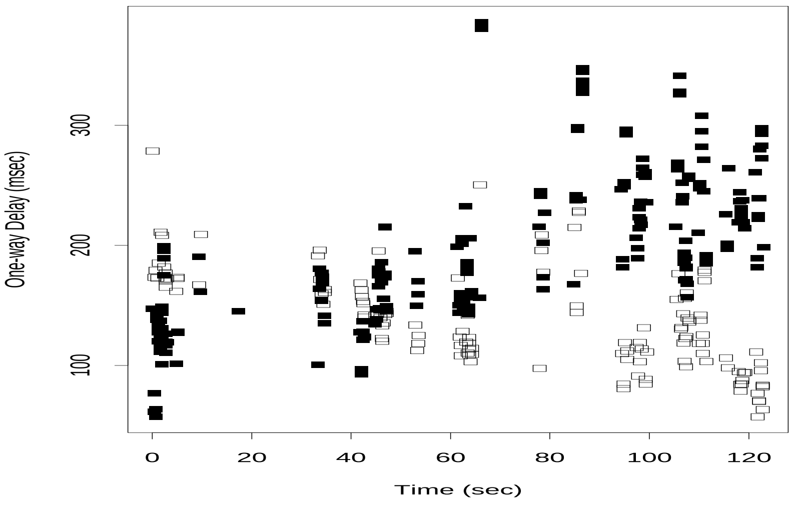

Figure 12.21: An OTT pair plot showing relative clock skew

of keeping fairly good absolute time, this is true, but, for purposes of assessing network dynamics, it is not.

To illustrate why skew is a crucial concern, consider evaluating OTTs between two hosts s and r, for which r's clock runs 0.01% faster than s's. That is, over the course of a day, r's clock will gain about 9 seconds relative to s's clock, not a particularly large error for many purposes. If, however, we are computing OTTs between s and r, then over the course of only 10 minutes r's clock will gain 60 msec over s's clock. *If we assume that variations in OTT reflect queueing delays in the network, then this minor clock drift could lead to a large false interpretation of growing congestion.* For example, if s sends 512 byte packets to r and the bandwidth of the path between them is T1 (§ 14.7.1), then a true 60 msec increase in delay reflects the equivalent of an additional 23 packets' worth of queueing. Thus, quite "minor" skew differences between the two endpoint clocks can lead to quite large, erroneous assessments of queueing delay.

Because we are very interested in accurately characterizing queueing time scales (§ 16.4), it is vital that we determine whether a given pair of clocks suffer from skew. The first issue is then to identify a skew "signature" similar to that for clock adjustments shown in Figure 12.12. Figure 12.21 shows an OTT pair plot that exhibits a clear skew signature: the OTTs in one direction show a steady overall increase, while those in the opposite direction show a steady decrease. Both changes have a magnitude of about 120 msec over the 2 minute course of the connection, consistent with the receiver's clock advancing about 0.1% faster than the sender's clock. It is difficult to see what sort of network dynamics could introduce such a true combined inflation and deflation of OTTs over a two-minute period, so we conclude that the OTT pair plot shows strong evidence of relative clock skew.

Two other clock skew signatures we investigated were differences in round-trip times (RTTs) reported by the endpoints in a connection, and strong negative correlations between the forward and reverse OTTs. The difficulty with evaluating RTT differences lies in limited clock resolution[7] and noise making the RTTs in the two directions slightly different even in the absence of clock skew. The difficulty with looking for strong negative correlations is the same as discussed in § 12.6.6 above, namely that except in instances of very strong clock skew, there is too much noise to obtain a reliable decision based on the strength of the correlations.

In the remainder of this section we develop robust algorithms for detecting and removing relative clock skew.

12.7.1 Defining canonical sender/receiver skew

Before we proceed with developing a method for identifying relative clock skew, we need to define exactly what quantity it is that we wish to estimate. First, we assume that the skew trends we identify will be *linear*. While we might possibly encounter non-linear skew, we did not find any clear examples of such in $\mathcal{N}_1$ or $\mathcal{N}_2$, except those shown in § 12.6.5. For linear skew, we can summarize the skew using a single value that reflects the excess rate at which one clock advances compared to the other.

To avoid ambiguity (in terms of which clock we are comparing to which), we will always quantify how C_r, the receiver's clock, advances with respect to C_s. Suppose C_r runs a factor η faster than C_s, by which we mean that, if C_s reports that an interval ΔT has elapsed, then C_r will have reported the same interval as having length $\eta \Delta T$. Likewise, we can say that C_s runs a factor $1/\eta$ faster than C_r (or, a factor of η slower).

The algorithms we develop for estimating relative skew all work in terms of linear trends in OTT measurements. These trends are estimated based on how OTT measurements expand or shrink with respect to time. It is important to recognize that the phrase "with respect to time" does *not* mean "with respect to true time," since we have no way of measuring true time. Instead, it means "with respect to the packet originator's clock," that is, the clock associated with tracing the TCP endpoint that sent the packet.

When discussing a linear trend in the measured OTTs of the packets sent by host s, we will quantify the trend in terms of G_s, the growth in the OTTs of the packets sent by s. Suppose packet p_1 is sent at time T_s^1, according to C_s, and arrives at time T_r^1, according to C_r. Likewise, suppose packet p_2 is sent at T_s^2 and arrives at T_r^2. Suppose further that the transit times of the packets are identical (no network-induced noise), so the only variation in their OTTs are due to clock skew.

The measured OTTs for the two packets are:

$$\begin{aligned} O_1 &= T_r^1 - T_s^1 \\ O_2 &= T_r^2 - T_s^2. \end{aligned}$$

As G_s quantifies the linear growth in measured OTTs over time, we have:

$$O_2 = O_1 + G_s(T_s^2 - T_s^1).$$

In the absence of relative skew between C_r and C_s, $G_s = G_r = 0.0$. If C_r runs faster than C_s, then the packets sent by s will exhibit *increasing* OTTs and those sent by r will exhibit *decreasing* OTTs, so we will have $G_s > 0$ and $G_r < 0$. Naturally, the reverse holds if C_r runs slower than C_s.

[7] For example, if the RTT is on the order of 100 msec, and the clock resolution is 1 msec, then only relative skews larger than 1% can be detected; these are very large.

We now relate G_r and G_s to η, the factor by which C_r runs faster than C_s. Continuing the example above, we have:

$$\begin{aligned}
G_s &= \frac{O_2 - O_1}{T_s^2 - T_s^1} \\
&= \frac{(T_r^2 - T_s^2) - (T_r^1 - T_s^1)}{T_s^2 - T_s^1} \\
&= \frac{(T_r^2 - T_r^1) - (T_s^2 - T_s^1)}{T_s^2 - T_s^1} \\
&= \frac{(T_r^2 - T_r^1)}{T_s^2 - T_s^1} - 1 \\
&= \eta - 1. \qquad (12.8)
\end{aligned}$$

It can similarly be shown that:

$$\begin{aligned}
G_r &= \frac{1}{\eta} - 1 \qquad (12.9) \\
&= \frac{1}{G_s + 1} - 1. \qquad (12.10)
\end{aligned}$$

For $\eta = 1 + \epsilon$, where $|\epsilon| \ll 1$, we have:

$$\begin{aligned}
G_s &= \epsilon, \\
G_r &\approx -\epsilon.
\end{aligned}$$

Because clock skews are often only a few parts per thousand or ten thousand, we are usually in this regime (but see § 12.7.7 below). Consequently, an easy inaccuracy to introduce is to assume that:

$$G_s = -G_r,$$

(i.e., the slopes are equal but opposite), since this often appears to be the case when inspecting OTT pair plots. To ensure full accuracy, we instead take care to always use Eqns 12.8 and 12.9 to express relative clock skew in terms of η, or Eqn 12.10 to translate G_r to G_s. We will refer to values of G_s and G_r that are consistent with respect to Eqn 12.10 as "equivalent but opposite" slopes.

12.7.2 Difficulties with noise

One particular problem with testing for clock skew is that one of the paths can have such highly variable OTTs due to queueing fluctuations that these completely mask the smaller-scale trend of OTT increase or decrease due to skew, even after de-noising. Figure 12.22 shows an example, in which congestion on the forward path completely obscures the relative clock skew, which is apparent from the enlargement of the return path shown in Figure 12.23. Such noise most often obscures the forward path (presumably due to extra queueing induced by the data packets), but it can also obscure the reverse path. Thus, we cannot always rely on the signature of *dual* equivalent-but-opposite OTT trends; sometimes we must settle instead for simply a compelling trend in one direction.

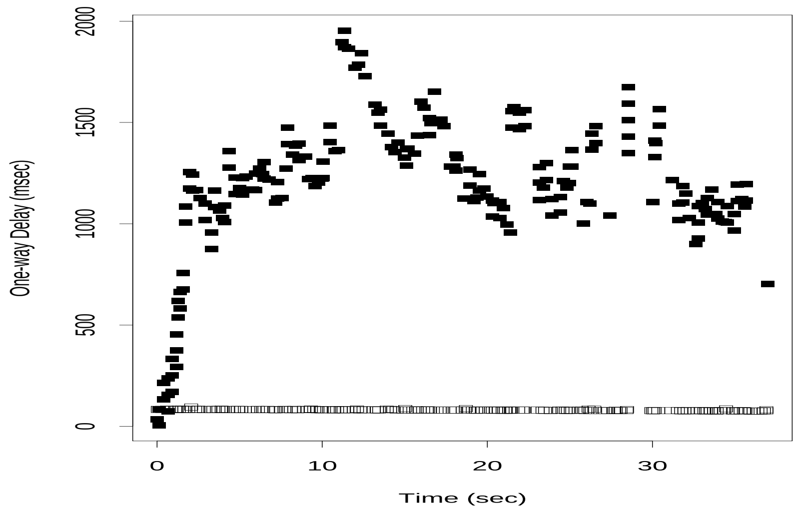

Figure 12.22: Clock skew obscured by network delays

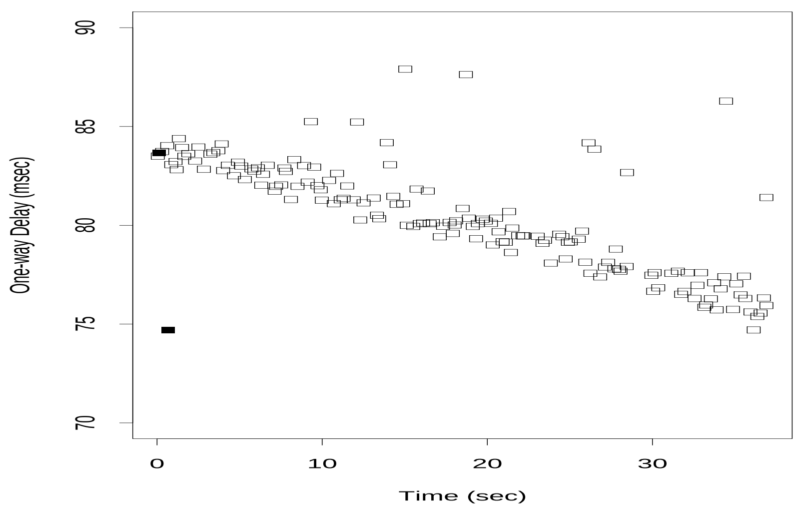

Figure 12.23: Enlargement of reverse path

12.7.3 Failure of line-fitting approaches

Our first attempt to detect relative skew was based on the idea of fitting lines to the OTT plots. We hoped that fits with equivalent and opposite slopes would indicate clock skew, and those without would indicate a lack of skew. One difficulty with this approach is cases of unidirectional noise, as illustrated in the previous section. For these, we can still try to find a very clean fit in one direction, and, if present, to then use it to deduce the presence of skew.

From Figure 12.21 it is clear that the raw OTT measurements are too noisy to hope for clean fitting, as was also the case when testing for clock adjustments. So, we again base our analysis on the de-noised OTT measurements, $\check{s}_t$ and $\check{r}_t$ (§ 12.6.2).

Even using de-noised measurements, least-squares fitting fails to provide solid skew detection, because residual noise in $\check{s}_t$ and $\check{r}_t$ makes it too difficult to reliably distinguish between a skewing trend and coincidental opposite queueing trends. All it takes is one period of elevated queueing at either end of the connection to throw off the fit.

We expected as much, but had high hopes for the robust linear fitting technique discussed in § 9.1.4 as a way of coping with the residual noise. Alas, even this approach fails to reliably detect clock skew. The difficulty lies in both false positives and false negatives generated due to queueing fluctuations. These fluctuations are sufficient to introduce frequent non-zero slopes for the robust fits, and sometimes these slopes happen to have equivalent magnitudes with opposite sign. Furthermore, the fluctuations are often significant enough to alter the slopes so they no longer have equivalent magnitude in the different directions, even though skew is present. Finally, the robust techniques do not offer much help in distinguishing between a genuine skew trend in one direction and noise in the other (§ 12.7.2), versus noise in both directions but no skew.

12.7.4 A test based on cumulative minima

Eventually we recognized that the most salient feature of relative clock skew is not simply the overall trend (slope) of the OTT measurements, but the fact that the smallest such measurements continually increase or decrease. This observation suggests the following statistical test, the strength of which is that it is relatively immune to transient increases in OTT measurements due to queueing buildups.

Suppose we have n observations X_{t_i}, $1 \le i \le n$, where t_i is the time of the observation and X_{t_i} is the value of the observation. We assume that the t_i's are monotone increasing, and that the X_{t_i} are distinct. Further, we assume without loss of generality that we wish to test for a negative trend in X_{t_i}. We discuss applying the same test for a positive trend in § 12.7.5 below.

Consider the indicator:

$$I_{t_j} = \begin{cases} 1, & \text{if } X_{t_j} < \min_{i<j} X_{t_i}, \text{ or if } j = 1, \text{ and} \\ 0 & \text{otherwise.} \end{cases}$$

That is, I_{t_j} is 1 if X_{t_j} represents a new "cumulative minimum" if we inspect X_{t_i} from 1 up to j (but not all the way up to n), and 0 if there is an earlier X_{t_i} that is less than X_{t_j}.

If the X_{t_i} are independent, then:

$$P[I_{t_j} = 1] = 1/j,$$

because the probability that any particular X_{t_i} out of j observations is the minimum of the group is simply $1/j$.

Consider now the function:

$$M_j = \sum_{i=1}^{j} I_{t_i},$$

which is the number of cumulative minima seen as we inspect X_{t_i} from the first value up to the jth value. The key observation we make is that, in the absence of a negative trend, the distribution of M_j will tend to be close to that for independent X_{t_i}; that is, we will find a few cumulative minima but not a great number; while, in the presence of a negative trend, we should find many cumulative minima, since the X_{t_i} tend to get smaller and smaller.

Suppose we find $M_n = k$, that is, the X_{t_i} exhibit k cumulative minima. We wish to compute the probability that we would have observed this many or more minima, given the independence assumption. If we find the probability sufficiently low, we will reject the null hypothesis that the X_{t_i} are independent. In its place we will accept the tentative hypothesis (which we will further test in § 12.7.6) that the X_{t_i} exhibit a negative trend.

Let:

$$R(n, k) = P[M_n \geq k].$$

Given $0 \leq k \leq n$, we can compute $R(n, k)$ recursively, as follows:

$$R(n,k) = \begin{cases} 1, & \text{if } k = 0, \\ 1/n!, & \text{if } k = n, \text{ and} \\ R(n-1,k-1)(1/n) + R(n-1,k)(1-1/n) & \text{if } k < n. \end{cases} \tag{12.11}$$

The first case is the degenerate one that grounds the recursive definition: the probability that there are at least 0 cumulative minima is always 1.

The second case corresponds to every single X_{t_i} being a cumulative minimum. This only occurs if the X_{t_i}'s are sorted in descending order, which, if they are independent, has probability $1/n!$, since there are $n!$ permutations of the X_{t_i}, only one of which is sorted (because the X_{t_i} are distinct).

The last case corresponds to conditioning on whether X_{t_n} is a cumulative minimum or not. For independent X_{t_i}, it will be a cumulative minimum with probability $1/n$. In this case, for the n points to exhibit at least k cumulative minima, the $n-1$ points prior to X_{t_n} must themselves exhibit at least $k-1$ cumulative minima, which occurs with probability $R(n-1, k-1)$. If, however, X_{t_n} is not a cumulative minimum, which occurs with probability $1-1/n$, then the $n-1$ prior points must exhibit at least k cumulative minima, which occurs with probability $R(n-1, k)$.

We can compute $R(n, k)$ in $O(n^2)$ time using straight-forward dynamic programming. Furthermore, if the dynamic programming is done using a "memo" function that remembers its previously-computed results in a table, then additional computations of $R(n, k)$ will benefit from earlier computations, and the evaluation becomes extremely cheap.

Figure 12.24 shows the distribution of $R(n, k)$ for $n = 15$. The key feature of the distribution that makes it a powerful test for a negative trend is the rapid fall-off in probability above a certain point, in this case around $k = 8$. Because if the X_{t_i}'s do indeed have a negative trend we should find k quite close to n, this means we can readily distinguish between the case of a negative trend and that of no trend, without requiring that *all* of the X_{t_i} be increasingly negative. Thus, we can accommodate considerable noise.

Finally, we take as for the size of the trend the slope computed by a robust linear fit (§ 9.1.4) to X_{t_i}'s minima. This corresponds to the value G_s or G_r discussed in § 12.7.1 above.

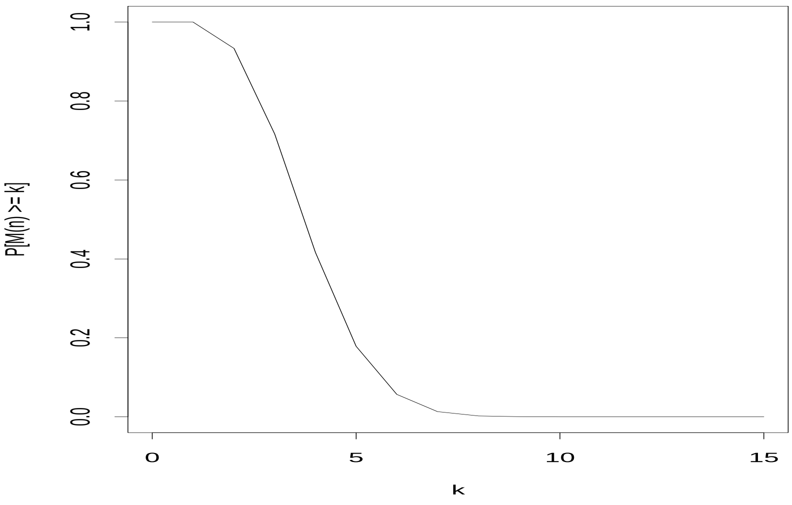

Figure 12.24: Distribution of $R(n,k)$ for $n = 15$

12.7.5 Applying the test to a positive trend

The test developed in $\S$ 12.7.4 for detecting a negative trend can also be applied to detecting a positive trend, with one subtlety. At first blush one might think that, to do so, one simply uses maxima in lieu of minima. This works in principle, but fails when applied to OTT sequences, because of the positive additive nature of OTT noise ($\S$ 12.6.2). That is, the maxima will be often dominated by the noisiest OTT values, rather than by OTT values that slowly rise due to skew, so the noise will obscure any positive trend due to clock skew. This remains a problem even after de-noising, since all it takes is a single period of elevated OTT values, long enough to span an entire de-noising interval, to pollute the de-noised values with what will in some cases by a global maximum. When searching for a negative trend, such an interval will, on the other hand, simply not include a minimum; but it will not prevent the test from finding other minima due to clock skew.

There is a simple fix for this problem, though. The key observation is that the smallest OTT values are in general those with the least noise. So we apply the cumulative minima test to $Y_{t_j} = X_{t_{n-j+1}}$, which is simply X_{t_i} viewed in reverse. The reversal converts a positive trend in X_{t_i} to a negative trend in Y_{t_j}, which the cumulative minima algorithm then readily detects.

Finally, for a given series X_{t_i} we need to decide whether to test it for a positive or negative trend. We do this by first performing a robust linear fit to the observations. If the slope of the fit is positive, we look for a positive trend; if negative, a negative trend; and if exactly zero, we decree there is no trend.

12.7.6 Identifying skew trends

With the cumulative minima test we finally have a robust algorithm for detecting trends. These trends, however, might not be due to clock skew but to networking effects, so we need to develop further *heuristic* checks to correctly detect linear skew.

Suppose we have two sequences of de-noised OTT measurements, $\check{s}_t$ and $\check{r}_t$, corresponding as usual to the full-sized data packets sent from the connection sender to the receiver, and the acks sent back from the receiver to the data sender. For each sequence, we first determine whether it is a *skew candidate* as follows.

Let u_t denote the given sequence. Let $R_u(n,k)$ be the probability that the sequence u_t matches the null hypothesis of no trend (independence) given by Eqn 12.11. We consider u_t a skew candidate if either:

1. $R_u(n,k) < 10^{-6}$ and u_t is either $\check{r}_t$, or u_t is $\check{s}_t$ and its trend is negative. This latter test is because queueing buildup due to the data packets sent along the forward path can often produce a strong positive trend; or

2. $R_u(n,k) < 10^{-3}$ and u_t is *tightly clustered* around the trend line. The goal here is to allow for a skew candidate if the u_t points fit quite closely to a (linear) trend, even though their cumulative minima probability is not so small. This can happen, for example, if we do not have a large number of points in u_t. For example, if we have only 7 points in u_t, then the smallest possible value of $R_u(n,k)$ is

$$R_u(n,n) = R_u(7,7) = \frac{1}{7!} \approx 2 \cdot 10^{-4},$$

   which will fail the $R_u(n,k) < 10^{-6}$ test in the previous item.

   Note that the limit of 10^{-3} precludes assuming a skew candidate if there are fewer than 7 points, since $1/6! \approx 1.4 \cdot 10^{-3}$ (but see below).

   It remains to define "tightly clustered." To do so, we compute the inter-quartile range (75th percentile minus 25th percentile, per § 9.1.4). If it is less than or equal to the larger of the joint clock resolution, $R_{s,r}$, or 1 msec, then a large number of the de-noised OTTs lie very closely to a pure linear trend.

We then proceed to determine whether either $\check{s}_t$ or $\check{r}_t$ is compelling enough by itself to accept as evidence of a skew trend; or if the pair form a *joint skew candidate* to be investigated further; or if there is insufficient evidence for a skew trend. To do so, we first consider which of them is individually a skew candidate, as follows:

1. If neither is a candidate, then we check to see whether $\max(R_s(n,k), R_r(n,k)) \leq 10^{-2}$. If so, then the joint probability that both have no trend (or, more precisely, are fully independent) is $\leq 10^{-4}$, which we consider sufficiently low to consider them as joint skew candidates and proceed as discussed below. If either probability exceeds 10^{-2}, then we reject the trace pair as a candidate for exhibiting a skew trend.

2. If $\check{r}_t$ is a skew candidate but $\check{s}_t$ is not, then we accept $\check{r}_t$ as reflecting clock skew quantified using the corresponding G_r. We do so because sometimes we have no hope of detecting a skew trend in $\check{s}_t$ due to queueing buildup, as illustrated in Figure 12.22 and Figure 12.23.

3. If $\check{s}_t$ is a skew candidate but $\check{r}_t$ is not, then we check the direction of $\check{s}_t$'s trend. If it is negative, then this goes against the networking tendency for a positive trend induced by the queueing of

the data packets along the forward path, and we accept $\check{s}_t$ as reflecting clock skew quantified using G_s.

If the trend is positive, we must proceed carefully to screen out a false skew trend due to queueing. First, we require

$$\sigma^2_{\check{s}_t} \leq \sigma^2_{\check{r}_t},$$

that is, the variance of the de-noised OTT values along the forward path is less than that in the reverse path. If this is not the case, then we reject the trace pair as a candidate for exhibiting a skew trend.

Next we split $\check{s}_t$ into two halves, $\check{s}_{t_1}$ and $\check{s}_{t_2}$, with the division coming at $\lfloor \frac{n}{2} \rfloor$ if s_t has n values. If $R(n,k)$ for either half exceeds 10^{-2}, or if the trends for the two halves do not agree in direction, then we also reject the possibility of a skew trend.

If $\check{s}_t$ passes these tests, then we consider $\check{s}_{t_1}$ and $\check{s}_{t_2}$ as comprising a joint skew candidate. We reverse $\check{s}_{t_2}$ so it now has the opposite trend of $\check{s}_{t_1}$, and proceed as discussed below.

4. If both $\check{s}_t$ and $\check{r}_t$ are skew candidates, then we consider them together a joint skew candidate.

If the above procedure yields a joint skew candidate, we then evaluate the candidate as follows:

1. If both candidates have the same trend direction, then we reject the possibility of a skew trend.

2. If not, then we translate the first candidate's skew quantification into terms of the second candidate using Eqn 12.10. Let G_1 and G_2 be the corresponding skew quantifications (one of which has been translated, so they can be directly compared). If

$$|G_1 - G_2| > \frac{G_1 + G_2}{2},$$

that is, the difference between the two exceeds their average, then we reject the pair as having too much variation in their slopes for them to be trustworthy indicators of skew. Otherwise, we accept the pair as indicative of a skew quantified as $G = \frac{G_1+G_2}{2}$.

12.7.7 Results of checking for skew

`tcpanaly` uses the method given in § 12.7.6 to check each trace pair it analyzes for clock skew. We found that 295 trace pairs in $\mathcal{N}_1$ out of 2,335 (13%) exhibited clock skews, and 487 out of 15,492 did so in $\mathcal{N}_2$ (3%). These proportions are high enough to argue for considerable caution when comparing timestamps from two different packet filters.

In both $\mathcal{N}_1$ and $\mathcal{N}_2$, about three-quarters of the skews were detected on the basis of $\check{r}_t$ alone, not particularly surprising since often a skew trend in $\check{s}_t$ will be lost in the OTT variations due to queueing induced by the data packets. The largest skew in $\mathcal{N}_1$ was a whopping $\eta = 5.5$, meaning that one clock ran *more than five times faster than the other*! Figure 12.25 shows how skew like this appears in an OTT pair plot. Note that the reverse path starts a time $T = -4$ sec because `tcpanaly` could not figure out any sort of useful relative clock offset. In the forward direction, the connection's elapsed time was only 2 sec, but in the reverse direction it took 10 sec!

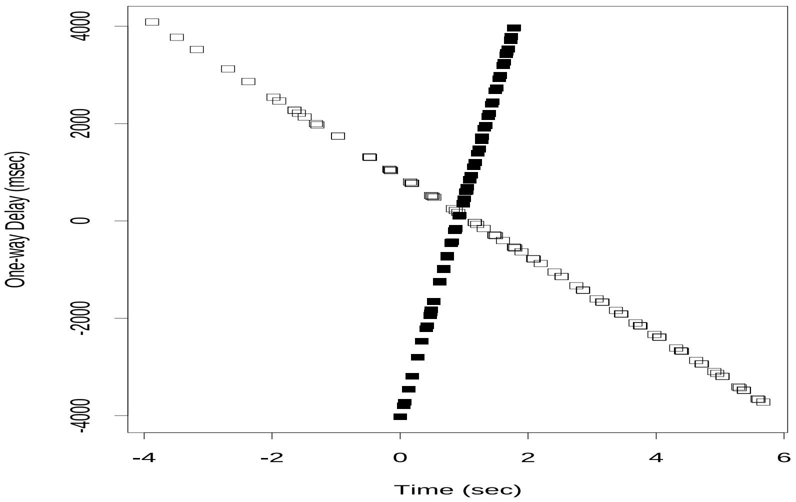

Figure 12.25: Example of extreme clock skew

This example is more than just an amusing curiosity. It occurred not once but 43 times in $\mathcal{N}_1$. Each time, the slower clock belonged to `austr`, and that was indeed the erroneous clock. We know it was the broken clock of the pairs exhibiting the problem not just because it was always one member of each problematic pair (which would be convincing by itself), but also because RTTs in those connections computed using its timestamps are physically impossible (too small) for the long distances traversed by the packets it sent and received. We likewise see the onset of this problem above in Figure 12.3. Note, however, that `austr`'s clock was one of the ones identified in § 12.5.3 as being *highly* synchronized with a number of the other sites, indicating care was being taken to keep accurate time with it (presumably using NTP). Thus, this clock's behavior is an compelling argument that *just because a clock is believed to be well-synchronized does not render it immune from extreme error!*

Aside from `austr`'s clock, the next largest skew we observed in $\mathcal{N}_1$ was $\eta = 0.991$, a frequency difference of about 0.9%. This led to an OTT change of about 70 msec during an 8 sec connection. All in all, after removing connections involving `austr`, in $\mathcal{N}_1$ the median skew had a magnitude of about 0.023%, and the mean 0.035%. These are small, but not negligible, as discussed at the beginning of § 12.7.

In $\mathcal{N}_2$, the prevalence of trace pairs exhibiting skew was significantly lower (3% versus 14%), perhaps due to the use among the participating sites of newer hardware with more reliable clocks. Apart from `oce`'s clock, which we discuss in § 12.7.8 below, the largest skews we observed were on the order of 6%. One of these was the example of clock adjustment using skew in Figure 12.15 above. Figure 12.26 shows another example. The pattern is quite striking, and clearly could lead to grossly inaccurate conclusions about network dynamics if undetected. Note that both sites involved in this connection, `nrao` and `ustutt`, were among those identified as closely synchronized in $\mathcal{N}_2$ (§ 12.5.3), again emphasizing that clocks that are *in general* well-synchronized can still exhibit very large errors.

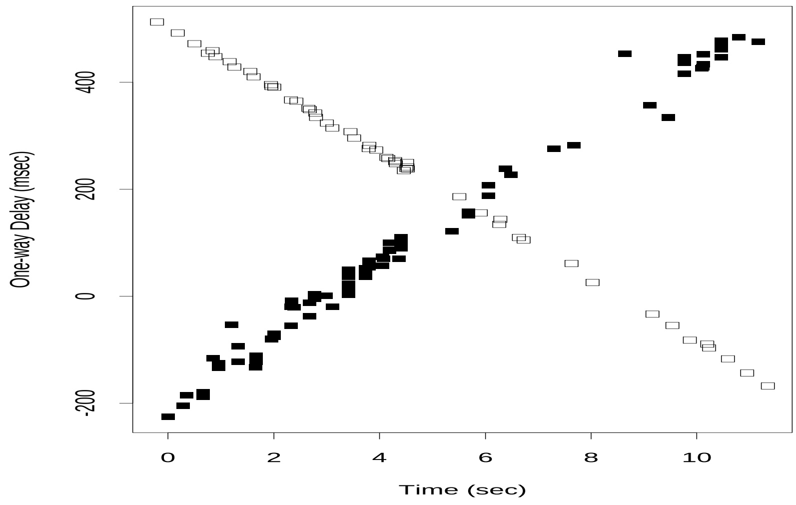

Figure 12.26: Strong relative clock skew of 6%

If we remove `oce`'s connections and those with skews larger than 1%, then the median skew magnitude of the remainder in $\mathcal{N}_2$ is about 0.011%, and the mean around 0.016%. These are a factor of two smaller than those in $\mathcal{N}_1$, but still not completely negligible for assessing queueing in longer-lived connections.

12.7.8 `oce`'s puzzling dynamics

When testing the $\mathcal{N}_2$ trace pairs for clock skew, we repeatedly encountered puzzling dynamics (or clock behavior) for some of the connections originated by `oce`, and, to a lesser degree, some of those in which `oce` was the receiver of the TCP transfer. (This did not occur for `oce` connections in $\mathcal{N}_1$.) Figures 12.27 and 12.28 show the general pattern of behavior. The connections have exceptionally high RTTs, more than 2 sec. These times far exceed the intrinsic propagation delay from the remote sites to `oce`. Furthermore, `traceroutes` from `oce` to other sites often show a first hop RTT on the order of 2 sec; thus, almost all of the delay is occurring right at `oce`'s border to the Internet.

Another part of the puzzle is the shift in OTTs from almost all of the total delay being incurred by the acks incoming to `oce`, to almost all of it being incurred by the data packets outbound from `oce`, back to the incoming acks again. The pattern is sometimes a bit different. Figure 12.29 shows a trace for which during most of the trace's 7.5 minute lifetime, the ack OTTs were virtually constant, while those for the data packets fluctuated enormously (1000's of msec). Then, at $T = 235$ sec, the ack OTTs suddenly begin to increase by a whopping 8 seconds, only to return to 1 sec again after a 75 second outage.

One possible explanation is that the network path between `oce` and the rest of the Internet exhibits what we term *half-duplex self-interference*. That is, somewhere on the path, probably at the first hop, there is a half-duplex link that does not fairly arbitrate between traffic in the two directions.

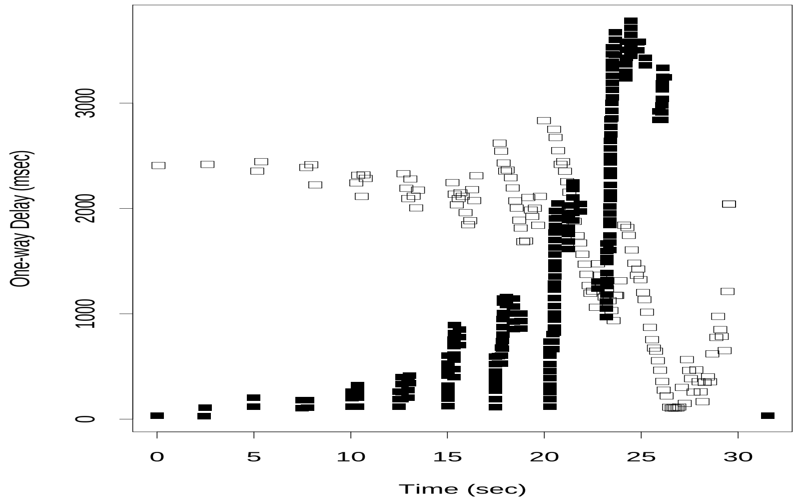

Figure 12.27: Example of puzzling `oce` behavior

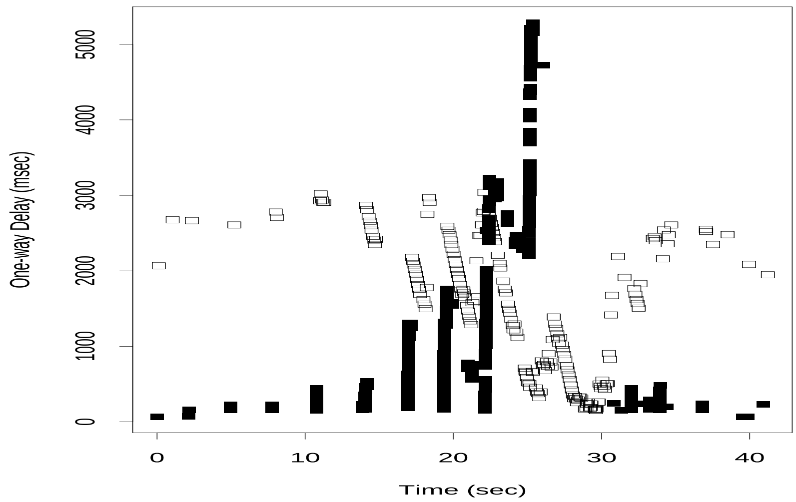

Figure 12.28: Another example of puzzling `oce` behavior

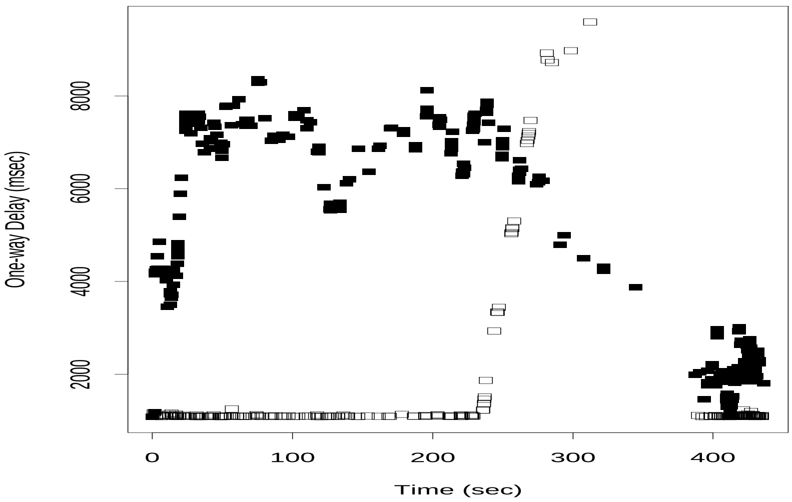

Figure 12.29: One more example of puzzling oce behavior

Initially, the data packets get first use of the link, and the acks must wait for their turn. Eventually, the phasing between which end of the link has preference shifts, so the acks gain preference and the data packets must wait, and with time it then shifts back.

One can imagine half-duplex self-interference occurring on any heavily-loaded half duplex link that does not explicitly guarantee fairness between the hosts using the link. For example, Ethernet networks can exhibit a "capture effect" in which the host using the network is unfairly able to continue using it longer than intended [RY94]. Another half-duplex networking technology that can exhibit unfairness on small time scales is FDDI, in which a single host can continue to use the ring for up to the "token holding time" [Jai90]. We have observed "ack compression" (§ 16.3.1) on high-speed network paths in which it appears that the compression is not due to network-layer queueing, but instead to link-layer delays, in which a TCP connection's acks wind up waiting for an FDDI token that is being hoarded by the same connection's data packets traveling in the opposite direction.

While half-duplex self-interference would explain the interplay between the oce forward and reverse OTT variations, it does not by itself explain the very large first-hop delay associated with the behavior. It may be that reversing the direction in which the link is being used is a very expensive operation (perhaps because of low-layer errors and retries; it seems unlikely such an expensive mechanism would be designed into a data link). The oce staff was unable to obtain an explanation for the phenomenon from their networking provider. oce does have a firewall in place through which the NPD traffic must transit, but it would be extremely poor performance for a firewall to add 2 seconds of latency to every packet it forwarded.

The final part of the puzzle concerns oce's clock. As discussed in § 12.5.3, its clock was the least-well synchronized in both $\mathcal{N}_1$ and $\mathcal{N}_2$. Even for those $\mathcal{N}_2$ oce connections that did not exhibit this sort of behavior (and many did not), the clock often exhibited skew. It is possible that oce's puzzling network dynamics makes synchronizing the clock difficult. But it is also quite possible that at least some of the puzzling dynamics are due to the clock itself (i.e., measurement artifacts), since the variations resemble quite closely the signature of a clock that is varying its rate over short time scales. The only problem with this explanation is the fact that the connections much more often start with elevated OTTs for the return path that then decrease as the forward path OTTs increase (Figure 12.27 and Figure 12.28) than the other way around (Figure 12.29). If the behavior were due to a variable-rate clock, then we would instead expect the clock to be equally likely to start the connection running at an elevated rate as at a depressed rate. For the OTT patterns to be due entirely to a misbehaving clock requires that somehow fluctuations in the clock's variable rate are tied with the host computer's network traffic. It is difficult to see what sort of mechanism could create this linkage, however.

Because the magnitude of the effect is sometimes so large, and because we could not rule out clock behavior as a source for the behavior or part of the behavior, we decided to eliminate all of the $\mathcal{N}_2$ `oce` connections from any analysis that involved timestamps produced by its clock. (But, for example, we still analyze its connections for statistics like proportion of packets lost, since these do not rely on timestamps.)

12.7.9 Removing relative skew

As discussed in the previous section, a non-negligible proportion of the trace pairs in our study suffer from relative clock skew. We would like to remove this skew so we can then reliably include those traces in our analysis of network dynamics. Fortunately, the skew almost always appears well-described as linear, which means it is straight-forward to remove it.

To remove skew of magnitude η, we simply modify all the timestamps t_i^r generated by C_r using:

$$t_i^{r\prime} = t_i^r + G_r(t_i^r - t_0^r), \tag{12.12}$$

where G_r is given by Eqn 12.9 and t_0^r is the first timestamp generated by C_r. To understand this transformation, recall from § 12.7.1 that G_r gives the trend in how OTTs for packets sent by r change with time. If $G_r > 0$, then the OTTs increase with time, indicating that C_r runs more slowly than C_s, and to adjust it we need to increase the timestamps it generates. If $G_r < 0$, then the OTTs decrease with time, and we need to decrease C_r's timestamps to effectively it slow down.

A key point is that applying Eqn 12.12 does *not* necessarily rectify C_r's skew with respect to *true time*. It only rectifies it with respect to C_s. It could be that the correct action to take in terms of true skew removal is to apply an analogous transformation to C_s's timestamps *instead*. We have no way of knowing which clock is in error, but by Eqn 12.12 we can at least make the two sets of timestamps consistent.

Indeed, both clocks could be skewed with respect to true time, in which case neither action will correct them in an absolute sense. But *for purposes of comparing the clocks' timestamps to compute OTTs and infer queueing delays from them, the most important consideration is that the two clocks have no relative skew.* Provided the absolute skew is small (say $< 1\%$), then its only effect is that the magnitude of the computed OTTs (and RTTs) will be off by an equally small amount. By correcting the relative skew, we remove potentially quite large, artificial OTT *trends*, and there lies our primary goal.

`tcpanaly` uses Eqn 12.12 to take out relative clock skew if its magnitude is less than 1%. If it is larger, then it flags the trace pair as having large relative skew and will not do any timing-based analysis.

Finally, after tcpanaly removes relative skew, it re-analyzes the clock. If it still detects relative skew, then either its initial assessment that the trace pair had relative skew was wrong, or the skew was not linear. It flags this case separately, and also then refrains from any further timing analysis. Thus, re-analysis provides a self-consistency test for the soundness of our skew detection. Only 1 of the 295 $\mathcal{N}_1$ trace pairs flagged as having relative skew failed this additional test, and only 10 of the 487 $\mathcal{N}_2$ trace pairs failed. Of these 13, three involved the puzzling oce behavior discussed in § 12.7.8, seven appear to have been false skew assessments due to network noise, and one had definite skew but enough noise along the reverse path to lead to misassessment of the magnitude of the skew.

12.8 Additional clock consistency checks

Along with testing the timestamps in trace pairs for clock adjustments and relative skew using the methods developed above, we apply two final self-consistency checks to the timestamps in an attempt to calibrate their accuracy.

12.8.1 Non-positive min-RTT$_{sr}$

We stated in § 12.5.1 that min-RTT$_{sr}$, as given by Eqn 12.7, should always be positive. tcpanaly flags any trace pair for which it is non-positive. It also checks for whether a non-positive min-RTT$_{sr}$ was the *only* indication of a clock problem, as this means that our main heuristics failed to detect a measurement problem. This happened four times in $\mathcal{N}_1$ and twelve times in $\mathcal{N}_2$, rarely enough to give us considerable confidence in our heuristics.

Most of the missed clock problems were due to one of the following: failing to detect skew in the presence of considerable noise; failing to detect adjustments due to noise or their occurrence at the edge of a connection (§ 12.6.5); or dealing with connections for which the RTT is on the order of the clock accuracy (some between sintef1 and sintef2).

Of the three remaining problems flagged only by the min-RTT$_{sr}$ check, one was due to tcpanaly failing to detect unreliable packet filter timestamps (§ 10.3.6), and the other two were due to a bizarre packet filter timing problem in which the filter appears to have waited many seconds before starting to timestamp packets at the beginning of a connection. Thus, for example, a connection between sdsc in San Diego and korea, on the other side of the Pacific, had packet filter timestamps from the korea tracing machine showing that the initial SYN handshake took only 4 msec to complete, while the San Diego packet filter reported it took 510 msec! Physically the first value is impossible, as the propagation time across the Pacific is much larger than 4 msec. Further inspection shows that packet timings on the korea end varied wildly at the beginning of the connection, yielding a swing of more than 10 seconds in the OTTs, after which they settled down and remained quite even. Figure 12.30 shows the corresponding OTT pair plot. Had this occurred in only one trace then we would have concluded the measurement had the bad luck to encounter a clock adjustment right at the connection's beginning, but it happened similarly in a second korea trace, indicating instead a packet filter timing problem associated with the beginning of a connection trace.

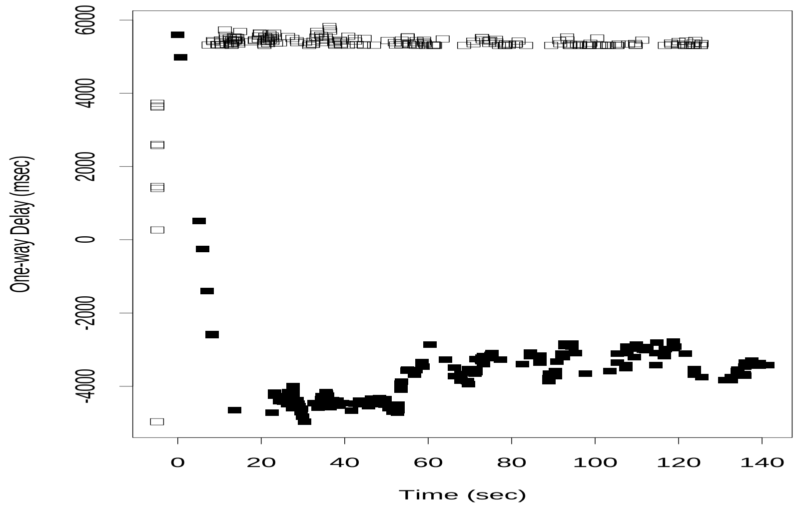

Figure 12.30: Initial packet filter timing glitch

12.8.2 Gap analysis

The final self-consistency check is based on the following observation. Suppose host s sends a packet at time s_1, measured by C_s, and it arrives at r at time r_1, according to C_r. Later, r sends a packet at r_2, arriving at s_2. It should always be the case that:

$$s_2 - s_1 > r_2 - r_1, \tag{12.13}$$

because r_1 reflects an event that occurred *after* s_1, and r_2 reflects an event that occurred *before* s_2. Put another way, if all of the timestamps were accurate, then we would have:

$$s_1 < r_1 < r_2 < s_2,$$

and, even if C_s and C_r have a relative offset $\Delta C_{r,s}$ between them, as long as the offset is fixed, then the inequality in Eqn 12.13 follows, since the subtractions remove the effects of the offset. Eqn 12.13 might *not* hold, however, if C_s is running slower than C_r, or if C_s is adjusted backward (or C_r forward) in between s_1 and s_2 (in between r_1 and r_2).

We term checking whether Eqn 12.13 holds as "gap analysis." Exhaustively testing all of the packet arrivals and departures for consistency with Eqn 12.13 requires $O(n^2)$ time for n packets, since each departure of a sender packet can be paired with the departures of any of the receiver's packets sent after it. To avoid this cost, `tcpanaly` instead employs a strategy of "burning the candle at both ends," namely it checks Eqn 12.13 for the first packet and the last ack; then for the next packet and the penultimate ack; and so on, until it works its way to the middle of the connection. Doing so reduces $O(n^2)$ time to $O(n)$, at the cost of perhaps missing some instances in which Eqn 12.13 fails to hold, though the strategy still spans a wide range of gap intervals. `tcpanaly` also does gap analysis from the receiver's perspectives (where s is the host generating acks and r the host

Dataset	Relative offset	Likelihood of adjustment
$\mathcal{N}_1$	$<$ 1 sec	1.4 %
$\mathcal{N}_1$	$\geq$ 1 sec	1.6 %
$\mathcal{N}_2$	$<$ 1 sec	0.75 %
$\mathcal{N}_2$	$\geq$ 1 sec	0.95 %

Table XVI: Relationship between relative clock accuracy and clock adjustments

generating subsequent data packets). It needs to check both perspectives in order to detect relative skew and adjustments in which *either* of the two clocks runs faster than the other.

Gap analysis finds some but by no means all of the clock adjustment and skew problems uncovered by the more robust techniques developed earlier. However, it also serves as a self-consistency check: we would like to know that the robust techniques find *all* of the clock problems, so we would hope that gap analysis never uncovers a problem missed by the others. It did so only once, the problem being a clock "hiccup" (§ 12.6.5) in which a connection with OTTs of about 3 msec (from `lbl` to `sandia`) had a single packet with an OTT of 430 μsec!

12.9 Clock synchronization vs. stability

We finish our study of clock calibration with an investigation into the question of whether highly-synchronized clocks tend to be free of problems such as adjustments and skew. We will term clocks free of such problems as "stable."

We might hope that highly-synchronized clocks would also be stable, because freedom from such problems would tend to greatly aid a clock in maintaining synchronization. On the other hand, if good synchronization is maintained by frequently adjusting an errant clock to match an external notion of accurate time, then such clocks might be *more* likely to exhibit adjustments or skew (§ 12.2), and hence be less stable than other clocks.

The issue is an important one because it is quite cheap to determine whether a remote clock's offset is close to that of a local clock (§ 12.5.1). If relative accuracy is a good indicator that the remote clock is stable, then we can quickly determine that we can rely on the soundness of the timestamps generated by the remote clock, without having to go through all the effort of the methods developed in this chapter for detecting adjustments and skew. Such a quick determination could prove invaluable for a transport protocol that needs to decide whether it can trust the timing feedback information being returned from a remote peer. The hope is that the protocol can do so by looking at just a few initial timestamps.

Table XVI shows the relationship between relative clock accuracy and the likelihood of observing a clock adjustment. We see that closely synchronized clocks, i.e., those with a relative offset under 1 sec, are only slightly less likely to exhibit a clock adjustment than less closely synchronized clocks. Thus, relative clock accuracy is not a good predictor of the absence of clock adjustments.

Table XVII shows the relationship between relative clock accuracy and the likelihood of

Dataset	Relative offset	Likelihood of skew
$\mathcal{N}_1$	< 0.01 sec	0.95%
$\mathcal{N}_1$	< 0.1 sec	5.6%
$\mathcal{N}_1$	< 1 sec	13 %
$\mathcal{N}_1$	≥ 1 sec	12 %
$\mathcal{N}_2$	< 0.001 sec	1.3 %
$\mathcal{N}_2$	< 0.01 sec	0.88 %
$\mathcal{N}_2$	< 0.1 sec	1.3 %
$\mathcal{N}_2$	< 1 sec	1.8 %
$\mathcal{N}_2$	≥ 1 sec	5.3 %

Table XVII: Relationship between relative clock accuracy and clock skew

observing relative clock skew.[8] For $\mathcal{N}_1$, clock synchronization only provides an advantage if the clocks are highly synchronized, with a relative offset under 100 msec and preferably under 10 msec. For $\mathcal{N}_2$, however, synchronization of under 1 sec provides a definite advantage in predicting a lower likelihood of skew, though much better synchronization provides little additional predictive power. For both $\mathcal{N}_1$ and $\mathcal{N}_2$, not even very close synchronization reduces the likelihood of encountering clock skew to a negligible level (i.e., appreciably lower than 1%).

In summary, we conclude that relative clock accuracy provides no benefit in assuring that clock adjustments will be unlikely, and some benefit in assuring that clock skew is less likely, but not to such a degree that we can ignore the possibility of clock skew when analyzing more than a handful of measurements.

In addition, we conjecture that the closely-synchronized hosts in our study are most likely synchronized using NTP. If so, then the use of NTP does *not* reduce the likelihood of clock adjustments introducing systematic errors when measuring packet transit times, and reduces but does not eliminate the likelihood of clock skew introducing systematic errors. This finding does *not* mean that NTP fails to keep good time. Rather, the timescales on which it does so significantly exceed those of our connections. NTP keeps good time on large time scales precisely by altering clock behavior on small time scales.

Thus, prudent large-scale measurement and analysis of packet timings should include algorithms such as those developed in this chapter as self-consistency checks to detect possible systematic errors, even in the presence of NTP-synchronization. We further argue that even pairs of clocks using a more direct external synchronization source such as GPS should be subjected to such checks, as a means of assuring that no timing errors have crept in between the original, highly accurate time source, and the timestamps ultimately produced by the packet filters.

[8] The percentages given in the table include the outlier sites of `austr` in $\mathcal{N}_1$ and `oce` in $\mathcal{N}_2$. However, these sites only affect the ≥ 1 sec row, since their relative offsets were large; and, it seems legitimate to leave them in the summaries since they are indeed instances of large relative offsets indicating an increased likelihood of clock skew.

Chapter 13

Network Pathologies

After correcting for packet filter errors (Chapter 10) and TCP behavior (Chapter 11), we next turn to analyzing network behavior we might consider "pathological," meaning unusual or unexpected. When we present a series of packets to the network for delivery to a remote endpoint, a number of things might happen. The network can:

(i) deliver them as we asked;

(ii) fail to deliver them at all (packet loss, cf. Chapter 15);

(iii) unduly delay them (packet delay, cf. Chapter 16), where "unduly" does not have a precise definition, except perhaps "causing unnecessary retransmission";

(iv) deliver them in a different order than sent (out-of-order delivery, § 13.1);

(v) deliver them more than once (packet replication, § 13.2);

(vi) deliver imperfect copies of them (packet corruption, § 13.3).

All but "deliver them as we asked" are in some sense unusual or unexpected, though to varying degrees. The first two unusual behaviors are in fact often expected; we devote two subsequent chapters to analyzing them in depth. The last three are less often expected, and we discuss them in the remainder of this chapter. It is important that `tcpanaly` recognize these sorts of pathological behaviors so that its subsequent analysis of packet loss and delay is not skewed by the presence of pathologies. For example, it is very difficult to perform any sort of sound queueing delay analysis in the presence of out-of-order delivery, since the latter indicates that a first-in-first-out (FIFO) queueing model of the network does not apply.

13.1 Out-of-order delivery

While Internet routers almost always employ FIFO queueing, the packet-switched nature of the network provides one common mechanism for reordering packets so that they arrive in a different order than sent: whenever the routes taken by two packets differ, and the second packet enjoys a sufficiently shorter transit time than the first, then reordering can occur [Mo92]. The designers of TCP were well aware of this fact, and engineered TCP for resilience in the face of out-of-order delivery, as well as the other pathologies enumerated above.

In the context of a transport protocol like TCP that sequences its data stream, we need to make a distinction between *out-of-order* delivery, which is caused by the network, and *out-of-sequence* delivery, which is caused by the either the network (due to packet loss), or the transport protocol (due to retransmission).

From a trace recorded at a TCP receiver, we cannot always distinguish between these two, though two heuristics often work well. The first is checking whether the IP "id" field (§ 10.3.5) of two packets exhibits a small backward skip. Since each IP packet sent by a host typically increments the field by one, a backward skip usually only occurs due to reordering. The second is to look at the length of time between the arrival of the first (out-of-order or out-of-sequence) packet and that of the second. If it is on the order of the round trip time (RTT) or higher, then it is likely that the first packet is a retransmission. If it is quite short, then it is likely due to network reordering.

Since we have traces recorded at both ends of each TCP connection, and since we can reliably pair departures recorded in one trace with arrivals in the other (§ 10.5), we can more directly detect network reordering. `tcpanaly` does this as follows.

13.1.1 Detecting out-of-order delivery

To analyze network reordering between endpoints s and r, with corresponding packet traces $\mathcal{T}_s$ and $\mathcal{T}_r$, we first check to see whether we have previously determined that r's packet filter suffers from resequencing (§ 10.3.6), or if we were unable to pair the packets in the two traces due to ambiguities (§ 10.5). If either of these occurred, we skip further analysis. Otherwise, we scan the packet arrivals in $\mathcal{T}_r$. For each arriving packet p_i recorded in the trace, we check whether it was sent after the last non-reordered packet, p_N. If so, then we set $p_N \leftarrow p_i$, and proceed to the next arrival.

If, however, p_i was sent before p_N, then we count p_i's arrival as an instance of a network reordering. So, for example, if a flight of ten packets all arrive in the order sent except the last one arrives before all of the others, we consider this to reflect 9 reordered packets rather than 1. Likewise, if the first arrives after all the others, and otherwise all arrivals are in order, we consider this as reflecting 1 reordered packet. Using this definition emphasizes "late" arrivals rather than "premature" arrivals. It turns out that counting late arrivals gives somewhat higher numbers than counting premature arrivals, but the difference is not that great ($\approx 25\%$).

`tcpanaly` further computes statistics on how many packets were sent between p_i and p_N, how many of these arrived prior to p_N, and how much time elapsed between the arrival of p_i and that of p_N. After analyzing packets sent from s to r, it then repeats the process for those sent from r to s.

13.1.2 Results of out-of-order analysis

Out-of-order packet delivery proved much more prevalent in the Internet than we had expected (prior to the routing pathology analysis in § 6). In $\mathcal{N}_1$, 36% of the traces included at least one packet (data or ack) delivered out of order, while in $\mathcal{N}_2$, 12% did. Overall, 2.0% of all of the $\mathcal{N}_1$ data packets and 0.61% of the acks arrived out of order, while in $\mathcal{N}_2$ the corresponding figures fell to 0.26% and 0.10%. It is not surprising that data packets are significantly more often reordered than acks, because they are frequently sent closer together than acknowledgements due to ack-every-other acking policies (§ 11.6.1), and so reordering for data packets requires less of a difference in transit times than reordering for acks.

We should *not* infer from the differences between reordering in $\mathcal{N}_1$ and $\mathcal{N}_2$ that reordering became less likely over the course of 1995, because out-of-order delivery varies greatly from site to site. For example, 15% of the data packets sent by `ucol` during $\mathcal{N}_1$ arrived out of order, far exceeding the average for the entire dataset. Likewise, reordering is highly asymmetric. For example, only 1.5% of the data packets sent *to* `ucol` during $\mathcal{N}_1$ arrived out of order. Furthermore, while for some sites out-of-order delivery of packets sent *from* the site strongly correlated with out-of-order delivery of those sent *to* the site, for other sites (such as `ucol` and `wustl`) the two directions were uncorrelated. This means a TCP cannot soundly infer whether the packets it sends are likely to be reordered, based on observations of the acks it receives. This is unfortunate, because, if a TCP could make this assumption, then it could more accurately determine the correct duplicate ack threshold to use for fast retransmission (see § 13.1.3 below).

The site-to-site variation in reordering directly matches our findings concerning route flutter (§ 6.6). In that analysis, we identified two sites as particularly exhibiting flutter, `ucol` and `wustl`. For the part of $\mathcal{N}_1$ during which `wustl` exhibited route flutter, 24% of all of the data packets it sent arrived out of order, a rather stunning degree of reordering. If we eliminate `ucol` and `wustl` from the analysis, then the proportion of all of the $\mathcal{N}_1$ data packets delivered out-of-order falls by a factor of two. Clearly, these two sites heavily dominate $\mathcal{N}_1$ reordering. Finally, we note that, in $\mathcal{N}_2$, data packets sent by `ucol` were reordered only 25 times out of nearly 100,000 sent, though 3.3% of the data packets sent *to* `ucol` arrived out of order, dramatizing how, over long time scales, site-specific effects can completely change.

Thus, we should not interpret the prevalence of out-of-order delivery summarized above as giving any sort of representative numbers for the Internet, but should instead form the rule of thumb: Internet paths are *sometimes* subject to a high incidence of reordering, but the effect is strongly site-dependent, and highly correlated with route fluttering.

The extremes of out-of-order delivery are interesting because they represent situations of network behavior far from normal. Such true pathologies sometimes illuminate unforeseen interactions between transport protocols and the network.

Figure 13.1 shows the single worst trace in our data in terms of out-of-order delivery, from `wustl` to `nrao` in $\mathcal{N}_1$. 74 packets out of 205 sent arrived out-of-order, a proportion of 36% (the worst in $\mathcal{N}_2$ was 28%). The plot includes a line linking adjacent packets to highlight the effect. Every time the line heads downward to the right it indicates an out-of-order delivery. It is interesting to note that while this connection endured major reordering, it did not suffer *any* packet loss, and only one needless retransmission, that due to the Solaris TCP's insufficiently large initial retransmission timeout (RTO), discussed in § 11.5.10. In particular, the timer *was* able to cope with significant fluctuations in round-trip time. This may appear surprising in light of the problems previously uncovered with the Solaris timer adaption algorithm (§ 11.5.1). However, out-of-order packets elicit *duplicate* acks from the network, corresponding to the temporarily missing packets. If the RTO adaptation only uses timings based on acks that advance the window, then it will tend to see timings reflecting the longer of the two routes over which the packets travel. This is, fortunately, exactly the right RTT timing to which one should adapt the RTO, since it represents the worst-case on how long it can take for a packet to traverse the network and be acknowledged.

While we found earlier in this section that data packets are significantly more likely to be

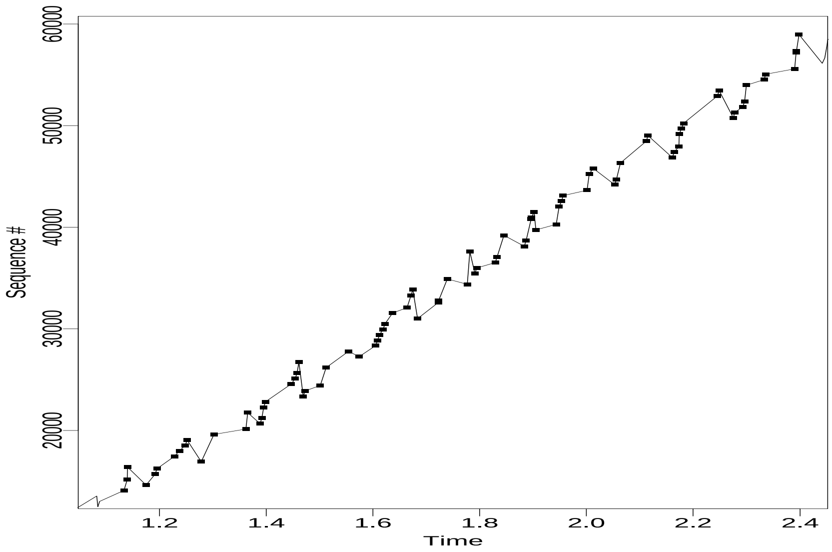

Figure 13.1: Sequence plot showing a connection with 36% of data packets delivered out-of-order

reordered than acks, this does not necessarily apply to the extremes of behavior. Indeed, in $\mathcal{N}_1$ we observed 12 connections in which 20% or more of the acks were reordered, with an extreme value of 33% reordered. (In $\mathcal{N}_2$, the extreme value was 13%.)

Figure 13.2 shows the *largest* out-of-order gap we found. In this $\mathcal{N}_2$ trace from `adv` to `harv`, all the packets shown in the plot were sent in sequence. After data packet 61,953 arrives, the next arrival is 89,601, sent 54 packets later!

While at first blush it might appear that the reordering in Figure 13.2 is due to a routing change at sequencing 89,601, the evidence indicates it is in fact due to a different effect. Figure 13.3 shows a similar massive reordering event. Here, however, the higher-sequence number packets nearly lie on a line. Indeed, fitting a line to them yields a data rate of a little over 170 Kbyte/sec. This rate is a compelling value because it agrees with a T1 bottleneck (§ 14.7.1). Furthermore, it agrees with the remainder of the trace, which is shown in its entirety in Figure 13.4. Indeed, from that figure it is clear both that the slope of the packets delivered *late* in Figure 13.3 is aberrant, and that the late packets were abnormally delayed, rather than the high-sequence packets arriving early due to a routing change. Finally, the slope of the late packets, if we factor in the number of high-sequence packets arriving in their midst, is just under 1 Mbyte/sec, consistent with an Ethernet bottleneck.

We analyze this behavior as follows. A router quite close to the receiver (such that the bottleneck bandwidth between the router and the receiver corresponds to Ethernet speed) stopped forwarding packets just as 72,705 arrived. The most likely explanation for its 110 msec lull is that it had a routing update to process, as these can take considerable time and many routers cease forwarding packets during the update [FJ94]. After the processing finished, which occurred just between the arrival of 91,137 and 91,649, the router began forwarding packets normally again. Thus, the higher-sequence packets, which arrived at the router at T1 speeds since that is the upstream bottleneck, continued through the router unaltered. Meanwhile, the router had queued some 35 packets

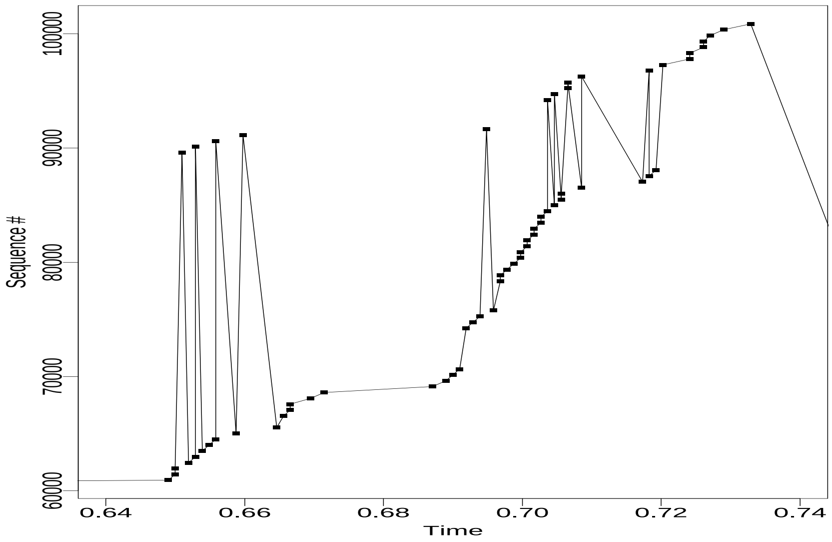

Figure 13.2: Sequence plot showing a connection with an out-of-order gap of 54 packets

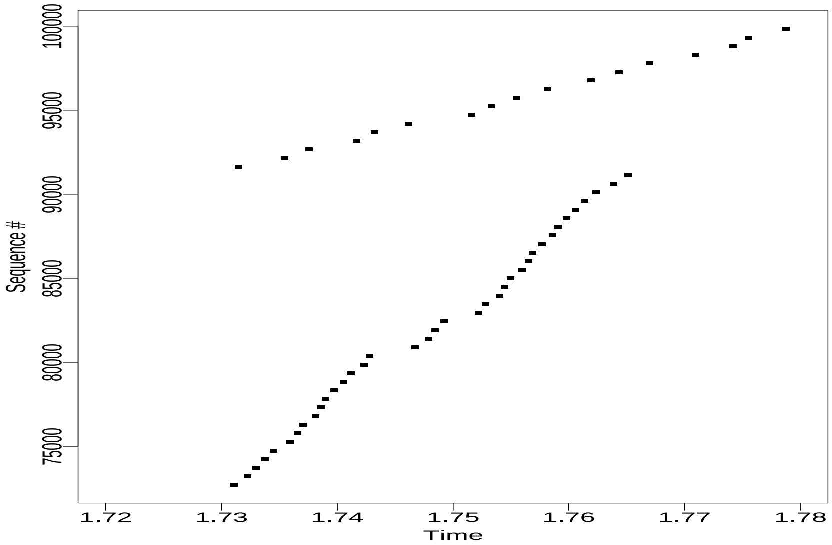

Figure 13.3: Out-of-order delivery with two distinct slopes

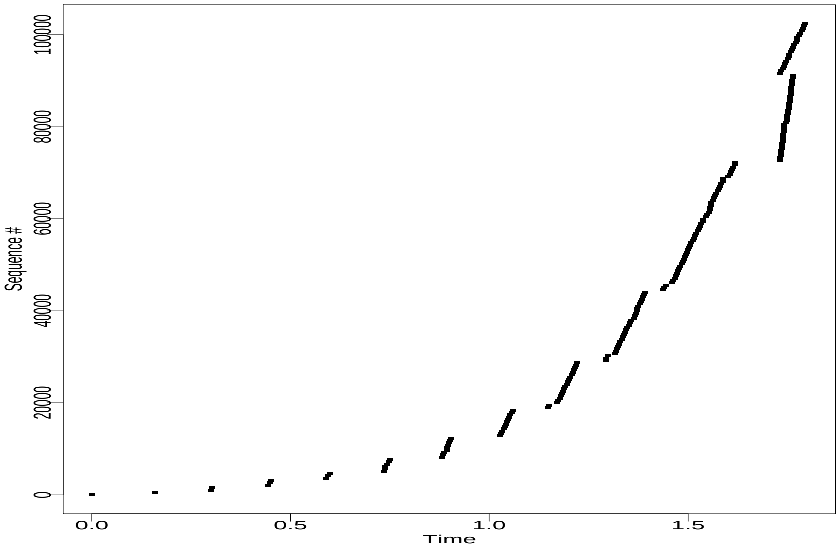

Figure 13.4: Sequence plot of entire connection shown in previous figure

while it processed the update, and these were now finally forwarded whenever the router had time (was not processing a newly arriving packet). Thus, they went out as quickly as possible, namely at Ethernet speed.

We observed this pattern a number of times in our data—not frequent enough to conclude that it is anything but a pathology, but often enough to suggest that significant momentary increases in networking delay can be due to effects different from both queueing and route changes; most likely due to router "pauses."

Striking reordering is not confined to data packets. Figure 13.5 shows a SYN-ack packet, still advertising a (relatively) small initial window (shown in the plot by the circle above the ack), arriving a full second after it was sent, after 19 subsequent acks have already arrived. Even more striking is the trace shown in Figure 13.6. Here, two acks, the first for 47,617 and the second for 48,129, arrive a full *twelve* seconds after they were sent (and long after the packets they acknowledged were needlessly retransmitted). Just where in the network they spent those 12 seconds, and what led to their eventual release, remains a mystery! One clue, however, is that they arrived with a remaining TTL of 40, while all the other acks had TTL's of 41 remaining. They may have taken a different route through the network. This is not certain, however, because the router that detained them may instead have additionally decremented the TTL field to reflect the long delay (§ 4.2.1).

13.1.3 Impact of reordering

While out-of-order delivery can violate one's assumptions about how the network works—in particular, the abstraction that the network is well-modeled as a series of FIFO queueing servers—it often is no more than a nuisance in terms of its impact on transport protocols such as TCP. For example, Figure 13.1 above shows the trace that endured the largest proportion of out-of-order packet delivery of the more than 20,000 traces we studied; yet it did not suffer any

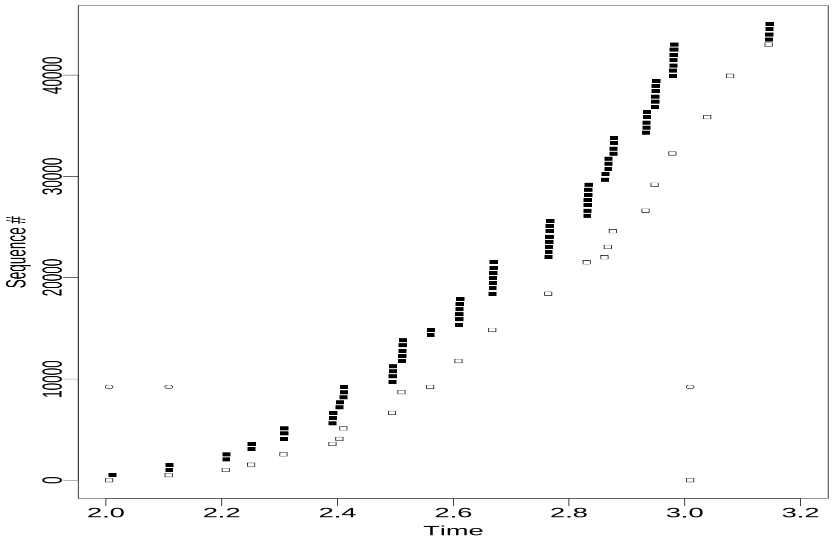

Figure 13.5: Sequence plot of ack delivered out-of-order

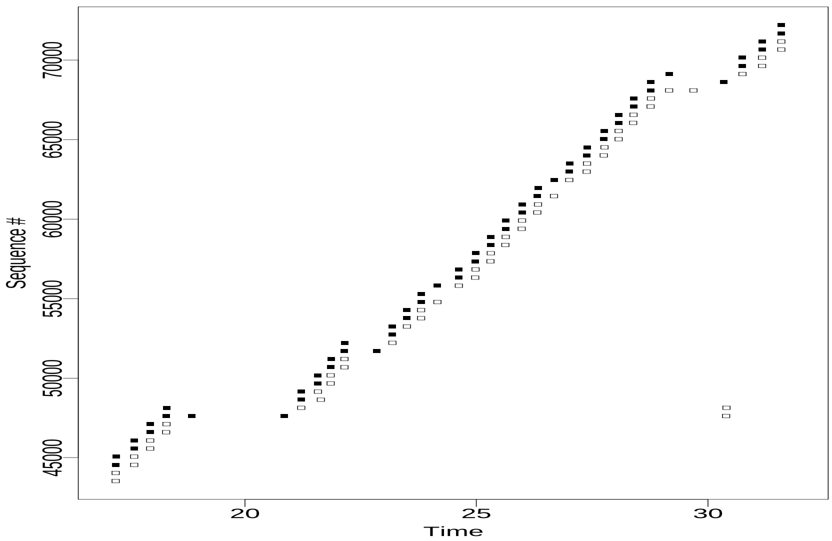

Figure 13.6: Sequence plot of two acks delivered out-of-order and very late

retransmissions, and in fact had its performance limited by the small advertised receiver window, rather than by any effects from the reordering.

Where reordering makes a difference, however, is when one wishes to make a quick decision whether or not to retransmit an unacknowledged packet.[1] In particular, if the network never exhibited reordering, then, as soon as the receiver observed the arrival of a packet that created a sequence "hole," it would know that the expected in-sequence packet had been dropped, and could signal this information to the sender to call for prompt retransmission. Because of reordering, however, the receiver does *not* know whether the packet in fact was dropped; it may instead have simply been reordered and will arrive shortly. In this latter case, the receiver should *not* call for retransmission, as retransmission is unnecessary and will thus needlessly consume network resources.

TCP addresses this problem as follows. When a TCP receives a packet above a sequence hole, it may generate a dup ack for the sequence hole. (Indeed, all TCPs in our study except SunOS generate such acks; see § 11.6.2.) If a TCP receives a certain threshold number N_d of dup acks, it then can enter a *fast retransmit* phase (§ 9.2.7). Presently, $N_d = 3$, a value chosen so that "false" dup acks generated by out-of-order delivery are unlikely to lead to spurious retransmissions.

The value of $N_d = 3$ was chosen primarily to assure that the threshold was conservative and needless retransmission avoided. Large-scale measurement studies were not available to further guide the selection of the threshold. In this section we examine whether the fast retransmit mechanism could be improved in two different ways: by delaying the generation of dup acks in order to better disambiguate packet loss from out-of-order delivery, and by choosing a different threshold value to improve the balance between increasing opportunities to retransmit quickly, and avoiding unneeded retransmissions due to out-of-order delivery.

We first look at packet reordering time scales to determine whether a TCP could profitably wait a short period of time upon receiving a packet above a sequence hole before generating a dup ack. We only look at the time scales of data packet reorderings, since ack reordering time scales do not affect the fast retransmission process. Indeed, since TCP acks are cumulative, out-of-order delivery of acks has essentially no effect on the performance of a TCP connection.

Figure 13.7 shows the distribution of the amount of time between an out-of-order data packet arrival and the later arrival of the last packet sent before it. The plot is log-scaled and thus reflects a wide range in reordering times. The distribution exhibits several artifacts meriting investigation. For example, the central step in the distribution occurring around 50% probability lies at exactly 10 msec, and corresponds to a common clock resolution (§ 12.4.2). Likewise, the smaller step a bit to the right of it is at 20 msec, another common resolution.

The skip at the upper right of the plot is more interesting, as it is not a measurement artifact per se. It lies right at 81 msec, which initially seems a strange value. However, one of the sites in our study was linked to the Internet during $\mathcal{N}_1$ via a 56 Kbit/sec link (`connix`). Using the methodology developed in Chapter 14, we found this site's bottleneck bandwidth was right around 6,320 user data bytes/sec. If a remote site is sending 512 byte packets, and if they are reordered upstream from the 56 Kbit/sec bottleneck link, then the packets can arrive *no closer* than:

$$\frac{512 \text{ bytes}}{6,320 \text{ bytes/sec}} = 81.0 \text{ msec.}$$

[1] It can also make a significant difference for a TCP receiver that does not retain above-sequence data, as we saw for Trumpet/Winsock in § 11.7.3. Such a TCP will force retransmission of every packet delivered out of order.

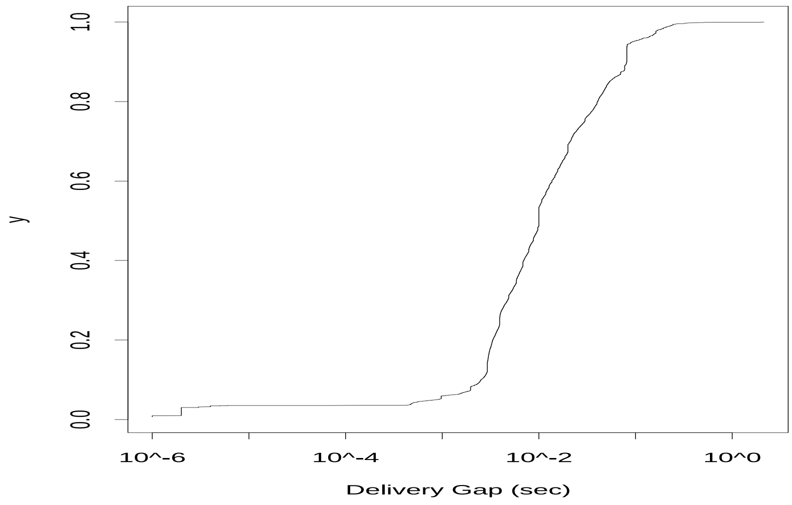

Figure 13.7: Distribution of out-of-order delivery interval for $\mathcal{N}_1$ data packets

Thus, we see that reordering can have associated with it a *minimum* time which can be quite large. This effect, however, will diminish with time as faster links replace slower ones.

Figure 13.8 shows the same distribution for $\mathcal{N}_2$ (solid), with $\mathcal{N}_1$ added (dotted) for comparison. It likewise exhibits timer resolution steps and the 56 Kbit/sec minimum reordering time, as well as a slightly smaller minimum time of 70 msec, corresponding to 64 Kbit/sec links delivering about 7,300 bytes/sec. The most noteworthy aspect of the plot, however, is the strong shift towards lower values. The median of the $\mathcal{N}_1$ intervals was 10 msec, and the geometric mean 9 msec, while for $\mathcal{N}_2$ these dropped by more than a factor of two, both to around 4 msec. We suspect the change is due to the deployment of faster links within the Internet infrastructure.[2] If so, then again we expect reordering times to diminish as the infrastructure is further upgraded.

Even with the $\mathcal{N}_1$ intervals, a strategy of waiting 20 msec would identify 70% of the out-of-order deliveries. For the $\mathcal{N}_2$ intervals, the same proportion can be achieved waiting 8 msec.

However, a more basic question is: are false fast retransmit signals due to out-of-order deliveries actually a problem? To find an answer, we added to `tcpanaly` analysis of duplicate acks[3] as follows. For each trace pair it analyzes, it inspects each series of duplicate acks arriving at the sending TCP and classifies the sequence as one of:

good: indeed due to a missing packet requiring retransmission;

[2]It is not due to better clock resolutions in $\mathcal{N}_2$ compared to those in $\mathcal{N}_1$. If we eliminate the 9–11 msec and 19–21 msec spikes in the distributions shown in Figure 13.7 and Figure 13.8, we still find a virtually identical shift between the two datasets.

[3]`tcpanaly` only considers an ack as a duplicate of the preceding ack if it *(i)* acknowledges the same sequence number; *(ii)* contains the same offered window; and *(iii)* is a “pure” ack packet, one not containing any data. This test can still mistake a series of acknowledgements for “zero window” probes as triggering a fast retransmit. However, such probes were exceedingly rare in our traces: only 6 instances in $\mathcal{N}_1$, and none in $\mathcal{N}_2$. Of the 6 in $\mathcal{N}_1$, only one persisted long enough to elicit more than a single ack in reply (it elicited two such acks).

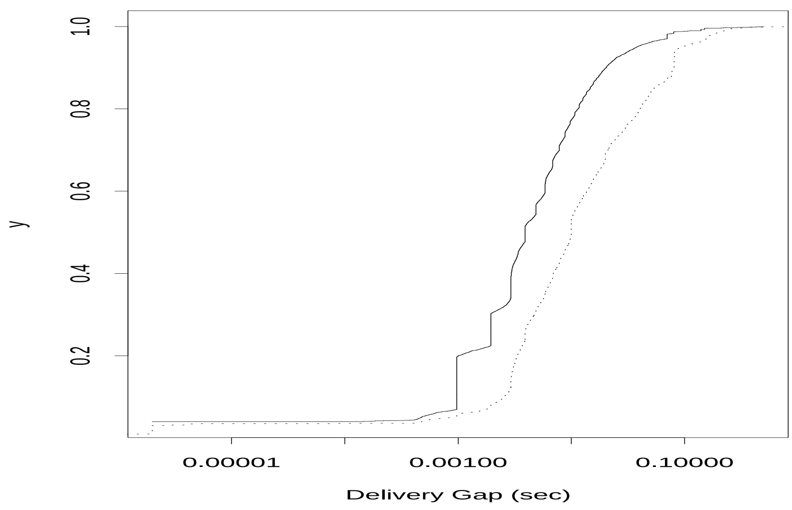

Figure 13.8: Distribution of data packet out-of-order delivery interval for $\mathcal{N}_1$ (dotted) and $\mathcal{N}_2$ (solid)

bad: actually due to a temporary sequence hole caused by out-of-order delivery; or,

top: corresponding to the top sequence number sent so far.

The terms **good** and **bad** reflect the perspective of using the series of duplicate acks as a signal for fast retransmission. **top** series reflect situations in which the TCP has already needlessly retransmitted. When a needless retransmission arrives at the receiver, because it is below-sequence it will immediately trigger the generation of a duplicate ack (§ 9.2.7). **top** series can lead to further needless retransmission (thus perpetuating the cycle), but the TCP can employ a simple heuristic to avoid these, discussed below.

In addition to classifying each duplicate ack series, `tcpanaly` assigns a length D corresponding to the number of duplicate acks in the series. For **good** duplicate ack series, `tcpanaly` also associates a *savings* S indicating how much time would have been saved if the fast retransmit threshold N_d had been equal to D, and thus the series had led to retransmission. For $D > 3$, S is often negative, because in fact the packet was already transmitted upon receipt of the third duplicate, rather than waiting for all D packets.

For **bad** duplicate ack series, `tcpanaly` associates a *waiting time* W, indicating how long the TCP would have had to wait in order to recognize that the sequence hole was due to out-of-order delivery rather than to packet loss.

When considering a refinement to the fast retransmission mechanism, our interest lies in the resulting ratio of **good** to **bad**, $R_{g:b}$, which is controlled by both N_d and $\widetilde{W}$, the minimum amount of time that the receiving TCP would wait prior to generating a duplicate; and the mean ensuing savings $\overline{S}$ of how much more quickly the TCP can retransmit as a result of the refinement.

We first consider the current state of affairs, in which $N_d = 3$ duplicates and $\widetilde{W} = 0$, namely duplicate acks are generated immediately as called for. In $\mathcal{N}_1$ we find $R_{g:b} = 22$, and in $\mathcal{N}_2$ $R_{g:b} = 300$! (That is, in $\mathcal{N}_1$, each incorrect fast retransmit was countered, overall, by 22 correct fast retransmits, and, in $\mathcal{N}_2$, by 300 correct retransmits.) The order of magnitude improvement between $\mathcal{N}_1$ and $\mathcal{N}_2$ is likely mostly due to the use in $\mathcal{N}_2$ of bigger windows (§ 9.3), and hence much more opportunity for **good** duplicate ack series. (We do not evaluate the savings S of the current mechanism, because it is what we are measuring against.)

Because the current scheme works well, we do not investigate increasing the threshold in detail. We note, however, that $N_d = 4$ improves $R_{g:b}$ by about a factor of 2.5, but diminishes the number of fast retransmit opportunities by about 30%, a significant loss.

We might instead consider whether the threshold can be safely lowered from 3 to 2. For $N_d = 2$, we gain about 65–70% more fast retransmit opportunities (i.e., **good** dup ack sequences), a hefty improvement. Furthermore, the mean savings $\overline{S}$ for these new opportunities is 1.65–1.73 sec, because we are avoiding retransmission timeouts. The cost, however, is that $R_{g:b}$ falls by about a factor of three, in both $\mathcal{N}_1$ and $\mathcal{N}_2$.

If, however, the receiving TCP waited $\widetilde{W} = 20$ msec before generating a second dup (avoiding doing so if the missing packet arrived, and immediately doing so if another out-of-order arrival called for a third dup), then, for $\mathcal{N}_1$, $R_{g:b}$ only falls from 22 to 15, while for $\mathcal{N}_2$ it does not fall at all.

Thus, the simplest change of just lowering N_d from 3 to 2 gains a large proportion of quicker retransmissions, but at the cost of three times as many unnecessary retransmissions. A companion change to TCPs to delay for $\widetilde{W} = 20$ msec when sending a second duplicate ack ameliorates almost all of the drawbacks of lowering N_d to 2. However, there are considerable deployment differences between these two modifications. The first is a one-line change in most TCP implementations and garners benefits (and drawbacks) even if only the *sending* TCP has been modified and it is communicating with an unmodified receiving TCP. The receiving TCP change involves additional timer management and so is not necessarily a simple change, and it only garners benefit if *both* the sending and receiving TCP have been modified (it does not do much harm if the sender has not, however). But lowering the retransmit threshold to two duplicate acks is only a sound change *if* deployed simultaneously with the $\widetilde{W} = 20$ msec change. Such widespread simultaneous deployment, however, is virtually infeasible due to the size of the Internet. Therefore, we would have to live with partial deployment for a lengthy period of time, and, for that time, significantly more unneeded retransmissions. In summary, if we require changing both the sender and the receiver, then, while the change is appealing, it is likely impractical considering the size of the Internet's installed base of TCP implementations.

Another approach would be to modify *senders* to wait 20 msec before responding to $N_d = 2$ duplicate acks with a fast retransmission. This pause would then generally allow, in the case of out-of-order delivery, sufficient time for another ack to arrive indicating that the temporarily missing data packet was successfully delivered. We do not evaluate this approach in detail here, but note that it has several drawbacks. First, it requires additional timer management, which, as mentioned above, is not always a simple change. Second, delay variations along the return path taken by the acks might require a significantly larger value of $\widetilde{W}$ to avoid unnecessary retransmissions. Third, if the ack return path suffers from loss, then the "clarifying" ack that identifies the first two dups as due to a reordering event might be lost, again leading to unnecessary retransmissions.[4]

[4] We show in § 15.2 that losses along the forward and reverse directions of an Internet path are, overall, nearly uncorrelated, so we could quite plausibly have a situation in which "clarifying" acks are dropped, but there is no loss along the forward path, and hence no retransmission necessary.

We note that the TCP *selective acknowledgement* ("SACK") option, now pending standardization, also holds promise for honing TCP retransmission [MMFR96]. SACK provides sufficiently fine-grained acknowledgement information that the sending TCP can generally tell which packets require retransmission and which have safely arrived (§ 15.6). To gain any benefits from SACK, however, requires that both the sender and the receiver support the option, so the deployment problems are similar to those discussed above. Furthermore, use of SACK aids a TCP in determining *what* to retransmit, but not *when* to retransmit. Because these considerations are orthogonal, investigating the effects of lowering N_d to 2 merits investigation, even in face of impending deployment of SACK.

Perhaps needless to say, lowering N_d all the way to a single dup ack is a disaster. $R_{g:b}$ falls by a factor of 10 from its value for $N_d = 3$. For $\mathcal{N}_2$, using a 20 msec delay before generating a dup ack wins back most of the loss (changing the factor to 1.5), but for $\mathcal{N}_1$, it still falls by a factor of 3.

The final category of duplicate ack series analyzed by `tcpanaly` is **top**. These are quite common, due primarily to broken retransmission timers (§ 11.5.10), but also due to imperfect recovery during retransmission. A **top** series occurs when the original ack (of which all the others are then duplicates) had acknowledged *all* of the outstanding data (hence, the **top** of the sequence space). When this occurs, subsequent duplicates for that ack are *always* due to an unnecessary retransmission arriving at the receiving TCP, until the sending TCP sends new data. Even when it does, subsequent duplicates are still due to redundant packets until at least a round-trip time has elapsed after sending the new data.

Figure 13.9 shows a retransmission event leading to a **top** series. The sender has opened a large window of about 50 packets when data packet 45,025 is lost, as are the 17 packets following it. A river of dup acks pours in as 54,673 and above successfully arrive. The third dup triggers fast retransmit, but since nearly half the window was lost, the many dup acks are not enough to induce fast recovery, so no more packets are in flight, and hence no more dups arrive signaling that 45,561 was also lost. Thus, 45,561 times out, and a slow-start sequence begins at $T = 2.46$.

The first four flights of this sequence all work to fill the large sequence hole due to the 18 dropped packets, but the fifth flight, considerably larger than the fourth, transmits almost entirely redundant data already safely received at the other end. The arrival of these unnecessary packets then causes another sequence of duplicate acks. Figure 13.10 shows the resulting **top** series. The first ack for 67,001 is not a dup but instead indicates that the sequence hole has been filled. *It also advances the window*, so 13 new packets are sent, beginning with 67,537. Shortly after, the first of the dups arrive, and, after three, 67,537 is sent *again* due to fast retransmission, and more packets are sent on the additional dups due to fast recovery. Since fast recovery is enabled, however, no more spurious retransmissions result, ending the cycle, and the connection proceeds normally once fast recovery terminates about time $T = 2.85$.

Top series are about 10 times rarer than **good** series, but that still makes them the cause of between 2 and 15 times as many unnecessary retransmissions than **bad** series due to out-of-order delivery. They are, however, preventable, using the following heuristic. Whenever a TCP receives an ack, it notes whether the ack covers all of the data sent so far. If so, it then ignores any duplicates it receives for the ack, otherwise it acts on them in accordance with the usual fast retransmission mechanism.

The only drawback to this method is if the TCP sends a flight of new data after receiving

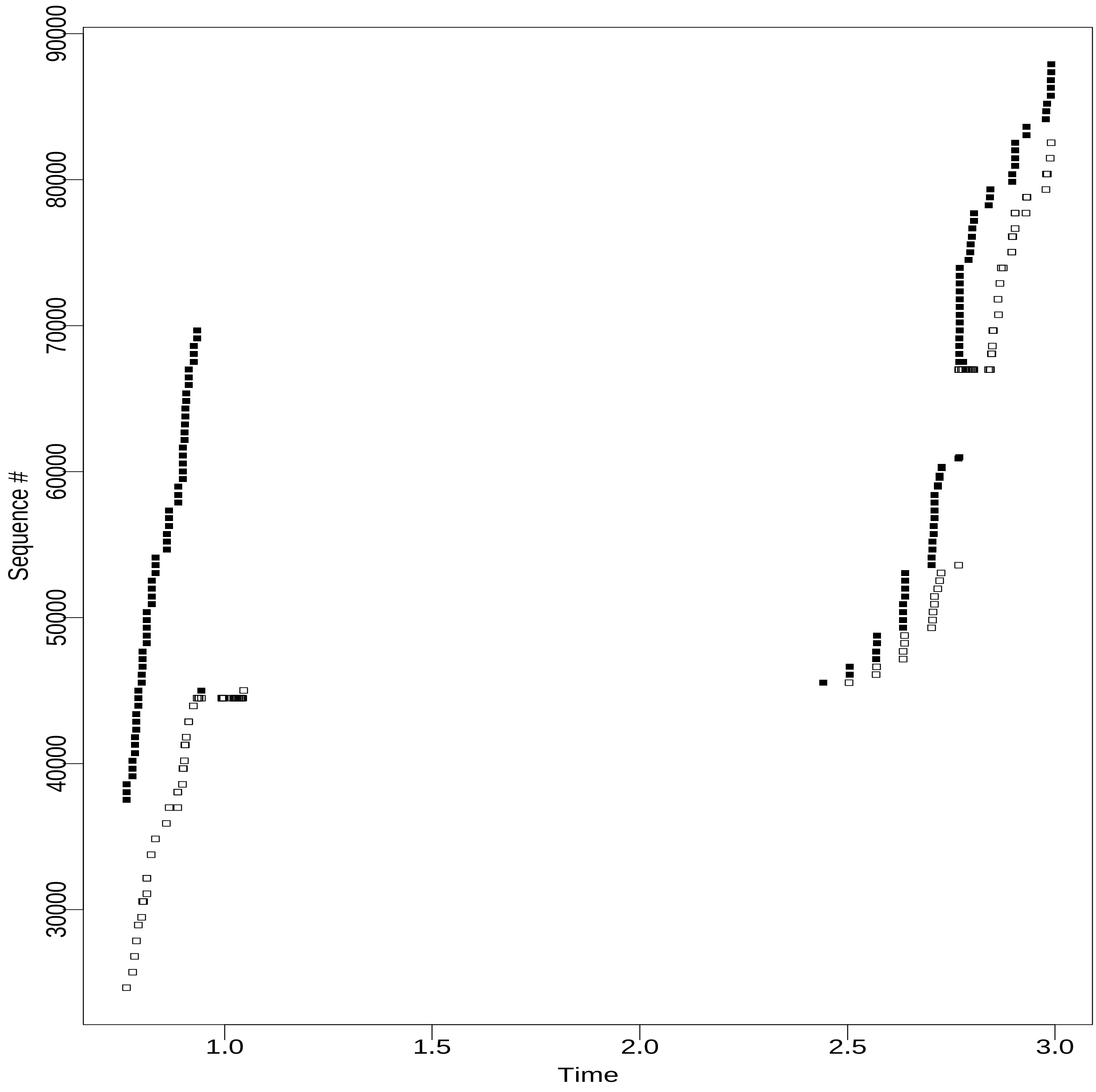

Figure 13.9: Sequence plot showing retransmission event leading to **top** duplicate ack series

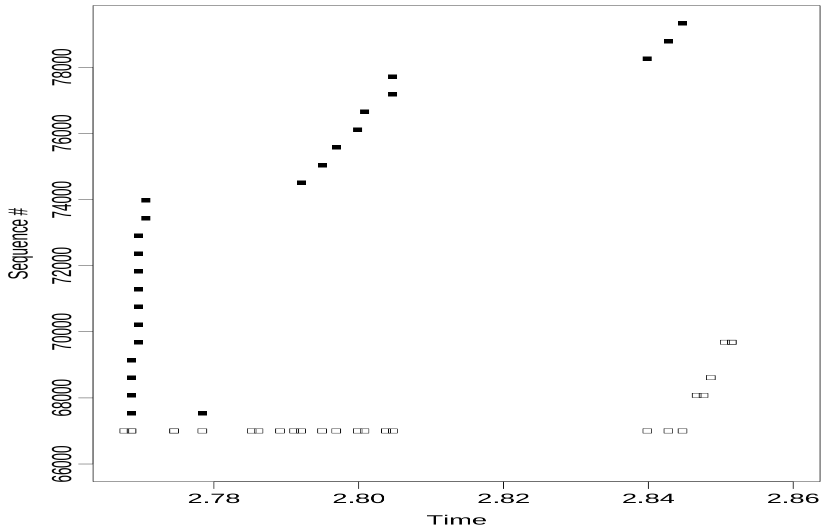

Figure 13.10: Enlargement of **top** duplicate ack series

the first top ack, and the first packet of the flight is lost, then the subsequent dups generated by the arrival of the remainder of the flight will fail to trigger fast retransmission for the missing packet, and so the connection will stall pending a timeout retransmission. This deficiency can be addressed by allowing the TCP to honor dup acks if they arrive at least one round-trip time (RTT) after the TCP sent new data. This requires, however, that the TCP maintain an estimate of the minimal RTT, which most present implementations do not. (The retransmission timeout is based on an estimate of the *maximum* RTT.) Use of SACK will also eliminate **top** dup ack series, since SACK allows the sender to disambiguate between dups due to needless retransmission and dups due to a genuine missing packet. But the heuristic we propose has the attractive benefit of not requiring that both the sender and receiver implement it. It works fine if just the sender uses it.

13.2 Packet replication

In this section we look at *packet replication*, meaning instances in which the network delivers multiple copies of the same single packet. While with out-of-order delivery we can readily picture a causal mechanism, namely uneven path delays, it is difficult to see how the network can replicate a packet given to it. Our imaginations notwithstanding, it does occur, albeit very rarely. We suspect the mechanism may involve links whose link-level technology includes a notion of retransmission, and for which the sender of a packet on the link incorrectly believes the packet was not successfully received, so it sends the packet again. A related mechanism, pointed out by Van Jacobson, would occur on a token ring network if the sender's network interface sometimes failed to promptly drain the packet from the ring, such that it made multiple circuits.

In $\mathcal{N}_1$, we observed only one instance of packet replication. Figure 13.11 shows the corresponding sequence plot, recorded at the data sender. Two acks, one for 43,009 and one for 44,033, arrive at $T = 1.86$. They then arrive again, and again, and again, for a total of 9 pairs of

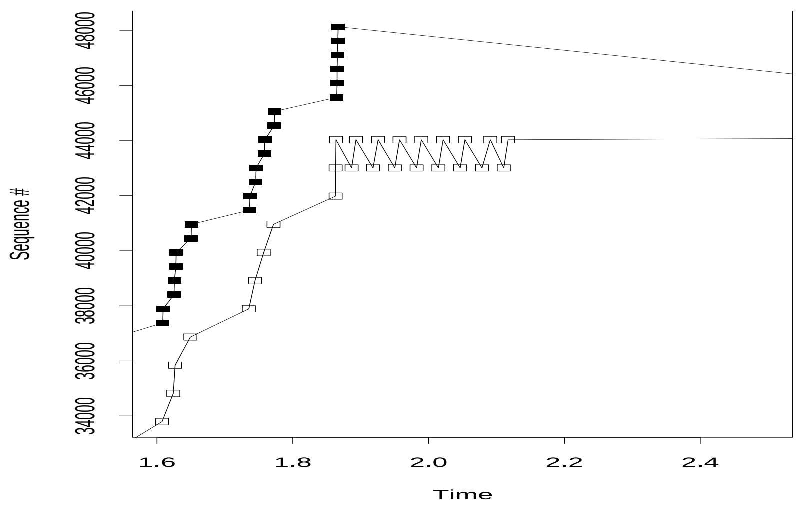

Figure 13.11: Two acks replicated 8 times each

arrivals, each pair coming 32 msec after the last. Since the replication involves *two* different acks, the multiple arrivals do not constitute a duplicate ack series, and so no fast retransmission occurs (§ 9.2.7). The fact that two packets were together replicated does not fit with the explanations offered above for how a single packet could be replicated, since link-layer effects would only replicate one packet at a time. Finally, the replication in Figure 13.11 was accompanied by a routing change along the path from the data sender to the receiver. It seems likely the two events were somehow related.

In $\mathcal{N}_2$, however, we observed 65 instances of the network infrastructure replicating a packet. Figure 13.12 shows the most striking of these, a single data packet 78,337 being replicated 22 times by the network (two extended blurs in the plot). The receiving TCP dutifully generates dup acks for each additional arrival, though it experiences a processing lull of about 7 msec while doing so.

All of the packet replications in $\mathcal{N}_2$ were of a single packet, indicating perhaps a different mechanism than that for $\mathcal{N}_1$'s lone replication event. Several sites dominated the $\mathcal{N}_2$ replication events: in particular, the two Trondheim sites, `sintef1` and `sintef2`, accounted for half of the events (almost all of these involving `sintef1`). Of the remainder, the two British sites, `ucl` and `ukc`, accounted for nearly half again. But after eliminating all of these, we still observed replication events among connections between 7 different sites, so the effect is not completely isolated to one or two locations.

Surprisingly, packets can also be replicated at the sender. Figure 13.13 shows an example. Here, the ack arriving in the lower left corner of the plot has liberated 19 new packets (the receiver is a Solaris system and the ack reflects the Solaris slow-start acking strategy discussed in § 11.6.1). The packets are sent at nearly Ethernet speed, but, 4 msec after it was first sent, packet 91,649 shows up again. The second occurrence is a replication and not a temporary routing loop, because

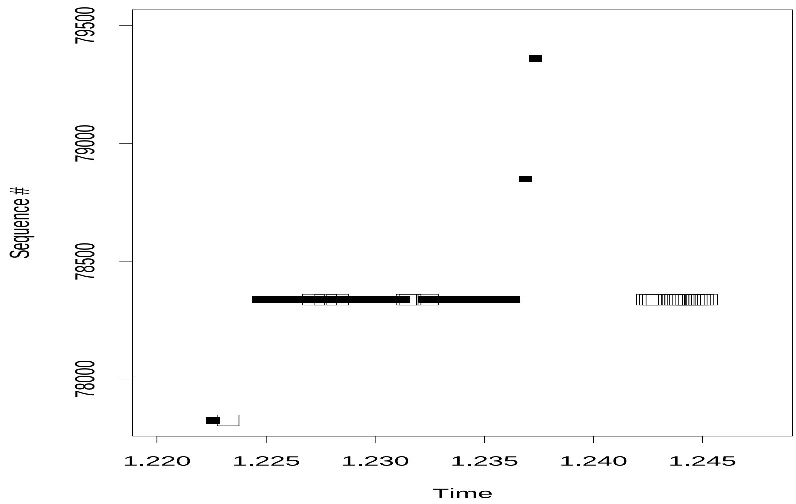

Figure 13.12: Data packet replicated 22 times

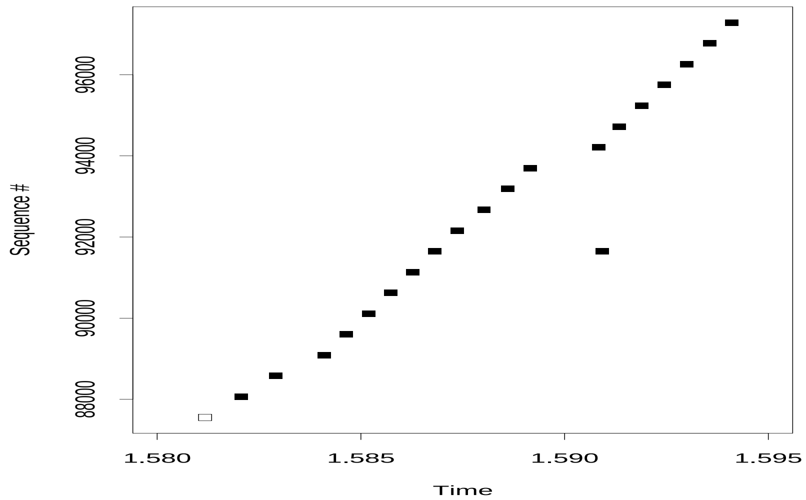

Figure 13.13: Data packet replicated at sender

both copies show up at the receiver.[5] Furthermore, the second copy had a TTL field one less than that in the first copy, indicating that the replicant did indeed take a slight detour before showing up again on the local link. While there were no sender-replicated packets in $\mathcal{N}_1$, $\mathcal{N}_2$ had 17 instances, 12 involving `sintef1` and the remainder involving `connix`. For both sites, the replicated packet was always out-bound, sometimes an ack and sometimes a data packet.

13.3 Packet corruption

The final pathology we look at is *packet corruption*, in which the network delivers to the receiver an imperfect copy of the original packet. Packet corruption is a well-known problem and a great deal of effort has been devoted to coding schemes and checksums in order to detect and correct for transmission errors. For TCP/IP, the IP header includes a 16 bit *header* checksum that is computed over the IP header bytes. It does *not* include the TCP header or the TCP data bytes. It is supposed to be checked at each forwarding hop (though it is not clear whether all high-speed routers do so). If the checksum fails to match the header, the packet is discarded, because it cannot be reliably forwarded (who knows what is the true destination address?).

TCP packets are further protected by a 16 bit checksum for the entire data contents of the packet, as well as the TCP header and part of the IP header. This checksum is intended as an *end-to-end* checksum, the merits of which are persuasively argued in [SRC84].

We discussed `tcpanaly`'s checksum analysis in § 11.2 and § 11.4.2. One issue we mentioned was the fact that what `tcpanaly` is actually detecting are packets ignored by the TCP receiver, which we then presume are due to checksum failures. An important point is that packets can be ignored due to other effects, such as the kernel having exhausted its available buffer space for keeping the packet until the TCP receiver can process it, or the network card dropping the incoming packet for the same sort of reason. In particular, the vantage-point problem (§ 10.4) can render the distinction between a checksum failure and other problems difficult to make.

We address this difficulty by observing that packet filters running on the same host as the TCP receivers should only see packets also seen by the receiver: if the network interface or kernel lacked resources for delivering the packet to the TCP, then the filter should not have received a copy, either.[6] Packet filters running on separate hosts, on the other hand, *will* see both kinds of receiver losses, those due to checksum failures and those due to other causes. Thus, if a significant portion of `tcpanaly`'s inferred checksum errors are actually packets discarded for a different reason, then we should find the sites with separate packet filter hosts more likely to detect purported checksum errors than those with the packet filter running on the same host as the TCP.

We do not, however, find much of a disparity: in $\mathcal{N}_2$, after eliminating `lbli` (see below), we find that 3.3% of the traces recorded by separate-host packet filters included a purported checksum error, while about 3.0% of those recorded by same-host filters did. Accordingly, we argue that the vast majority of checksum errors inferred by `tcpanaly` are indeed due to packet corruption.

We now present analysis based on this assumption. In $\mathcal{N}_1$, `tcpanaly` flagged 75 traces (2.9%) as exhibiting a total of 105 checksum errors, with an overall proportion of 0.02% of the

[5] We verified that both copies include the same value in the IP "id" field.

[6] It might be possible that, on some systems, the kernel may find it has sufficient resources to give a copy of a packet to the packet filter, but not a separate copy to the TCP receiver. We would expect, though, that this sort of borderline case would manifest itself only rarely.

received packets corrupted by checksum errors. In $\mathcal{N}_2$, however, the figures climbed to 748 traces (4.4%) exhibiting 1,982 checksum errors, for an overall proportion of 0.06% of the received packets.

The apparent trend, however, is not significant. It is all due to an increase in the checksum errors seen for data packets received by `lbli`. In $\mathcal{N}_1$, only 4% of the traces with data checksum errors were to `lbli` as the receiver. In $\mathcal{N}_2$, however, 33% were. Furthermore, `lbli` in $\mathcal{N}_2$ was particularly prone to checksum bursts like those shown in Figure 11.3. If we eliminate from our analysis those $\mathcal{N}_2$ traces with `lbli` as the receiver, then the proportion of traces with errors falls to 3.0% and the proportion of received packets falls to 0.02%, essentially the same as in $\mathcal{N}_1$. After doing so, no particular site stands out as being exceptionally plagued by checksum errors. Thus, the evidence is good that, as a rule of thumb, the proportion of Internet data packets corrupted in transit is around 1 in 5,000.[7]

A corruption rate of 1 packet in 5,000 is low but certainly not negligible, because TCP protects its data with a mere 16-bit checksum. Consequently, on average one bad packet out of 65,536 will be erroneously accepted by the receiving TCP, resulting in *undetected data corruption*. If the rates in our study are typical, which seems plausible (but see below), then about one in every 300 million Internet packets is accepted with corruption. As the Internet carries far more data than 300 million packets per day,[8] it appears likely that bad data is being accepted by a number of TCPs around the Internet every day.[9] Thus, these statistics argue that TCP's 16-bit checksum is no longer adequate, if the goal is that globally in the Internet there are very few corrupted packets accepted by TCP implementations.

We noted above that `lbli` showed a strong increase in the prevalence of corrupted data packets received between $\mathcal{N}_1$ and $\mathcal{N}_2$. Since `lbli`'s Internet link is via an ISDN line, it appears quite likely that the change is due to an increase in noise on the ISDN channels. That the errors most likely occur on an ISDN link also suggests why we observe bursts of checksum errors. The link in question uses SLIP compression (CSLIP) in order to transmit the TCP/IP header information very succinctly over the link [Jac90]. CSLIP works by encoding the header as differences with respect to the header of the connection's previous packet. Thus, if the link suffers an undetected error, not only will the current packet be corrupted, but so will every subsequent packet whose header is expressed in terms of differences with respect to the current packet's corrupted header. CSLIP consequently produces a stream of corrupted packets until the compression is reset (which happens when the originally-corrupted packet is retransmitted). This is exactly the behavior seen in Figure 11.3—the errors stop as soon as the first corrupted packet is retransmitted. (We frequently see this pattern with checksum bursts.) This means that, at the *physical* layer, probably only one error occurs, but the use of compression magnifies this error and turns it into a burst. From a networking perspective, this is quite unfortunate, as it results in a spate of what should have been unneeded retransmissions. The correct fix for this problem is probably to ensure that the link layer uses a strong checksum, so it can discard corrupted packets without even presenting them to CSLIP for decompression; and to

[7] If we assume single-bit uniformly-distributed errors, along with 512 byte data packets having 40 bytes of TCP/IP header, then this corruption rate corresponds to a Bit Error Rate of about $4.5 \cdot 10^{-8}$.

[8] A 37 minute trace of the busy Internet exchange point FIX-WEST captured on June 21, 1995, logged slightly under 1,000,000 packets per minute [`http://www.nlanr.net/Flowsresearch/fixstats.21.6.html`].

[9] This analysis assumes that corruptions result in uniformly-distributed checksum alterations. See [PHS95] for a more detailed analysis of data corruption checksum patterns, which can make the failure rate for accepting bad data significantly higher. In general, our data does not enable us to check for these other patterns, since our traces do not include packet contents.

ensure that CSLIP can resynchronize its compression state in the presence of such discards.

Finally, we note that the data checksum error rate of 0.02% of the packets is much higher than that found for pure acks (§ 11.2). For pure acks, we found only 1 corruption out of 313,730 acks in $\mathcal{N}_1$, and 26 out of 1,839,824 acks in $\mathcal{N}_2$. Of the 26 in $\mathcal{N}_2$, however, 25 were received by `lbli`, which we removed from our analysis above since it showed a clear prevalence of checksum errors far exceeding any other site. We thus need to reconcile an error rate of $2 \cdot 10^{-4}$ for data packets versus one of between $3 \cdot 10^{-6}$ ($\mathcal{N}_1$) and $6 \cdot 10^{-7}$ ($\mathcal{N}_2$) for pure acks, a ratio of between 60:1 and 300:1.

A first question to address is whether part of the difference is due to a tendency for data packet corruptions to come in bursts, as discussed above. However, other than `lbli`, this is not the case—for other sites, corruption events were usually confined to isolated packets.

If we assume that corruption is due to uniformly distributed single bit errors, then a packet's likelihood of corruption will be directly proportional to the packet's size. Since pure acks have 40 bytes of TCP/IP header while data packets in our study were usually about 14 times larger (though sometimes as much as 37 times), the difference in size alone does not appear to reconcile the discrepancy.

Note, however, that the IP header has its own checksum, which is supposedly verified at each hop taken by a packet. We add the caveat "supposedly" because it is not clear whether all high-speed routers verify checksums, a potentially costly packet-forwarding step as it requires inspecting the entire IP header, which might otherwise be avoidable.

Thus, if a packet is corrupted on a link so that its IP header is altered, then the router receiving the packet is supposed to discard it. Furthermore, if either of the 16-bit port fields in the TCP header are corrupted, then the packet filter used in our study would have rejected the packet, so we would not have had an opportunity to observe the checksum error. The net effect is that, from the perspective of the number of corruptible-yet-observable bits, pure acks have a size of only 16 bytes. (The number of corruptible-yet-observable bits in data packets likewise diminishes, but by a much smaller fraction.) This effect, plus the factor of 14 difference in size, reduces the weighted error rate ratio to between about 2:1 and 10:1.

In addition, if a compression technique such as that in CSLIP is used, then pure acks as transmitted on a link can take much less than 40 bytes (as little as 5 bytes using CSLIP), while data packets take only slightly less than their full TCP/IP size. The size difference can therefore expand from 14:1 to 100:1 or even larger. However, it is not clear whether CSLIP is used on any but quite slow links, since for faster links, the performance cost of compressing and decompressing the packet headers might outweigh the gains due to the reduced transmission times.

Another possibility is that errors are *not* uniformly distributed across the bits in a packet. We could imagine a scenario, for example, in which each time a new packet is sent, the beginning of the transmission of a packet on a link serves to synchronize the sender and receiver on the link. It could then be that for longer packets there is more opportunity for the sender and receiver to drift out of synchronization, adding noise to the signals used to communicate the bits. Investigating this possibility, however, is beyond the scope of our study—doing so would require capturing entire packets in order to assess the distribution of errors within them.

In summary, we can make a somewhat plausible, but not compelling, argument that we can reconcile the discrepancies in checksum failure rates. If we accept the argument, then the compression effect's large role in reconciling the two error rate estimates suggests that errors tend to occur most often on point-to-point links, since those are the ones for which compression is widely used; and furthermore, most likely on slow point-to-point links, as those are the ones for which it is particularly appealing to use compression. Such links might also plausibly be relatively more prone to link errors, since the underlying technology will be pushed hard to try to squeeze out as much bandwidth as possible.

Finally, we note that packet corruption combined with CSLIP can produce surprising errors. Because CSLIP highly compresses the representation of the IP and TCP headers, but does not utilize an additional checksum to protect the compact representation, a bit error can result in packets that appear in many respects perfectly reasonable, albeit different than what was originally sent! We refer to these as "desynchronization errors," since one of the elements leading to them is that the CSLIP sender and receiver have lost agreement upon their common state.

One benign form of desynchronization error exhibits itself as a change in the IP "id" field (§ 10.3.5). This has virtually no effect upon the packet's integrity as far as TCP is concerned, though it can introduce ambiguities when attempting to match up packets in pairs of traces (§ 10.5).

A considerably nastier form of desynchronization error occurs when a packet alters in a plausible fashion. If undetected by the checksum, these packets will often match what the TCP receiving them expects, leading to a fundamental mismatch between the connection state at the two TCP endpoints. We observed several such instances, all in $\mathcal{N}_2$ and all involving packets sent to or from `lbli`. In one, an acknowledgement for sequence 1 (corresponding to an ack for the receiver's SYN-ack) arrived at the receiver with an ack for sequence 33 instead, and similarly for the next packet; then two more packets after those arrived with acknowledgements for sequence 65. Needless to say, the receiver had never sent any of this data! In others, packets sent without any data arrived with 512 bytes of in-sequence data, and other packets changed size in flight. All of these failed their checksum tests. But the ability of a CSLIP link to turn bit errors into plausible header fields, which is somewhat inevitable due to its clever, heavy use of compression, means that, when a corrupted packet finally *does* pass the checksum test, it is considerably more likely to both be accepted by the receiving TCP as valid and to desynchronize the TCP's state with respect to that of its remote peer.

Chapter 14

Bottleneck Bandwidth

In this chapter we discuss one of the fundamental properties of a network connection, the *bottleneck bandwidth* that sets the upper limit on how quickly the network can deliver the sender's data to the receiver. In § 14.1 we discuss the general notion of bottleneck bandwidth and why we consider it a fundamental quantity. § 14.2 discusses "packet pair," the technique used in previous work, and § 14.3 discusses why for our study we gain significant benefits using "receiver-based packet pair," in which the measurements used in the estimation are those recorded by the receiver, rather than the ack "echoes" that the sender later receives.

While packet pair often works well, in § 14.4 we illustrate four difficulties with the technique, three surmountable and the fourth fundamental. Motivated by these problems, we develop a robust estimation algorithm, "packet bunch modes" (PBM). To do so, we first in § 14.5 discuss an alternative estimation technique based on measurements of the "peak rate" (PR) achieved by the connection, for use in calibrating the PBM technique, which we then develop in detail in § 14.6. In § 14.7, we analyze the estimated bottleneck bandwidths for the Internet paths in our study, and in § 14.8 we finish with a comparison of the efficacy of the various techniques.

14.1 Bottleneck bandwidth as a fundamental quantity

Each element in the end-to-end chain between a data sender and the data receiver has some *maximum rate* at which it can forward data. These maxima may arise directly from physical properties of the element, such as the frequency bandwidth of a wire, or from more complex properties, such as the minimum amount of time required by a router to look up an address to determine how to forward a packet. The first of these situations often dominates, and accordingly the term *bandwidth* is used to denote the maximum rate, even if the maximum does not come directly from a physical bandwidth limitation.

Because sending data involves forwarding the data along an end-to-end *chain* of networking elements, the *slowest* element in the entire chain sets the *bottleneck bandwidth*, i.e., the maximum rate at which data can be sent along the chain. The usual assumption is that the bottleneck element is a network *link* with a limited bandwidth, although this need not be the case.

Note that from our data we cannot say anything meaningful about the *location* of the bottleneck along the network path, since our methodology gives us only end-to-end measurements (though see § 15.4). Furthermore, there may be multiple elements along the network path, each limited to the same bottleneck rate. Thus, our analysis is confined to an assessment of the bottleneck bandwidth as an end-to-end path property, rather than as the property of a particular element in the path.

We must make a crucial distinction between *bottleneck* bandwidth and *available* bandwidth. The former gives an upper bound on how fast a connection can *possibly* transmit data, while the less-well-defined latter term denotes how fast the connection in fact *can* transmit data, or in some cases how fast it *should* transmit data to preserve network stability, even though it could transmit faster. Thus, the available bandwidth never exceeds the bottleneck bandwidth, and can in fact be much smaller. Bottleneck bandwidth is often presumed to be a fairly static quantity, while available bandwidth is often recognized as intimately reflecting current network traffic levels (congestion). Using the above terminology, the bottleneck location(s), if we were able to pinpoint them, would generally not change during the course of a connection, unless the network path used by the connection underwent a routing changes. But the networking element(s) limiting the available bandwidth might readily change over the lifetime of a connection.

TCP's congestion avoidance and control algorithms reflect an attempt to confine each connection to the available bandwidth. For this purpose, the bottleneck bandwidth is essentially irrelevant. For connection *performance*, however, the bottleneck bandwidth is a fundamental quantity, because it indicates a limit on what the connection can hope to achieve. If the sender tries to transmit any faster, not only is it guaranteed to fail, but the additional traffic it generates in doing so will either lead to queueing delays somewhere in the network, or packet drops, if the overloaded element lacks sufficient buffer capacity.

We discuss available bandwidth further in § 16.5, and for the remainder of this chapter focus on assessing bottleneck bandwidth.

The bottleneck bandwidth is further a fundamental quantity because it determines what we term the *self-interference time-constant*, Q_b. Q_b measures the amount of time required to forward a given packet through the bottleneck element. Thus, Q_b is identical to the service time at the bottleneck element; we use the term "self-interference time-constant" instead because of the central role Q_b plays in determining when packet transit times are necessarily correlated, as discussed below.

If a packet carries a total of b bytes and the bottleneck bandwidth is ρ_B byte/sec, then:

$$Q_b = \frac{b}{\rho_B} \tag{14.1}$$

in units of seconds. We use the term "self-interference" because if the sender transmits two b-byte packets with an interval $\Delta T_s < Q_b$ between them, then the second one is guaranteed to have to wait behind the first one at the bottleneck element (hence the use of "Q" to denote "queueing").

We use the notation Q_b instead of the more functional notation $Q(b)$ because we will assume unless otherwise stated that, for a particular trace pair, b is fixed to the maximum segment size (MSS; § 9.2.2). We note that full-sized packets are *larger* than MSS, due to overhead from transport, network, and link-layer headers. However, while it might at first appear that this overhead is known (except for the link-layer) and can thus be safely added into b, if the bottleneck link along a path uses *header compression* (§ 13.3) then the header as transmitted might take much less data than would appear from tallying the number of bytes in the header. Since many of the most significant bottleneck links in our study also use header compression, we decided to perform all of our analysis of the bottleneck bandwidth in terms of the maximum rate at which a connection can transmit *user data*.

For our measurement analysis, accurate assessment of Q_b is critical. Suppose we observe a sender transmitting p_1 and p_2, both b bytes in size, and that they are sent an interval ΔT_s apart. If

$$\Delta T_s < Q_b,$$

then we know that p_2 had to wait a time $Q_b - \Delta T_s$ at the bottleneck element B while p_1 was being forwarded across B. (This assumes that p_1 and p_2 take the same path through the network, a point we address in detail later in this chapter.)

Thus, if $\Delta T_s < Q_b$, the delays experienced by p_1 and p_2 are *perforce correlated*. If $\Delta T_s \geq Q_b$, then if p_2 experiences greater delay than p_1, the increase is not due to self-interference but some other source (such as additional traffic from other connections, or processing delays).

We use Q_b to analyze packet timings and remove self-interference effects in Chapter 16. In this chapter, we focus on sound estimation of Q_b, as we must have this in order for the subsequent timing analysis to be likewise sound.

14.2 Packet pair

The fundamental idea behind the *packet pair* estimation technique is that, if two packets are transmitted by the sender with an interval $\Delta T_s < Q_b$ between them, then when they arrive at the bottleneck they will be spread out in time by the transmission delay of the first packet across the bottleneck: after completing transmission through the bottleneck, their spacing will be exactly Q_b. Barring subsequent delay variations (due to downstream queueing or processing lulls), they will then arrive at the receiver spaced not ΔT_s apart, but $\Delta T_r = Q_b$. The size b then enables computation of ρ_B via Eqn 14.1.[1]

The principle of the bottleneck spacing effect was noted in Jacobson's classic congestion paper [Ja88], where it in turn leads to the “self-clocking” mechanism (§ 9.2.5). Keshav subsequently formally analyzed the behavior of packet pair for a network in which all of the routers obey the “fair queueing” scheduling discipline, and developed a provably stable flow control scheme based on packet pair measurements [Ke91].[2] Both Jacobson and Keshav were interested in estimating *available* rather than *bottleneck* bandwidth, and for this *variations* from Q_b due to queueing are of primary concern (§ 16.5). But if, as for us, the goal is to estimate ρ_B, then these variations instead become noise we must deal with.

To use Jacobson's self-clocking model to estimate bottleneck bandwidth requires an assumption that delay variation in the network is small compared to Q_b. Using Keshav's scheme requires fair queueing. Internet paths, however, often suffer considerable delay variation (Chapter 16), and Internet routers do not employ fair queueing. Thus, efforts to estimate ρ_B using packet pair must deal with considerable noise issues. The first step in dealing with measurement noise is

[1] If the two packets in the pair have different sizes b_1 and b_2, then which to use depends on how we interpret the timestamps for the packets. If the timestamps reflect when the packet *began* to arrive at the packet filter's monitoring point, then b_1 should be used, since that is how much data was transmitted between the timestamps of the two packets. If the timestamps reflect when the packet *finished* arriving, then b_2 should be used. In practice, a packet's timestamp is recorded some time *after* the packet has finished arriving, per § 10.2, and so if $b_1 \neq b_2$, `tcpanaly` uses b_2.

[2] Keshav also coined the term “packet pair.”

to analyze as large a number of pairs as feasible, with an eye to the tradeoff between measurement accuracy and undue loading of the network by the measurement traffic.[3]

Bolot used a stream of packets sent at fixed intervals to probe several Internet paths in order to characterize delay and loss behavior [Bo93]. He measured round-trip delay of UDP echo packets and, among other analyses, applied the packet pair technique to form estimates of bottleneck bandwidths. He found good agreement with known link capacities, though a limitation of his study is that the measurements were confined to a small number of Internet paths. One of our goals is to address this limitation by determining how well packet pair techniques work across diverse Internet conditions.

Recent work by Carter and Crovella also investigates the utility of using packet pair in the Internet for estimating bottleneck bandwidth [CC96a]. Their work focusses on `bprobe`, a tool they devised for estimating bottleneck bandwidth by transmitting 10 consecutive ICMP echo packets and recording the arrival times of the corresponding replies. `bprobe` then repeats this process with varying (and carefully chosen) packet sizes. Much of the effort in developing `bprobe` concerns how to filter the resulting raw measurements in order to form a solid estimate. `bprobe` currently filters by first widening each estimate into an interval by adding an (unspecified) error term, and then finding the point at which the largest number of intervals overlap. The authors also undertook to calibrate `bprobe` by testing its performance for a number of Internet paths with known bottlenecks. They found in general it worked well, though some paths exhibited sufficient noise to sometimes produce erroneous estimates. Finally, they note that measurements made using larger echo packets yielded more accurate estimates than those made using smaller packets, which bodes well for our interest in measuring Q_b for $b = \text{MSS}$.

One limitation of both studies is that they were based on measurements made only at the data sender (§ 9.1.3). Since in both studies the packets echoed back from the remote end were the same size as those sent to it, neither analysis was able to distinguish whether the bottleneck along the forward and reverse paths was the same, or whether it was present in only one direction. The bottleneck could differ in the two directions due the packets traversing different physical links because of asymmetric routing (§ 8), or because some media, such as satellite links, can have significant bandwidth asymmetries depending on the direction traversed [DMT96].

For the study in [CC96a], this is not a problem, because the authors' ultimate goal was to determine which Web server to pick for a document available from a number of different servers. Since Web transfers are request/response, and hence bidirectional (albeit potentially asymmetric in the volume of data sent in each direction), the bottleneck for the combined forward and reverse path is indeed a figure of interest. For general TCP traffic, however, this is not always the case, since for a unidirectional transfer—especially for FTP transfers, which can sometimes be quite huge [PF95]—the data packets sent along the forward path are much larger than the acks returned along the reverse path. Thus, even if the reverse path has a significantly lower bottleneck bandwidth, this is unlikely to limit the connection's maximum rate. However, for estimating bottleneck bandwidth by measuring TCP traffic a second problem arises: if the only measurements available are those at the sender, then ack compression (§ 16.3.1) can significantly alter the spacing of the small ack packets as they return through the network, distorting the bandwidth estimate. We investigate the degree of this problem below.

[3] Gathering large samples, however, can conflict with another goal, that of forming an estimate *quickly*, briefly discussed at the end of the chapter.

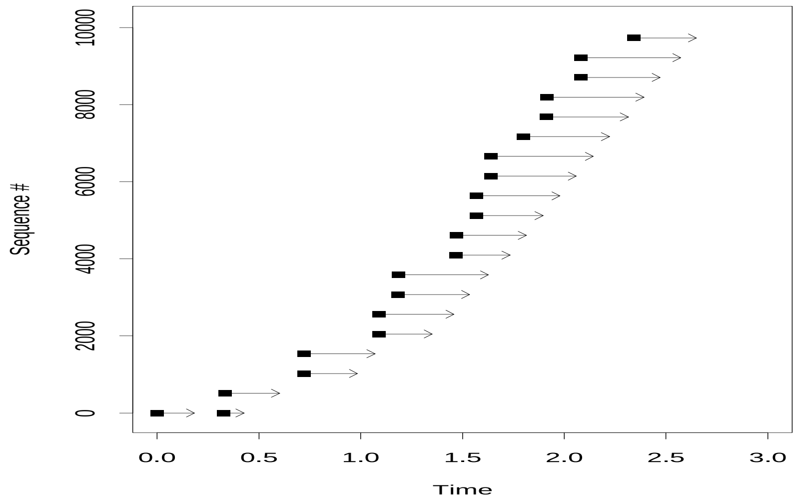

Figure 14.1: Paired sequence plot showing timing of data packets at sender (black squares) and when received (arrowheads)

14.3 Receiver-based packet pair

For our analysis, we consider what we term *receiver*-based packet pair (RBPP), in which we look at the pattern of data packet arrivals at the receiver. We also utilize knowledge of the pattern in which the data packets were originally sent, so we assume that the receiver has full timing information available to it. In particular, we assume that the receiver knows when the packets sent were *not* stretched out by the network, and can reject these as candidates for RBPP analysis.

RBPP is considerably more accurate than sender-based packet pair (SBPP; cf. § 14.2), since it eliminates the additional noise and possible asymmetry of the return path, as well as noise due to delays in generating the acks themselves (§ 11.6.4). Figure 14.1 shows a *paired sequence plot* for data transferred over a path known to have a 56 Kbit/sec bottleneck link. The centers of the filled black squares indicate the times at which the sender transmitted the successive data packets, and the arrowheads point to the times at which they arrived at the receiver. (We have adjusted the relative clock offset per the methodology given in § 12.5). The packet pair effect is quite strong: while the sender tends to transmit packets in groups of two back-to-back (due to slow start opening the congestion window), this timing structure has been completely removed by the time the packets arrive, and instead they come in at a nearly constant rate of about 6,200 byte/sec.

Figure 14.2 shows the same trace pair with the acknowledgements added. They are offset slightly lower than the sequence number they acknowledge for legibility. The arrows start at the point in time at which the ack was generated by the receiver, and continue until received by the sender. We can see that some acks are generated immediately, but others (such as 4,096) are delayed. Furthermore, there is considerable variation among the transit times of the acks, even though *they* are almost certainly too small to be subject to stretching at the bottleneck link along the return path. If we follow the ack arrowheads by eye, it is clear that the strikingly smooth pattern in Figure 14.1

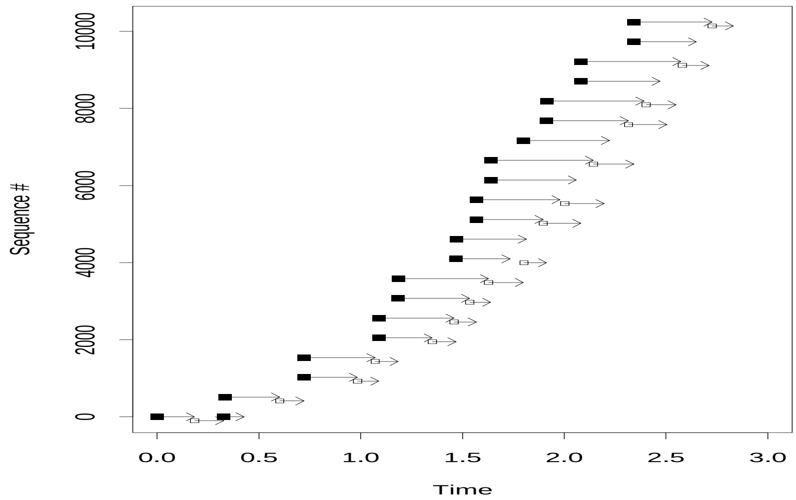

Figure 14.2: Same plot with acks included

has been blurred by the ack delays, which have nothing to do with the quantity of interest, namely Q_b on the forward path.

14.4 Difficulties with packet pair

As shown in the Bolot and Carter/Crovella studies ([Bo93, CC96a]), packet pair techniques often provide good estimates of bottleneck bandwidth. We are interested both in estimating the bottleneck bandwidth of the Internet paths in our study, and, furthermore, whether the packet-pair technique is robust enough that an Internet transport protocol might profitably use it in order to make decisions based on Q_b.

A preliminary investigation of our data revealed four potential problems with packet pair techniques, even if receiver-based. Three of these can often be satisfactorily addressed, but the fourth is more fundamental. We discuss each in turn.

14.4.1 Out-of-order delivery

The first problem stems from the fact that, for some Internet paths, out-of-order packet delivery occurs quite frequently (§ 13.1). Clearly, packet pairs delivered out of order completely destroy the packet pair technique, since they result in $\Delta T_r < 0$, which then leads to a negative estimate for ρ_B. The receiver sequence plot in Figure 14.3 illustrates the basic problem. (Compare with the clean arrivals in Figure 14.1.)

Out-of-order delivery is symptomatic of a more general problem, namely that the two packets in a pair may not take the same route through the network, which then violates the assumption that the second queues behind the first at the bottleneck. In a sense, out-of-order delivery is a blessing, because the receiver can usually *detect* the event (based on sequence numbers, and

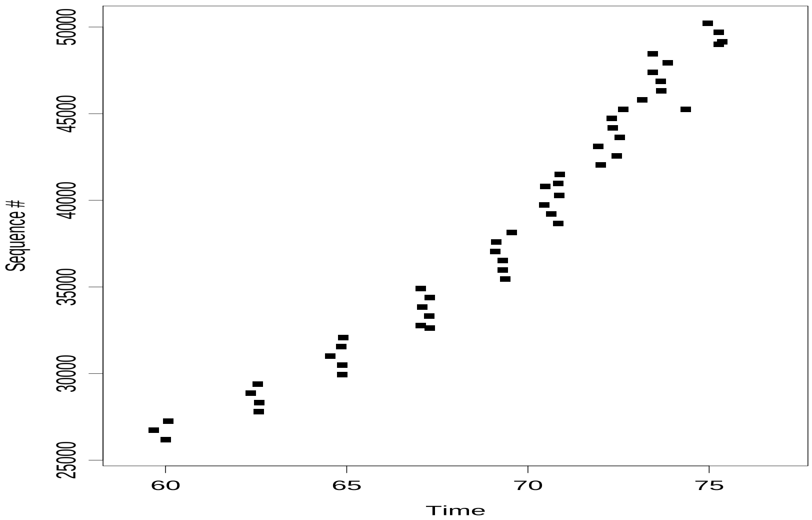

Figure 14.3: Receiver sequence plot illustrating difficulties of packet-pair bottleneck bandwidth estimation in the presence of out-of-order arrivals

possibly IP "id" fields for retransmitted packets; cf. § 10.5). More insidious are packets pairs that traverse different paths but still arrive in order. The interval computed from their arrivals may have nothing to do with the bottleneck bandwidth, and yet it is difficult to recognize this case and discard the measurement from subsequent analysis. We discuss a particularly problematic instance of this problem in § 14.4.4 below.

14.4.2 Limitations due to clock resolution

Another problem relates to the receiver's clock resolution, C_r (§ 12.3). C_r can introduce large margins of error around estimates of ρ_B. Suppose two b-byte packets arrive at the receiver with a spacing of ΔT_r. We want to estimate ρ_B from Eqn 14.1 using

$$\begin{aligned} \Delta T_r &= Q_b \\ &= \frac{b}{\rho_B}, \end{aligned}$$

and hence

$$\rho_B = \frac{b}{\Delta T_r}. \tag{14.2}$$

However, we cannot measure ΔT_r exactly, but only estimate an interval in which it lies, using:

$$\max(\Delta\widetilde{T}_r - C_r, 0) \leq \Delta\widehat{T}_r \leq \Delta\widetilde{T}_r + C_r, \tag{14.3}$$

where $\Delta\widetilde{T}_r$ is the value reported by the receiver's clock for the spacing between the two packets. Combining Eqn 14.2 with Eqn 14.3 gives us:

$$\begin{aligned} \widehat{\rho}_b &= \frac{b}{\Delta\widehat{T}_r}, \\ \frac{b}{\Delta\widetilde{T}_r + C_r} \le \ \widehat{\rho}_b &\le \frac{b}{\max(\Delta\widetilde{T}_r - C_r, 0)}. \end{aligned} \tag{14.4}$$

In the case where $\Delta\widetilde{T}_r \le C_r$, i.e., the two packets arrived with the clock advancing at most once, we cannot provide any upper bound on $\widehat{\rho}_b$ at all. Thus, for example, if $C_r = 10$ msec, a common value on older hardware (§ 12.4.2), then for $b = 512$ bytes, from the arrival of a single packet pair we cannot distinguish between

$$\rho_B = \frac{512}{0.010 \text{ sec}} = 51,200 \text{ byte/sec},$$

and

$$\rho_B = \infty.$$

This means we cannot distinguish between a fairly pedestrian T1 link of under 200 Kbyte/sec, and a blindingly fast (today!) OC-12 link of about 80 Mbyte/sec.

For $C_r = 1$ msec, the threshold rises to 512,000 byte/sec, still much too low for meaningful estimation for high-speed networks. For today's networks, $C_r = 100$ μsec almost allows us to distinguish between T3 speeds of a bit over 5 Mbyte/sec and higher speeds. Since some of the clocks in our study had finer resolution, we view this problem as tractable with today's (better) hardware. It is not clear, however, whether in the future processor clock resolution will grow finer at a rate to match how network bandwidths grow faster (and thus Q_b decreases).

While some of today's hardware provides sufficient resolution for packet-pair analysis, other platforms do not, so we still need to find a way to deal with low-resolution clocks. In line with the argument in the previous paragraph, doing so also potentially benefits measurement of future networks, since their bandwidth growth may outpace that of clock resolution.

A basic technique for coping with poor clock resolution is to use packet *bunch* rather than packet pair.[4] The idea behind packet bunch, in which $k \ge 2$ back-to-back packets are used, is that bunches should be less prone to noise, since individual packet variations are smoothed over a single large interval rather than $k - 1$ small intervals. This idea has not been thoroughly tested, and one might argue the opposite: if packets are occasionally subject to large transient delays due to bursts of cross traffic, than the larger k is, the greater the likelihood that a bunch will be disrupted by a significant delay, leading to underestimation of ρ_B. We investigate this concern below. However, another benefit of packet bunch is that the overall time interval ΔT_r^k spanned by the k packets will be about $k - 1$ times larger than that spanned by a single packet pair. Accordingly, by choosing sufficiently large k we can diminish the adverse effects of poor clock resolution, except for the problem mentioned above of encountering spurious delays and underestimating ρ_B as a result.

[4] The term "packet bunch" has been in informal use for at least several years; however, we were unable to find any appearance of it in the networking literature. The *notion* appears in [BP95a], in the discussion of the "Vegas-*" variant, which attempts to estimate available bandwidth using a four-packet version of packet pair; and in [Ho96], which uses an estimate derived from the timing of three consecutive acks.

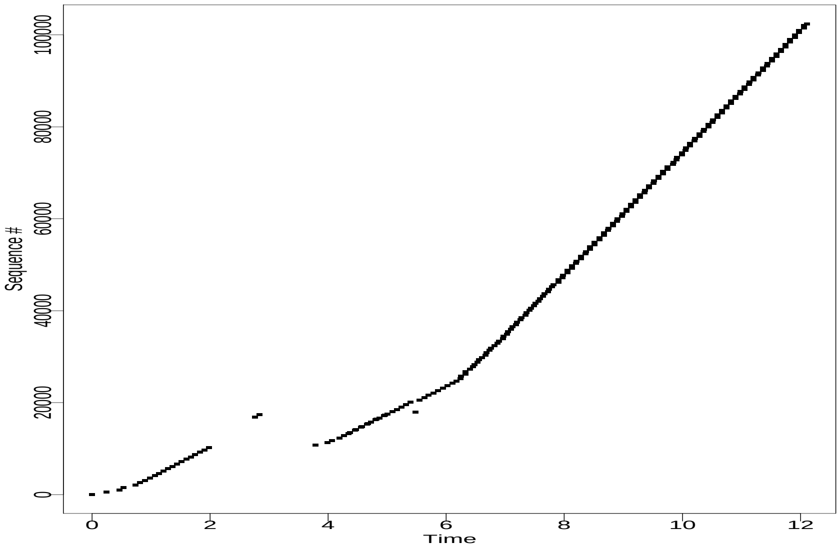

Figure 14.4: Receiver sequence plot showing two distinct bottleneck bandwidths

14.4.3 Changes in bottleneck bandwidth

Another problem that *any* bottleneck bandwidth estimation must deal with is the possibility that the bottleneck *changes* over the course of the connection. Figure 14.4 shows a trace in which this happened. We have shown the entire trace, but only the data packets and not the corresponding acks. While the details are lost, the eye immediately picks out a transition between one overall slope to another, just after $T = 6$. The first slope corresponds to about 6,600 byte/sec, while the second is about 13,300 byte/sec, and increase of about a factor of two.

For this example, we know enough about one of the endpoints (`lbli`) to fully describe what occurred. `lbli`'s Internet connection is via an ISDN link. The link has two *channels*, each nominally capable of 64 Kbyte/sec. When `lbli` initially uses the ISDN link, the router only activates one channel (to reduce the expense). However, if `lbli` makes sustained use of the link, then the router activates the second channel, doubling the bandwidth.

While for this particular example the mechanism leading to the bottleneck shift is specific to the underlying link technology, the *principle* that the bottleneck can change with time is both important and general. It is important to detect such an event, because it has a major impact on the ensuing behavior of the connection. Furthermore, bottlenecks can shift for reasons other than multi-channel links. In particular, routing changes might alter the bottleneck in a significant way.

Packet pair studies to date have focussed on identifying a *single* bottleneck bandwidth [Bo93, CC96a]. Unfortunately, in the presence of a bottleneck shift, any technique shaped to estimate a single, unchanging bottleneck will fail: it will either return a bogus compromise estimate, or, if care is taken to remove noise, select one bottleneck and reject the other. In both cases, the salient fact that the bottleneck shifted is overlooked. We attempt to address this problem in the development of our robust estimation algorithm (§ 14.6).

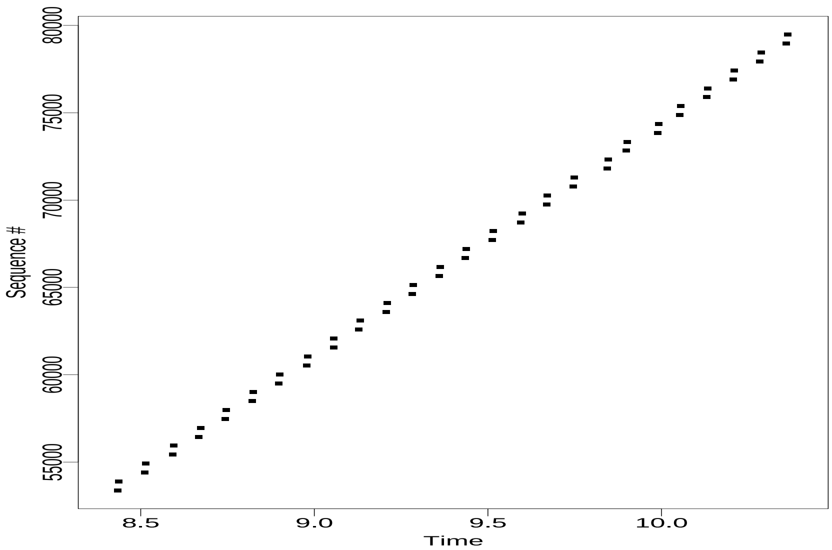

Figure 14.5: Enlargement of part of the previous figure

14.4.4 Multi-channel bottleneck links

We now turn to a more fundamental problem with packet-pair techniques, namely bottleneck estimation in the face of multi-channel links. Here we do not concern ourselves with the problem of detecting that the bottleneck has *changed* due to the activation or deactivation of the link's additional channel (§ 14.4.3). We instead illustrate a situation in which packet pair yields *incorrect overestimates* even in the absence of any delay noise.

Figure 14.5 expands a portion of Figure 14.4. The slope of the large linear trend in the plot corresponds to 13,300 byte/sec, as earlier noted. However, we see that the line is actually made up of pairs of packets. Figure 14.6 expands the plot again, showing quite clearly the pairing pattern. The slope between the pairs of packets corresponds to a data rate of about 160 Kbyte/sec, even though we know that the ISDN link has a hard limit of 128 Kbit/sec = 16 Kbyte/sec, a factor of ten smaller! Clearly, an estimate of

$$\widehat{\rho}_b \approx 160 \text{ Kbyte/sec}$$

must be wrong, yet that is what a packet-pair calculation will yield.

The question then is: where is the spacing corresponding to 160 Kbyte/sec coming from? A clue to the answer lies in the number itself. It is not far below the user data rates achieved over T1 circuits, typically on the order of 170 Kbyte/sec. It is as though every other packet were immune to queueing behind its predecessor at the known 16 Kbyte/sec bottleneck, but instead queued behind it at a downstream T1 bottleneck.

Indeed, this is exactly what is happening. As discussed in § 14.4.3, the bottleneck ISDN link has two channels. These operate in *parallel*. That is, when the link is idle and a packet arrives, it goes out over the first channel, and when another packet arrives shortly after, it goes out over the *other* channel. If a third packet then arrives, it has to wait until one of the channels becomes free. Effectively, it is queued behind not its immediate predecessor but its predecessor's predecessor, the

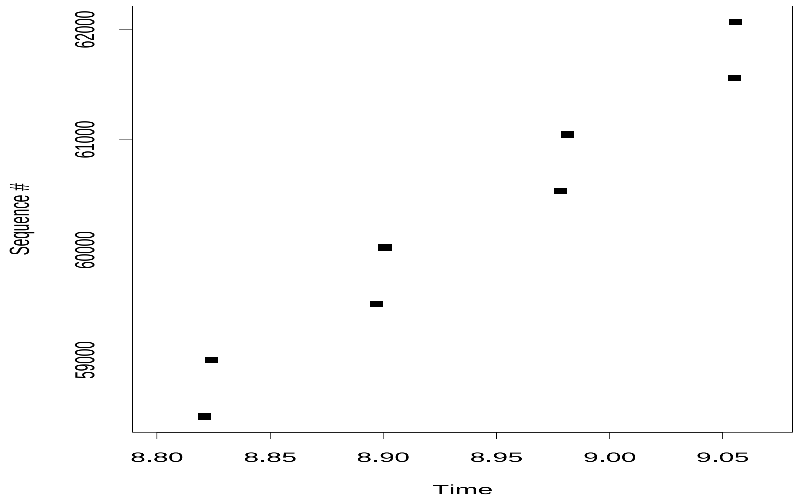

Figure 14.6: Enlargement of part of the previous figure

first packet in the series, and it is queued not for a 16 Kbyte/sec link but for an 8 Kbyte/sec channel making up just part of the link.

As queues build up at the router utilizing the multi-channel link, often both channels will remain busy for an extended period of time. In this case, additional traffic arriving at the router, or processing delays, can alter the "phase" between the two channels, meaning the offset between when the first begins sending a packet and when the second does so. Thus, we do not always get an arrival pattern clearly reflecting the downstream bottleneck as shown in Figure 14.6. We can instead get a pairing pattern somewhere between the downstream bottleneck and the true bottleneck. Figure 14.7 shows an earlier part of the same connection where a change in phase quite clearly occurs a bit before $T = 8$. Here the pair slope shifts from about 23 Kbyte/sec up to 160 Kbyte/sec. Note that the overall rate at which new data arrives at the receiver has not changed at all during this transition, only the fine-scale timing structure has changed.

We conclude that, in the presence of multi-channel links, packet-pair techniques can give completely misleading estimates for ρ_B. Worse, these estimates will often be much too high. The fundamental problem is the assumption with packet pair that there is only a single path through the network, and that therefore packets queue behind one another at the bottleneck.

We should stress that the problem is more general than the circumstances shown in this example, in two important ways. First, while in this example the parallelism leading to the estimation error came from a single link with two separate (and parallel) physical channels, the exact same effect could come from a router that balances its outgoing load across two different links. If these links have different propagation times, then the likely result is out-of-order arrivals, which can be detected by the receiver and removed from the analysis (§ 14.4.1). But if the links have equal or almost equal propagation times, then the parallelism they offer can completely obscure the true bottleneck bandwidth.

Second, it may be tempting to dismiss this problem as correctable by using packet bunch

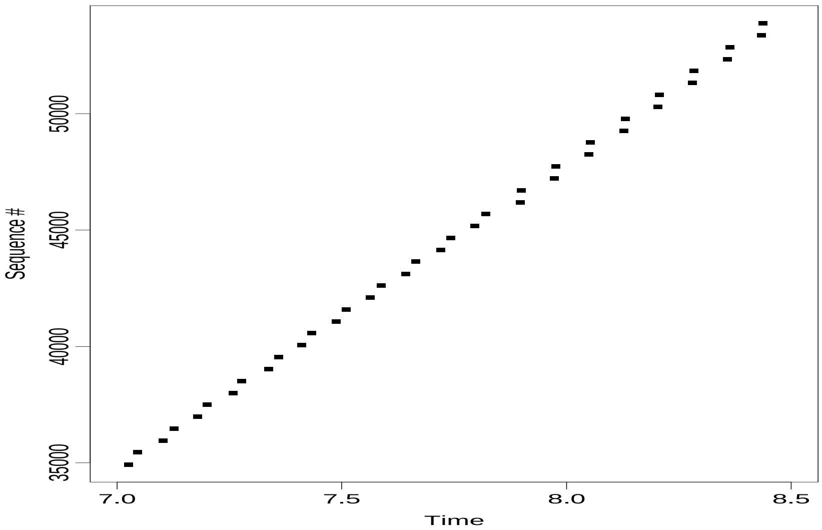

Figure 14.7: Multi-channel phasing effect

(§ 14.4.2) with $k = 3$ instead of packet pair. This argument is not compelling without further investigation, however, because packet bunch is potentially more prone to error; and, more fundamentally, $k = 3$ only works if the parallelism comes from *two* channels. If it came from *three* channels (or load-balancing links), then $k = 3$ will still yield misleading estimates.

We now turn to developing techniques to address these difficulties.

14.5 Peak rate estimation

In this section we discuss a simple, cheap-to-compute, and not particularly accurate technique for estimating the bottleneck bandwidth along a network path. We term this technique *peak rate* and subsequently refer to it as PR. Our interest in PR lies in providing *calibration* for the robust technique developed in the next section, based on packet-bunch modes ("PBM"). We develop two PR-based estimates, a "conservative" estimate, $\widehat{\mathrm{PR}}^c$, very unlikely to be an overestimate, and an "optimistic" estimate, $\widehat{\mathrm{PR}}^o$, which is more likely to be accurate but is also prone to overestimation. Armed with these estimates, we then can compare them with results given by PBM. If the robust technique yields an estimate less than $\widehat{\mathrm{PR}}^c$, or higher than $\widehat{\mathrm{PR}}^o$, then the discrepancy merits investigation. If they generally agree, then perhaps we can use the simpler PR techniques instead of PBM without losing accuracy (though it would be surprising to find that PR techniques suffice, per the discussion below).

PR is based on the observation that the peak rate the connection ever manages to transmit along the path should give a lower bound on the bottleneck rate. PR is a necessarily *stressful* technique in that it requires loading the network to capacity to assure accuracy. As such, we would prefer not to use PR as an active measurement methodology, but it works fine for situations in which the measurements being analyzed are due to traffic initiated for some reason other than bottleneck measurement. Thus, PR makes sense as a candidate algorithm for adding to a transport protocol.

In contrast, packet pair and PBM do not necessarily require stressing the network for accuracy, so they are attractive both as additions to transport protocols to aid in their decision-making, and as independent network analysis tools.

At its simplest, PR consists of just dividing the amount of data transferred by the duration of the connection. This technique, however, often grossly underestimates the true bottleneck bandwidth, because transmission lulls due to slow-start, insufficient window, or retransmission timeouts can greatly inflate the connection duration.

To reduce the error in PR requires confining the proportion of the connection on which we calculate the peak rate to a region during which none of these lulls impeded transmission. Avoiding slow-start and timeout delays is easy, since these regions are relatively simple to identify. Identifying times of insufficient window, however, is more difficult, because the correct window is a function of both the round-trip time (RTT) and the available bandwidth, and the latter is shaped in part by the bottleneck bandwidth, which is what we are trying to estimate.

If the connection was at some point not window-limited, then by definition it achieved a sustained rate (over at least one RTT) at or exceeding the available capacity. Since the hope embodied in PR is that at some point the available capacity matched the bottleneck bandwidth, we address the problem of insufficient window by forming our estimate from the maximum rate achieved over a single RTT.

`tcpanaly` computes a PR-based estimate by advancing through the data packet arrivals at the TCP receiver as follows. For each arrival, it computes the amount of data (in bytes) that arrived between that arrival and the next data packet coming just *beyond* the edge of a temporal window equal to the minimum RTT, $\mathrm{RTT}_{\mathrm{min}}$. ($\mathrm{RTT}_{\mathrm{min}}$ is computed as the smallest interval between a full-sized packet's departure from the sender and the arrival at the sender of an acknowledgement for that packet.) Suppose we find B bytes arrived in a total time $\Delta T_r > \mathrm{RTT}_{\mathrm{min}}$, and that the interval spanned by the departure of the packets when transmitted by the sender is ΔT_s.[5] Finally, if any of the packets arrived out of order, then we exclude the group of packets from any further analysis.

Otherwise, we compute the *expansion factor*

$$\xi_{s,r} = \frac{\Delta T_r + C_r}{\Delta T_s + C_s}, \tag{14.5}$$

where C_s and C_r are the resolutions of the sender's and receiver's clocks (§ 12.3). $\xi_{s,r}$ measures the factor by which the group of packets was spread out by the network. If less than 1, then the packets were *not* spread out by the network and hence not shaped by the bottleneck. Thus, calculations based on their arrival times should not be used in estimating the bottleneck. In practice, however, two effects complicate the simple rule of rejecting timings if $\xi_{s,r} < 1$. The first is that, if C_s is considerably different (orders of magnitude larger or smaller) than C_r, then $\xi_{s,r}$ can vary considerably, even if the magnitudes of ΔT_r and ΔT_s are close. The second problem is that sometimes due to "self-clocking" (§ 9.2.5), a connection rapidly settles into a pattern of transmitting packets at very close to the bottleneck bandwidth, in which case we might find $\xi_{s,r}$ slightly less than 1 even though it allows for a solid estimate of ρ_B. To address these concerns, we use a slightly different definition

[5] Here, B does *not* include the bytes carried by the first packet of the group, since we assume that the packet timestamps reflect when packets *finished* arriving, so the first packet's bytes arrived before the point in time indicated by its timestamp. Also see the footnote in § 14.2.

of $\xi_{s,r}$ than that given by Eqn 14.5:

$$\widetilde{\xi}_{s,r} = \frac{\Delta T_r + C_r}{\Delta T_s + C_r}, \tag{14.6}$$

namely, C_r is used in both the numerator and the denominator, which eliminates large swings in $\xi s, r$ due to discrepancies between C_r and C_s. This is a bit of a "fudge factor," and in retrospect a better solution would have been to use $C_r + C_s$; but, we find it works well in practice. The other fudge factor is that `tcpanaly` allows estimates for $\widetilde{\xi}_{s,r} \geq 0.95$, to accommodate self-clocking effects.

After taking into account these considerations, we then form the PR-based estimate:

$$\widehat{\mathrm{PR}}^c = \frac{B}{\Delta T_r + C_r}. \tag{14.7}$$

The ^{c} superscript indicates that the estimator is *conservative*. Since it requires $\Delta T > \mathrm{RTT_{min}}$, it may be an underestimate if the connection never managed to "fill the pipe," which we illustrate shortly.

For the same group of packets, `tcpanaly` also computes an "optimistic" estimate corresponding to the group minus the final packet (the one that arrived more than $\mathrm{RTT_{min}}$ after the first packet):

$$\widehat{\mathrm{PR}}^o = \frac{B^-}{\Delta T_r^- + C_r}, \tag{14.8}$$

where B^- is the number of bytes received after subtracting those for the last packet in the group, and ΔT_r^- is likewise the interval over which the group arrived, excluding the final packet. (Thus, we always have $\Delta T_r^- \leq \mathrm{RTT_{min}}$.) `tcpanaly` does not place any restriction on the expansion factor for the packets used in this estimate, because sometimes the data packets were in fact compressed by the network ($\xi_{s,r}^- < 1$) but still give reliable estimates, because they queued at the bottleneck link behind earlier packets transmitted by the sender. `tcpanaly` does require, however, that either $\Delta T_r^- > \frac{1}{2}\mathrm{RTT_{min}}$, or that B is equal to the offered window (i.e., the connection was certainly window-limited), to ensure that compression of a small number of packets does not skew the estimate.[6]

We compute the final estimates as the maxima of $\widehat{\mathrm{PR}}^c$ and $\widehat{\mathrm{PR}}^o$. Note that the algorithms described above work best with cooperation between the sender and the receiver, in order to detect out-of-order arrivals, and to form a good estimate for $\mathrm{RTT_{min}}$, which can be quite difficult to assess from the receiver's vantage point because it cannot reliably infer the sender's congestion window.

Figure 14.8 illustrates the difference between computing $\widehat{\mathrm{PR}}^c$ and $\widehat{\mathrm{PR}}^o$ for a window-limited connection. $\mathrm{RTT_{min}}$ is about 110 msec. 8 packets arrive, starting at $T = 1.5$. The optimistic estimate is based on the 3,584 bytes arriving 22 msec after the first packet, for a rate of about 163 Kbyte/sec. The conservative estimate includes the 9th packet arriving significantly later than the first 8 (due to the window limit). The corresponding estimate is 4,096 bytes arriving in 115 msec, for a rate of about 36 Kbyte/sec. In this case, the optimistic estimate is much more accurate, as the limiting bandwidth is in fact that of a T1 circuit, corresponding to about 170 Kbyte/sec of user data. In this example, the connection is limited by the *offered* window, which is easy to detect. Very often, however, connections are instead limited by the congestion window, due earlier retransmission

[6]The precise method used is a bit more complicated, since it includes the possibility of different-sized packets arriving.

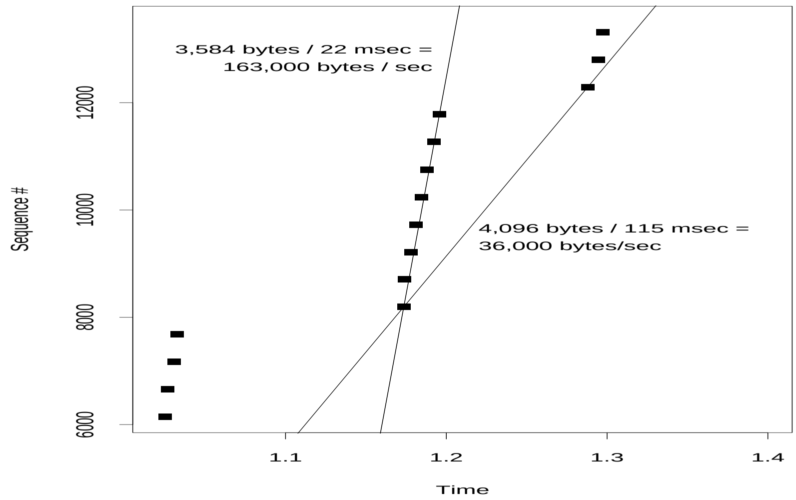

Figure 14.8: Peak-rate optimistic and conservative bottleneck estimates, window-limited connection

events. This limit is more difficult for the receiver to detect. Thus, $\widehat{\mathrm{PR}}^c$ often forms a considerable underestimate.

On the other hand, Figure 14.9 shows an instance in which $\widehat{\mathrm{PR}}^o$ is a large overestimate. The optimistic and conservative estimates for this trace both occurred for the group of packets arriving at time $T = 1.5$, in the middle of the figure. As can be seen from the surrounding groups, the true bottleneck capacity is about 170 Kbyte/sec (T1). The packet group at $T = 1.5$, however, has been *compressed* by the network (cf. § 16.3.2), and it all arrives at *Ethernet* speed. Thus, PR forms a gross overestimate for $\widehat{\mathrm{PR}}^o$. Furthermore, *even if* $\xi^-_{s,r}$ *were checked when forming this estimate*, the estimate would have been accepted, since the packets *left* the sender at Ethernet speed, too! In addition, $\widehat{\mathrm{PR}}^c$ is again a serious underestimate because the connection is again window-limited.

Thus, while PR is fairly simple to compute, it often fails to provide reliable estimates. We need a more robust estimation technique.

14.6 Robust bottleneck estimation

Motivated by the shortcomings of packet pair and PR estimation techniques, we developed a significantly more robust procedure, "packet bunch modes" (PBM). The main observation behind PBM is that dealing with the shortcomings of the other techniques involves both forming a range of estimates based on different packet bunch sizes, and to analyze the result with the possibility in mind of finding more than one bottleneck value.

By considering different bunch sizes, we can accommodate limited receiver clock resolutions (§ 14.4.2) and the possibility of multiple channels or load-balancing across multiple links (§ 14.4.4), while still avoiding the risk of underestimation due to noise diluting larger bunches, or window limitations (§ 14.5), since we also consider small bunch sizes.

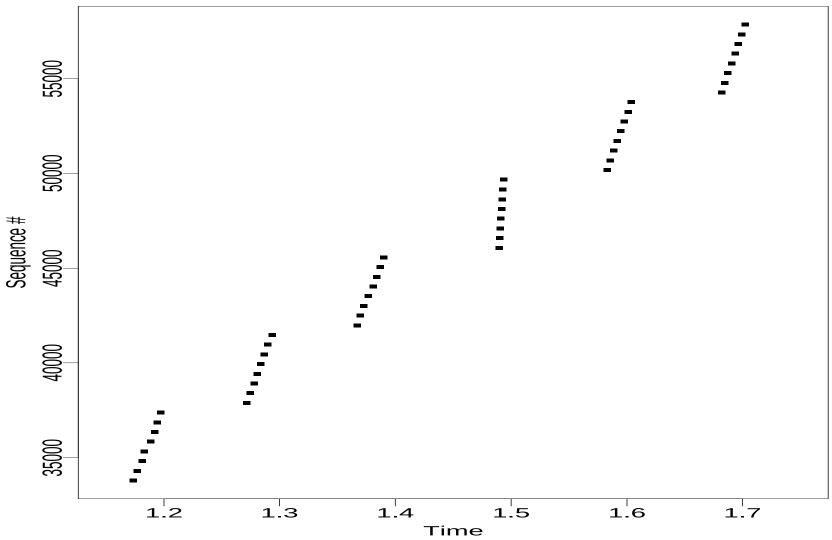

Figure 14.9: Erroneous optimistic estimate due to data packet compression

By allowing for finding multiple bottleneck values, we both again accommodate multi-channel (and multi-link) effects, and also the possibility of a bottleneck *change* (§ 14.4.3). Furthermore, these two effects can be distinguished from one another: multiple bottleneck values due to multi-channel effects *overlap*, while those due to bottleneck changes fall into separate regions in time.

In the remainder of this section we discuss a number of details of PBM. Many are heuristic in nature and evolved out of an iterative process of refining PBM to avoid a number of obvious estimation errors. It is unfortunate that PBM has a large heuristic component, as it makes it more difficult to understand. On the other hand, we were unable to otherwise satisfactorily deal with the considerable problem of noise in the packet arrival times. We hope that the basic ideas underlying PBM—searching for multiple modes and interpreting the ways they overlap in terms of bottleneck changes and multi-channel paths—might be revisited in the future, in an attempt to put them on a more systematic basis.

14.6.1 Forming estimates for each "extent"

PBM works by stepping through an increasing series of packet bunch sizes, and, for each, computing from the receiver trace all of the corresponding bottleneck estimates. We term the bunch size as the *extent* and denote it by k. For each extent, we advance a window over the arrivals at the receiver. The window is nominally k packets in size, but is extended as needed so that it always includes $k \cdot$ MSS bytes of data (so we can include less-than-full packets in our analysis). We do not, however, do this extension for $k = 1$, as that can obscure multi-channel effects.[7]

[7]For higher extents ($k > 1$), this extension does not obscure multi-channel effects, because we detecte multi-channel bottlenecks based on comparing estimates for $k = 1$ with estimates for $k = m$, where m is the number of multiple channels. Thus, the main concern is to not confuse the $k = 1$ estimate.

We also extend the window to include more packets if $\Delta T_r < C_r$, that is, if all the arrivals occurred without the receiver's clock advancing.

If any of the arrivals within the window occurred out of order, or if they were transmitted due to a timeout retransmission, we skip analysis of the group of packets, as the arrival timings will likely not reflect the bottleneck bandwidth.

If when the last packet in the group was sent, the sender had fewer than k packets in flight, then some unusual event occurred during the flight (such as retransmission or receipt of an ICMP source quench), and we likewise skip analysis of the group.

We next compute bounds on ΔT_r, using Eqn 14.3:

$$\begin{aligned} \Delta T_r^- &= \max(\Delta T_r - C_r, 0) \\ \Delta T_r^+ &= \Delta T_r + C_r. \end{aligned}$$

We also compute two *expansion factors* associated with the group, similar to that in Eqn 14.6. The first is more conservative:

$$\xi_{s,r}^c = \frac{\Delta T_r - C_r}{\Delta T_s + C_r}, \tag{14.9}$$

where ΔT_s again is the difference in time between the departure of the last packet and that of the first. The additional conservatism comes from using $\Delta T_r - C_r$ in the numerator. The second is likely to be overall the more accurate, but subject to fluctuations due to limited clock resolution:

$$\xi_{s,r}^o = \frac{\Delta T_r}{\Delta T_s + C_r}.$$

We term it "optimistic" since it yields expansion factors larger than $\xi_{s,r}^c$.

If the last packet group we inspected spanned an interval of $\Delta T_r'$, then we perform a heuristic test. If:

$$\frac{\Delta T_r + C_r}{\Delta T_r' + C_r} > 2, \tag{14.10}$$

then this group was spaced out more than twice as much as the previous group, and we skip the group (after assigning $\Delta T_r' \leftarrow \Delta T_r$), because it is likely to reflect sporadic arrivals. In some cases, this decision will be wrong; in particular, after a compression event such as that shown in Figure 14.9, we will often skip the immediately following packet group. However, this will be the only group we skip after the event, so, unless a trace is riddled with compression, our estimation does not suffer.

We then test whether $\xi_{s,r}^o \geq 0.95$ (where use of 0.95 rather than 1 is again an attempt to accommodate the self-clocking effect, per the discussion of Eqn 14.6). If so, we "accept" the group, meaning we treat it as providing a reliable estimate. (We will further analyze the accepted estimates, as discussed below.) Let B denote the number of bytes in the group (excluding those in the first packet, as also done in § 14.5). With the ith such estimate (corresponding to the ith acceptable group), we associate six quantities:

1. p_i^f, an index identifying the first packet in the group;
2. p_i^l, an index identifying the last packet in the group;
3. $\rho_i = B/\Delta T_r$, the bandwidth estimate;

4. $\rho_i^- = B/\Delta T_r+$, the lower bound on the estimate due to the clock resolution C_r;
5. $\rho_i^+ = B/\Delta T_r-$, the upper bound on the estimate; and
6. ξ_i^c, the conservative expansion factor corresponding to that given by Eqn 14.9.

We will refer to this set of quantities collectively as ψ_i.

One unusual, additional heuristic we use is that, if $\xi_{s,r}^o < 0.2$, i.e., the data packets were grossly compressed, then we *also* accept the estimate given by the corresponding group. (So we reject the estimate if $0.2 \le \xi_{s,r}^o < 0.95$.) This reasoning behind this heuristic is the same as that accompanying the discussion of Eqn 14.8, namely, that data packets can be highly compressed but still reflect the bottleneck bandwidth due to queueing at the bottleneck behind earlier packets transmitted by the sender. Finally, we note that this heuristic does not generally lead to problems accepting estimates based on compressed data that would otherwise be rejected, because the compression needs to be rampant for PBM to erroneously accept it as a bona fide estimate.

Finally, from a computational perspective, we would like to have an upper bound on the maximum extent k for which we do this analysis. The nominal upper bound we use is $k = 4$. If, however, the bounds on the estimates obtained for $k < 4$ are unsatisfactorily wide due to limited clock resolution, or if we found a new candidate bottleneck for $k = 4$, then we continue increasing k until both the bounds become satisfactory and we have not produced any new bottleneck candidates. These issues are discussed in more detail in the next section.

14.6.2 Searching for bottleneck bandwidth modes

In this section we discuss how we reduce a set of bottleneck bandwidth estimates into a small set of one or more values. Let $\Psi(k)$ be the set of bottleneck estimates formed using the procedure outlined in the previous section, for an extent of k packets. Let n_k denote the number of estimates, and N the total number of packets that arrived at the receiver. If:

$$n_k < \max(\frac{N}{4}, 5),$$

then we reject further analysis of $\Psi(k)$ because it consists of too few estimates. Otherwise, consider $\Psi(k)$ as comprising a sound set of estimates, and turn to the problem of extracting the best estimate from the set.

Previous bottleneck estimation work has focussed on identifying a single best estimate [Bo93, CC96a]. As discussed at the beginning of § 14.6, we must instead accommodate the possibility of forming multiple estimates. This then rules out the use of the most common robust estimator, the median, since it presupposes unimodality. We instead turn to techniques for identifying *modes*, i.e., local maxima in the density function of the distribution of the estimates. Using modal techniques gives PBM the ability to distinguish between a number of situations (bottleneck changes, multi-channel links) that previous techniques cannot.

Clustering the estimates

Because modes are properties of density functions, in trying to identify them we run into the usual problem of estimating density from a finite set of samples drawn from an (essentially) continuous distribution. [PFTV86] gives one procedure for doing so, based on passing a size-k window over sorted samples $X_{(i)}$ to see where $X_{(i+k-1)} - X_{(i)}$ is minimal. $[X_{(i)}, X_{(i+k-1)}]$ then corresponds to the region of highest density, since it packs the most datapoints into the least change in X. We experimented with this algorithm but found the results it produced for our estimation unsatisfactory, because there is no obviously correct choice for k, and different values yield different estimates.

We then devised an algorithm based on a similar principle of conceptually passing a window over the sorted data. Instead of parameterizing the algorithm with a window size k, we use an "error-factor," σ, for $\sigma > 1$. We then proceed through the sorted data, and, for each $X_{(i)}$, we search for an l satisfying $i \leq l < n$ such that:

$$X_{(l)} \leq \sigma X_{(i)} < X_{(l+1)}.$$

In other words, we look ahead to find two estimates that straddle the value of a factor σ larger than $X_{(i)}$. The first estimate, with index (l), is within a factor σ of $X_{(i)}$, while the second, $(l+1)$, is beyond it. If there is no such l (which can only happen if $X_{(n)} \leq \sigma X_{(i)}$), then we consider $X_{(n)}$ as the end of the range of the modal peak.

We term $C_i = l - i + 1$ the *cluster size*, as it gives us the number of points that lie within a factor of σ of $X_{(i)}$. If $C_i \leq 3$, then we consider the cluster *trivial*, and disregard it. Otherwise, we take as the cluster's mode its central observation, i.e., $X_{(i+\frac{C_i}{2})}$. If this is identical to that of a previously observed cluster, we *merge* the two clusters.[8] We then continue advancing the window until we have defined m cluster tuples. The final step is to prune out any clusters that overlap with a larger cluster.

We now turn to how to select σ. We decided to regard as consistent any bottleneck estimates that fall within $\pm 20\%$ of the central bottleneck estimate. We found that using smaller error bars (less than $\pm 20\%$) can lead to PBM finding spurious multiple peaks, while larger ones can wash out true, separate peaks.

Consequently, we will accept as falling within the estimate's bounds

$$X_{(i)} = 0.8 \cdot X_{(i+\frac{C_i}{2})},$$

and

$$X_{(l)} = 1.2 \cdot X_{(i+\frac{C_i}{2})}.$$

However, σ is in terms of the ratio between $X_{(l)}$, the high end of the bottleneck estimate's range, and $X_{(i)}$, the low end. It is easy to show that the above two relationships can hold if $\sigma = 1.5$, so that is the value we choose. Note, though, that we do not define the estimate's bounds in terms of $\pm 20\%$, but as

$$[\min(X_{(i)}, \rho_c^-), \ldots, \max(X_{(l)}, \rho_c^+)], \tag{14.11}$$

where ρ_c^- is the minimum bound on $X_{(i+\frac{C_i}{2})}$ due to clock resolution limits, and ρ_c^+ is the maximum such bound. In the absence of clock resolution limits, the bounds will often be tighter than $\pm 20\%$; but in the presence of such limits, they will often be wider.

The final result is $\Phi(k)$, a list of disjoint, non-trivial clusters associated with $\Psi(k)$, sorted by descending cluster size, and each with associated error bars given by Eqn 14.11.

[8] This can happen because of repeated observations yielding the same bottleneck estimates, due to clock resolution granularities and constant packet sizes.

Reducing the clusters

It is possible that $\Phi(k)$ is empty, because $\Psi(k)$ did not contain any non-trivial clusters. This can happen even if n_k is large, if the individual estimates differ sufficiently. In this case, we consider the extent-k analysis as having failed, and proceed to the next extent, or stop if $k \geq 4$.

Otherwise, we inspect the estimate reflected by each cluster to determine its suitability, as follows. First, we compute $\xi_i^{c(50)}$ and $\xi_i^{c(95)}$ as the 50th and 95th percentiles of the conservative expansion factors ξ_i^c associated with each of the estimates ψ_i within the cluster (per Eqn 14.9).

We next examine all of the estimates that fall within the cluster's error bars (nominally, $\pm 20\%$), to determine the cluster's *range*: where in the trace we first and last encountered packets leading to estimates consistent with the cluster. When determining the cluster's range, we only consider estimates for which $\xi_i^c \geq \min(\xi_i^{c(50)}, 1.1)$, to ensure that we base the cluster's range on sound estimates (those derived from definite expansion, if present very often; otherwise, those in the upper 50% of the expansions). Without this filtering, a cluster's range can be artificially inflated due to self-clocking and spurious noise, which in turn can mask a bottleneck change.

We next inspect all of the extent-k estimates derived from packets falling within ψ_i's inner range, to determine η_i, the proportion of these estimates consistent with the cluster (within the error bars given by Eqn 14.11). η_i is the cluster's *local proportion*, and reflects how well it captures the behavior within its associated range. A value of η_i near 1 indicates that, over its range, the evidence was very consistent for the given bottleneck estimate, while a lower value indicates the evidence for the bottleneck was diluted by the presence of numerous inconsistent measurements. If $\eta_i < 0.2$, or if $k = 2$ (i.e., we are looking at packet pair estimates) and $\eta_i < 0.3$, we reject the estimate reflected by the cluster as too feeble. This heuristic prunes out the vast majority of estimates that have made it this far in the process, since most of them are due to spurious noise effects. It keeps, however, those that appear to dominate the region over which we found them.

It at first appears that a threshold of 0.2 or 0.3 is considerably too lenient, but in fact it works well in practice, and using a higher threshold runs the risk of failing to detect multi-channel effects, which can split the estimates into two or three different regions. For example, in Figure 14.7 we can readily see that a number of different slopes emerge.

An estimate that has made it this far is promising. The next step is to see whether we have already made essentially the same estimate. We do so by inspecting the previously accepted ("solid") estimates to see whether the new estimate overlaps. If so, we consolidate the two estimates. The details of the consolidation are numerous and tedious.[9] We will not develop them here, except to note that this is the point where a solid estimate with a large error interval ($\rho_i^- \ll \rho_i^+$) can tighten its error interval based on the observation that we have independent evidence for the same estimate at a different extent, and the new evidence has a smaller associated error (due to the higher extent). This is also the point where we determine whether to increase the *maximum extent* associated with an estimate. Doing so is important when hunting for multi-channel bottleneck links, as these should exhibit one bandwidth estimate with a maximum extent exactly equal to the number of parallel channels.

If we do not consolidate a new estimate with any previous solid ones, then we add it to the set of solid estimates.

[9] And can be gleaned from the `tcpanaly` source code.

Forming the final estimates

After executing the process outlined in the previous two subsections, we have produced Υ, a set of "solid" estimates. It then remains to further analyze Υ to determine whether the estimates indicate the presence of a multi-channel link or a bottleneck change. Note that in the process we may additionally merge some of the estimates; we have not yet constructed the set of "final" estimates!

If Υ is empty, then we failed to produce any solid bandwidth estimates. This is rare but occasionally happens, for one of the following reasons:

1. so many packet losses that too few groups arrived at the receiver to form a reliable estimate;

2. so many retransmission events that the connection never opened its congestion window sufficiently to produce a viable stream of packet pairs;

3. such a small receiver window that the connection could never produce a viable stream of packet pairs; or,

4. the trace of the connection was so truncated that it did not include enough packet arrivals (§ 10.3.4).

In $\mathcal{N}_1$, we encountered 37 failures; in $\mathcal{N}_2$, only 1, presumably because the bigger windows used in $\mathcal{N}_2$ (§ 9.3) gave more opportunity of observing a packet group spaced out by the bottleneck link. Interestingly, no estimation failed on account of too many out-of-order packet deliveries. Even those with 25% of the arrivals occurring out of order provided enough in-order arrivals to form a bottleneck estimate.

Assuming Υ is not empty, then if it includes more than one solid estimate, we compare the different estimates as follows. First, we define the *base estimate*, ρ^*, as the first solid estimate we produced. No other estimate was formed using a smaller extent than ρ^*, since we generated estimates in order of increasing extent.

If ρ^* was formed using an extent of $k = 2$, and if Υ includes additional estimates that were only observed for $k = 2$ (i.e., for higher extents we never found a compatible estimate with which to consolidate them), then we assess whether these estimates are "weak." An estimate is weak if it is low compared to ρ^*; the overall proportion of the trace in accordance with the estimate is small; and the estimate's expansions $\xi_i^{c(50)}$ and $\xi_i^{c(95)}$ are low. If these all hold, then the estimate fits the profile of a spurious bandwidth peak (due, for example, to the relatively slow pace at which duplicate acks clock out new packets during "fast recovery", per § 9.2.7), and we discard the estimate.

We now can (at last!) proceed to producing a set of final bandwidth estimates. We begin with the base estimate, ρ^*. We next inspect the other surviving estimates as follows. For each estimate, we test to see whether its range overlaps any of the final estimates. If so, then we check whether the two estimates might reflect a two-channel bottleneck link, which requires:

1. One of the estimates must have a maximum extent of $k = 2$ and the other must have a minimum extent of $k \geq 3$. Call these E_2 and E_3. This requirement splits the estimates into one that reflects the downstream bottleneck, which is only observed for packet pairs ($k = 2$, since for $k > 2$ the effect cannot be observed for a two-channel bottleneck), and the other that reflects the true link bandwidth (which can only be observed for $k > 2$, since $k = 2$ is obscured by the multi-channel effect).

2. E_3 must span at least as much of the trace as E_2. It may span more due to phase effects, as illustrated in Figure 14.7.

3. Unless E_3 spans almost the entire trace, we require that:

$$\xi_3^{c(95)} \geq \min(\frac{3}{4}\xi_2^{c(95)}, 2).$$

   This requirement assures that E_3 was at least occasionally observed for a considerable expansion factor, or, if not, then neither was E_2. The goal here is to not be fooled by an E_3 that was only generated by self-clocking (i.e., no opportunity to observe a higher bandwidth for an extent $k > 2$).

4. The bandwidth estimate corresponding to E_3 must be at least a factor of 1.5 different than that from E_2, to avoid confusing a single very broad peak with two distinct peaks.

If the two estimates meet these requirements, then we classify the trace as exhibiting a multi-channel bottleneck link.

We originally performed the same analysis for (E_3, E_4), that is, for overlapping estimates, one with extent $k = 3$ and one with $k \geq 4$. A three-channel bottleneck would produce estimates for both. We did not find any traces that plausibly exhibited three-channel bottleneck links, though, and did endure a number of false findings, so we omit three-channel analysis from PBM. If we have the opportunity in the future to obtain traces from paths with known three-channel bottlenecks, then we presume we could devise a refinement to the present methodology that would reliably detect their presence.

If two estimates overlap but fail the above test for a multi-channel bottleneck, and if either has both a higher bandwidth estimate and accords with twice as many measurements as the other, then we discard the weaker estimate and use the stronger in its place.

If they overlap but neither dominates, then if one has a minimum extent larger than the other's maximum extent, and larger than $k = 3$ (to avoid erroneously discarding multi-channel estimates), then we discard it as almost certainly reflecting spurious measurements.

If two estimates overlap and none of the three procedures above resolve the conflict, then PBM reports that it has found conflicting estimates. This never happened when analyzing $\mathcal{N}_1$. For $\mathcal{N}_2$, we found only 10 instances. 7 involve `lbli`, which frequently exhibits both a bottleneck change and a multi-channel bottleneck, per Figures 14.4 and 14.5. The other three all exhibit a great deal of delay variation, leading to the conflicting estimates.

If the newly considered estimate does not overlap, then, after some final sanity checks to screen out spurious measurements (which can otherwise remain undetected, if they happen to occur at the very beginning or end of the trace, and thus do not overlap with the main estimate), we add it to the collection of final estimates. At this point, we conclude that the trace exhibits a bottleneck change.

Completing the above steps results in one or more final estimates. For each final estimate ρ_B, we then associate bounds:

$$\rho_B^- < \rho_B < \rho_B^+, \tag{14.12}$$

where ρ_B^- and ρ_B^+ reflect Eqn 14.11, i.e., the smallest and largest estimates within $\pm 20\%$ of ρ_B, or the bounds on ρ_B itself due to limited clock resolution (§ 14.4.2), if larger. In the latter case, we term the estimate as *clock-limited*.

Results of estimation	$\mathcal{N}_1$ #	$\mathcal{N}_1$ %	$\mathcal{N}_2$ #	$\mathcal{N}_2$ %
Single bottleneck	2,018	90%	14,483	94%
Estimate failure	37	1.7%	1	—
Broken estimate	46	2.1%	72	0.05%
Ambiguous estimate:	139	6.2%	779	5.1%
change	94	4.2%	594	3.9%
multi-channel	74	3.3%	506	3.3%
conflicting	0	0.0%	11	0.07%
Total trace pairs	2,240	100%	15,335	100%

Table XVIII: Types of results of bottleneck estimation for $\mathcal{N}_1$ and $\mathcal{N}_2$

Results of estimation	$\mathcal{N}_1$ #	$\mathcal{N}_1$ %	$\mathcal{N}_2$ #	$\mathcal{N}_2$ %
Single bottleneck	1,929	95%	14,134	98%
Estimate failure	37	1.8%	1	—
Broken estimate	19	0.9%	61	0.04%
Ambiguous estimate:	48	2.3%	204	1.4%
change	7	0.34%	67	0.47%
multi-channel	41	2.0%	135	0.9%
conflicting	0	0.0%	3	0.02%
Total trace pairs	2,033	100%	14,400	100%

Table XIX: Types of results after eliminating trace pairs with `lbli`

14.7 Analysis of bottleneck bandwidths in the Internet

We applied the bottleneck estimation algorithms developed in § 14.5 and § 14.6 to the trace pairs in $\mathcal{N}_1$ and $\mathcal{N}_2$ for which the clock analysis described in Chapter 12 did not uncover any uncorrectable problems. These comprised a total of 2,240 and 15,335 trace pairs, respectively. Table XVIII summarizes the types of results we obtained. "Single bottleneck" refers to traces for which we found solid evidence for a single, well-defined bottleneck bandwidth. An "estimate failure" occurs when PBM is unable to find any persuasive estimate peaks (§ 14.6.2). "Broken estimate" summarizes traces for which PBM yielded a single uncontested estimate, but subsequent queueing analysis found counter-evidence indicating the estimate was inaccurate. (We describe this self-consistency test in § 16.2.6.) "Ambiguous estimate" means that the trace pair did not exhibit a single, well-defined bottleneck: it included either evidence of a bottleneck change, or a multi-channel bottleneck link, or both; or it had conflicting estimates, already discussed in § 14.6.2.

The ambiguous estimates were clearly dominated by `lbli`, no doubt because its ISDN link routinely exhibited both bottleneck changes and multi-channel effects (since when it activates the second ISDN channel, the bandwidth doubles and a parallel path arises). Table XIX summarizes the types of results after removing all trace pairs with `lbli` as sender or receiver. We see that PBM almost always finds a single bottleneck. The results also exhibit a general trend between $\mathcal{N}_1$ and $\mathcal{N}_2$ towards fewer problematic estimates. We suspect the difference is due to two effects: the lower prevalence of out-of-order delivery in $\mathcal{N}_2$ compared to $\mathcal{N}_1$, and the use of bigger windows in $\mathcal{N}_2$ (§ 9.3), which provides more opportunity for generating tightly-spaced packet pairs and packet bunches.

In the remainder of this section, we analyze each of the different types of estimated bottlenecks.

14.7.1 Single bottlenecks

Far and away the most common result of applying PBM to our traces was that we obtained a single estimated bottleneck bandwidth. Unlike [CC96a], we do not *a priori* know the bottleneck bandwidths for many of the paths in our study. We thus must fall back on self-consistency checks in order to gauge the accuracy of PBM. Figures 14.10 and 14.11 show histograms of the estimates formed for $\mathcal{N}_1$ and $\mathcal{N}_2$, where the histogram binning is done using the logarithms of the estimates, so the ratio of the sizes of adjacent bins remains constant through the plot.

There are a number of readily apparent peaks. In $\mathcal{N}_1$, we find the strongest at about 170 Kbyte/sec, and another strong one at 6.5 Kbyte/sec. Secondary peaks occur at about 100, 330, 80, and 50 Kbyte/sec, with lesser peaks at 30 Kbyte/sec, 500 Kbyte/sec, and at a bit over 1 Mbyte/sec. The pattern in $\mathcal{N}_2$ is a bit different. The 170 Kbyte/sec peak clearly dominates, and the 6.5 Kbyte/sec peak has shifted over to about 7.5 Kbyte/sec. The peaks between 50 and 100 Kbyte/sec are no longer much apparent, and the 330 Kbyte/sec peak has diminished while the 30, 500 and 1 Mbyte/sec peaks have grown. Finally, a new, somewhat broad peak has emerged at 13–14 Kbyte/sec.

We calibrate these peaks using a combination of external knowledge about popular link speeds, and by inspecting which sites tend to predominate for a given peak. Several common slower link speeds are 56, 64, 128, and 256 Kbit/sec. Common faster links are 1.544 Mbit/sec ("T1"—primarily used in North America), 2.048 Mbit/sec ("E1"—used outside North America), and 10 Mbit/sec (Ethernet). Certainly faster links are in use in the Internet, but we omit discussion of them since none of the bottlenecks in our study exceeded 10 Mbit/sec; we note, however, that it is the use of faster wide-area links that enables a local-area limit such as Ethernet to wind up as a connection's bottleneck.

The link speeds discussed above reflect the *raw* capacity of the links. Not all of this capacity is available to carry user data. Often a portion of the capacity is permanently set aside for framing and signaling. Furthermore, transmitting a packet of user data using TCP requires encapsulating the data in link-layer, IP, and TCP headers. The size of the link-layer header varies with the link technology. The IP and TCP headers nominally require at least 40 bytes, more if IP or TCP options are used. Use of IP options for TCP connections is rare, and none of the connections in our study did so. TCP options are common, especially in the initial SYN packets. Thus, we might take 40 bytes as a solid lower bound on the TCP/IP header overhead. An exception, however, is links utilizing *header compression* (§ 13.3), which, depending on the homogeneity of the traffic traversing the link, can greatly reduce the bytes required to transmit the headers. Header compression works by leveraging off of the large degree of redundancy between the headers of a connection's successive packets. For example, under optimal conditions, CSLIP compression can reduce the 40 bytes to

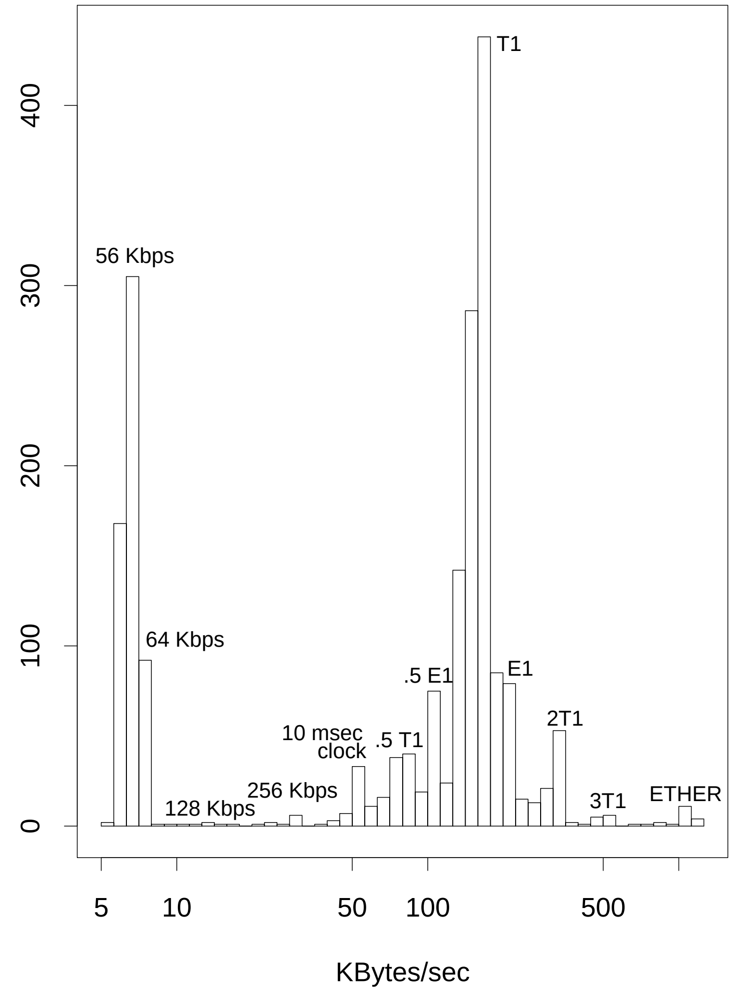

Figure 14.10: Histogram of different single-bottleneck estimates for $\mathcal{N}_1$

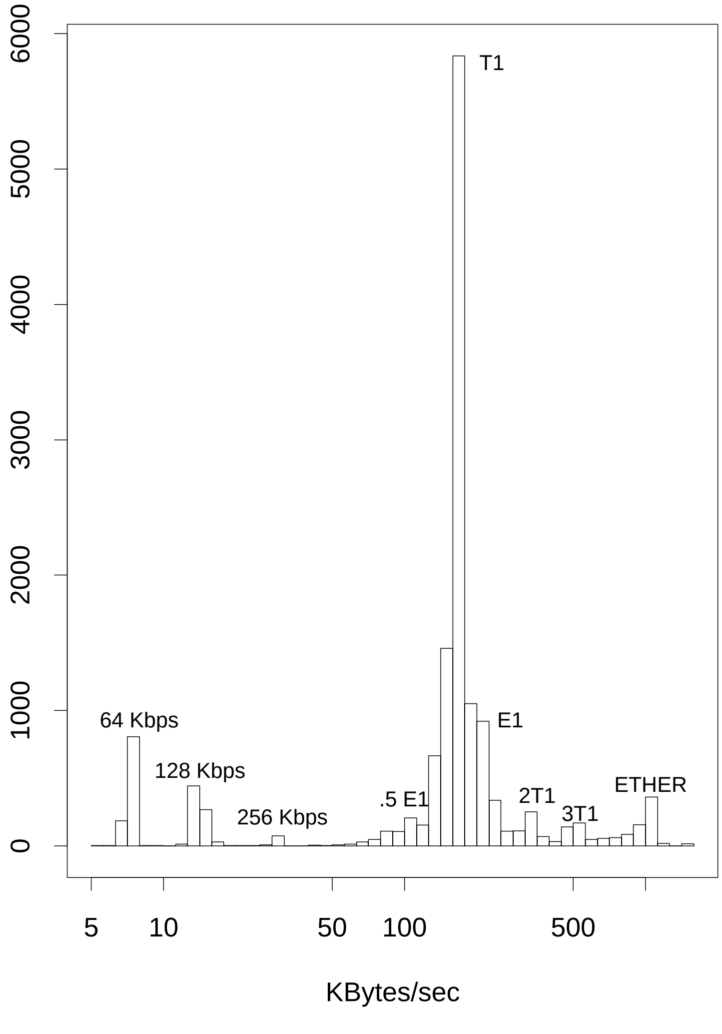

Figure 14.11: Histogram of different single-bottleneck estimates for $\mathcal{N}_2$

Raw rate (ρ_R)	User data rate (ρ_U)	Notes
56 Kbit/sec	$\approx$ 6.2 Kbyte/sec	
64 Kbit/sec	$\approx$ 7.1 Kbyte/sec	
128 Kbit/sec	$\approx$ 14.2 Kbyte/sec	
256 Kbit/sec	$\approx$ 28.4 Kbyte/sec	
1.544 Mbit/sec	$\approx$ 171 Kbyte/sec	T1
2.048 Mbit/sec	$\approx$ 227 Kbyte/sec	E1
10 Mbit/sec	$\approx$ 1.1 Mbyte/sec	Ethernet

Table XX: Raw and user-data rates of different common links

5 bytes. Finally, some links use *data compression* techniques to reduce the number of bytes required to transmit the user data. We presume these techniques did not affect the connections in our study because NPD sends a pseudo-random sequence of bytes (to avoid just this effect).

Given these sundry considerations, we do not hope to nail down a single figure for each link technology reflecting the user data rate it delivers. Instead, we make "ballpark" estimates, as follows. For high-speed links, the framing and signaling overhead consumes about 4.5% of the raw bandwidth [Ta96]. We compromise on the issues of header compression versus additional bytes required for link-layer headers and TCP options by assuming 40 bytes of overhead for each TCP/IP packet. Finally, we assume that a "typical" data packet carries 512 bytes of user data. This is the most commonly observed value in our traces, though certainly not the only one. Given these assumptions, the user data rate available from a link with a raw rate of ρ_R is:

$$\begin{aligned}\rho_U &\approx (.955)\left(\frac{512}{512+40}\right)\rho_R \\ &\approx .886\rho_R.\end{aligned}$$

Table XX summarizes the corresponding estimated user-data rates for the common raw link rates discussed above. From the table, it is clear that the strong 170 Kbyte/sec peak in Figure 14.10 and Figure 14.11 reflect T1 bottlenecks. Likewise, the 6.5 Kbyte/sec peak reflects 56 Kbit/sec links, and may be slightly higher than the estimate in the Table due to the likely use of header compression. Its shift to 7.5 Kbyte/sec reflects upgrading of 56 Kbit/sec links to 64 Kbit/sec. The 13–14 Kbyte/sec peak reflects 128 Kbit/sec links and the 30 Kbyte/sec peak, 256 Kbit/sec. The 1 Mbyte/sec peaks are clearly due to Ethernet bottlenecks.

These identifications still leave us with some unexplained peaks from the bottleneck estimates. We speculate that the 330 Kbyte/sec peak reflects use of two T1 circuits in parallel, 500 Kbyte/sec reflects three T1 circuits (not half an Ethernet, since there is no easy way to subdivide an Ethernet's bandwidth), and 80 Kbyte/sec comes from use of half of a T1.

We then have only two unexplained peaks remaining: 50 and 100 Kbyte/sec. The 50 Kbyte/sec peak is only prominent in $\mathcal{N}_1$. We find that this peak in fact reflects vagueness due to limited clock resolution: in § 14.4.2 we showed that, for packet pair, the fastest bandwidth a 10 msec clock can yield for 512 byte packets is 51.2 Kbyte/sec. Thus, the 50 Kbyte/sec peak is a measurement artifact, though it also indicates the presence of connections for which PBM was unable to tighten its bottleneck estimate using higher extents (which would reduce uncertainties due to clock resolution), presumably because the connection rarely had more than two packets delivered to the receiver at the bottleneck rate, due to extensive queueing noise.

The 100 Kbyte/sec peak, on the other hand, most likely is due to splitting a single E1 circuit in half. Indeed, we find non-North American sites predominating these connections, as well exhibiting peaks at 200–220 Kbyte/sec, above the T1 rate and just a bit below E1. This peak is, however, absent from North American connections. (See also Figure 14.12 and accompanying discussion, below.)

In summary, we believe we can offer plausible explanations for all of the peaks. Passing this self-consistency test in turn argues that PBM is indeed detecting true bottleneck bandwidths. We next turn to variation in bottleneck rates. We would expect to observe strong site-specific variations in bottleneck rates, since some of the limits arise directly from the speed of the link connecting the site to the rest of the Internet.

Figure 14.12 clearly shows this effect. The figure shows a "box plot" for $\log_{10}$ of the bottleneck estimates for each of the $\mathcal{N}_2$ receiving sites. In these plots, we draw a box spanning the inner two quartiles (that is, from 25% to 75%). A dot shows the median and the "whiskers" extend out to 1.5 times the inter-quartile range. The plot shows any values beyond the whiskers as individual points. The horizontal line marks 171 Kbyte/sec, the popular T1 user data rate (Table XX).

The plot clearly shows considerable site-to-site variation. While all sites reflect some 64 and 128 Kbit/sec bottlenecks, we quickly see that `austr2` has virtually only 128 Kbit/sec bottlenecks, indicating it almost certainly uses a link with that rate for its Internet connection. (`austr`, on the other hand, has at least E1 connectivity.) `lbli` generally does not have a single bottleneck above 64 Kbit/sec (it often has a bottleneck *change* that includes 128 Kbit/sec, but in this section we only consider traces exhibiting a single, unchanged bottleneck). The `lbli` estimates tend to be quite sharply defined. Of those larger than 7 Kbyte/sec, 96% lay within a 30 byte/sec range centered about 7,791 byte/sec. The other site with a narrow bottleneck bandwidth region is `oce`, which has a 64 Kbit/sec link to the Internet, as clearly evidenced by the plot, except for a cluster of outliers at 17 Kbyte/sec. All of the outliers were localized to a 1 day period, perhaps a time when `oce` momentarily enjoyed faster connectivity.

In the main, the plot exhibits a large number of sites with median bottlenecks at T1 rate. A few have slightly higher median bottlenecks, and these tend to be non-North American sites, consistent with E1 links. Two sites have occasional values just below $\log_{10} = 1.5$, corresponding to 256 Kbit/sec links. These sites are `ucl` and `ukc`, both located in Britain, so we suspect these bottlenecks reflect a British circuit or set of circuits. Some sites also exhibit a fair number of bottlenecks exceeding 1 Mbyte/sec: `bnl`, `lbl`, `mid`, `near`, `panix`, and `wustl` (as well as, more rarely, a number of others), indicating these all enjoyed Ethernet-limited Internet connectivity.

We next investigate the stability of bottleneck bandwidth over time. We confine this investigation to $\mathcal{N}_2$, since it includes many more connections between the same sender/receiver pairs, spaced over a large range of time. We begin by constructing for each sender/receiver pair two sequences, $\Delta\mathcal{T}_{s,r}$ and $\mathcal{R}_{s,r}$, giving the difference in time between the beginning of successive connections from the sender to the receiver, and the ratio of the estimated bottleneck rate for the second of the connections to that of the first.

As noted in § 9.3, we varied the mean time between successive connections between sender/receiver pairs, and, in addition, our methodology would sometimes include "revisiting" a pair at a later date. Accordingly, $\Delta\mathcal{T}_{s,r}$ exhibits considerable range: its median is 8 minutes, its 90th

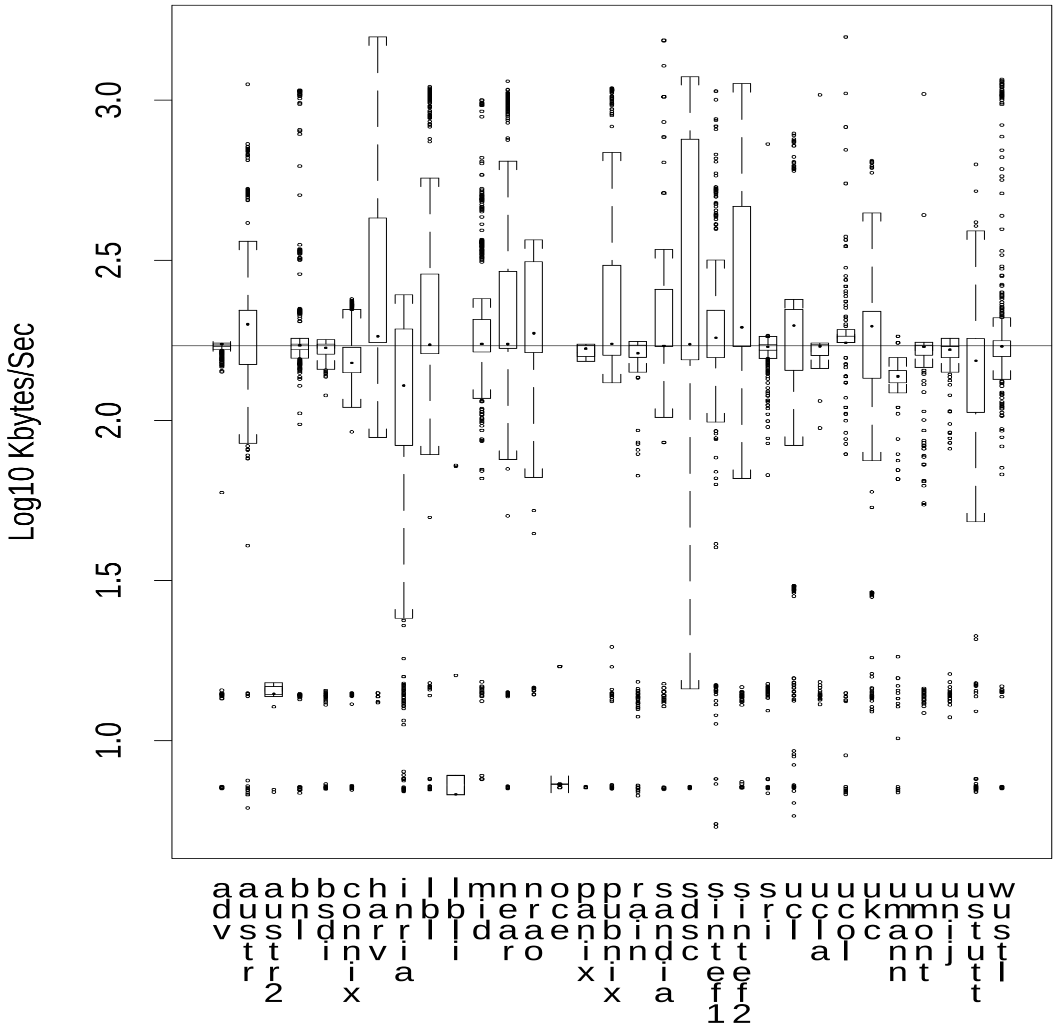

Figure 14.12: Box plots of bottlenecks for different $\mathcal{N}_2$ receiving sites

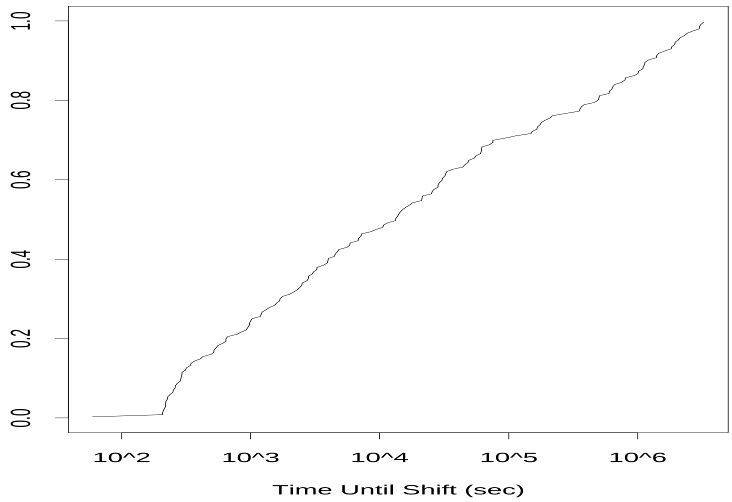

Figure 14.13: Time until a 20% shift in bottleneck bandwidth, if ever observed

percentile is 104 minutes, but its mean is about 7 hours, due to revisiting.

The bottleneck ratio $\mathcal{R}_{s,r}$ overall shows little variation. Its median is exactly 1.0. Evaluating $\mathcal{R}_{s,r}$'s distribution directly can be misleading, because it will tend to be < 1 as often as > 1, depending on whether the second of a pair of estimates was lower or higher than the first. What is more relevant is the "magnitude" of the ratio between successive estimates, which we define as:

$$|\mathcal{R}|_{s,r} \equiv \exp[|\log \mathcal{R}_{s,r}|],$$

that is, the ratio of the larger of the two estimates to the smaller. The median of $|\mathcal{R}|_{s,r}$ is 1.0175, indicating that 50% of the successive estimates differ by less than 1.75% from the previous estimate. We find that 80% of the successive estimates differ by less than 10%, and 98% differ by less than a factor of two.

We consider two different assessments of the stability of the bottleneck rate over time. First, we examine the correlation between $|\mathcal{R}|_{s,r}$ and $\Delta\mathcal{T}_{s,r}$. If bottlenecks fluctuate significantly over time, then we would expect the magnitude of the ratio to correlate with the time separating the connections. If fluctuations are mainly due to measurement imprecision, then the two should be uncorrelated.

For $\Delta\mathcal{T}_{s,r} < 1$ hour (85% of the successive measurements), we find very slight positive correlation between $|\mathcal{R}|_{s,r}$ and $\Delta\mathcal{T}_{s,r}$, with a coefficient of correlation equal to 0.03. We obtain a coefficient of about this size regardless of whether we first apply logarithmic transformations to either or both of $|\mathcal{R}|_{s,r}$ and $\Delta\mathcal{T}_{s,r}$ in an attempt to curb the influence of outliers. For $\Delta\mathcal{T}_{s,r} \geq 1$ hour, the coefficient of correlation rises to about 0.09. This is still not strong positive correlation, and indicates that bottleneck bandwidth is quite stable over periods of time ranging from minutes to days (the mean of $\Delta\mathcal{T}_{s,r}$, conditioned on it exceeding 1 hour, is 52 hours).

We can also assess stability in terms of the time required to observe a significant change. To do so, for each sender/receiver pair we take the first bottleneck estimate as a "base measurement" and then look to see when we find two consecutive later estimates that both differ from the base measurement by more than 20%, and that both agree in terms of the direction of the change (20% larger or smaller). We look for consecutive estimates to weed out spurious changes due to isolated measurement errors. We find that only about a fifth of the sender/receiver pairs *ever* exhibited a shift of this magnitude. Furthermore, the amount of time between the first measurement and the first of the pair constituting the shift has a striking distribution, shown in Figure 14.13. The distribution appears almost uniform, except that the x-axis is logarithmically scaled, indicating that shifts in bottleneck bandwidth occur over a wide range of time scales. This finding qualitatively matches that in Chapter 7 that the time over which different routes persist varies over a wide range of scales. We would expect general agreement since one obvious mechanism for a shift in bottleneck bandwidth is a routing change, though some routing changes will not alter the bottleneck.

The last property of bottleneck bandwidth we study in this section is its symmetry: how often is the bottleneck from host A to host B the same as that from B to A? We know from Chapter 8 that Internet routes often exhibit major routing asymmetries, with the route from A to B differing from the reverse of B to A by at least one city about 50% of the time in $\mathcal{N}_2$. It is quite possible that these asymmetries will also lead to bottleneck asymmetries, an important consideration because sender-based "echo" bottleneck measurement techniques such as those explored in [Bo93, CC96a] will observe the *minimum* bottleneck of the two directions.

Figure 14.14 shows a scatter plot of the *median* bottleneck rate estimated in the two directions for the hosts in our study. The plot uses logarithmic scaling on both axes to accommodate the wide range of bottleneck rates. For each pair of hosts A and B for which we had successful measurements in both directions, we plot a point corresponding to A's median estimate on the x-axis, and B's median estimate on the y-axis. The solid diagonal line has slope one and offset zero. Points falling on it have equal estimates in the two directions. The dashed diagonal lines mark the extent of estimates 20% above or below the solid line. About 45% of the points fall within $\pm 5\%$ of equality, and 80% within $\pm 20\%$ (i.e., within the dashed lines). But about 20% of the estimates differ by considerably more. For example, some paths are T1 limited in one direction but Ethernet limited in the other, a major difference.

Of the considerably different estimates, the median ratio between the two estimates is 40% and the mean is 65%. In light of these variations, we see that sender-based bottleneck measurement provides a good rough estimate, but will sometimes yield quite inaccurate results.

14.7.2 Bottleneck changes

We now turn to analyzing how frequently the bottleneck bandwidth changes during a single TCP connection. From the results in the previous section, we expect such changes to occur only rarely, and indeed this is the case. If we disregard `lbli`, which, as noted in § 14.4.3, frequently exhibits a bottleneck change due to the activation of its second ISDN channel, then, as shown in Table XIX, only about 1 connection in 250 (0.4%) exhibited a bottleneck change. The changes are all large, by definition (since we merge bottleneck estimates with minor differences), with the median ratio between the two bottlenecks in the range 3-6.

Figure 14.15 illustrates one of the smaller changes. At about $T = 2.3$, the bottleneck decreases from an estimated 168 Kbyte/sec to an estimated 99 Kbyte/sec. The effect here is not self-clocking, as the one-way delays of the packets show a considerable increase at $T = 2.3$ as well. Contrast this behavior with that at about $T = 2.1$, where we see a momentary decrease. In this

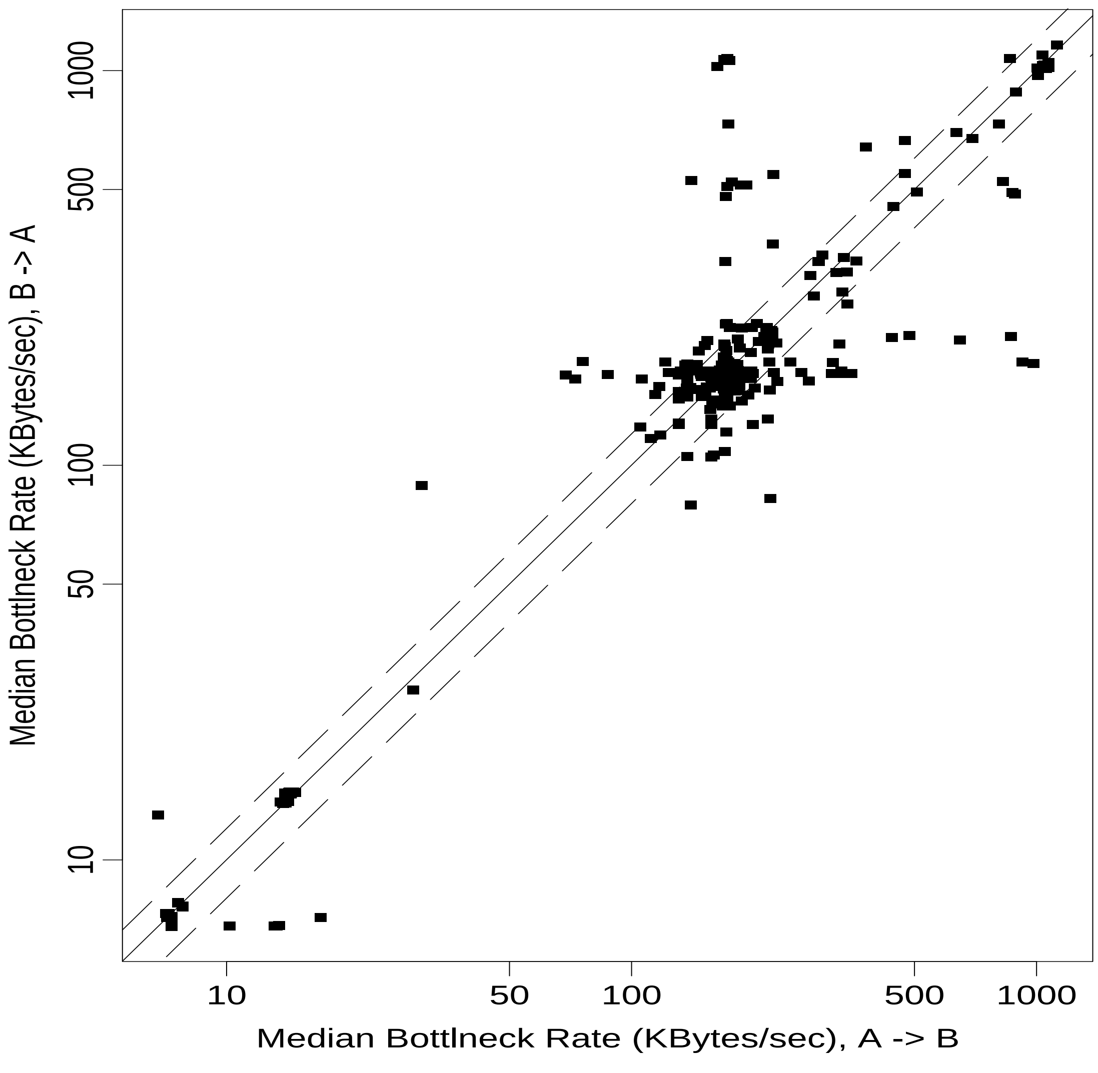

Figure 14.14: Symmetry of median bottleneck rate

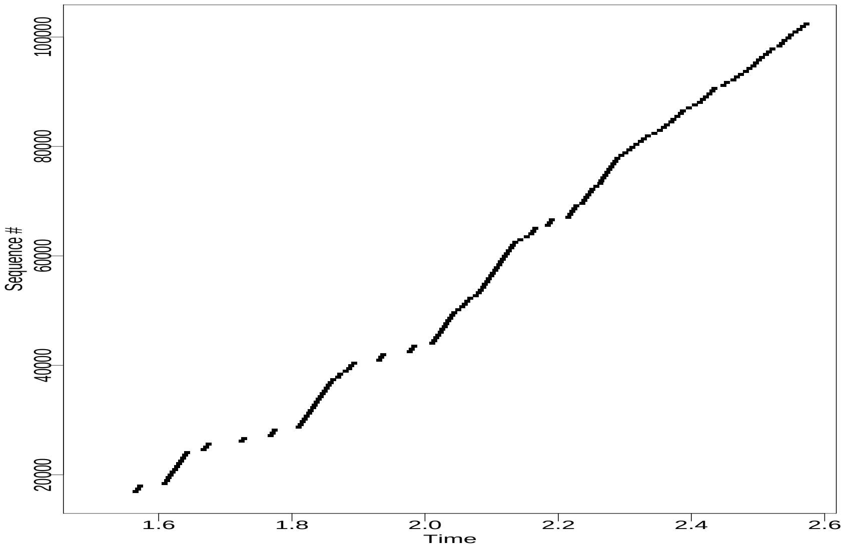

Figure 14.15: Sequence plot reflecting halving of bottleneck rate

case, the slow-down is not accompanied by an increase in transit time, and is instead a self-clocking "echo" of the slow-down at $T = 1.9$.

Since 99 Kbyte/sec is not a particularly compelling link rate vis-a-vis Table XX, we might consider that the bottleneck rate did not in fact change, but instead at $T = 2.3$ a *constant-rate* source of competing traffic began arriving at the bottleneck link, diluting the bandwidth available to our connection and hence widening the spacing between arriving data packets. This may well be the case. We note, however, that *effectively* this situation is the same as a change in the bottleneck rate: if the additional traffic is indeed constant rate, and not adaptive to the presence of our traffic, then we might as well have suffered a reduction in the basic bottleneck link rate, since that is exactly the effect our connection will experience. So we argue that, in this case, we *want* to regard the change as due to a bottleneck shift, rather than due to congestion.

A few of the bottleneck "changes" appear spurious, however. These apparently stem from connections with sufficient delay noise to completely wash out the true bottleneck spacing, and which coincidentally produce a common set of packet spacings that lead to a false bottleneck peak. Most changes, however, appear genuine. In both datasets, about 15% of the changes differ by about a factor of two, suggesting that a link had been split or two sub-links merged following a failure or repair at the physical layer.

14.7.3 Multi-channel bottlenecks

The final type of bottleneck we analyze are those exhibiting the *multi-channel* effect discussed in § 14.4.4. As shown in Table XIX, except for connections involving `lbli`, known to have a 2-channel bottleneck link, we found few multi-channel bottlenecks. However, after excluding `lbli`, we still found a tendency for a few sites to predominate among those exhibiting multi-channel bottlenecks: `inria`, `ukc`, and `ustutt`, in both datasets, and `wustl` in $\mathcal{N}_1$. The presence of this last

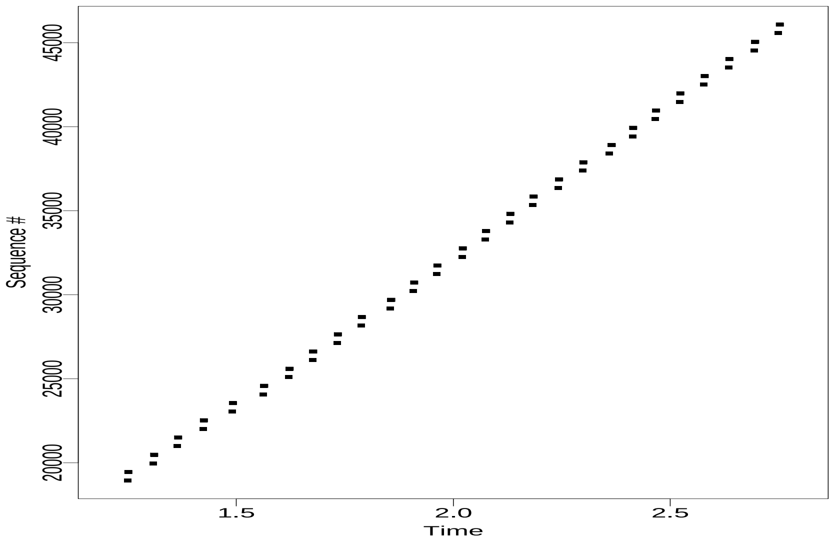

Figure 14.16: Excerpt from a trace exhibiting a false "multi-channel" bottleneck

site in the list is not surprising, since we know that due to route "flutter" many of its connections used two very different paths to each remote site (§ 6.6).

However, we cannot confidently claim that any of the non-`lbli` purported multi-channel bottlenecks are in fact due to multi-channel links, since we find that very often the trace in question is plagued with delay noise, and lacks the compelling pattern shown in Figure 14.6. The ratios between the nominal bandwidths of extent $k = 2$ and $k \geq 3$ bunches also generally tend to be < 2, which from our experience often instead indicates excessive measurement noise smearing out the bottleneck signature.

Even when the measurements appear quite clean, we must exercise caution. Figure 14.16 shows a portion of an $\mathcal{N}_1$ trace from `ukc` to `ucl` with a pattern very similar to that in Figure 14.6. Most of the trace looks exactly like the pattern shown. PBM analyzes this trace as exhibiting a multi-channel bottleneck with an upper rate of 477 Kbyte/sec and a slower rate of 18 Kbyte/sec. However, detailed analysis of the trace reveals a few packet bunches with $k \geq 3$ that arrived spaced at 477 Kbyte/sec, evidence that either the bunches were *compressed* (§ 16.3.2) subsequent to the multi-channel bottleneck, or the bottleneck is in fact not multi-channel. Further analysis reveals that the sending TCP was limited by a sender-window (§ 11.3.2), and that the ack-every-other policy used by the receiver led to almost perfect self-clocking of flights of two packets arriving at the true bottleneck rate, followed by a self-clocking lull, followed by another flight of two, and so on. While PBM includes heuristics based on $\widetilde{\xi}_{s,r}$ (Eqn 14.6) that attempt to discard traces like these as multi-channel candidates, this one passed the heuristic due to some unfortuitous packet bunch expansion early in the trace. Had the sending TCP not been window-limited, it would have continued expanding the window as the self-clocking set in, leading to numerous flights of $k \geq 3$ packets all arriving at the faster link rate, and PBM would have determined that in fact the link was not multi-channel.

In summary, we are not able to make quantitative statements about multi-channel bottle-

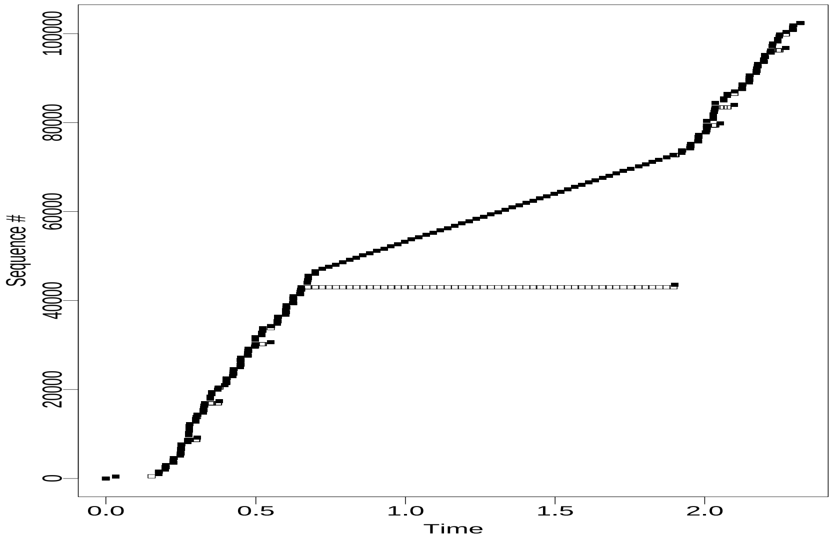

Figure 14.17: Self-clocking TCP "fast recovery"

necks in the Internet, except that in any case they are quite rare; that at least one link technology (ISDN) definitely exhibits them; and that some sites exhibit either true such links, or at least noise patterns resembling the multi-channel signature.

14.7.4 Estimation errors due to TCP behavior

In the previous section, we noted how TCP "self-clocking" can lead to a packet arrival pattern that matches that expected for a multi-channel bottleneck link quite closely, even though the bottleneck link is not in fact multi-channel. In this section we briefly illustrate another form of TCP behavior that can lead to false bottleneck estimates. Figure 14.17 shows a sequence plot of a connection clearly dominated by an unusually smooth and slow middle period.

What has occurred is that a single packet was dropped at about $T = 0.7$. Enough additional packets were in flight that 4 duplicate acks came back to the sender. The first 3 sufficed to trigger "fast retransmit" (§ 9.2.7), and the congestion window was such that the 4th led to the transmission of an additional packet carrying new data via the "fast recovery" mechanism (§ 9.2.7). However, the first packet retransmitted via fast retransmit was also dropped, while the fast-recovery packet carrying new data arrived successfully. This meant that the TCP receiver still had a sequence hole reflecting the original lost packet, so it sent another dup ack. The arrival of that duplicate then liberated another packet via fast recovery, and the cycle repeated 50 more times, until the original lost packet was finally retransmitted again, this time due to a timeout. Its retransmission filled the sequence hole and the connection proceeded normally from that point on.

Since the connection had an RTT of about 22 msec and only one fast recovery packet or dup ack was in flight at any given time, during the retransmission epoch the connection transmitted using “stop-and-go,” with an effective rate of:

$$\frac{512 \text{ bytes}}{0.22 \text{ sec}} = 23 \text{ Kbyte/sec}.$$

PBM finds this peak rather than the true bottleneck of 1 Mbyte/sec, because the true bottleneck is obscured by the receiver's 1 msec clock resolution.

The TCP dynamics shown in the figure are quite striking. We note, however, that use of the SACK selective-acknowledgement option [MMFR96], now in the TCP standardization pipeline, will give the sender enough information to avoid situations like this one. We also note that, while this sort of TCP behavior is not exceptionally rare, this was the only such trace that we know PBM to have misanalyzed.

14.8 Efficacy of other estimation techniques

We finish with a look at how other, simpler bottleneck estimation techniques perform compared to PBM. Since PBM is quite complex, it would be useful to know if we can use a simpler method to get comparably sound results. In this context, the development of PBM serves as a way to calibrate the other methods. We confine our analysis to those traces for which PBM found a single bottleneck, as the other techniques all assume such a situation to begin with.

We further associate error bars with each PBM estimate. These either span the range of “consistent” estimates we found, where estimates are considered consistent if they lie within $\pm 20\%$ of the main PBM estimate (§ 14.6.2); or, if larger, the error bars reflect the inherent uncertainty in the PBM estimate due to limited clock resolution (§ 14.4.2). If another technique produces an estimate lying within the error bars, then we consider it as performing as well as PBM, otherwise not.

14.8.1 Efficacy of PR

In this section we evaluate the “conservative” and “optimistic” peak-rate (PR) estimators developed in § 14.5. These estimators were developed primarily as calibration checks for PBM, and we noted in their discussion that they will tend to underestimate the true bottleneck rate. Still, since they are simple to compute, it behooves us to evaluate their efficacy. We only evaluate them for $\mathcal{N}_2$, since they rely on the sending TCP enjoying a large enough window that it could “fill the pipe” and send at a rate equal to or exceeding the bottleneck rate (§ 9.3).

As we might expect, we find that the conservative estimate $\widehat{\text{PR}}^c$ given by Eqn 14.7 often underestimates the bottleneck: 60% of the time in $\mathcal{N}_2$, $\widehat{\text{PR}}^c$ was below the lower bound given by PBM; 39% of the time, it was in agreement; and 2% of the time it exceeded the upper bound, due to packet compression effects (§ 16.3).

Unfortunately, the more optimistic estimate $\widehat{\text{PR}}^o$ given by Eqn 14.8 only fares slightly better, underestimating 43% of the time, agreeing 52%, and overestimating 5% of the time.

We conclude that neither peak-rate estimator is trustworthy: they both often underestimate, because connections fail to fill the pipe due to congestion levels high enough to preclude an RTT's worth of access to the full link bandwidth.

14.8.2 Efficacy of RBPP

Receiver-based packet pair (§ 14.3) is equivalent to PBM with the extent limited to $k = 2$. (That is, it uses PBM's clustering algorithm to pick the best $k = 2$ estimate.) Consequently, we would expect it to do quite well in terms of agreeing with PBM, with disagreement potentially arising only due to clock resolution limitations for $k = 2$ (§ 14.4.2); delay noise on very short time scales such that pairs of packets are perturbed and do not yield a clear bandwidth estimate peak, but larger extents do; and multi-channel bottlenecks (not further evaluated in this section), one of the main motivations for PBM in the first place.

We find the RBPP estimate is almost always within $\pm 20\%$ of PBM's, disagreeing in $\mathcal{N}_1$ and $\mathcal{N}_2$ by more only 2-3% of the time. The two estimates are identical about 80% of the time, indicating PBM was usually unable to further hone RBPP's estimate by considering larger extents. Thus, if (1) PBM's general clustering and filtering algorithms are applied to packet pair, (2) we do packet pair estimation at the *receiver*, (3) the receiver benefits from sender timing information, so it can reliably detect out-of-order delivery and lack of bottleneck “expansion,” and (4) we are not concerned with multi-channel effects, then packet pair is a viable and relatively simple means to estimate the bottleneck bandwidth.

14.8.3 Efficacy of SBPP

We finish with an evaluation of one form of *sender*-based packet pair (SBPP). SBPP is of considerable interest because a sender can use it without any cooperation from the receiver. This property makes SBPP greatly appealing for use by TCP in the Internet, because it works with only *partial deployment*. That is, SBPP can enhance a TCP implementation's decision-making for every transfer it makes, even if the receiver is an old, unmodified TCP. We expect SBPP to have difficulties, though, due to noise induced by networking delays experienced by the acks, as well as variations in the TCP receiver's *response delays* in generating the acks themselves (§ 11.6.4).

The bottleneck bandwidth estimators previously studied are both sender-based [Bo93, CC96a]. They differ from how sender-based TCP packet pair would work in that those schemes use “echo” packets. As noted in the discussion of Figure 14.14, Internet paths do not always have symmetric bottlenecks in the forward and reverse directions. Consequently, echo-based techniques will sometimes perforce give erroneous answers for the forward path's bottleneck rate. For TCP's use, however, the “echo” is the acknowledgement of the data packet. Except for connections sending data in both directions simultaneously, which are rare, these echoes are therefore returned in quite small ack packets. Consequently, bottleneck asymmetry will not in general perturb SBPP for TCP. Another significant difference is that, for TCP, usually an echo is only generated for every other data packet (§ 11.6.1). Consequently, the interval between each pair of acks arriving at the sender echoes the difference in time between the arrivals of *two* data packets at the receiver, rather than the arrivals of consecutive data packets. Because of this loss of fine-scaled timing information, TCP SBPP cannot detect the presence of multi-channel links, since doing so requires observing per-packet timing differences. (It will instead see timings corresponding to an extent of $k = 4$, which, for 2-channel and 3-channel links, is in fact the true bottleneck rate.)

To fairly evaluate SBPP, we assume use by the sender of the following considerations for generating “good” bandwidth estimates:

1. The sender always correctly determines how many user data bytes arrived at the receiver

between when it sent the two acks.

2. The sender does not consider pairs of acks if the first ack was for all the outstanding data, as such a pair is guaranteed to have a spurious RTT delay between the first and second ack.

3. The sender never bases an estimate on an ack that is for only a single packet's worth of data (MSS), as these often are delayed acks, and the sender lacks sufficient information to remove the timer-induced additional delay.

4. The sender never bases an estimate on an ack that does not acknowledge new data. This prevents the sender from using inaccurate timing information due to packet loss or reordering.

5. The sender keeps track of the sending times for its data packets, so it can determine the *sender expansion factor* (§ 14.5):
$$\widetilde{\xi}_{s,s} = \frac{\Delta T_a + C_s}{\Delta T_d + C_s},$$
where ΔT_a is the elapsed time between the arrival of successive acks, ΔT_d is the elapsed time between the departure of the first and last data packet being acknowledged, and C_s is the sender's clock resolution.

   The sender rejects an estimate if $\widetilde{\xi}_{s,s} < 0.9$. We use 0.9 instead of 1.0 as a "fudge factor" to account for self-clocking, which sometimes occurs at exactly the bottleneck rate.

The sender also computes "acceptable" estimates, which are those that do not conform to all of the above considerations, but at least conform to the first two. (These estimates will be used if SBPP cannot form enough "good" estimates.)

After collecting "good" and "acceptable" estimates for the entire trace, we then see whether we managed to collect 5 or more "good" estimates. If so, we take their 95th percentile as the bottleneck estimate (allowing for the last 5% to have been corrupted by ack compression, per § 16.3.1). If not, then we take the median of the "acceptable" estimates as our best guess.

We find, unfortunately, that SBPP does not work especially well. In both datasets, the SBPP bottleneck estimate lies within $\pm 20\%$ of the PBM estimate only about 60% of the time. About one third of the estimates are too low, reflecting inaccuracies induced by excessive delays incurred by the acks on their return, with the median amount of underestimation being a factor of two (and the mean, more than a factor of four). The remaining 5–6% are overestimates, reflecting frequent ack compression (§ 16.3.1), with an $\mathcal{N}_1$ median overestimation of 60% and a mean of 175%, though in $\mathcal{N}_2$ these dropped to 45% and 75%.

A final interesting phenomenon in $\mathcal{N}_2$ is that, about 2% of the time, SBPP was unable to form *any* sound estimate. These all entailed connections to receivers that generated only one ack for each entire slow-start "flight" (§ 11.6.1). Since one of the considerations outlined above requires that the first ack of a pair not be an ack for all outstanding data (to avoid introducing a round-trip time lull that has nothing to do with the bottleneck spacing), if the network does not drop any data packets, then such a receiver will *only* generate acks for all outstanding data, so the SBPP algorithm above fails to find any acceptable measurements.

14.8.4 Summary of different bottleneck estimators

In our evaluation of the different bottleneck rate estimators, we found that PBM overall appears quite strong. It produces many bandwidth estimates that accord with known link speeds, and produces few erroneous results, except for a tendency to misdiagnose a multiple-channel bottleneck link in the presence of considerable delay noise.

Using PBM then as our benchmark, we found that the stressful "peak rate" (PR) techniques perform poorly, frequently underestimating the bottleneck, as we surmised they probably would when developing them in § 14.5. They did, however, serve as useful calibration tests when developing PBM, since they pointed up traces for which we needed to investigate why PBM produced an estimate less than that of the conservative PR technique, or greater than that of the optimistic PR technique.

We also found that receiver-based packet pair (RBPP) performs virtually identically to PBM, provided that we observe the requirements outlined in § 14.8.2, and are not concerned with detecting multi-channel bottleneck links. Unfortunately, one requirement for RBPP is sender cooperation in timestamping the packets it sends, so the receiver can detect out-of-order delivery and data packet compression. We have not investigated the degree to which these requirements can be eased, but this would be a natural area for future work.

We unfortunately found that sender-based packet pair (SBPP) does not fare nearly as well as RBPP. Even taking care to use only measurements the sender can deduce should be solid, SBPP suffers from ack arrival timings perturbed by queueing delays and ack compression. As a result, it renders accurate results less than 2/3's of the time.

Thus, receiver-based bottleneck measurement appears to hold intrinsic advantages over sender-based measurement, and fairly simple receiver packet pair techniques, with sender cooperation, gain all of the advantages of the more complex PBM, unless we are concerned with detecting multi-channel bottleneck links.

Finally, a particularly interesting question for future work to address is how *quickly* these techniques can form solid estimates. If we envision a transport connection using an estimate of the bottleneck bandwidth to aid in its transmission decisions, then we would want to form these estimates as early in the connection as possible, particularly since most TCP connections are short-lived and hence have little opportunity to adapt to network conditions they observe [DJCME92, Pa94a].

Chapter 15

Packet Loss

In a packet-switched network that does not provide mechanisms for reserving resources within the network on behalf of a particular packet "flow", loss is inevitable under conditions of load. The Internet is such a network. According to traditional network traffic theory, based on Poisson models that emphasize at most fleeting correlations between packet arrivals, one can generally engineer a packet-switched network to have as low a packet loss rate as desired. Operational experience, however, has been quite contrary and brutal to the Poisson framework [JR86, G90, FL91, DJCME92, PF95], which appears woefully inadequate for accurately predicting actual network behavior. Recent years have seen the rise of *self-similar* traffic models, in which correlations are extremely long-lived and have a fractal structure, leading to "burstiness on all time scales" [LTWW94]. Fractal models predict that packet loss is extremely hard to avoid, due to the great burstiness of network traffic, and, more generally, due to the lack of a single *burst time scale* for which one can then engineer the network to accommodate.

We should note that packet loss is not unequivocally a problem. TCP makes splendid use of packet loss as an *implicit* signal that the network is under stress and the TCP sender should reduce its sending rate [Ja88]. If the network had immense buffering within it to avoid packet loss, this over-engineering would defeat TCP's congestion signal. Furthermore, such buffering does *not* guarantee that the network can promise to always deliver useful throughput [Na87], and, actually, things would be worse off, since TCP senders then could not adapt their transmission rates to the limited capacity of the bottleneck link.

In this chapter we look at what our measurements tell us about packet loss in the Internet: how frequently it occurs and with what general patterns (§ 15.1); differences between loss rates of data packets and acks (§ 15.2); the degree to which it occurs in bursts (§ 15.3); the degree to which losses occur at the bottleneck link (§ 15.4); how loss rates evolve over time (§ 15.5); and how well TCP retransmission matches genuine loss (§ 15.6).

15.1 Loss rates

A fundamental issue in measuring packet loss is to avoid confusing measurement drops with genuine losses. Doing so can often be difficult unless the measurement apparatus takes pains to accurately report measurement drops. As we saw in § 10.3.1, some do and some do not. Here is one of the analysis areas where the effort to ensure that `tcpanaly` understands the details of the many

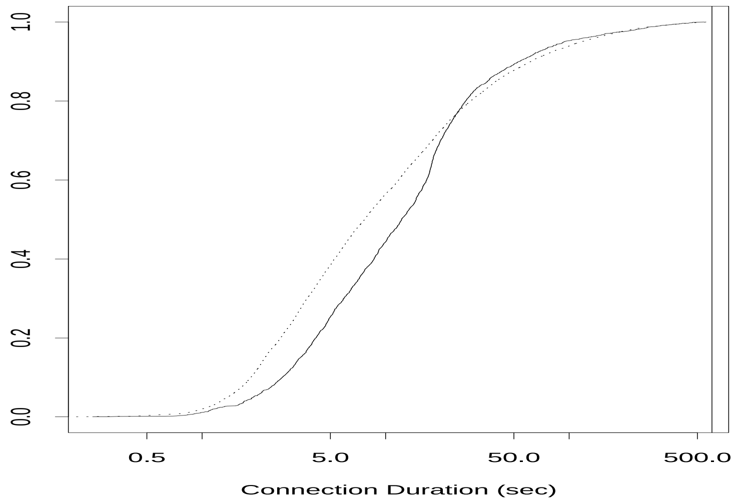

Figure 15.1: Connection durations for $\mathcal{N}_1$ (solid) and $\mathcal{N}_2$ (dotted)

TCP implementations in our study pays off extremely well. Because we can determine whether traces suffer from measurement drops, we can exclude those that do from our packet loss analysis and avoid what could otherwise be significant inaccuracies. Since, for the most part, measurement drops will be uncorrelated with the presence of true network drops, excluding these tainted traces should not bias our subsequent analysis. An exception would be if the measurement drops are due to large bursts of traffic on the local network overrunning the packet filter's ability to record the burst, and if such bursts were coupled with true loss on the local network. Since our interest lies in loss in the Internet-in-the-large, and not in loss in local networks (even though local loss also contributes to the end-to-end chain), we regard this source of bias as minor.

Our measurements do, however, suffer from one form of bias: due to their limited duration (§ 9.3), we will fail to successfully measure and analyze connections that suffered such high packet loss rates that they required more than 10 minutes to transfer 100 Kbyte. When these measurement attempts reach the 10-minute lifetime without having successfully completed, the entire measurement attempt is aborted, and *no* trace data is retrieved from the NPDs conducting the measurement.

Unfortunately, due to the centralized control of the experiment, we cannot accurately assess how often a measurement failed for this reason, and how often for a different reason, such as a loss of connectivity between `npd_control` and one of the remote NPDs (§ 5.2, § 9.3). Thus, the statistics presented in this section will *underestimate* Internet packet loss rates somewhat.

We argue, however, that the bias is, overall, fairly small. Figure 15.1 shows the distribution of the connection durations for $\mathcal{N}_1$ (solid line) and $\mathcal{N}_2$ (dotted line). The vertical line on the righthand side of the plot marks the 10-minute maximum duration. The x-axis is logarithmically scales, so we see that a large number of the connections in our study completed much sooner than the 10 minute upper lifetime. This in turn suggests that the lifetime was generally not a limitation. At the end of this section, however, we show that it *did* significantly bias European loss rates towards underestimation.

We begin our analysis with looking at aggregate packet loss over the course of entire connections. In $\mathcal{N}_1$, out of about 714 thousand packets (data and ack) transmitted, 3.0% failed to arrive at the other end. In $\mathcal{N}_2$, for 4.66 million packets, the figure rose to 4.6%, a significant increase that merits further investigation.

On immediate question is whether the use of additional sites in $\mathcal{N}_2$ (and the absence of a few of the $\mathcal{N}_1$ sites) skewed these basic numbers. Indeed, it did, but *towards underestimating the increase*! Of the sites in common, in $\mathcal{N}_1$, 2.7% of the packets were lost, while in $\mathcal{N}_2$, this figure nearly doubled to 5.2%. Conventional wisdom among TCP researchers holds that a loss rate of 5% has a significant adverse effect on TCP performance, because it will greatly limit the size of the congestion window and hence the transfer rate, while 3% is often substantially less serious. Thus, it behooves us to try to understand the circumstances and details of the increase as much as possible.

First, we need to address the question of whether the increase in loss rate was due to the use of bigger windows in $\mathcal{N}_2$ than in $\mathcal{N}_1$ (§ 9.3). Such could easily be the case, since with larger windows the transfers will often have significantly more data in flight, and, consequently, will load the router queues along the path much more. We can assess the impact of larger windows by looking at loss rates of *data* packets versus those for *ack* packets. Data packets contribute to queueing and having more in flight stresses the forward path. On the other hand, the rate at which a TCP transmits data packets *adapts* to current conditions. Ack packets contribute almost no additional load along the reverse path, other than occupying a buffer when queued, so having more of them in flight at one time should not significantly alter the loss rate they suffer. They do not adapt to current conditions, except during periods of heavy congestion, when an entire window's worth of acks is lost, forcing a timeout retransmission.[1] Thus, to compare changes in loss rates between $\mathcal{N}_1$ and $\mathcal{N}_2$, using the ack loss rates should eliminate the bias caused by the different window sizes. We discuss more issues concerning data packet loss versus ack loss in § 15.2.

Overall, in $\mathcal{N}_1$, acks were actually slightly more likely (3.16%) to be lost than data packets (2.96%), while in $\mathcal{N}_2$ the ordering is the opposite (4.25% for acks versus 4.75% for data packets). Restricting the comparison to the sites in common, however, changed the discrepancy between data packets and acks, with 2.88% for acks versus 2.65% for data packets in $\mathcal{N}_1$, and 5.14% versus 5.28% for the same $\mathcal{N}_2$ figures. So, even if we restrict ourselves to the ack loss rates for the common sites, which should be quite sound to compare, we observe a 78% increase in the loss rate, from 2.88% to 5.14%.

Another interesting loss rate figure is how the rate changes if we condition on observing at least one loss during the connection. Here we make a tacit assumption that a network path has two basic states, "quiescent," during which connections tend to not suffer any loss, and "busy," during which they tend to suffer loss. The first corresponds to, overall, light or steady enough load that the router buffers suffice to avoid packet loss, and the second to sufficient load, overall, to occasionally overflow the buffers. We would expect to find that "busy" states coincide with the usual peak usage times of working hours, and quiescent states with off-peak times. We return to this point below, in the discussion of Figure 15.3 and Figure 15.4.

In $\mathcal{N}_1$, 52% of the connections between the common sites did not lose a single ack packet. However, only 28% of the connections losing at least one ack lost exactly one. For $\mathcal{N}_2$, the corre-

[1]The transmission rate of acks can also adapt to current conditions if the loss conditions along both directions of the path are correlated, since the rate at which a TCP transmits acks reflects the rate at which it receives data packets. In § 15.2 below, however, we find that loss rates in the two directions are nearly uncorrelated.

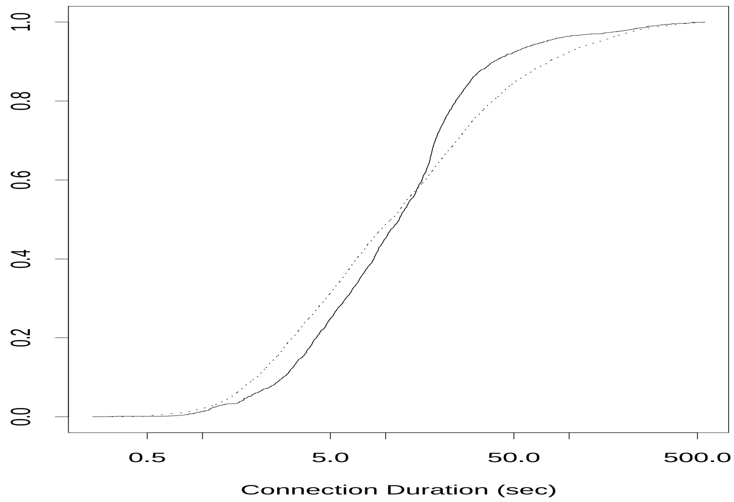

Figure 15.2: Connection durations for sites common to $\mathcal{N}_1$ (solid) and $\mathcal{N}_2$ (dotted)

sponding figures are 49% and 20%. We see that part of the change in the higher $\mathcal{N}_2$ ack loss rates stems from *greater loss during busy periods*. The proportion of quiescent periods remains virtually unchanged. Similarly, for the common sites, if we condition on a connection suffering at least one loss, then the ack loss rate for an $\mathcal{N}_1$ connection climbs from 2.88% to 5.69%, while for $\mathcal{N}_2$ the increase goes from 5.14% to 9.16%. Thus, even in $\mathcal{N}_1$, if the network path was busy (using our simplistic definition above), loss rates were quite high, and for $\mathcal{N}_2$ they shot upward to a level that in general will seriously impede TCP performance.

These increases give us strong evidence that networking conditions in one important respect *degraded* during the course of 1995, similar to our earlier finding that several aspects of Internet routing degraded during 1995 (§ 6.10, § 8.5). Since bottleneck link rates generally *increased* during 1995 (§ 14.7.1), we cannot tell from just the loss rate statistic whether users perceived the network as delivering better or worse service. A basic measure of perceived level of service is how long it takes to transfer a given amount of data. However, when comparing such durations we need to keep in mind that the use of bigger windows in $\mathcal{N}_2$ gave $\mathcal{N}_2$ connections more opportunity both to "fill the pipe" and to utilize fast retransmission (§ 9.2.7), which gives them performance advantages that have little to do with how the network service changed. (For the sites in common, in $\mathcal{N}_1$ the mean number of fast retransmissions was 0.98, while in $\mathcal{N}_2$ it climbed to 1.64.)

Still, we find the comparison illuminating. Figure 15.2 shows the distribution of the durations of connections between sites common to both $\mathcal{N}_1$ (solid line) and $\mathcal{N}_2$ (dotted line). For the sites in common, the median connection duration diminished from 11.8 sec in $\mathcal{N}_1$ to 10.7 sec in $\mathcal{N}_2$, a rather modest improvement. That single figure does not tell the entire story, though, since we see from the figure that the distribution of durations did not unilaterally slide a bit to the left. Instead, $\mathcal{N}_2$ connections were likely to be 20% shorter than those in $\mathcal{N}_1$ *if they were short*, meaning that we condition on the duration being < 12 sec; and 50% longer if we condition on the duration being > 12 sec. It seems likely that the differences are due to a higher prevalence of fast retransmission

Region	# $\mathcal{N}_1$	# $\mathcal{N}_2$	$\mathcal{N}_1$ loss rate	$\mathcal{N}_2$ loss rate	Δ
Europe	104	734	2.8%	2.8%	−03%
North America	641	2,405	1.3%	1.6%	+23%
(`umont`)	75	562	1.5%	5.8%	+287%
Into Europe	255	1,243	6.2%	11.7%	+88%
Into North America	320	1,544	3.5%	3.2%	−08%
All regions	1,395	6,488	2.8%	4.6%	+63%

Table XXI: Ack loss rates for different connection geographies

in $\mathcal{N}_2$ aiding short transfers, while a higher packet loss rate led to more frequent timeouts for those connections that failed to open their congestion windows enough to facilitate fast retransmission.[2]

So far, we have treated the Internet as a single aggregated network in our loss analysis. Geography, however, plays a crucial role in the prevalence of packet loss. To study geographic effects, we partition the connections between the sites common to $\mathcal{N}_1$ and $\mathcal{N}_2$ into four primary groups: "European," "North American," "Into Europe," and "Into North America." European connections are those with both a European sender and a European receiver. North American have both sender and receiver in Canada or the United States (but see below). "Into Europe" are connections with European data *senders* and North American data *receivers*. The terminology is backwards here because what we will assess are *ack* loss rates, and these are generated by the receiver. Hence, "Into Europe" loss rates reflect those experienced by packet streams traveling from North America into Europe. Similarly, "Into North America" are connections with North American data senders, European data receivers, and ack streams traveling from Europe into North America.

This partition does not include connections to or from Australia, because we had only one Australian site common to both $\mathcal{N}_1$ and $\mathcal{N}_2$, so it would be difficult to gauge the generality of loss rates involving it. We note, however, that it experienced a rise of more than a factor of two in the loss rates of ack traveling into and out of Australia, from 3.3% in $\mathcal{N}_1$ to 7.8% in $\mathcal{N}_2$.

While the above grouping was our original intent, upon examining the data we made one further distinction. The sole Canadian site, `umont`, was a major outlier for packet loss in $\mathcal{N}_2$, so large that its presence as one of the 13 North American hosts sufficed to significantly skew the overall North American findings. (It was not, however, an outlier in $\mathcal{N}_1$.) Since we had no other Canadian sites in our study, we cannot gauge whether this reflects a problem unique to `umont` or a more general problem with Canadian Internet service. Consequently, we removed `umont` from our notion of "North America" as described above; so, in fact, all of the North American sites discussed below are in the United States. We also summarize below connections from U.S. sites to or from `umont`, to illustrate its atypical loss rates.

Table XXI shows the loss rates of ack packets for the different regions. The second and third columns give the number of $\mathcal{N}_1$ and $\mathcal{N}_2$ connections that occurred in the region. There were 6 common European sites and 12 North American sites plus `umont`. The fourth and fifth columns give the overall loss rate for the ack packets sent during all of the region's connections, and the final column indicates the loss rate change between $\mathcal{N}_1$ and $\mathcal{N}_2$. Clearly:

[2]Note that, had we not restricted ourselves to the sites common to the two datasets, but instead interpreted Figure 15.1 in this regard, then we would have drawn a considerably different, less sound conclusion.

Region	$\mathcal{N}_1$ quies.	$\mathcal{N}_2$ quies.	$\mathcal{N}_1$ cond. loss	$\mathcal{N}_2$ cond. loss	Δ
Europe	48%	58%	5.3%	5.9%	+11%
North America	66%	69%	3.6%	4.4%	+21%
(`umont`)	60%	15%	3.7%	6.8%	+81%
Into Europe	40%	31%	9.8%	16.9%	+73%
Into N.A.	35%	52%	4.9%	6.0%	+22%
All regions	53%	52%	5.6%	8.7%	+54%

Table XXII: Conditional ack loss rates for different connection geographies

- Europe suffered considerably higher packet loss rates than did North America, but the loss rate appears stable. However, we show below that the European figures are *biased* towards *underestimation*;
- North American loss rates were fairly low and, while the trend is increasing, it is not doing so at an ominous rate;
- `umont` suffered a tremendous increase in packet loss rate, although we lack sufficient data to tell if this is a general problem for Canadian networks or specific to the University of Montreal or its local region;
- the trans-Atlantic links carrying European traffic to North America had fairly high loss rates, but the situation is perhaps improving; and
- the links carrying North American traffic to Europe were a compounding disaster. We note that since Europe's rates are significantly lower than those of trans-Atlantic traffic heading into Europe, it must be the case that most traffic between two European sites stays inside Europe, rather than transiting through North America, even though we sometimes observed such routes in § 6.9.

Table XXII looks at loss rates for the same regions, but now with conditioning on whether any acks were lost. The second and third columns give the proportion of quiescent connections, where "quiescent" is defined as above to mean connections that did not lose any acks. We see that, except for `umont` and the trans-Atlantic links going into North America, the proportion of quiescent connections was fairly stable, suggesting that perhaps changes in loss rate are confined to already-loaded "busy" periods of heavy load. We investigate this possibility in more detail shortly.

The fourth and fifth columns list the proportion of acks lost, given that at least one ack was lost, and the final column summarizes the relative change. None of the conditional loss rates is especially heartening, and the trends are *all* increasing. During $\mathcal{N}_2$, the trans-Atlantic links into Europe were close to unusable during busy periods, with a loss rate of nearly 17%. This matches anecdotal reports such as requests the author received to mail hardcopies of papers to European researchers since they could not viably retrieve them over the network. In summary, we note that, *for every region, loss rates for busy connections increased between* $\mathcal{N}_1$ *and* $\mathcal{N}_2$.

Within regions, we find considerable site-to-site variation in loss rates, as well as variation between loss rates for packets inbound to the site and those outbound (§ 15.2). We did not, however,

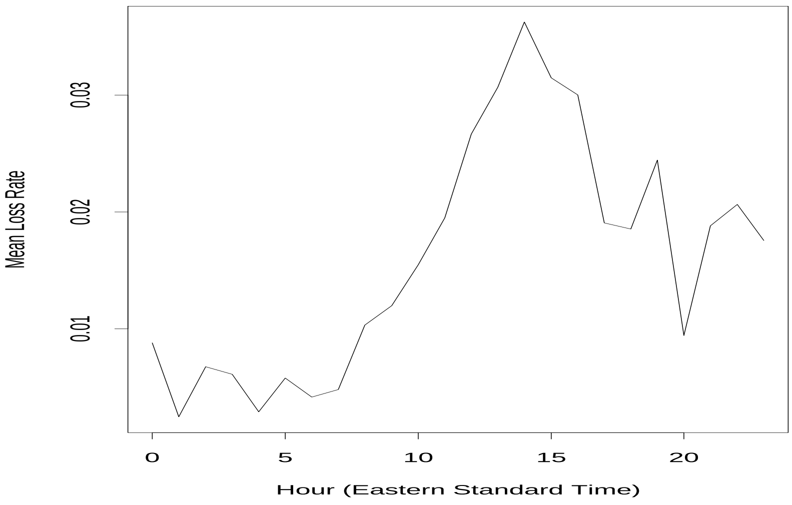

Figure 15.3: Hourly variation in ack loss rate for North American connections

find any other outliers as dramatic as `umont` in $\mathcal{N}_2$, so we kept the regions otherwise intact.

The last aggregate loss statistics we look at are variations of loss rate over the course of the day. We expect to find a diurnal cycle, as numerous studies have noted significant hourly variation in connection and packet arrival rates ([PF95] and many others). It was this expectation that led us to postulate that the distinction made above between "busy" and "quiescent" connections is broadly meaningful.

Figures 15.3 and 15.4 show the hourly loss rates for the $\mathcal{N}_2$ connections internal to North America and Europe, respectively. The North American loss rates, with the x-axis reflecting the hour in the Eastern Standard Time zone, clearly follow the oft-observed pattern of activity increasing over the morning hours and falling off during the late afternoon. [PF95] notes a pickup in evening FTP traffic, which agrees with the secondary peak. One unusual facet of Figure 15.3 is that it does not exhibit a noon-time "dip." However, this is almost certainly due to the North American traffic spanning three time zones, effectively spreading out lunch-related lulls over several hours. The apparent discontinuity between the 23rd hour at the right and the midnight hour at the left, however, is puzzling. We have verified that as one approaches midnight, the rates come closer together. We do not, though, have an explanation as to why midnight EST would serve as such a sharp transition point, given that it corresponds to 9PM Pacific Standard Time, when presumably we still see considerable user activity.

Figure 15.4 differs considerably from Figure 15.3. Here the x-axis reflects the hour in the Greenwich Mean Time zone. We observe a morning rise in loss rate, but a considerable noontime dip lasting several hours, followed by a striking increase in the late afternoon. Again, the evening hours are elevated compared to the early morning hours, with a sharp transition occurring around midnight. The late afternoon hours may in part reflect increasing background traffic from North American sites, too, since late afternoon GMT coincides with noon and early afternoon EST, which we see in Figure 15.3 is the peak North American period.

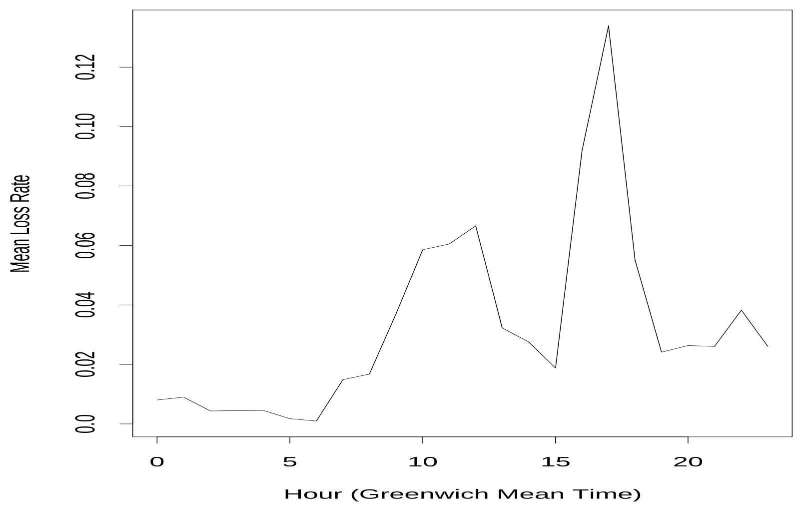

Figure 15.4: Hourly variation in ack loss rate for European connections

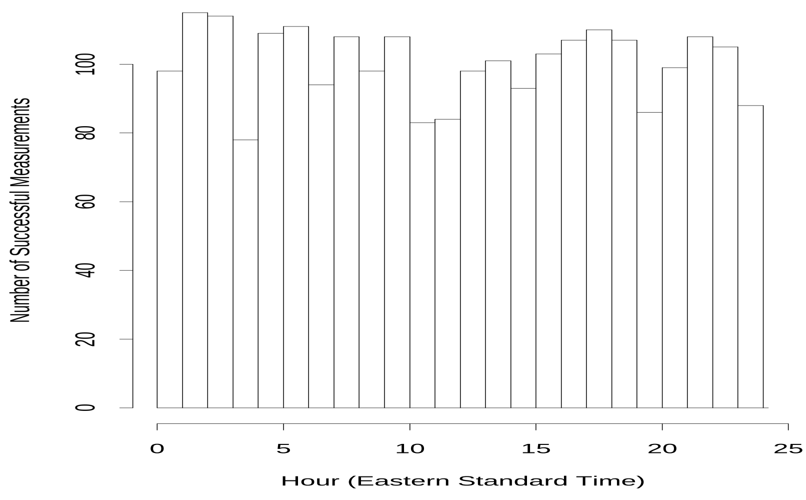

Figure 15.5: Successful North American measurements, per hour

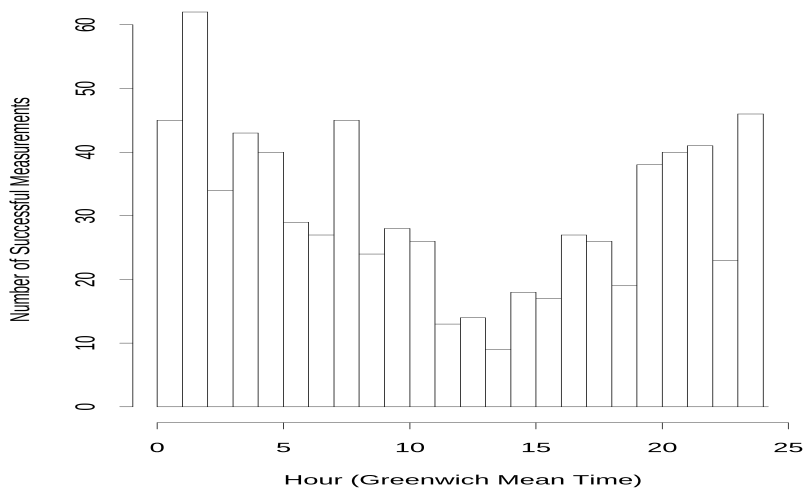

Figure 15.6: Successful European measurements, per hour

We must exercise caution, however, in interpreting Figure 15.4, due to our measurement bias against very long-lived connections (discussed at the beginning of this section). We can test for the presence of this bias by examining how many successful measurements we made for each hour of the day. Because of our Poisson sampling methodology (§ 9.1), measurement *attempts* were uniformly distributed over the course of the day. Figure 15.5 shows a histogram of the number of successful North American measurements made for each distinct hour of the day. The distribution appears fairly even, and, indeed, the measurement times pass the powerful Anderson-Darling A^2 goodness-of-fit test for uniformity [DS86], using 5% significance (and, indeed, for higher significance).

Figure 15.6 shows the same histogram for the European measurements. The bias towards the less busy early morning and late evening hours immediately stands out. The distribution fails A^2 at all significance levels, as one might expect. The bias is strongest against the 11AM to 1PM periods, and eases somewhat in the later afternoon, so the apparent difference between the two corresponding peaks in Figure 15.4 may be simply due to measurement bias and not reflect a true underlying difference. However, we can certainly conclude based on Figure 15.6 that our analyses of European loss rates are in general *underestimates*.

15.2 Data packet loss vs. ack loss

We noted in the previous section that analyzing data packet loss rates can be complicated because the size of the data packets and the tendency for them to be sent closely together both add to queueing load along the network path. We expect that this load in turn leads to a greater likelihood of the data packets being lost, though, because TCP can unfairly distribute available bandwidth [FJ92], this is not necessarily the case. We saw in § 15.1 that, in $\mathcal{N}_1$, acks were actually slightly more likely to be lost than data packets, though, in $\mathcal{N}_2$, the pattern reverses, which we (at least partially) attribute to the use of bigger windows in $\mathcal{N}_2$ (§ 9.3).

In this section we take a closer look at the loss rates of data packets versus those of acks. We consider any packet carrying one or more bytes of user data as a data packet. We would expect to observe some differences between different-sized data packets. Unfortunately, it would prove difficult to explore this effect with our data. Some of the sites in our study always used a maximum segment size (MSS) of 512 bytes, the common default value, while others used larger sizes whenever the opportunity to do so arose. But the site-specific nature of the MSS used means that, for each site, the samples of data packet loss rates generally reflect only a small number of packet sizes, sometimes only one. Since in § 15.1 we showed that ack loss rates exhibit strong regional variation, we could easily conflate a spurious MSS size effect in data loss rates with a genuine, separate effect due to the regions.

Thus, we confine ourselves to a simple definition of "data packet" as one carrying any user data whatsoever. But in addition, we make a key distinction between "loaded" and "unloaded" data packets. A "loaded" data packet is one that presumably queued at the bottleneck link behind one of the connection's previous packets, while an unloaded data packet is one that we know did not *have* to queue at the bottleneck behind a predecessor. Here we are abstracting the intricate, multi-element network path to a presumably equivalent model of a single element that forwards at the bottleneck rate, and at which all significant queueing occurs.

To tell if a packet is unloaded, we first form an estimate of the bottleneck bandwidth using the methodology developed in Chapter 14. If the methodology indicates a bottleneck change or the possible presence of a multi-channel bottleneck, then we refrain from further packet-loss analysis.

If, however, the methodology produces a single bottleneck estimate, ρ_B, as is generally the case, then the methodology also associates lower and upper bounds with ρ_B (Eqn 14.12):

$$\rho_B^- < \rho_B < \rho_B^+. \tag{15.1}$$

This equation in turn gives us the maximum amount of time required for a b-byte packet to transit across the bottleneck, namely:

$$\tau_b^+ = b/\rho_B^- \text{ sec.} \tag{15.2}$$

Let T_i^s be the time at which the sender transmits the ith data packet. We then sequentially associate a *maximum load* λ_i^+ with the packet as follows. The first packet has a load equal to

$$\lambda_1^+ = \tau_b^+, \tag{15.3}$$

where b is the size of the packet. Subsequent packets have a load

$$\lambda_i^+ = \tau_b^+ + \max\left[(T_{i-1}^s + \lambda_{i-1}^+) - T_i^s, 0\right]. \tag{15.4}$$

λ_i^+ thus reflects the maximum amount of extra delay the ith packet incurs due to its own transmission time across the bottleneck link, plus the time required to first transmit any preceding packets across the bottleneck link, if i will arrive at the bottleneck before they completed transmission. The latter will be the case if

$$T_i^s < T_{i-1}^s + \lambda_{i-1}^+, \tag{15.5}$$

because this condition means that packet i was sent shortly enough after packet $i-1$ that packet $i-1$ would not yet have cleared the bottleneck link by the time packet i arrived at the bottleneck.

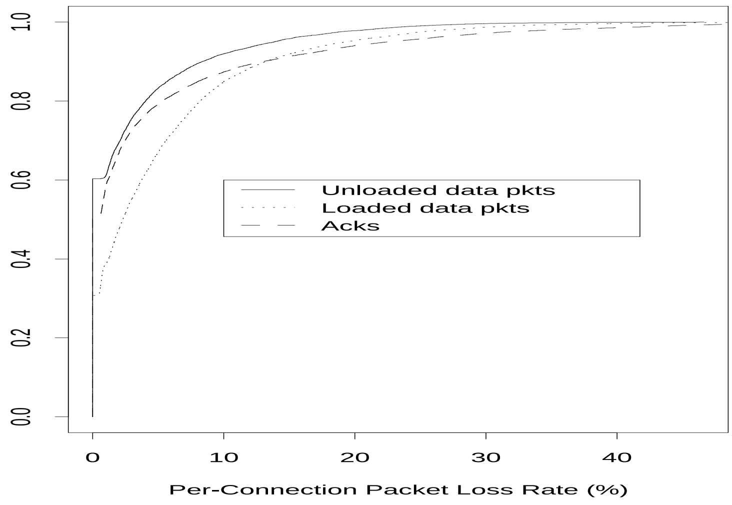

Figure 15.7: $\mathcal{N}_2$ loss rates for data packets and acks

If Eqn 15.5 applies to packet i, then we will say that packet i was "loaded," meaning that it had to wait for pending transmission of earlier packets. Otherwise, we term it "unloaded."

The development of the maximal load λ_i^+ has natural analogs λ_i^- and λ_i for the minimal and central loads associated with each packet, by using ρ_B^+ or ρ_B (from Eqn 15.1) in Eqn 15.2 to compute analogous "self-interference" time constants τ_b^- and τ_b.[3] Similarly, we can define different flavors of "loaded" and "unloaded" depending on whether we use the maximal, central, or minimal definitions for λ_i. In this section, we exercise conservatism and only consider a packet as unloaded for the definition in terms of the maximal λ_i^+.

Presently, our interest in whether a packet is loaded or unloaded comes just from analyzing whether the two types have different loss patterns. In § 16.2.6 we look in more detail at the coupling between λ_i and the variation in packet transit times.

In both $\mathcal{N}_1$ and $\mathcal{N}_2$, about 2/3's of the data packets were loaded. We might at first expect more loaded packets in $\mathcal{N}_2$, due to its use of bigger windows. Window size, however, determines whether the bottleneck link might continuously *remain* loaded. Even for a relatively small window size, the TCP sender will transmit a number of packets (equal to the window size) over a fairly short amount of time, and all of these but the first will be loaded. Once the entire window is in flight, then a lull comes equal to the mismatch between the small window and the bandwidth-delay product corresponding to the bottleneck rate and the RTT. Then the acks for the window arrive in short order, and self-clocking leads to another flight of all-but-one loaded packets. Thus, window size does not have a great deal of impact on the proportion of loaded packets.

Figure 15.7 shows the distributions of loss rates during $\mathcal{N}_2$ for unloaded data packets, loaded data packets, and acks. All three distributions show considerable probability of zero loss.[4]

[3] ρ_B^+ is associated with λ_i^- because of the inverse relationship given by Eqn 15.2; the higher the bottleneck bandwidth, the lower the time required for a packet to transit across the bottleneck, so the less load associated with the packet.

[4] Each curve also shows a horizontal shift just above a loss rate of 0%. These reflect the fact that the loss rate is

From the figure, we immediately see that loaded packets are much more likely to be dropped than unloaded packets, as we would expect. In addition, we see that acks are consistently more likely than unloaded packets to be dropped, but generally less likely to be dropped than loaded packets, except during times of severe loss, above about 14%, which make up the upper 10% of the distributions. We interpret the difference between ack and data loss rates as reflecting the fact that, while an ack stream presents a much lighter load to the network than a data packet stream (particularly a series of loaded data packets), the ack stream does *not* adapt to the current network conditions, while the data packet stream does. Thus, unloaded data packets gain the twin benefits of traveling at a time when the connection is not itself significantly contributing to load along the network path, and also lowering their transmission rate during times of congestion. Loaded data packets stress the network path, but at least they adapt, and, during periods of heavy congestion, their adaptive behavior outweighs the advantages of ack streams that otherwise favor acks during periods of lower congestion.

The equivalent set of distributions for $\mathcal{N}_1$ is qualitatively the same, though the distance between the three distributions is narrower. This likely reflects both the overall lower loss rates in $\mathcal{N}_1$ (§ 15.1) and the use of smaller windows limiting loss rates for loaded packets.

It is interesting to note the extremes that packet loss can reach. In $\mathcal{N}_2$, the largest unloaded data packet loss rate we observed was about 47%. For loaded packets it climbed to 65%. As we would expect, these connections suffered egregiously, achieving overall data throughput rates in the low hundreds of bytes per second due to lengthy, backed-off timeout periods. However, they *did* manage to successfully complete their transfers within their alloted ten minutes, a testimony to TCP's tenacity. For both of these extremes, *no* acks were lost in the reverse direction! The largest ack loss rate was even higher, 68%. Starved for confirmation of forward progress, this connection also managed only a few hundred bytes per second. Ironically, *no* data packets were lost in the forward direction!

As indicated by these extreme cases, clearly packet losses on the forward and reverse paths are sometimes completely independent. Indeed, the coefficient of correlation between combined (loaded and unloaded) data packet loss rates and ack loss rates in $\mathcal{N}_1$ was about 0.21, with the correlation for connections within North America falling to 0.13. In $\mathcal{N}_2$, however, the loss rates become uncorrelated (coefficient of -0.02), perhaps due to the greater prevalence of significant routing asymmetry (Chapter 8).

Another form of asymmetry is the degree to which loss correlates with the connection's throughput. We would expect that data packet loss rates correlate more strongly, and negatively, with throughput, since each loss requires a retransmission that subsequently cuts the sender's transmission rate, and perhaps entails a lengthy timeout lull. Ack loss, on the other hand, may go unnoticed, if light, since acks are cumulative, and, if another ack arrives shortly, the connection will not stall for any appreciable amount of time.

To fairly gauge the correlation, we need to first account for the different maximum throughput rates due to the different bottleneck bandwidth rates. We do so by dividing the achieved throughput over the entire connection (total bytes transferred divided by total duration) by the estimated bottleneck bandwidth. We then compute θ, the coefficient of correlation, between the *logarithm* of the normalized throughput and the loss rates of interest, where the logarithmic transforma-

computed in terms of k packets lost out of a total of n, hence $1/n$ is the minimum possible positive loss rate. Since, for most connections, $n \approx 200$ packets, we observe a minimum possible loss rate of around 0.5%.

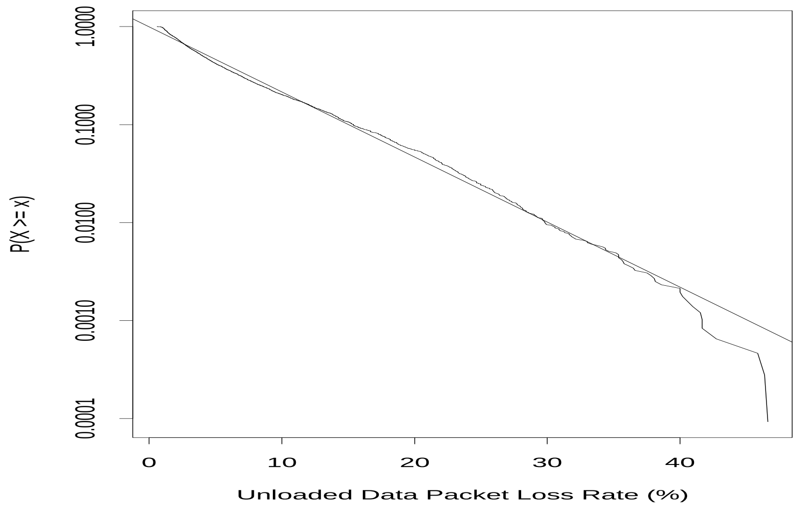

Figure 15.8: Complementary distribution plot of $\mathcal{N}_2$ unloaded data packet loss rate

tion is to reduce the otherwise dominating effect of throughput outliers.

For $\mathcal{N}_2$, we find that θ for the overall data loss rate is quite large, about -0.52, with unloaded loss rates a bit more strongly correlated than loaded loss rates. Presumably this latter effect is because backed-off timeout retransmissions, which have the greatest deleterious effect on connection throughput, always generate unloaded data packets, and further back-off occurs when these packets are then lost. The corresponding θ for ack loss rates also indicates a fairly strongly correlation, with a value of -0.42. Since these figures are for $\mathcal{N}_2$, this correlation is *not* due to any coupling between the ack loss rate and the data packet loss rate, because the two are generally uncorrelated, as shown previously. Instead, the correlation is probably due to the coupling between the ack loss rate and the possibility of losing an entire flight's worth of acks, which then unavoidably leads to a timeout retransmission (§ 15.6).

The significant correlation between ack loss rates and normalized connection throughput indicates that, when attempting to predict a connection's throughput along a particular forward path, it pays to have information about conditions along the reverse path, too. For the North American region (as defined in § 15.1), the correlations weaken somewhat, to -0.40 for data packet loss rates and -0.25 for acks. Thus, we must recognize that the strength of the correlations varies considerably.

The distributions in Figure 15.7 have shapes suggestive of exponential distributions, if we ignore the considerable zero portion of each distribution. Further investigating the distributions, one striking feature we find is that the non-zero portion of both the unloaded and loaded data packet loss rates is almost exactly exponential, while that for acks is not nearly so close a match.

Figures 15.8, 15.9, and 15.10 show logarithmically scaled complementary distribution plots of the unloaded, loaded, and ack loss rates, conditioned on observing at least one loss. A straight line on such a plot corresponds to an exponential distribution. We have added least-squares fits to each plot. We see that, for both unloaded and loaded data packets, the loss rate distribution is

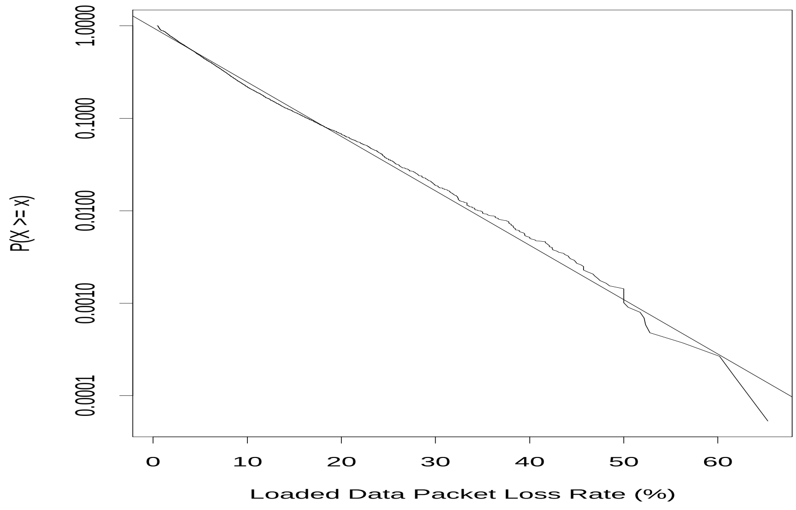

Figure 15.9: Complementary distribution plot of $\mathcal{N}_2$ loaded data packet loss rate

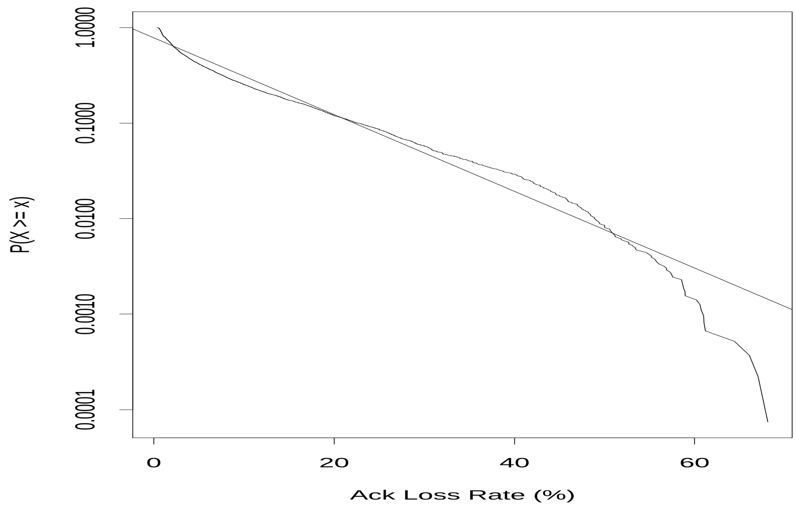

Figure 15.10: Complementary distribution plot of $\mathcal{N}_2$ ack loss rate

quite close to exponential, but for acks it deviates considerably more. The effect is widespread: it is also present for $\mathcal{N}_1$, and for the North American and European subsets of $\mathcal{N}_2$.

While striking, interpreting the fit to the exponential distribution is difficult. If, for example, packet loss occurs independently and with a constant probability, then we would expect the loss rate to reflect a binomial distribution, but that is *not* what we observe. (We also know from the results in § 15.1 that there is *not* a single Internet packet loss rate, or anything approaching such a situation.)

It seems likely that the better exponential fit for both loaded and unloaded data loss rates than ack loss rates holds a clue. The most salient difference between the transmission of data packets and that of acks is that the rate at which the sender transmits data packets *adapts* to the current network conditions, and furthermore it adapts *based on observing data packet loss*. Thus, if we passively *measure* the loss rate by observing the fate of a connection's TCP data packets, then we in fact are making measurements using a mechanism whose goal is to lower the value of what we are measuring (by spacing out the measurements). Consequently, we need to take care to distinguish between measuring overall Internet packet loss rates, which is best done using *non-adaptive* sampling, versus measuring loss rates *experienced* by a transport connection's packets—the two can be quite different.

15.3 Loss bursts

In this section we look at the degree to which packet loss occurs in *bursts* of more than one consecutive loss. Analytic models of network behavior often assume individual packet losses occur at a fixed rate but independently from other losses, as this assumption aids in keeping the models tractable. Accordingly, to gauge the strength of these models we need to address the issue of the soundness of this assumption.

As with loss rates, we expect that the size of loss bursts depends on whether we analyze losses of loaded data packets, unloaded data packets, or acks. These each correspond to a different transmission rate, and, furthermore, the first two are generated at a rate dynamically adapted to the frequency of previously observed packet loss, while acks are not.

The first question we address is the degree to which packet losses are well-modeled as independent. In [Bo93], Bolot investigated this question by comparing the unconditional loss probability, which we denote as P_l^u (*ulp* in Bolot's paper), with the conditional loss probability, P_l^c (*clp*), where P_l^c is conditioned on the fact that the previous packet was also lost. He found that $P_l^c \geq P_l^u$ always held, which one would expect, as it would be surprising if loss of the previous packet made loss of the next packet less likely. He investigated the relationship between P_l^u and P_l^c for different packet spacings δ, ranging from 8 msec to 500 msec. He found that P_l^c approaches P_l^u as δ increases, indicating that loss correlations are short-lived, and concluded that "losses of probe packets are essentially random as long as the probe traffic uses less than 10% of the available capacity of the connection over which the probes are sent." He also observed that P_l^u stabilized at about 10%, quite a high loss rate, though the path being studied included a heavily loaded trans-Atlantic link, and also a mid-level network known to have previously experienced 3% loss rates unrelated to congestion.

Table XXIII summarizes P_l^u and P_l^c for the different types of packets and our two datasets. P_l^c conditions on whether the connection's previous packet was lost, even if it is a different type than its successor (e.g., a loaded packet lost followed by an unloaded). Clearly, for TCP packets (which

Type of loss	P_l^u		P_l^c	
	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_1$	$\mathcal{N}_2$
Loaded data pkt	2.8%	4.5%	49%	50%
Unloaded data pkt	3.3%	5.3%	20%	25%
Ack	3.2%	4.3%	25%	31%

Table XXIII: Unconditional and conditional loss rates for different packet types

have a large range of interarrival intervals), we must discard the assumption that loss events are well-modeled as independent. Even for the low-burden, relatively low-rate ack packets, the loss probability jumps by a factor of seven if the previous ack was lost. We would expect to find the disparity strongest for loaded data packets, as these must contend for buffers with the connection's own previous packets, as well as any additional traffic, and indeed this is the case. We find the effect least strong for unloaded data packets, which accords with these not having to contend with the connection's previous packets.

It is interesting to observe that loaded packets are unconditionally less likely to be lost than unloaded packets. We suspect this reflects the fact that lengthy periods of heavy loss or outages will lead to timeout retransmissions, and these are unloaded, so they contribute to the loss probability of unloaded packets rather than loaded packets.

The relative differences between P_l^u and P_l^c in Table XXIII all exceed those computed by Bolot by a large factor. His greatest observed ratio of P_l^c to P_l^u was about 2.5:1. However, his P_l^u's were all much higher than those in Table XXIII, even for $\delta = 500$ msec, suggesting that the path he measured differed considerably from a "typical" path in our study.

(We also note that, since TCP packet loss events are not well-modeled as independent, it behooves us in general to avoid discussing unconditional packet loss in terms of *probability*, since for networking analysis this stochastic term often carries with it an implicit assumption of independence among the events. We advocate instead consistent use of the term packet loss *rate*, since this term downplays the implication of independence.)

Given that packet losses occur in bursts, the next natural question is: how big? To address this question, we grouped successive packet losses into *outages* and computed for each outage the number of packets lost and the duration of the outage in terms of the difference between the sending times of the two successfully arriving packets delimiting the outage. (Note that a data packet outage can encompass both loaded and unloaded packets.)

Figure 15.11 shows the distributions of the outage durations for data packets and acks in $\mathcal{N}_1$ and $\mathcal{N}_2$, using a logarithmic x-axis. We see considerable variation for the length of small outage durations. Our definition of duration as the time between two successfully arriving packets spanning the outage means the durations are both *upper bounds*,[5] and hence will be considerably skewed, for small values, by variations in the inter-packet spacing. The distributions are really only solid for larger values. Above 200 msec, the distributions agree quite closely, except that $\mathcal{N}_2$ data packet

[5]However, for large estimates the degree of overestimation is limited by the retransmit timer backoff (§ 9.2.3), and hence the estimated duration is off by at most a factor of two. Since we analyze the distributions using logarithmic x-axes, this factor at most results in a translation of the distribution's body—it does not appreciably alter the shape of the log-transformed distribution.

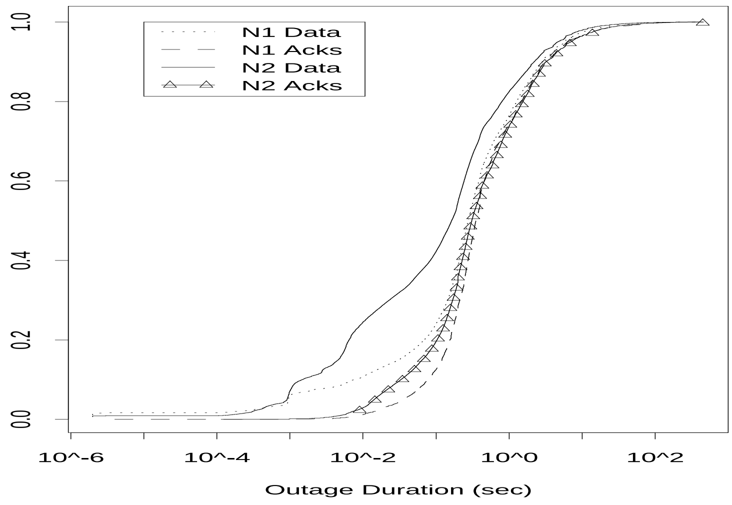

Figure 15.11: Distribution of packet loss outage durations

outages are considerably shorter lived, no doubt because, in $\mathcal{N}_2$, the connections often had many more data packets in flight (§ 9.3), and so had significantly more opportunity to observe short-lived outages.

Figure 15.12 shows the distributions conditioned on the outage exceeding 200 msec, which removes the effect of the $\mathcal{N}_2$ data packets observing more short-lived outages. (The x-axis extends only to 50 sec even though all of the distributions have some larger points. The plotting truncation lets us focus on the main body of the distribution in more detail than we could if we included the entire upper tail.) We see that, for outages of this length or longer, all four distributions agree fairly closely.

It is clear from Figure 15.11 that outage durations span several orders of magnitude. For example, 10% of the $\mathcal{N}_2$ ack outages were 33 msec or shorter, while another 10% were 3.2 sec or longer, a factor of a hundred larger. Furthermore, the upper tails of the distributions are consistent with those of Pareto distributions. Figure 15.13 shows a complementary distribution plot of the duration of $\mathcal{N}_2$ ack outages, for those lasting more than 2 sec (about 16% of all the outages). Both axes are log-scaled, so a straight line on the plot corresponds to a Pareto distribution. We see the long outages fit quite well to a Pareto distribution with shape parameter $\alpha = 1.06$, except for the extreme upper tail, to which we will return in a moment.

A shape parameter $\alpha \leq 2$ means that the distribution has *infinite variance*, indicating immense variability. Pareto distributions for activity and inactivity periods play key roles in some models of self-similar traffic [WTSW95, WP97, WPT97]. We do not attempt further analysis here of the possible role of packet loss outages in contributing to self-similar correlations in aggregate network traffic, but note that it may prove a fruitful area for further research.

However, it is clear in the plot that the extreme upper tail does not fit the same Pareto distribution. This discrepancy could simply be because the uppermost tail is subject to truncation, due to the 600-second lifetime to which our connections were limited (§ 9.3). But the discrepancy could in-

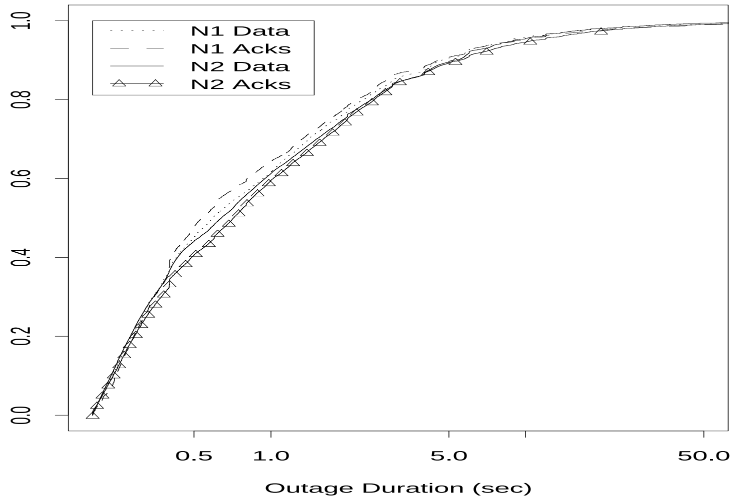

Figure 15.12: Distribution of packet loss outage durations exceeding 200 msec

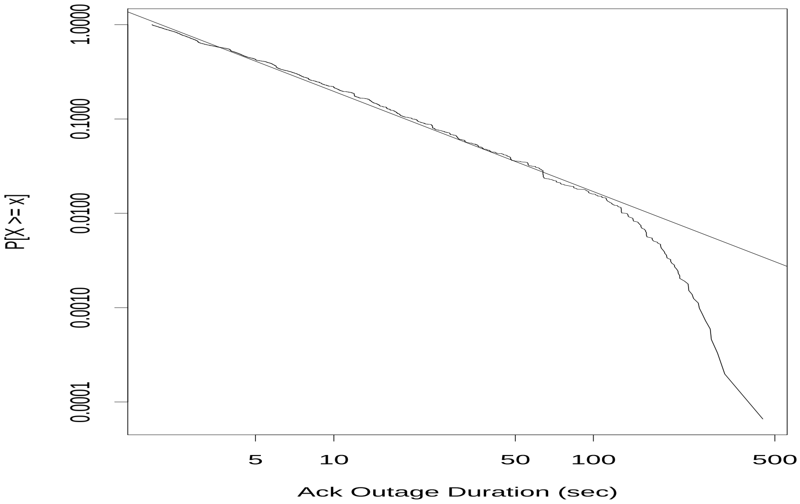

Figure 15.13: Log-log complementary distribution plot of $\mathcal{N}_2$ ack outage durations

stead reflect two different loss mechanisms. We showed in § 6.8 that "temporary outages" observed by `traceroute` measurements appear well-described using *exponential* distributions, which are much less volatile than Pareto distributions. That analysis, however, was confined to time scales of 30 sec or longer, and, for $\mathcal{R}_2$ (corresponding in time to $\mathcal{N}_2$), we found a mixture of exponentials, with the second only fitting to outages exceeding 75 sec in duration. This latter fit corresponds to the extreme upper tail in Figure 15.13. This in turn leads us to speculate that the distribution of outage durations might reflect a Pareto distribution for losses due to heavy congestion, and an exponential distribution for losses due to routing outages. We could test this hypothesis by gathering packet loss measurements made over longer periods of time, which would eliminate the ambiguities presented by the 600-second lifetime truncating the upper tail of our measurements.

We might also consider analyzing the *number* of lost packets in an outage, rather than the duration of the outage. This value, however, is much more subject to fluctuation due to the particulars of how many packets the TCP had in flight prior to the outage, or how many acks it had to generate during the outage in response to incoming data packets. We note that the mean number of packets lost during an outage was around 1.5, slightly lower for acks and higher for data packets. The loss extremes we observed were 68 consecutive data packets and 40 consecutive acks (most of which were dups in response to a large number of incoming packets). These extremes are less interesting than the extreme outage durations, because the former are specific to the structure of the TCP connections—both occurred due to very large numbers of data packets in flight,

We also note that the patterns of loss bursts we observe might be greatly shaped by use of "drop-tail" queueing. With the drop-tail policy, a router queues incoming packets until the available buffer space is exhausted, and then drops any additional arrivals until sufficient space becomes available again. Routers using drop-tail comprise the vast majority of Internet routers, no doubt because it is very simple to implement.

Simulations show that drop-tail leads to large bursts of losses when a flight of closely-spaced packets arrive at a router with no available buffers, and the entire flight is dropped [FJ93]. Related to this problem is a basic *unfairness* in how packets are dropped: a connection may suffer a large number of losses because a *different* connection is occupying all of the router's buffer. In response to these problems, [FJ93] developed the "Random Early Drop" (RED) policy, in which the router drops (or marks) incoming arrivals before all of the buffer has been exhausted. These drops are made with probabilities reflecting the proportion of the router's resources used by the connection, so the policy is much more fair than drop-tail. Because RED *spreads out* losses over time, widespread deployment of RED could significantly alter loss patterns and the corresponding connection dynamics.

A final loss burst pattern we investigated was the presence of *periodic* losses: outages occurring a fixed interval apart. Floyd and Jacobson observed periodic losses and described how they could arise due to global synchronization of the times at which routers exchange updates [FJ94]. They showed how fixed-interval timers such as thirty second update periods act as resonant frequencies, which can synchronize in phase to other events occurring at the same frequency. Periodic losses are thus possibly symptomatic of widespread synchronization in the network, which can have debilitating effects on network performance, especially since large loss periods can in turn synchronize all of the TCP senders that suffer a loss during the period.

Unfortunately, our measurements are ill-suited to detecting periodic loss. Rather than having fixed intervals between our loss "probes" (i.e., the individual packets of a single TCP connection), which would then lend themselves nicely to frequency-domain analysis, we have variable intervals. Furthermore, we used much larger, variable intervals between groups of measurements (connections), precisely to avoid problems with the measurements synchronizing to any periodicities present in the network. Thus, while we can analyze the timing of all of the lost packets in our measurements, the measurements themselves are sparse, and also are cluttered with a great deal of loss that is clearly not periodic.

We attempted to analyze for periodic loss by first identifying a North American subset of our sites with clocks highly synchronized to each other. We identified the day with the most connections between those sites and extracted from the traces a dataset giving the times L_i of each packet loss during those connections. We then constructed plots of L_i versus $L_i \bmod \mu$, and varied μ through the range of $1, 2, \ldots, 120$ sec. We hoped to find a μ for which many of the $L_i \bmod \mu$ clustered about a particular value. However, no compelling modulus emerged. We repeated the analysis for data packets sent to Europe, shown in Table XXII as the most loss-prone Internet path, to test whether perhaps their heavy losses are due in part to a periodic component rather than congestion. Again, we did not find persuasive evidence of frequent periodic losses.

We conclude that periodic losses do not *strongly* dominate TCP packet losses. However, the mismatch between our measurements and those needed to thoroughly examine the question of periodic losses is great enough that we cannot from our evidence conclude that such losses do not regularly occur.

15.4 Loss location

We discussed in Chapter 14 how each network path contains one (or more) “bottleneck” element(s) that limit the maximum rate a connection using the path can achieve. It is natural to assume that this bottleneck element is also the point of congestion along the path, because it has the least amount of one of the network's most important resources, namely bandwidth. Consequently, for a given load in terms of volume of packets to forward along a network path, the bottleneck elements will be the most stressed of those along the path, since they require the most time to service the load. With this assumption, we are again (as in § 15.2) abstracting the intricate, multi-element network path to a presumably equivalent model of a single element that forwards at the bottleneck rate, and at which all significant queueing occurs.

One might think that, with only end-to-end measurements, one lacks sufficient information to verify whether in fact loss occurs at the bottleneck or at some other element. Sometimes, however, we can, as illustrated by Figure 15.14 and Figure 15.15. Both sequence plots reflect data packet arrivals at the receiver, with the packets flowing in steadily at the bottleneck rate. In each plot, one packet has been lost, and the circle indicates where it would have arrived had it not been lost, and had it likewise arrived at the bottleneck rate. In Figure 15.14, its successor arrives in the position where the lost packet would have otherwise arrived. This indicates its successor did *not* queue behind the lost packet, but instead behind the lost packet's predecessor; hence the lost packet must never have made it across the bottleneck link. In Figure 15.15, however, the successor arrives in the same position that it would have, had the lost packet safely arrived too. Thus, the successor *did* queue behind the lost packet at the bottleneck, and we conclude that the lost packet did indeed make it across the bottleneck link, only to be dropped later.

In general, we prefer that packets are dropped *before* the bottleneck, so they do not fruit-

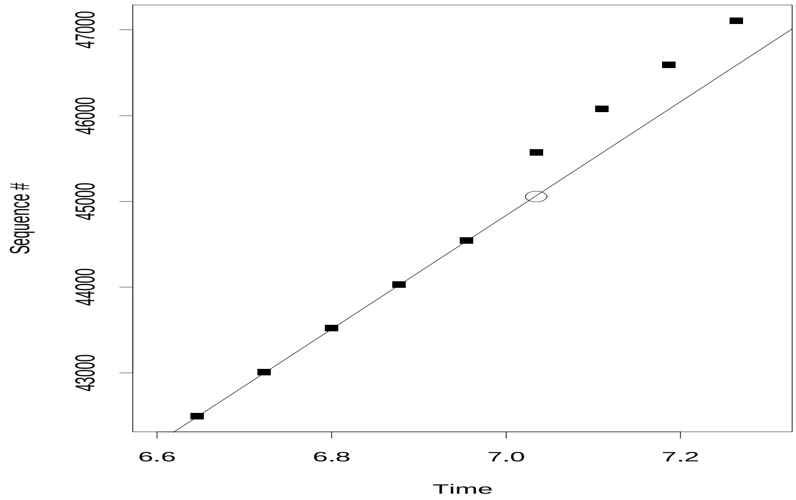

Figure 15.14: Receiver sequence plot showing packet lost at or before bottleneck link

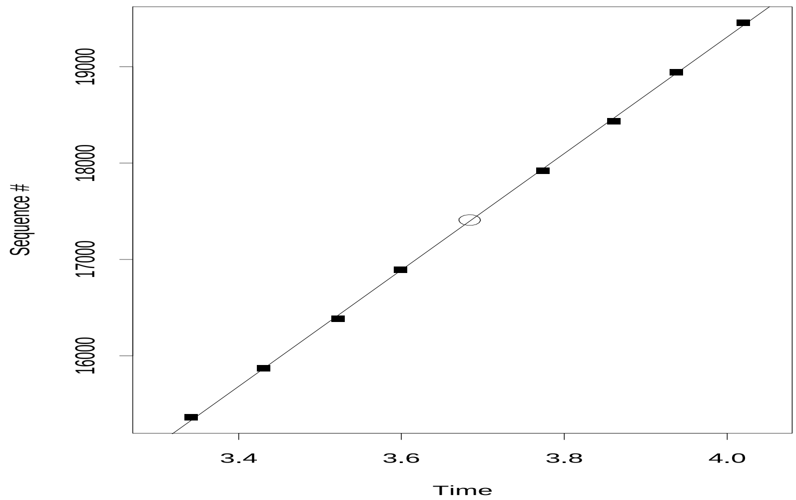

Figure 15.15: Receiver sequence plot showing packet lost after bottleneck link

lessly consume the (usually) scarce bottleneck resource. In this section, we analyze how often this occurs. We first clarify our terminology. We will refer to a packet lost after it has been successfully forwarded by the bottleneck element as occurring "after the bottleneck," while one lost earlier as occurring "before the bottleneck." These latter may have been lost because of a full queue just before the bottleneck element, or may have been lost further upstream. Some network paths may have *multiple* bottlenecks, meaning a number of elements with the same limiting rate. Since our analysis is based on the patterns in Figures 15.14 and 15.15, in the case of multiple bottlenecks we consider only loss after or before the first of the bottlenecks. Loss prior to subsequent bottlenecks will still appear at the receiver as in Figure 15.15, since the data packets will have already been spread out by the first bottleneck.

Our analysis is doomed to be inexact, since effects such as data packet compression (§ 16.3.2) and spurious extra delay often obscure the patterns so clearly evinced in Figures 15.14 and 15.15. But we still aspire to attempt some sort of meaningful analysis, since the basic question of position of loss is an intriguing one, with the potential to reshape our abstractions when analyzing networks.

We proceed as follows. For each lost data packet, we check whether both its predecessor and successor arrived successfully. If not, then we ignore the packet for our analysis, which removes from our possible results the effects of loss bursts. Since we know from § 15.3 that loss bursts are not uncommon, the resulting bias means our results will at best be only qualitative. (We attempted to extend the analysis to include loss bursts, but the ambiguities of whether the next successful packet had to queue behind only *some* of the packets lost in the burst proved too difficult to remove.)

If both predecessor and successor arrived, then we check whether the lost packet was sufficiently "loaded" (§ 15.2) that, upon arriving at the bottleneck, it would find its predecessor waiting in the queue, not yet having begun its service. If not, then we again ignore the packet for our analysis. Doing so assures we only analyze lost packets that would nominally have occupied a full "slot" at the queue, and not a partial slot due to arriving while its predecessor was in the process of transmission across the bottleneck.

If the lost packet was sufficiently loaded, then we check whether its successor was sent soon enough after that, had the lost packet queued at the bottleneck, its successor would have arrived at the bottleneck before the lost packet began *its* bottleneck transmission, and thus the successor would have been delayed a full "slot" in the queue, too. If the successor was sent too late, we again ignore the lost packet for our analysis.

If the successor was sent sufficiently soon after the lost packet, then we next inspect the arrival time of the successor. If it is within $\pm 25\%$ of the time expected had the lost packet never been transmitted (no bottleneck "load" incurred), then we consider the lost packet as having been dropped before the bottleneck. If the successor arrives within $\pm 25\%$ of the time expected had the lost packet indeed loaded the bottleneck, then we consider the loss as occurring after the bottleneck. If the successor's arrival is between these two ranges, then its arrival is "ambiguous," and if its arrival is after (or before) both ranges, then its arrival is "inconsistent," meaning the simple packets-arriving-at-the-bottleneck-rate scenario we envision is inadequate, probably due to downstream queueing.

In both $\mathcal{N}_1$ and $\mathcal{N}_2$, about a third of the losses fit the "inconsistent" category, and almost none were "ambiguous." Of the remaining two-thirds, we find that, in $\mathcal{N}_1$, fully 48% of the losses occurred after the bottleneck. In $\mathcal{N}_2$, the figure falls to 28%. These figures, however, are less than solid in two important ways. First, if a packet is lost before the bottleneck, but its successor queues behind a packet from *another connection* at the bottleneck, then we will still obtain the signature of an after-bottleneck loss. It is difficult to see how to quantify the frequency of this effect given only end-to-end measurement data. Second, our analysis is somewhat skewed by the presence of sites in our study with low-speed Internet connections. For connections involving these sites, the bottleneck will often be immediately at the sender (or before the receiver), so there is little opportunity for loss before (or after) the bottleneck. If we restrict our analysis to only connections with a bottleneck rate exceeding 100 Kbyte/sec, then in $\mathcal{N}_1$ we find 36% of the losses occur after the bottleneck, and 26% in $\mathcal{N}_2$.

From this analysis, we conclude that, for isolated packet losses (not bursts), the assumption that loss occurs at or before the bottleneck link is certainly true more often than not. But if loss position is critical to some analysis, then one must accommodate the possibility of loss occurring after the bottleneck. We also conclude that perhaps 25% of packet loss occurs regretfully late in the network path, meaning that an upstream bottleneck link spent its scarce resources carrying a doomed packet.

15.5 Evolution of packet loss rate

In this section we look at how packet loss rates along an Internet path evolve over time. Our goal is to determine how fruitful it might be to cache packet loss information for Internet paths to better estimate the service we might expect from the paths in the future. For each path in our study, we analyze the evolution of the ack loss rate along the path in several different ways. Clearly, there will be great variation among some of the paths in how the loss rate evolves over time. But we presently limit ourselves to investigating overall patterns of loss rate evolution, aggregated over all of the $\mathcal{N}_2$ connections. We do not analyze the $\mathcal{N}_1$ connections because few of the $\mathcal{N}_1$ paths were measured frequently enough to allow solid analysis.

We first look at how well observing no loss along the path for a 100 Kbyte connection predicts experiencing no loss along the path for another such connection at some point in the future. For each zero-loss connection, c, we compute the pair $\langle \Delta T_c, I_c^z \rangle$, where ΔT_c is the time between that connection and the next successful connection, c', we observed along that path; and I_c^z is an indicator function with a value of 1 if c' also experienced no loss, and 0 if it did.

After constructing these pairs, we sort them on ΔT_c and then compute the probability $P^z(\Delta T)$ that a connection that comes an interval ΔT after a zero-loss connection will also be zero-loss, as follows. Let $I_{(i)}^z$ denote the ith indicator, sorted on ΔT_c. Beginning with $\widehat{P}_0^z = 1$, we run an exponentially-weighted moving average (EWMA) with $\alpha = 0.01$ through the sorted indicators, where the ith value of the average is computed as

$$\widehat{P}_i^z = (1 - \alpha)\widehat{P}_{i-1}^z + \alpha I_{(i)}^z.$$

Let $\widehat{P}^z(\Delta T)$ then be $\widehat{P}_i^z$ for the value of i corresponding to the interval ΔT.

Using $\alpha = 0.01$ means that $\widehat{P}_i^z$ is dominated by the preceding 100 values of I^z, though earlier values still contribute to the smoothing. Our goal is to turn the indicator values into meaningful probability estimates, while still allowing for effects that are localized to different time intervals.

Figure 15.16 shows how $\widehat{P}^z(\Delta T)$ evolves with time. The x-axis gives the time between the first zero-loss connection and the subsequent connection, logarithmically scaled, and the y-axis gives the smoothed probability that the subsequent connection is also zero-loss.

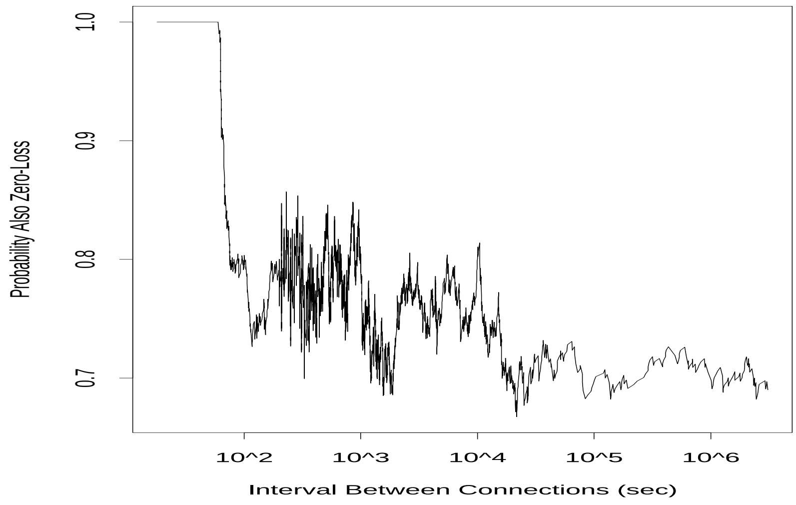

Figure 15.16: Evolution of how well observing a zero-loss connection predicts that a future connection will also be zero-loss

We had very few successive connections in our study separated by less than 60 sec, because the NPDs reuse TCP connection identifiers (to aid in filtering the traffic, per § A.2), and most TCP implementations set a minimum waiting interval on reusing identifiers of 1 minute or more.[6]

Because of the combination of exponential smoothing and very few closely-spaced successive connections, the leftmost portion of the plot exhibits an artifact in terms of a steep dip from probability 1.0 to probability 0.8. Had we instead used an initial probability of $\widehat{P}_0^z = 0.8$, then this spike would disappear. Putting aside the spike, we see that the probability of again observing a zero-loss connection stays at about 0.75 for intervals on the order of a few minutes to a few hours. Above about 6 hours, it approaches what appears to be a "steady state" of 0.70, which continues all the way out to several weeks. Thus, observing a zero-loss connection remains a good predictor of observing future zero-loss connections, even for points in time quite far in the future.

Figure 15.17 shows the same evolution except for the predictive power of observing a non-zero-loss connection rather than a zero-loss connection. The pattern is similar, though the steady state shows signs of declining on time scales of weeks. The "notch" at about 6 hours (21,600 sec) is somewhat puzzling, though it is perhaps simply an artifact, as the region surrounding the notch contains only about 200 points. The notch at four minutes is likewise puzzling: it contains 20% of all of the points, and hence is clearly not spurious, but it is difficult to see what mechanism would lead to less correlation between connections 3-5 minutes apart compared to those further apart. (The comparable notch in Figure 15.16 occurs instead at two minutes, and contains only 3% of the points, so it is perhaps spurious.)

The final aspect of packet loss evolution we look at is how loss rates change over time. For each connection, we compute $\langle T_c, \lambda_c \rangle$, where T_c is the time when the connection began and λ_c

[6]The TCP specification sets this time at 4 minutes, though it provides exceptions for which it can be bypassed [Br89].

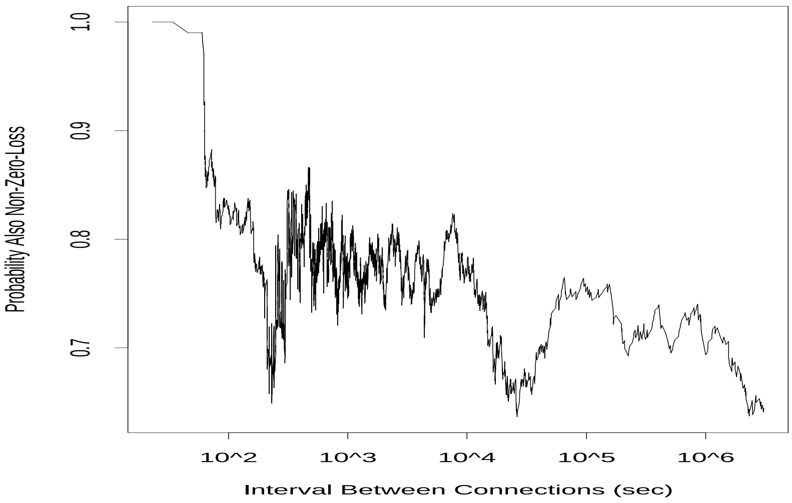

Figure 15.17: Evolution of how well observing a non-zero-loss connection predicts that a future connection will also be non-zero-loss

is the ack loss rate. We then compute for consecutive connections c_1 and c_2 along the same path the pair $\langle \Delta T_{1,2}, \Lambda_{1,2} \rangle$, where:

$$\begin{aligned} \Delta T_{1,2} &= T_{c_2} - T_{c_1}, \\ \Lambda_{1,2} &= |\lambda_{c_2} - \lambda_{c_1}|. \end{aligned}$$

Thus, $\Lambda_{1,2}$ gives the magnitude of the difference in loss rates between the two connections.

Figure 15.18 shows how the EWMA of $\Lambda_{1,2}$ evolves as $\Delta T_{1,2}$ increases, where the smoothing is done with $\alpha = 0.01$ and with an initial value of $\Lambda_{0,0} = 0$. We see an almost immediate jump to a mean difference of $\pm 2\%$ in loss rate, followed by a steady climb up to a difference of $\pm 4\%$ at about 10 hours, followed by a jump to the $\pm 6 - 8\%$ level for larger time intervals, where the variation for very large time scales (weeks) at the righthand edge of the plot may be spurious, due to an exceedingly small number of samples.

From Figures 15.16, 15.17, and 15.18, we conclude that observing no loss along a path is a good predictor that we will continue to not observe loss along the path, even far into the future; that the same holds almost as strongly for observing loss predicting we will observe future loss; but that the farther into the future we wish to project, the more difficult it is to accurately assess the *magnitude* of the loss rate based on the magnitude of the currently observed loss rate. These findings support the notion developed earlier in this chapter that network paths have two general states, a tendency towards loss-free connections ("quiescent"), and a tendency towards lossy connections ("busy"), and provide evidence that both states are long-lived, on time scales of hours, presumably because they are functions of whether the path has adequate capacity for the aggregate traffic delivered to it, and aggregate traffic rates generally change on time scales of hours [PF95]. We also find that, while we may predict future loss rates fairly accurately for time scales of minutes to hours, as time scales grow beyond, our predictive power diminishes.

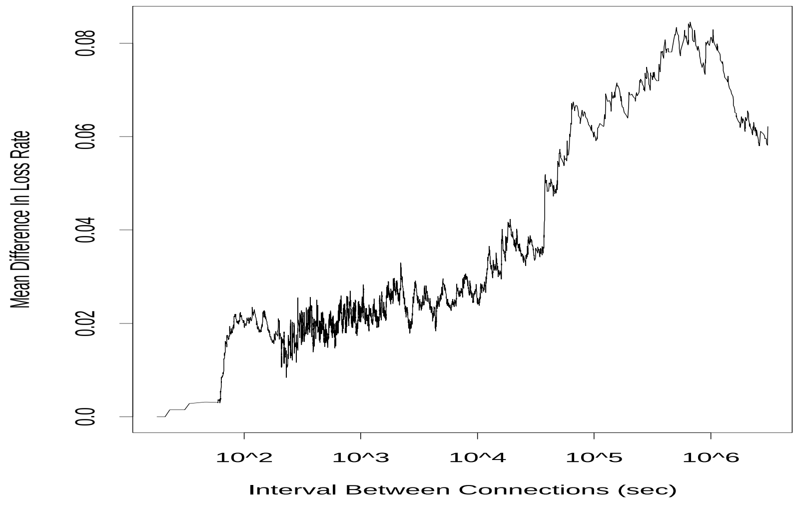

Figure 15.18: Evolution of the mean difference in loss-rate between successive connections along the same path

15.6 Efficacy of TCP retransmission

The final aspect of packet loss we investigate is how efficiently TCP deals with it. Ideally, TCP retransmits any lost data until it is successfully received, but never retransmits unnecessarily, as that would waste network resources. However, the transmitting TCP lacks perfect information, and consequently will sometimes indeed retransmit unnecessarily. For example, TCP acknowledgements are *not* transmitted reliably; so, if a flight of data packets all arrive successfully at the receiver, but all of the corresponding acknowledgements are lost, then the TCP has no choice but to retransmit when the retransmission timer expires.

We analyzed the efficacy of retransmission by the different TCPs in our study as follows. For each connection, we examine each retransmitted packet P_r to see if the data contained in P_r had already been successfully sent.[7] Note that the earlier, successful transmission may not have arrived yet at the receiver at the time of the retransmission; we consider it successful, however, if an earlier transmission of the data *ever* arrives at the receiver.

If P_r contained data that had not previously been successfully transmitted to the receiver, then we term P_r "necessary," otherwise we term it "redundant." *In both $\mathcal{N}_1$ and $\mathcal{N}_2$, about 40% of the retransmissions were redundant!* As an aggregate statistic, this is not a happy number. It means that two times out of five, the TCP should (1) not have retransmitted, and (2) not have cut its congestion window, if the retransmission led it to do so. However, we need to investigate the 40% figure better, since there are a number of different reasons why a TCP might send redundant retransmissions (RRs):

[7]The exact test is whether *all* of the data in P_r had been successfully sent. This fine point can be important if different portions of P_r's data were earlier sent in different packets.

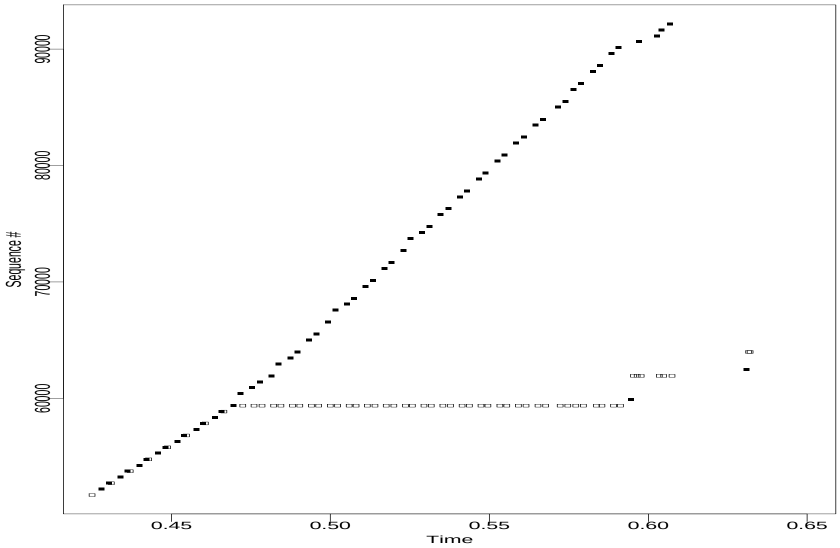

Figure 15.19: Receiver sequence plot showing large number of sequence holes

unavoidable We mentioned earlier that, if the network drops all of the acks for a flight of data packets, then the TCP sender has no choice but to retransmit, since no further feedback will be forthcoming from the receiver.

pathological The packet was a timeout retransmission, but the interval between the data's earlier transmission and this packet's was less than the minimum round-trip time ever seen. Hence, the retransmission timeout used by the TCP was absolutely broken—the receiver did not even have a chance to acknowledge the data—and, furthermore, a simple test by the TCP to make sure that at least the minimum RTT had elapsed would have prevented the redundant retransmission.

coarse feedback Since TCP acknowledgements simply give the highest data sequence number received in-order, when a TCP retransmits with a window larger than one packet (such as during slow-start after a timeout), it may transmit unnecessary packets because the receiver lacks a fine enough feedback mechanism to tell it which above-sequence packets have already arrived. Figures 15.19 and 15.20 illustrate the problem. In the first sequence plot (measured at the data receiver), we see that the sender has a large amount of data in flight, which until about $T = 0.47$ has steadily streamed in. At that point, however, the packet with sequence number 59,905 is lost. Many more packets continue streaming in, but they contain numerous holes where some were lost. The new arrivals generate a torrent of duplicate acks in response. Since, however, the acks only provide coarse feedback to the sender, all the sender really knows is that sequence 59,905 was lost, and many more packets safely arrived—but it does not know which.

The sender retransmits the first missing packet via fast retransmission (§ 9.2.7), and this packet arrives at the receiver just before $T = 0.6$. The receiver duly acknowledges up to the next hole, and even generates some duplicate acks for new data arriving at sequence 90,625

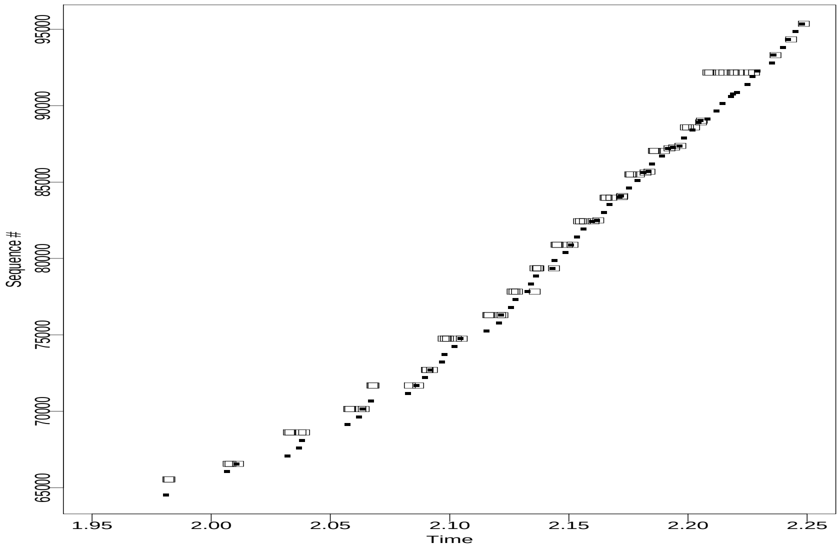

Figure 15.20: Redundant retransmissions subsequent to previous figure

and above (sent due to fast recovery). These in turn lead to a fast retransmission for the next hole, arriving at $T = 0.63$. At this point, however, the sender does not see any more incoming acks allowing it to send more data via fast recovery (and it has halved its congestion window twice, once per fast retransmission event, so it will take a while for more dup acks to inflate the window far enough to enable fast recovery). Consequently, self-clocking ceases and the sender stalls until a retransmission timeout occurs.

Until now, the retransmissions have all been necessary. The retransmissions after the timeout, however, are a disaster, as shown in Figure 15.20. The first packet retransmitted after the timeout was also necessary. Unfortunately, the acks generated by it (shown as large squares in the plot) rapidly open the sender's congestion window due to slow start, and it sends larger and larger flights of packets. Nearly all of these retransmitted packets are unnecessary—all that is really needed is to fill the sporadic holes shown in Figure 15.19. Every duplicate ack in Figure 15.20 corresponds to an unnecessary retransmission, yet because the sender lacks fine-grain information regarding which above-sequence packets the receiver already has, it continues retransmitting to fill the known holes (as indicated by the latest ack it has received), as well as pouring additional, unnecessary packets into the network—23, all told.

The TCP research community has long known about this problem and is in the midst of standardizing a TCP extension to remedy it. With the extension, a "selective acknowledgement" (SACK) option, acks can carry additional information concerning above-sequence packets that have arrived at the receiver (§ 13.1.3). The sender then uses this information to select which packets require retransmission.

We consider an RR as reflecting TCP's "coarse feedback" problem if it occurred *after* the arrival of an ack that itself was sent after the original copy of the data arrived at the receiver. Presumably, had we used SACK, this ack could have conveyed to the sender that the data had

Type of RR	$\mathcal{N}_1$ total	$\mathcal{N}_2$ total	$\mathcal{N}_1$ Solaris	$\mathcal{N}_2$ Solaris	$\mathcal{N}_1$ Other	$\mathcal{N}_2$ Other
% all packets	2%	3%	6%	6%	1%	2%
% retransmissions	43%	38%	66%	59%	26%	28%
Unavoidable	25%	25%	14%	33%	44%	17%
Pathological	2%	7%	3%	11%	0%	2%
Coarse feedback	18%	41%	1%	1%	51%	80%
Bad RTO	55%	28%	81%	55%	4%	1%

Table XXIV: Proportion of redundant retransmissions (RRs) due to different causes

already arrived, and the sender would have avoided the RR.

bad RTO If the RR was prompted by a timeout, and if an acknowledgment for the previously sent data arrives after the timeout retransmission, then the TCP selected too low a value for its retransmission timeout (RTO). The RR could have been avoided simply by waiting longer.

Table XXIV summarizes the prevalence of the different types of RRs in $\mathcal{N}_1$ and $\mathcal{N}_2$. The second and third columns give the overall percentage of the $\mathcal{N}_1$ and $\mathcal{N}_2$ RRs due to each type. The fourth and fifth columns give the same figures if we restrict the analysis to just Solaris TCP senders, since in § 11.5.10 we discussed how it is prone to underestimating RTO and consequently retransmitting too early, so we would expect it to exhibit a higher frequency of "pathological" and "bad RTO" types of RRs than the other TCPs in our study. The final two columns summarize the frequency of each type of retransmission for the non-Solaris TCPs.[8]

We see that a fair proportion of the RRs were unavoidable. (Some of these might, however, have been avoidable had the receiving TCP generated more acks.) We note that for $\mathcal{N}_2$, which, with its bigger windows (§ 9.3), had more opportunity to successfully transmit an ack for part of the window, only about 1/6 of the RRs for non-Solaris TCPs were unavoidable. Clearly it is worth our efforts to first eliminate the avoidable 5/6's.

Pathological RRs could be eliminated with a simple test: if the packet being retransmitted was previously transmitted (or retransmitted) less than one RTT in the past, then simply do not retransmit it. Aside from Solaris, most pathological RRs occur within retransmission epochs, during which earlier RRs lead to enough duplicate acks that the TCP resends data it sent shortly before due to the window advancing. For Solaris, many occurred due to the problems the Solaris TCP timer has with adapting to the true round-trip time, cf. § 11.5.10 and § 11.5.1.

"Coarse feedback" RRs would presumably all be fixed using SACK. The increase in non-Solaris coarse feedback RRs in $\mathcal{N}_2$ is no doubt due to the use of bigger windows in $\mathcal{N}_2$, and hence more opportunity for acks (and, thus, finer feedback) to potentially inform the sending TCP of what

[8]In § 11.5.8 we identified the Linux 1.0 TCP as suffering from many RRs due to its practice of retransmitting all the unacknowledged packets rather than just the first. However, in § 10.5 we discussed how many of the Linux traces could not be unambiguously paired in terms of packet departures and arrivals, precisely because of this retransmission problem. In this section, we confine our retransmission analysis to those traces that we could unambiguously pair, so we can distinguish between the different types of RRs (in particular, "coarse feedback," which depends on whether the original data arrived before a subsequently transmitted and received ack). Consequently, we analyzed very few Linux 1.0 traces and thus their presence does not significantly affect the statistics in Table XXIV.

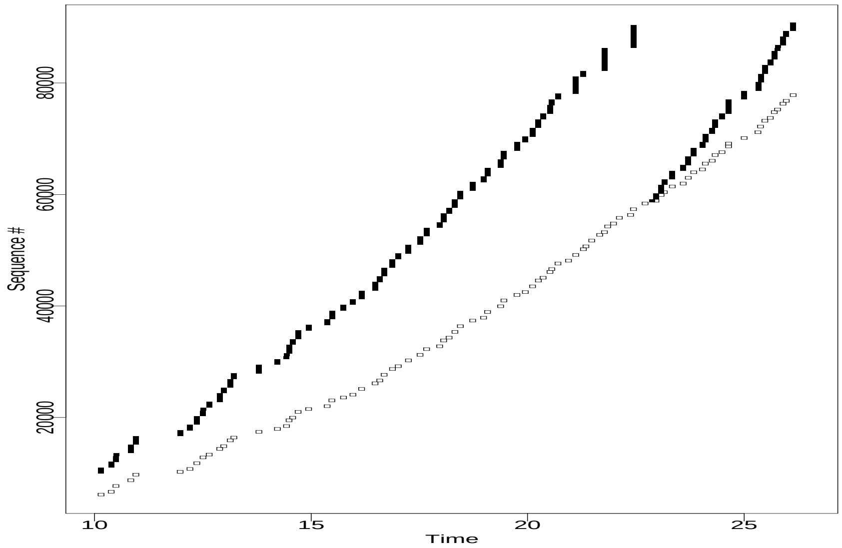

Figure 15.21: Sender sequence plot showing failure of RTO adaption

packets the receiver already has. It is encouraging to see that, aside from Solaris TCPs, deployment of SACK remedies almost all of the avoidable RRs. It makes almost no difference for Solaris TCP, since many of its RRs occur before any ack for the previous transmission of data has arrived from the receiver, due to the Solaris timer adaption problems.

"Bad RTO" RRs indicate that the TCP's computation of the retransmission timeout was erroneous. These are the bane of Solaris TCP, as noted above. More than half of its RRs were due to miscalculating the timeout. Fixing the calculation eliminates 4-5% of *all* of the data traffic generated by the TCP.[9]

The TCP standard requires use of Jacobson's exponentially-weighted moving average (EWMA) round-trip time (RTT) estimate and associated variance estimate ([Br89, 4.2.2.15] and [Ja88]), along with Karn's algorithm for eliminating ambiguous RTT estimates [KP87]. If we assume that the non-Solaris TCPs do in fact implement this algorithm, then from Table XXIV we see that it performs quite well.

Figure 15.21 shows an instance where it failed, or at least where HP/UX 9.05's implementation of it failed. Here the receiving TCP is offering a very large window, to which the sending TCP is rapidly opening its congestion window in the face of no packet loss. The bottleneck link, however, can only support about 7.3 Kbyte/sec, and so the window represents a large mismatch with the correct window size needed to avoid overloading the bottleneck. Consequently, the RTT rises rapidly as packets queue behind their predecessors. During the last five round trips, starting at time $T = 10$, the RTT increases by about 1 second during each trip. The RTO estimation algorithm fails to track this rapid increase, and at time $T = 23$ a retransmission timeout occurs, even though the corresponding ack is just about to arrive. Subsequent acks for the first transmissions of the data then rapidly feed the slow-start sequence begun by the timeout retransmission, and the sending TCP

[9] We note that this problem has already been fixed in Solaris 2.5.1.

promptly resends 63 packets, all redundant. However, we found pathological behavior like that shown in the figure quite rare.

While the standard RTO estimation almost never leads to an unnecessary timeout retransmission, a separate question, unanswered by these statistics, is whether it could be safely modified to be less conservative. At present the timeout often occurs after much more than an RTT elapses. A more aggressive RTO algorithm could potentially lead to higher connection throughput, because timeout lulls would be less costly than they currently are. Yet, if the more aggressive algorithm leads to excessive retransmission during times of RTT fluctuation, then it could contribute to congestion collapse, a major disaster.

Answering the question of how the RTO estimation might be reengineered is a complex problem. The current timer uses coarse-grained (as much as 500 msec granularity) measurements with some subtle adjustments to compensate for the granularity, as well as timing only one packet per flight. A revised timer might take advantage of both higher-resolution clocks and the opportunity to time multiple packets per flight. The first affects the adjustment factors used by the current algorithm, and the second changes the constant used in the EWMA estimator. Because the issues are complex, we leave this interesting question for future work.

In summary: assuring standard-conformant RTO calculations and deploying the SACK option together eliminate virtually all of the avoidable redundant retransmissions. The few remaining RRs are rare enough to not present, overall, any serious performance problems.

The last aspects of TCP retransmission we investigate are the patterns of packet loss during fast recovery sequences. The TCP fast recovery mechanism, described in § 9.2.7, works best when only a single packet out of a flight is lost. When multiple packets in one flight are lost, the fast recovery mechanism generally will not suffice to retransmit all of the missing packets, and the TCP transfer will subsequently stall until a retransmission timeout, seriously diminishing throughput [FF96, Ho96]. It was this problem that motivated the development of the SACK option, which allows a TCP to efficiently recover from multiple losses.

A separate fast recovery problem occurs when the retransmitted packet is also lost.[10] When this happens, the TCP will again stall until a retransmission timeout expires. In some circumstances, and depending on the algorithm used by a TCP to act upon information it acquires by using the SACK option, a TCP using SACK can avoid this timeout by determining that the retransmitted packet was itself lost, and retransmitting it again.

While these problems have been recognized for quite a while, no hard data has been available in order to gauge the degree to which they actually present difficulties for Internet connections. We analyzed the $\mathcal{N}_1$ and $\mathcal{N}_2$ measurements to provide such data, as follows. For each packet retransmitted using the fast recovery mechanism, we tallied whether the retransmitted packet was lost or successfully arrived at the receiver, and also counted the number of outstanding (unacknowledged) packets at the time of the retransmission that were lost.

In $\mathcal{N}_1$, out of 1,178 packets retransmitted using fast recovery, only 3.9% were themselves lost. In $\mathcal{N}_2$, 15,444 packets were retransmitted using fast recovery (a significantly higher proportion of all of the retransmissions than in $\mathcal{N}_1$, due to the use of bigger windows in $\mathcal{N}_2$, per § 9.3). Of these, only 4.5% were also lost. (These proportions are quite close to the unconditional loss rates we examined in § 15.1, and much lower than the conditional loss rates examined in § 15.3, indicating

[10] This problem also occurs for TCPs that implement "fast retransmit" (§ 9.2.7) but not fast recovery. However, for simplicity, we will only use the term "fast recovery" in our discussion.

that congestion often drains on time scales of RTTs.) Thus, we conclude that the second concern discussed above is, in practice, not an especially serious problem.

However, in both $\mathcal{N}_1$ and $\mathcal{N}_2$, one third of the time more than one packet was lost in the flight prior to a fast recovery, and about 15% of the time, more than two packets were lost. These proportions are high enough to give solid support for refining the fast recovery mechanism (such as by adding SACK, or the modifications discussed by Hoe [Ho96]) in order to better cope with multiple packet losses within a single flight.

Chapter 16

Packet Delay

The final aspect of Internet packet dynamics we analyze is that of packet delay. Delay variation is arguably the most complex element of network behavior to analyze—with loss, for example, the packet either shows up at the receiver or it does not, while with delay there are many shades of possibility and meaning in the time required for a packet to arrive. Likewise, delay variation is potentially the richest source of information about the network, as one of the principle elements contributing to delay is queueing within the network, which is of vital importance in understanding how network capacities evolve over time.

Any accurate assessment of delay must first deal with the issue of clock accuracy, as all delay measurement stems from clock measurements. Unless we tightly calibrate the clocks used for delay measurement, or, equally important, recognize which clocks cannot be well calibrated and discard the corresponding measurements, we cannot know that the subsequent analysis reflects true network behavior and not spurious or misleading clock artifacts. It was these considerations that led us to the lengthy efforts developed in Chapter 12.

We proceed as follows. In § 16.1 we briefly discuss round-trip time (RTT) variation in our measurements, which plays a central role in transport protocol behavior. From the point of view of network path analysis, however, a packet's one-way transit time (OTT) is more fundamental, particularly since RTT measurements conflate delays along the forward and reverse path. Consequently, we devote the remainder of the chapter to OTT analysis. In § 16.2, we discuss OTT variation in large-scale terms. We then in § 16.3 turn to packet timing *compression*—network events in which a group of packets arrive at the receiver more closely spaced together than when they were sent. Compression is a significant event because it introduces potentially misleading discrepancies between the timing of events at the sender and at the receiver, clouding the ability of one endpoint to assess conditions perceived at the other. In § 16.4 we then tackle estimation of the amount of queueing packets encounter during their transit. We attempt to determine the *time scales* associated with queueing, but find wide variation. Finally, in § 16.5 we look at the relationship between queueing delays and *available bandwidth*—the transfer rate the network can sustain for a connection, given the network's current load.

16.1 RTT variation

16.1.1 The role of RTTs

A transport connection's round-trip time (RTT) plays a central role in the connection's behavior. First, a reliable transport protocol such as TCP needs to decide how long to wait for an acknowledgement of data it has sent before retransmitting the data. There is a basic tension between wanting to wait long enough to assure that the protocol does not retransmit unnecessarily, versus not wanting to wait too long so as to unduly delay the connection when in fact retransmission is needed. Our analysis of the Solaris 2.3/2.4 TCP in § 11.5.10 highlights how unfortunate it can be to err on the side of retransmitting too quickly. Network researchers have made considerable efforts in studying how to set a connection's retransmission timeout (RTO), and early problems with TCP's RTO computation identified by Zhang [Zh86] have for the most part been rectified by the work of Karn and Partridge in eliminating ambiguous RTT measurements [KP87], and by that of Jacobson in introducing exponentially-weighted moving averages to estimate both RTT and its variance [Ja88].

The second way in which a connection's RTT influences the connection's behavior concerns the important notion of *bandwidth-delay product* (BDP). A connection's BDP is the product of ρ_A, the available bandwidth, measured in bytes/sec, with τ, the RTT, measured in seconds. The result is a number $B = \rho_A \cdot \tau$ of bytes indicating how much data the connection must have in flight to fully utilize the available bandwidth. A simple way to understand this relationship is to consider that, to fully utilize the available bandwidth, the connection must send ρ_A bytes every second, and thus it must send $\rho_A \cdot \tau$ bytes every round-trip time in order to achieve this goal. A round-trip time, however, exactly corresponds to one cycle of send-and-receive feedback. This relationship, in turn, is directly reflected in the connection's *window* (§ 9.2.2)—the current window controls how much data the connection can have in flight at any given moment, and the window can only change due to feedback for the currently in-flight packets after one RTT has elapsed, since no feedback can arrive sooner than that. Thus, B gives the size of the window the connection must use to fully utilize a bandwidth of ρ_A.

We must, however, make a crucial distinction between these two different roles of RTT in a connection's behavior. For the first role, regulating retransmission, the RTT of interest is how long it might take for a packet to reach the receiver and the corresponding acknowledgement to return to the sender—the *maximum* RTT. For the second role, the RTT of interest is the *minimum* time required for packets to traverse the network path to the receiver and for acks to return. The *larger* values possibly observed for the *actual* amount of time required in general reflect *queueing* along the network path. It does *not* improve a connection's throughput to use such a larger RTT when computing B; it instead only adds to queueing along the path. This observation motivated the development of TCP Vegas [BOP94], in which a significant increase in measured RTT is interpreted as due to using too large a window and adding to queueing along the path, and thus calling for a decrease in the window size to diminish the queueing.

16.1.2 RTT measurement considerations

When discussing RTT times, we must bear in mind that larger packets require larger transmission times, proportional to the bottleneck bandwidth. The effect, naturally, is most apparent on slow links. Accordingly, we need to make sure we do not confuse RTT variation due to packet size with RTT variation due to queueing.

Another consideration is that, if we measure RTT as simply the difference in time between when a packet is sent and when a corresponding reply returns, then we will include in the measurements "response delays" at the receiver (§ 11.6.4). For many purposes, doing so is appropriate, since the roles played by RTT above both concern quantifying the *feedback* time scale, and this includes both the network's delays and those of the receiver. If, however, we wish to discuss only the network's contribution to the feedback time scale, then we need to deduct the response delay from the measured RTT. `tcpanaly` can do this since it knows how to pair packets with their responses. However, we argue that the network's contribution to delay is best studied in terms of one-way transit times (OTT), since doing so allows for the possibility of asymmetries along the two directions of the network path, which we find in § 16.2.3 are in fact common. So, for our RTT analysis, we do not deduct response delays from the measurements, that we might study the entire "closed loop."

Finally, we note that RTT can be measured in two different ways: as the amount of time elapsed between when a TCP sends a packet and when it receives an acknowledgement in response to that packet, or as the time between when a TCP sends an acknowledgement and when it receives the packet liberated by that acknowledgement (§ 11.3.1). As we might expect, overall we find these two values to be very close to one another, except for variations due to "response delays" (§ 11.6.4). (They also can appear different if the clocks at the sender and receiver run at significantly different rates, per § 12.7.7.) In the remainder of this section, we confine our analysis to RTTs measured at the sender.

16.1.3 RTT extremes

Extremes of network behavior are always interesting to consider, since they sometimes challenge the assumptions made by our mental models of how networks "really" work. For example, some might find RTTs larger than a few hundred milliseconds exceedingly unlikely—where could a packet spend all that time?—and thus best treated as pathological events rather than part of the regime we must accommodate as "normal." (We saw how dangerous this can be in Figure 11.9.)

Our data is inappropriate for exploring the full range of RTTs one finds in the Internet, since the set of sites in our study is small, and we would expect RTT extremes to be governed for the most part by geography. This is especially the case for network paths that include satellite links, as these can add hundreds of milliseconds due to the propagation delays up to and back down from the satellite.

However, while geography certainly dominates upper RTT extremes, it is not the only factor. To our surprise, we found that one site in our study, `oce`, experiences extremely high delays for many of its connections. 50% of its connections had a minimum RTT of over 1 sec.

`oce` is sited in the Netherlands. One striking connection came from `wustl` in North America. It never observed an RTT less than 4.4 sec![1] Another came from `unij`, never experiencing an RTT below 2.3 sec—yet `unij` is also in the Netherlands! A `traceroute` from `unij` to `oce` reveals that the route stays wholly within the Netherlands. Furthermore, it shows that all of the delay occurs at the hop between NLNet, the Netherlands Internet backbone, and the `oce` site itself. The

[1] Alas, `wustl` is a Solaris 2.4 site. Its RTO timer had great difficulty accommodating the large RTT, per Figure 11.9. During the first minute of the connection, before the timer finally adapted, it sent 31 new data packets and 51 retransmissions, all but one unnecessary. One packet was retransmitted seven times!

cause of this large delay, which we discussed in § 12.7.8, remains unexplained, despite investigative efforts by staff at the `oce` site. It highlights, however, how commonplace—and often correct—assumptions concerning network behavior can be violated in unexpected ways.

Even after eliminating `oce`, we still find some striking RTT extremes. Connections involving `austr2` experienced minimum RTTs as high as 1.85 sec (to a host in California).[2] If we remove `austr2`, then, curiously, the next highest extremes involved not international traffic but connections with both endpoints in the United States. One, from `wustl` to `adv`, never saw an RTT lower than 1.2 sec, even though a connection ten minutes earlier had a minimum of 156 msec, and one 25 minutes later was back to the typical value of 47 msec. Unfortunately, we do not have a `traceroute` measured right at the time of the anomalous connection. Ones fifteen minutes earlier and 80 minutes later show no anomalies and both report an RTT of about 44 msec.

The most extreme RTT connection in $\mathcal{N}_1$ involved not `korea`, for which we might expect high RTTs (and, indeed, it had plenty), but `nrao` and `bsdi`, in Virginia and Colorado. This connection had a minimum RTT of 1.4 sec and a median value of 2.1 sec. While in § 6.9 we gave an example of a circuitous route involving `bsdi`, `traceroute` reported its RTT as only about 160 msec, much less than observed by this connection; so, we do not have an explanation for what took the packets so long.

So far in this section we have focussed on the *minimum* RTT observed during a connection, which is important for correctly determining B, the bandwidth-delay product. For computing RTO, the connection's retransmission timeout, we instead are interested in the *maximum* RTT, which we now look at briefly. (As discussed in § 15.6, we do not undertake a detailed analysis of how we might modify TCP's RTO algorithms to increase their performance, as this is a complex problem.)

We would expect that RTT maxima can rise very high for connections with slow bottleneck links and many available buffers at the bottleneck. In such cases, the sending TCP will not receive a packet loss signal until it has exhausted the available buffer. For a slow link, a significant amount of buffer can translate into a huge delay as packets finally wend their way through the queue.

The largest apparent RTT we ever observed was 23.8 sec, for a SYN packet and its accompanying SYN-ack. This was not, however, a true RTT: the receiving SunOS 4.1 TCP generating the SYN-ack was retransmitting it in an attempt to establish the connection, and its timer backed off first to 6 sec and then to 24 sec. At the same time, the sender, also a SunOS 4.1 TCP, was backing off its retransmission timer for the original SYN. The two timers were slightly out of phase. Consequently, just before the sender reached the 24 sec retransmission, a SYN-ack arrived from the receiver, leading to the huge apparent RTT. We mention this anomaly because some modifications to TCP such as Hoe's in [Ho96] suggest using the RTT timing for the SYN packet as a quick estimate of the path's true RTT. Such schemes must take care not to get fooled by SYN-ack retransmissions. In this particular case, use of Karn's algorithm would have discarded the RTT measurement as ambiguous [KP87]. However, had the retransmitted SYN-ack arrived just before the *first* retransmission of the SYN (i.e., just before the 6 sec timer expired), then even Karn's algorithm would have accepted the measurement, since the algorithm is predicated upon the assumption that acks are not retransmitted. Finally, we note that Hoe's scheme uses the RTT to estimate B, the bandwidth-delay product. Using a value of 6 sec instead of the correct value of 220 msec would grossly overestimate B, leading to the connection overestimating the window it should use. Hoe's scheme, however, could be easily modified to use a more robust initial RTT estimate, since it does not make any decisions based on

[2] `austr2`, alas, is also a Solaris 2.4 site . . .

B until it has received a flight of 3 closely spaced acks. At that point, there should have been ample opportunity to estimate RTT better.

Putting aside anomalies due to SYN-ack retransmissions, we find that the largest true RTT in our study was 15.1 sec, for a connection involving `oce`. We discussed above `oce`'s peculiarly large RTTs, and in § 12.7.8 the puzzling interplay between the transit times of packets and acks in its connections, so we will not further analyze `oce`-connection RTTs here. If we eliminate `oce`, then we find the next largest RTT comes from the 12-second packet reordering event discussed in § 13.6. Putting aside this pathology, we finally find a "normal" extreme RTT, not due to any unusual network dynamics, of 7.9 sec (involving a connection to `lbli`, which has a low-speed Internet link with a lot of buffer space). A few others range above 6 sec, including one from a high-speed connection between `sintef2` in Norway and `austr` in Australia.

16.1.4 RTT variation during a connection

Another way to characterize RTT extremes is in terms of the variation we observe in RTT over the course of a connection. Our interest lies in whether we can develop a "rule of thumb" such as "it is rare to observe a maximum RTT more than double the minimum RTT." This sort of empirical finding would aid in considering how transport protocols can best adapt to network conditions.

We first note that connections with slow bottlenecks can often experience great swings in RTT as their own packets pile up at the queue for the bottleneck (§ 16.1.3). While such connections are an important consideration for general-purpose transport protocols, for our purposes we eliminate any connection with an estimated ρ_B less than 100 Kbyte/sec, so that we might focus on RTT variations not heavily dominated by the connection's own behavior. We also eliminate connections between `sintef1` and `sintef2`, as they are sited very close together and thus much more easily exhibit large relative swings in RTT, even though in absolute terms the swings are quite small.[3]

After these eliminations, in $\mathcal{N}_2$ we are left with 12,486 connections. Figure 16.1 shows the distribution of the ratio between their maximum RTT and minimum RTT, log-scaled. We compute RTTs from the TCP sender's perspective, using the time required to receive an acknowledgement for a full-sized packet.

The distribution shows great variation, with a median ratio of 2.2:1 (mean of 3:1), but the upper 5% have ratios of 6.7:1 and higher. The entire upper 50% fits closely to a Pareto distribution with $\alpha = 2.1$, shown with a log-log complementary distribution plot in Figure 16.2 (we discussed these plots in § 15.3). A value of $\alpha > 2$ means that the ratio has finite variance, and this is probably due to the fact that the maximum RTT is bounded by the amount of buffer space available along the network path. However, the great degree of variation means that, without additional information, we cannot accurately predict the relationship between the minimum and maximum RTTs.

The ratios exhibit one other striking distribution. If we instead consider the ratio of the *minimum* RTT to the *maximum*, then the corresponding distribution is very nearly normal. Figure 16.3 shows this distribution, with a normal distribution fitted to the mean and variance shown by a dotted line. Figure 16.4 shows a Q-Q plot of the same fitted normal, with the line corresponding to slope 1 and offset 0. Clearly, the agreement is quite good except in the tails. Unfortunately, an

[3]The pair `lbl` and `lbli` do not exhibit this problem because `lbli`'s low-bandwidth ISDN link leads to fairly large RTTs between the two sites.

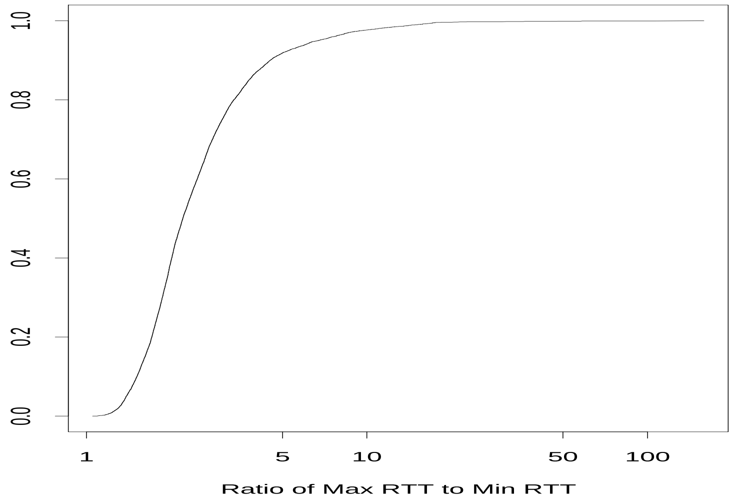

Figure 16.1: Distribution of the ratio between a connection's maximum RTT to minimum RTT

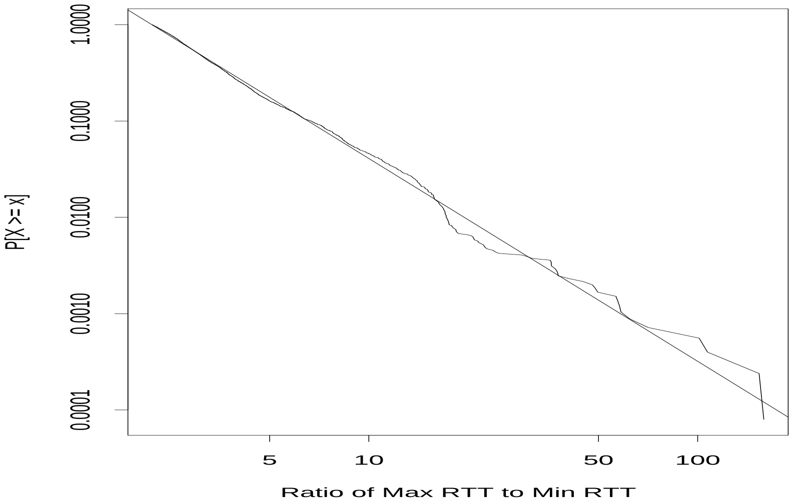

Figure 16.2: Log-log complementary distribution plot of max-min RTT ratio

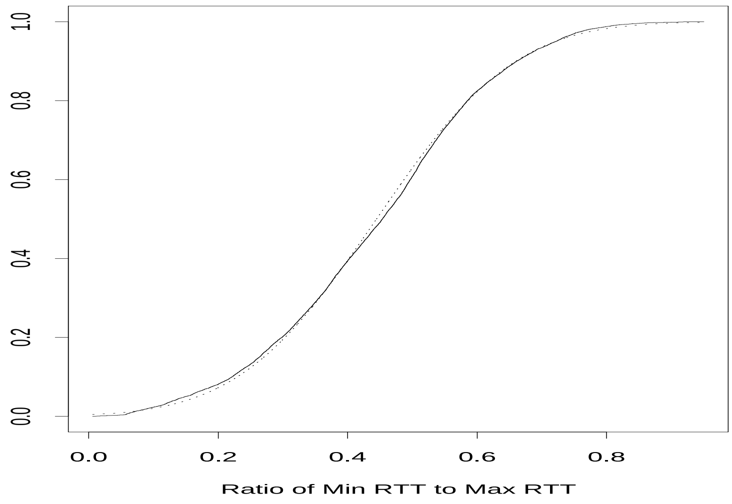

Figure 16.3: Distribution of inverse ratio (minimum RTT to maximum RTT)

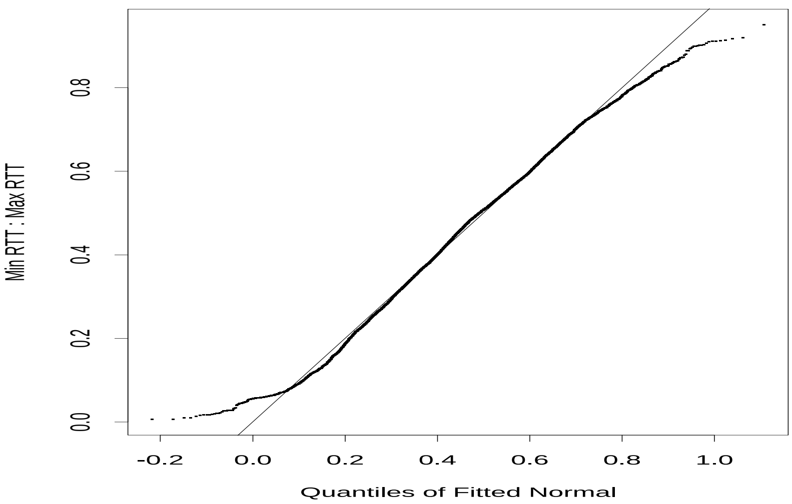

Figure 16.4: Q-Q plot of ratio of minimum RTT to maximum RTT (y-axis) versus fitted normal distribution (x-axis)

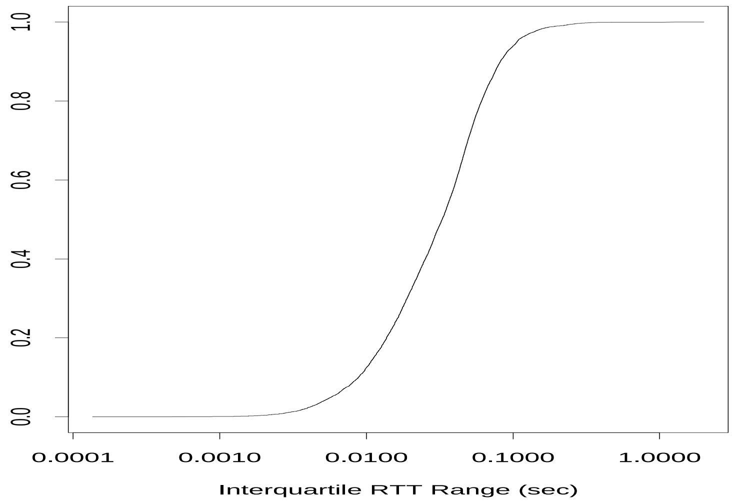

Figure 16.5: Distribution of RTT interquartile range

interpretation for this fit (or for the corresponding Pareto fit) eludes us. As with the elusive exponential fit to data packet loss rates (§ 15.2), we mention the fit here in hopes that it might stimulate further research.

We finish with a look at less extreme RTT variation: the interquartile range (75th percentile minus 25th percentile), IQR. This range gives a much more robust statistic in the sense of being insensitive to extreme values. We are particularly interested in IQR as an aid in estimating the maximum RTT, as this has immediate applications for computing retransmission timeouts (RTOs).

Figure 16.5 shows the distribution of IQR, and Figure 16.6 shows the distribution if we normalize to the minimum RTT. Both plots use a logarithmic scale on the x-axis. We see a wide range of variation, with the lower and upper 5% tails of the absolute range spanning 6 msec up to 106 msec, and, with normalization, the same tails range from a factor of 0.046 up to a factor of 1.23.

The interquartile range is in many ways analogous to a robust version of standard deviation [Ri95]. Consequently, we interpret the wide range of variation as supporting the argument that RTT estimation (for purposes of computing timeouts, for example) must include a notion of variation in addition to estimating the mean or minimum value. Jacobson's estimator does exactly this for TCP [Ja88].

In Figure 16.2 we found that maximum RTTs often are much larger than minimum RTTs. We might wonder, though, whether this discrepancy can be reduced if expressed in terms of RTT variation. For example, it could be the case that the maximum is generally less than n times IQR above the minimum. Unfortunately, this does not appear to be the case. Figure 16.7 shows the distribution of the difference between the maximum and minimum RTT, normalized by dividing by IQR. Again, the x-axis is scaled logarithmically, indicating a wide range of variation. Furthermore, normalization has diminished but not eliminated the Pareto distribution for the upper tail. Instead of occupying a full 50% of the distribution, it now occupies the upper 20%, with $\alpha = 1.84$, within the domain of infinite variance. Finally, these results do not change appreciably if we look at the

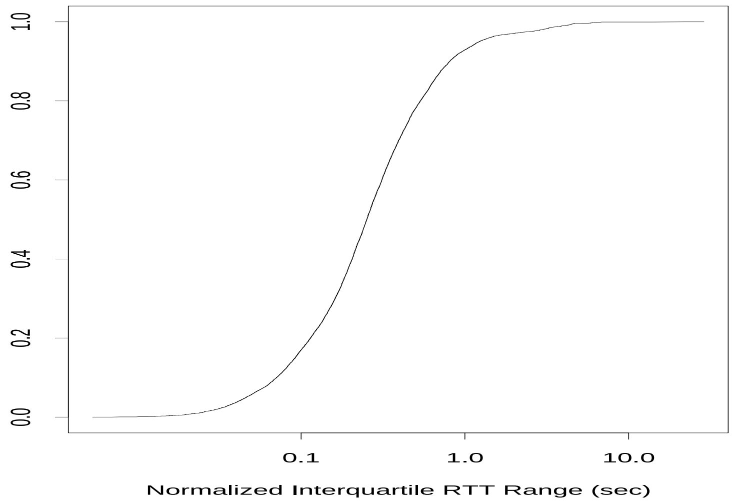

Figure 16.6: Distribution of RTT interquartile range, normalized to minimum RTT

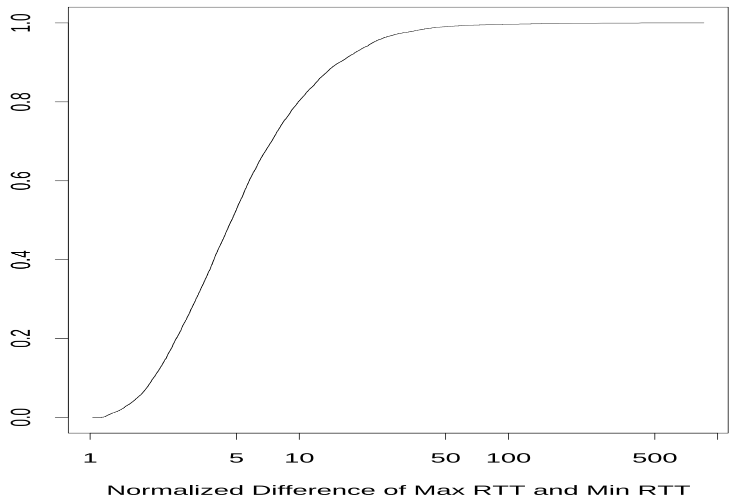

Figure 16.7: Distribution of difference between maximum RTT and minimum RTT, normalized by interquartile range

normalized difference between the maximum RTT and the *median* RTT, rather than the minimum RTT.

From Figure 16.7 it appears that the combination of minimum RTT and interquartile range is inadequate for estimating maximum RTT. TCP RTO estimation is based on similar information, i.e., the estimated RTT mean and standard deviation. Yet, we should not conclude from this that TCP's estimation algorithm cannot work, because the algorithm *updates* its estimates as the connection progresses, using exponentially-weighted moving averages to incorporate new information. Consequently, it has opportunities to *adapt*, while the preceding analysis is *static*. Again, as discussed in § 15.6, we do not undertake here a detailed analysis of how well TCP's RTT estimation algorithm performs, as doing so involves a number of subtle issues.

16.2 OTT variation

For the remainder of this chapter, we focus on one-way transit times (OTTs). Any accurate assessment of delay must first deal with the issue of clock accuracy, from which all delay measurement stems. This problem is particularly pronounced when measuring OTTs since doing so involves comparing measurements from two separate clocks. It was primarily to this end that we undertook the efforts described in Chapter 12 attempting to assure that we can soundly gauge the trustworthiness of the packet timestamps. The subsequent analysis we discuss was always done after first using these algorithms to reject or adjust traces with clock errors.

OTT variation was previously analyzed by Claffy and colleagues in a study of four Internet paths [CPB93a]. They found that mean OTTs are often *not* well approximated by dividing RTTs in half, and that variations in the paths' OTTs are often asymmetric. From our data we cannot confirm their first finding, but we discuss the asymmetry finding shortly.

16.2.1 Why we do not analyze OTT extremes

We do not investigate extreme OTT variation, as we did for RTTs in § 16.1.3, for two reasons. First, most of the RTT extremes are due to network delays, and, in particular, extreme OTTs, so the OTT results are very similar to the RTT results.[4] Second, our absolute OTT values were derived using the approximation that we could rectify clocks in our study by dividing RTTs in half (Eqn 12.5 in § 12.5.1). We know from the Claffy et al. study, and from our earlier results on routing asymmetries (§ 8), that this approximation is often erroneous, and we noted in the derivation that consequently we must refrain from analyzing the absolute OTT values themselves.

16.2.2 Range of OTT variation

Our measurements do, however, let us accurately assess *variations* in OTT. In doing so, we will always distinguish between ack OTTs and data packet OTTs, as we expect the latter to show significantly more variation due to their queueing load. Figure 16.8 shows the distributions of IQR and max-min variations in OTTs for $\mathcal{N}_2$ data packets and acks. Again, we have limited our

[4] This would not have been the case if RTT extremes were due to delays by the TCP endpoint, or combined increases in delay along the two directions of the network path. But neither of these is the dominant effect.

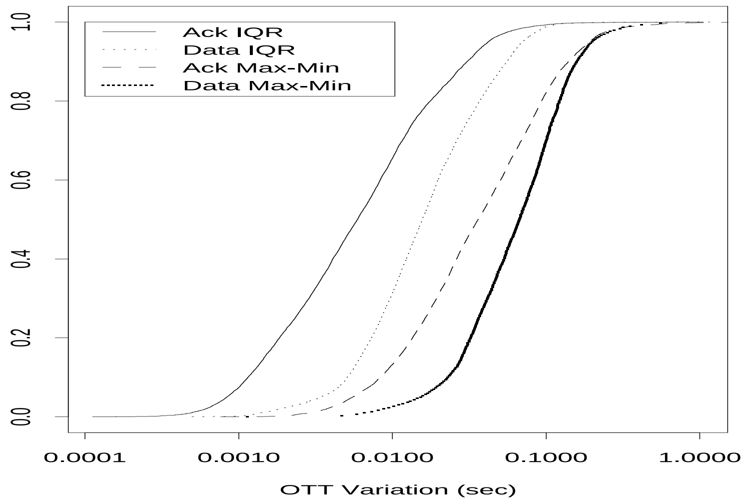

Figure 16.8: Distribution of interquartile and max-min OTT variation

analysis to connections with a bottleneck bandwidth exceeding 100 Kbyte/sec, and have removed those between `sintef1` and `sintef2`.

The x-axis reflects logarithmic scaling; so, as with many aspects of RTT variation, we see a wide range of variation. For example, for data packets the median ratio between the max-min variation and IQR is 3.5:1, and the upper 5% tail exceeds 13:1. For acks, the numbers are higher, the median being 5:1 and the upper 5% tail at 29:1. The difference lies in data packets having a larger IQR to begin with, due to OTT variation caused by the connection's own queueing; for acks, IQR is fairly tame, so the same absolute OTT extreme will be relatively larger when compared to the IQR.

As with normalized RTT variation (Figure 16.7), much of the distribution of the ratio between maximum OTT variation and IQR fits well to a Pareto distribution, for both data packets and acks. Here, the fit is to the entire upper 50% of the distribution, and the α's are well below 2, reflecting sometimes enormous variation.

16.2.3 Path symmetry of OTT variation

We now turn to the relationship between OTT variation on the forward path and that on the reverse path. For $\mathcal{N}_2$, we find that the coefficient of correlation, η, between the max-min OTT variations of the data packets and the corresponding acks is about 0.1—quite weak, though not negligible. For IQR, it drops to 0.06, and for the max-min variation divided by IQR, it drops still further, to 0.02.

However, these statistics do not tell the whole story. As noted above, the forward path is often perturbed by the queueing load of the connection's data packets. We can instead look at OTT variation for only unloaded packets (where a packet is considered unloaded if it does not satisfy Eqn 15.5). Such packets did *not* queue behind their predecessors, unless cross traffic delayed their

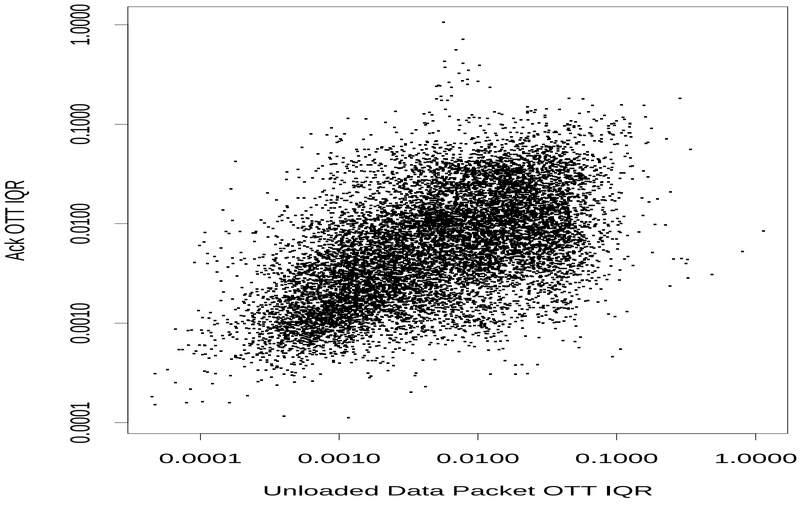

Figure 16.9: Scatter plot of interquartile ranges of unloaded data packet OTT variations (x-axis) versus acks (y-axis)

predecessors. If we analyze only unloaded packets on the forward path, then η between the IQRs of the forward and reverse variations rises to 0.18, considerably more substantial. η for the logarithms of the IQRs is 0.55, indicating that the order of magnitude of the variation along one path is a good predictor of the order of magnitude of the variation encountered along the other.[5] Figure 16.9 shows a scatter plot of the forward path IQR variation for unloaded data packets, versus the ack IQR variation. Note that both axes are log-scaled.

The correlations appear to indicate that delay variations along both directions of an Internet path are indeed coupled, albeit weakly. However, we must investigate a bit further. It could be the case that only *some* Internet paths have coupled variations, while most do not. In particular, we found in § 15.1 that European sites have higher loss rates than those in the United States, and that the paths from Europe to the U.S., and, particularly, from the U.S. to Europe, have the highest loss rates. So it could easily be that traffic between the U.S. and Europe, which traverses in each direction the highly congested trans-Atlantic links, experiences similar delay variations in both directions; while other traffic does not.

To test this effect, we repeated the above analysis with only those $\mathcal{N}_2$ connections between two sites in the U.S. We found that the correlations were only slightly weaker, indicating that the effect has only a mild influence.

In summary: if we know the OTT variation along one direction of a path, then we can fairly well predict the order of magnitude of the variation along the other direction. Predicting the variation to a finer degree is difficult. However, if we are interested not in the intrinsic delays along the path, but the delays actually experienced by a TCP connection, which include variations induced by the connection's load (i.e., its packets queueing behind their predecessors), then prediction is very

[5] If we normalize the IQRs by the round-trip times, the coefficients of correlation do not change much (rising to 0.22 and falling to 0.50, respectively).

difficult: the two directions are nearly uncorrelated.

16.2.4 Relationship between loss rate and OTT variation

It is natural to expect that delay variation might be closely correlated with packet loss, because, whenever packets are delayed in the network, they must be stored somewhere, and that storage will have a finite capacity. Thus, if delay climbs high enough, loss ensues as buffers become exhausted. However, this relationship can be obscured if routers have enough buffers to absorb considerable delay variations. It can also be obscured because delay variation derives from the *end-to-end* concatenation of variations at each hop along a path, while loss is presumed to be governed by one or perhaps a few overloaded elements along the path. Hence, many elements will contribute to delay variation but not to loss.

To investigate the relationship between delay variation and loss, we look at how the IQR of ack OTT variation correlates with the loss rate experienced by the acks. (We confine our analysis to acks to avoid the complications introduced by higher data packet loss rates due to the load they present to the forward path, per $\S$ 15.2.)

Overall, we find $\eta = 0.22$, indicating a definite, but not overly strong, linkage. However, much of the linkage comes from low OTT variation being coupled with experiencing *no loss*, a situation we referred to in $\S$ 15.1 as "quiescence." If we confine our analysis to those connections experiencing at least one loss ("busy"), then η drops to 0.12. Figure 16.10 shows the corresponding scatter plot. The plot shows some apparent structure: the region corresponding to a very low loss rate (on the y-axis) appears separate from the rest of the plot. However, this difference is a granularity artifact. The log scale highlights the difference between losing a single ack and losing two ack, since the latter corresponds to twice the ack loss rate of the former. Setting aside this artifact, we conclude that there is no strong relationship between OTT variation and loss rate.

If we log-transform both the IQR and the loss rate, then η climbs to 0.35, indicating that the order of magnitude of the IQR is a fairly good predictor of the order of magnitude of the loss rate, but nothing finer. These statistics are virtually unchanged if we confine our analysis to connections between U.S. sites, so the effect is not being skewed by the trans-Atlantic or European sites, which differ in their loss patterns ($\S$ 15.1).

Finally, if we normalize the delay variation IQR by the connection's round-trip time, then correlation *decreases*, and, for "busy" connections, the two become uncorrelated, with $\eta = -0.02$.

We conclude that the linkage between delay variation and loss is weak, though not negligible. Unfortunately, from our data it is difficult to discern which of the two effects mentioned at the beginning of this section weakens the linkage: routers having large amounts of buffer space, or the end-to-end chain accruing a number of small variations into a single, considerably larger variation.

16.2.5 Evolution of OTT variation

We now look briefly at how OTT variation evolves with time. To do so, we follow the methodology used in $\S$ 15.5 to assess how loss rates evolve with time. For each connection c between the same source and destination, we compute the pair $\langle \Delta T_c, |\Delta\sigma_c| \rangle$, where ΔT_c is the time between that connection and the next successful connection, c', we observed along that path; and $|\Delta\sigma_c|$ is the absolute value of the difference between the IQRs of the ack OTT variations for c and c', where each IQR is normalized by the connection's round-trip time.

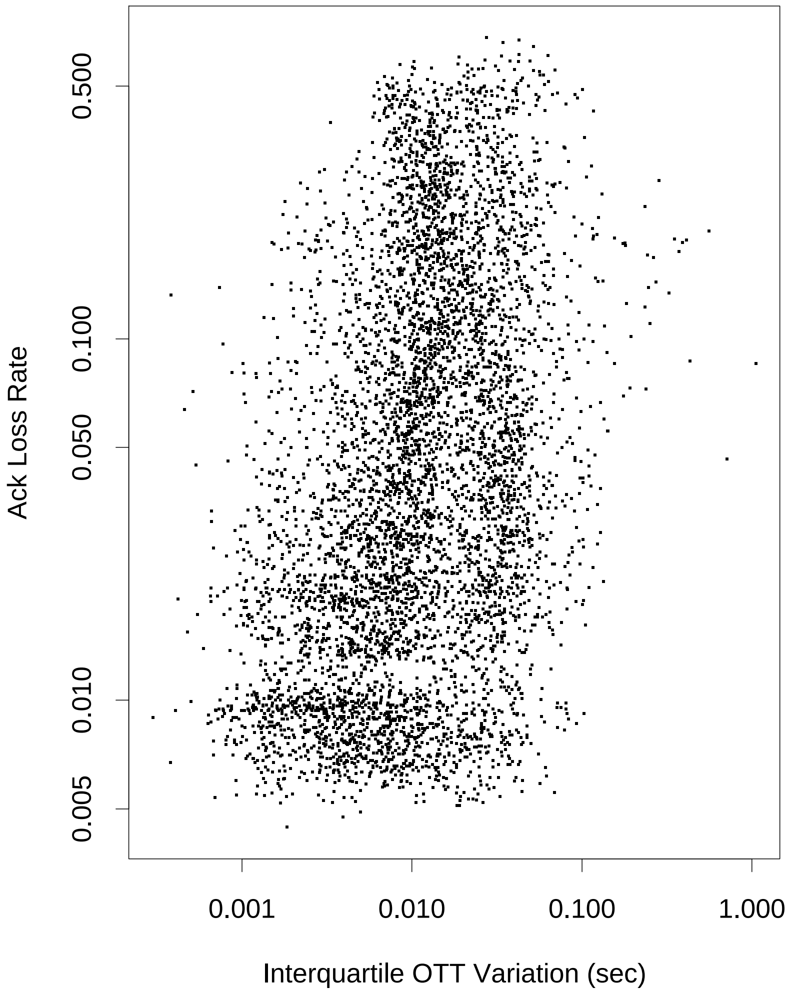

Figure 16.10: Scatter plot of ack loss rate versus interquartile ack OTT variation, for $\mathcal{N}_2$ connections that lost at least one ack

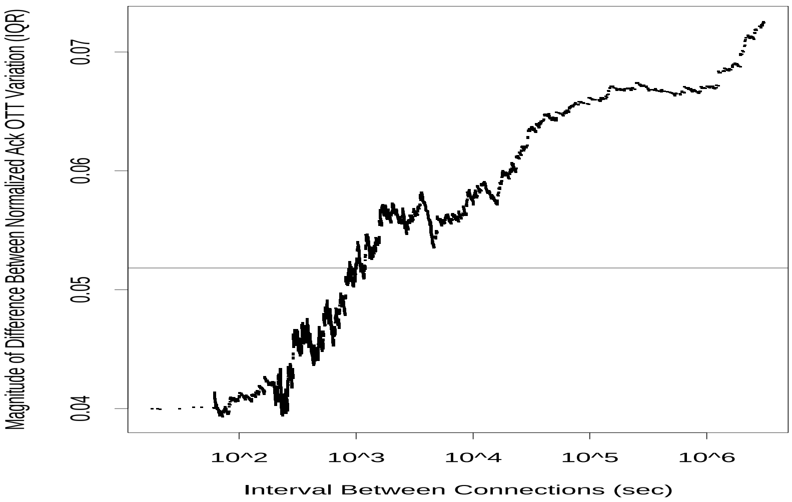

Figure 16.11: Evolution of how the interquartile range of normalized ack OTT variation differs with time

After constructing these pairs, we sort them on ΔT_c and then use an exponentially-weighted moving average (EWMA) with $\alpha = 0.01$ to smooth how $|\Delta\sigma_c|$ evolves as a function of ΔT_c. We first computed the EWMA with an initial value of 0, but inspecting the resulting plot indicated that, even for very small ΔT_c's, $|\Delta\sigma_c|$ was around 0.04, so we used 0.04 for the initial value in our final computation.

Figure 16.11 shows how the smoothed $|\Delta\sigma_c|$ evolves with time. The horizontal line corresponds to the median normalized ack OTT interquartile range for a single connection: a bit over 5% of the RTT. Note that the y-axis ranges from only 0.04 to 0.07. Thus, the change in normalized variation slowly ranges from a bit below the median variation to a bit above, across a wide range of time scales. Figure 16.12 shows the same plot except for "raw" ack OTT variations, that is, the IQR of the variations without normalizing by dividing by the connection's round-trip time. Again, we see a rapid rise followed by a slowly-increasing regime between 6-9 msec (keep in mind that this plot is heavily averaged; some paths have IQR variations far higher than 10 msec). The horizontal line corresponds to the median IQR variation for a single connection—just under 6 msec—which is quickly exceeded.

Since even the minimum $|\Delta\sigma_c|$ is not a great deal below the median normalized OTT variation, and the raw IQR differences rapidly exceed the median raw OTT variation, we conclude that a connection's ack OTT variation is *not* a very good predictor of future variation. This compares with Figure 15.18, which shows that a connection's loss rate is not a very good predictor of its future loss rate, either. Both figures argue that caching detailed network path information will prove beneficial only in the near-term, meaning on the order of a few minutes into the future.

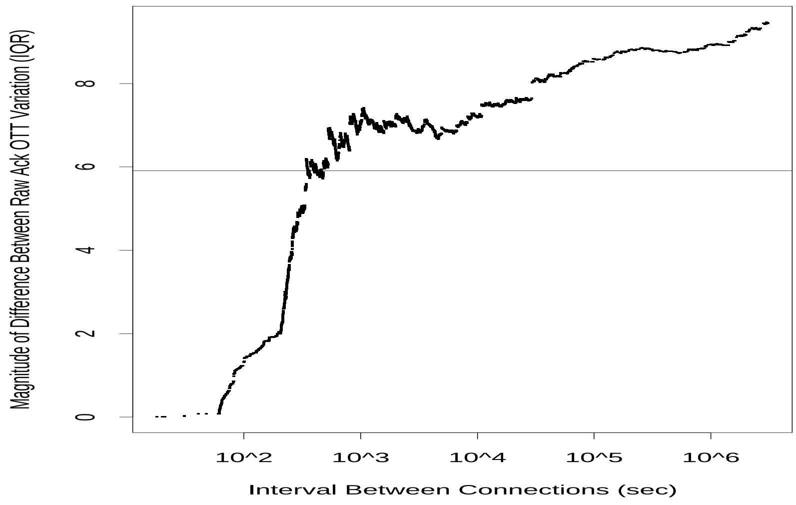

Figure 16.12: Evolution of how the interquartile range of raw ack OTT variation differs with time

16.2.6 Removing load from OTTs

In § 15.2 we developed the notion of "loaded" data packets, namely those which would have to queue behind their predecessors at the bottleneck due to the spacing between the time of their transmission and that of their predecessors. In this section we look at the subtle problem of removing the packet's load, as given by Eqns 15.3 and 15.4. The main problem we face in doing so is that the estimated bottleneck bandwidth given by Eqn 14.12 in Chapter 14 is *inexact*. In particular, our methodology produces an error *range* associated with the estimate.

Depending on which value within this range we use, Eqns 15.3 and 15.4 (or, more accurately, their counterparts for the particular estimate we use) can in some circumstances produce considerably different values for a packet's load. Thus, if we subtract that load from the packet's OTT we can easily under- or over-estimate the packet's "true" OTT, meaning its OTT if it did not have to queue behind its predecessors at the bottleneck.

We can partially address this uncertainty using a self-consistency check for the estimated bottleneck bandwidth. In particular, we can test the soundness of the central estimate of the bottleneck bandwidth, ρ_B, as follows. We first compute for each connection ΔQ, the difference between the minimum and maximum OTTs for the connection's loaded data packets. (The difference is presumably due to queueing, hence the notation ΔQ.) We then subtract out each packet's load (as given by Eqns 15.3 and 15.4 when using ρ_B rather than ρ_B^-, per Eqn 14.12), and compute $\Delta\widetilde{Q}$, the residual difference between the minimum and maximum OTTs. $\Delta\widetilde{Q}$ is thus the counterpart to ΔQ for the loaded packet OTTs, after adjusting for the connection's own contribution to the delays. We would expect to find

$$\Delta\widetilde{Q} \leq \Delta Q,$$

since a connection's extra, self-induced delay should only increase the OTT extremes it experienced.

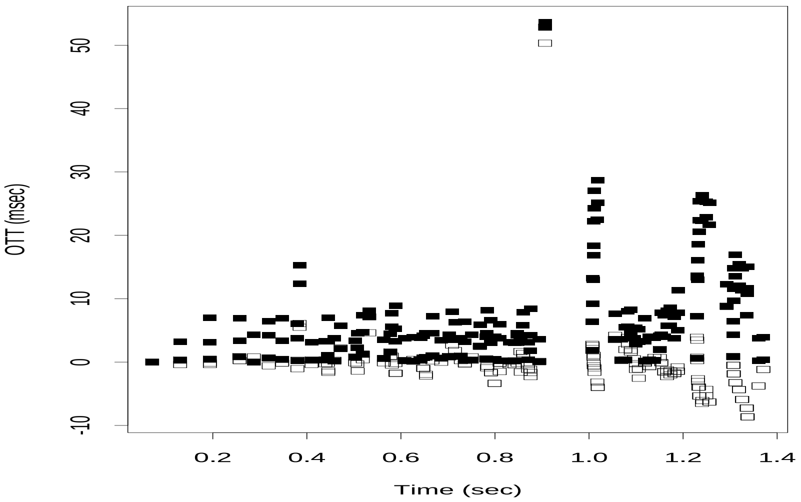

Figure 16.13: OTT plot revealing "broken" bottleneck estimate: one that is too low. Solid squares mark unadjusted OTTs, hollow squares mark OTTs adjusted to remove load based on bottleneck estimate.

If, however, we find that

$$\Delta\widetilde{Q} > 1.1 \cdot \Delta Q, \tag{16.1}$$

and if the difference between the two is also larger than twice the joint clock resolution $R_{s,r}$ (§ 12.3) (to assure that it is not just due to measurement noise), then we consider the bandwidth estimate ρ_B as *broken*: likely wrong, since using it to subtract out queueing effects actually increases the range of OTTs we observe.

This check is not foolproof. It can generate both false positives and false negatives. For example, it may be that the packet with the greatest OTT had little load to subtract out, while that with the least OTT happened to have more load, leading to an erroneous determination that ρ_B is broken. Using a factor of 1.1 in Eqn 16.1 helps avoid the possibility of these sorts of false positives, by only flagging a ρ_B estimate as broken if using it leads to a significant increase in adjusted delay.

The check might also fail, generating a false negative, if ρ_B is indeed quite inaccurate, but subtracting out inaccurate loads from the OTTs still happens to reduce their range. We find that these false negatives are much more likely to occur if ρ_B is too high, since an overestimate leads to relatively little (but still some) load being subtracted. If ρ_B is an underestimate, then excessive load is removed, which tends to lead to some packets having grossly under-adjusted OTTs, widening $\Delta\widetilde{Q}$.

The test is worth making because it detects two situations of interest. First, as noted above, if ρ_B is too low, then the calculated packet loads will be too high, and subtracting them out will often expand the range. Figure 16.13 shows an instance of this occurring. The solid squares show the OTTs of the connection's data packets, and the hollow squares correspond to the OTTs adjusted for the (erroneously too small) bottleneck bandwidth. The trend towards progressively lower adjusted OTTs indicates that the low estimate leads to removing more and more spurious load

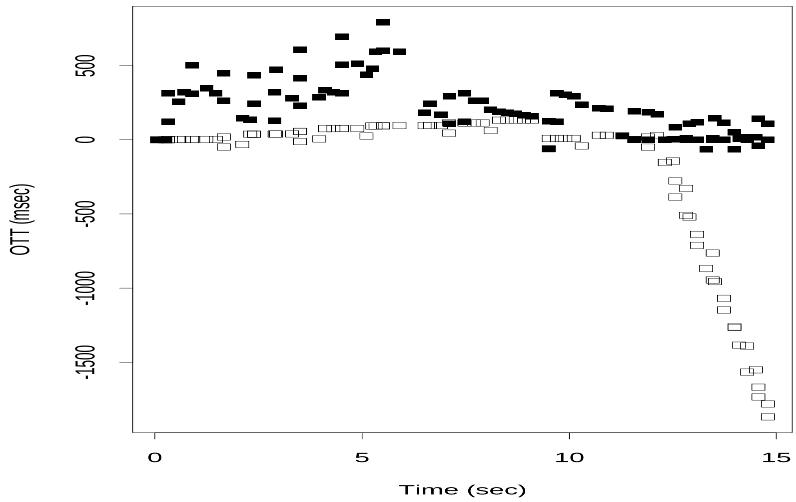

Figure 16.14: Another OTT plot revealing a "broken" bottleneck estimate: one that failed to detect a change in the bottleneck rate. Solid squares mark unadjusted OTTs, hollow squares mark OTTs adjusted to remove load based on bottleneck estimate.

as the connection transmits more packets that are erroneously judged to queue behind one another.

We particularly want to detect the case of ρ_B too low, because later in this chapter we will use load computations as a basis for determining the degree of available bandwidth in the network (§ 16.5), and we want these computations based on solid estimates of packet loads. The other case that the test can detect is the presence of an undiagnosed bottleneck change. If ρ_B corresponds to the slower of the two bottleneck rates, then `tcpanaly` will compute excessive loads for the packets transmitted during the era of the faster bottleneck rate. Figure 16.14 illustrates this happening. The estimated ρ_B is fairly accurate for most of the trace (a bit too high, as indicated by the slowly rising adjusted OTTs—not enough load is being removed). However, at $T = 12$ sec, when the bottleneck rate doubles, the estimate becomes much too low, and leads to removing too much load.[6]

Table XVIII in § 14.7 summarizes how often this check detected a broken bottleneck rate estimate in $\mathcal{N}_1$ and $\mathcal{N}_2$. It was not very often, which contributes to our faith in the PBM algorithm for detecting bottleneck rates (§ 14.6), but it did detect some problems, indicating it is worth the effort to perform the test.

As the lefthand portion of Figure 16.14 indicates, a slight mismatch in ρ_B can lead to definite, spurious trends in the adjusted OTTs. Such trends are apparent even when the estimated ρ_B is quite good. Figure 16.15 shows an OTT plot in which the bottleneck estimate is clearly quite good, as it accounts for virtually all of the variation in the OTTs (the adjusted times, shown with hollow squares, are nearly constant). Yet, if we zoom in on just the adjusted OTTs, shown in Figure 16.16, we see a clear downward trend in the adjusted OTTs. The trend corresponds to 500 μsec over about 300 msec, or about 1 part in 600. Consequently, we see that, even though our

[6] PBM does not detect this bottleneck change because it comes so close to the end of the trace.

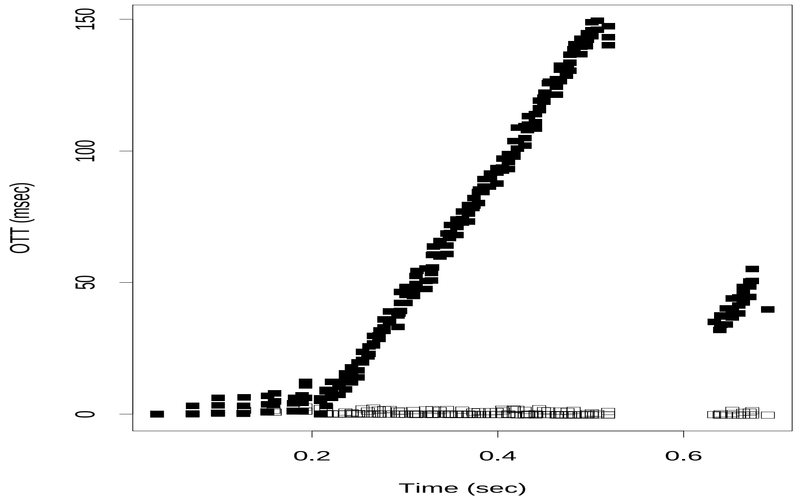

Figure 16.15: OTT plot showing virtually all OTT variation due to connection's own queueing load

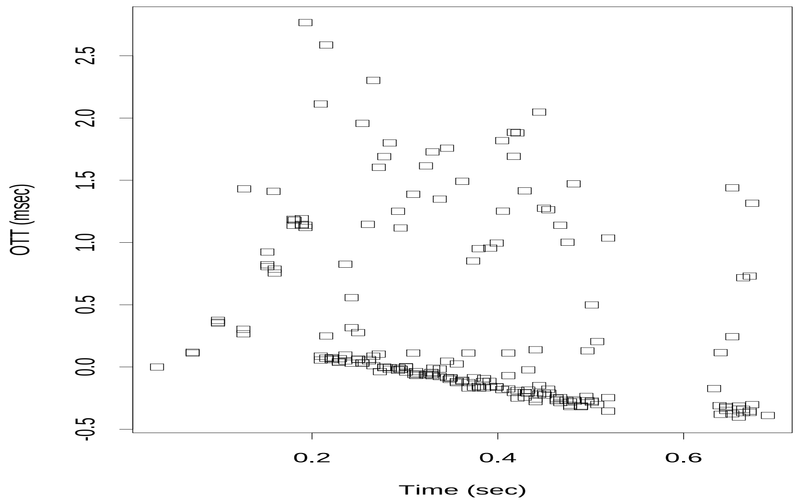

Figure 16.16: Enlargement of adjusted OTTs from previous figure

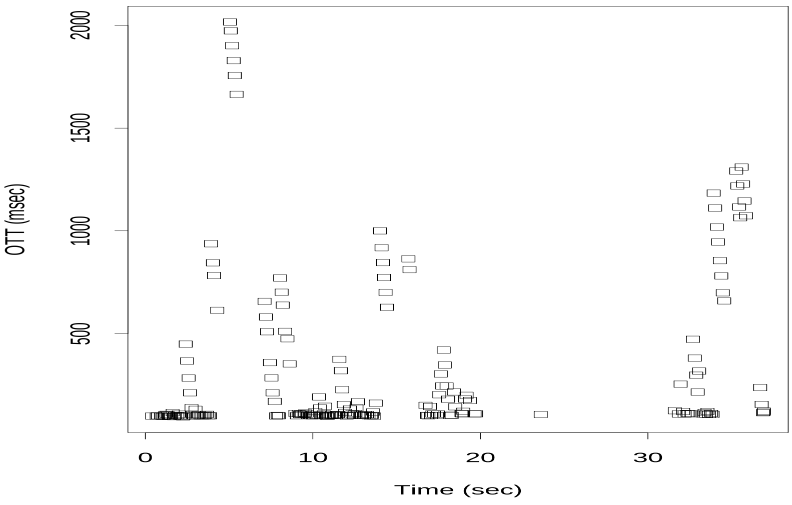

Figure 16.17: Ack OTT plot showing 10-sec periodicities

estimated ρ_B is quite good, it is not sufficiently exact to avoid introducing an artificial trend in the adjusted OTTs.

Because of this problem, we abandoned our original goal of trying to treat loaded packets the same as unloaded packets by adjusting their OTTs, as doing so requires extremely precise estimation of ρ_B. If the estimate is off, we introduce systematic errors that could easily be confused with genuine network effects.

16.2.7 Periodicity in OTTs

In § 15.3 we discussed our efforts at testing whether packet loss patterns exhibit periodicity. We might expect them to do so due to synchronization effects known to sometimes plague Internet routers, resulting in periodic packet forwarding outages. These lead to lengthy delays and perhaps loss, if buffers become exhausted during the outage [FJ94]. In this section we briefly discuss evidence in our data for periodic variations in packet delays.

In attempting to assess delay periodicities, we run into the same problems as when assessing loss periodicities: our data unfortunately are not suited for a proper investigation of the question. § 15.3 outlines the reasons for this and we will not repeat them here. We did, however, attempt the same analysis as in § 15.3: we selected connections between the North American sites exhibiting the highest degree of clock synchronization, singled out the busiest day among them, and analyzed their connections to determine the time at which the connection's largest delay occurred. We then studied plots of the peak delay time versus the same time modulo different possible periodicity intervals. This effort did not find any conclusive evidence of global periodicities.

However, phenomenological inspection of other traces shows that delay periodicities definitely do occur. Figure 16.17 shows a plot of ack OTT times for a connection from `connix` to `ucl`. The distance between the first OTT peak at about 1000 msec and the second such peak (we are ignoring the striking, 2000 msec peak) is 10.07 sec, while that between the second peak and the third such peak (at about 1200 msec) is 19.92 sec. Furthermore, two acks *were* sent about 10 sec after the second peak, but both were lost (hence, they do not appear on the plot). Thus, this trace exhibits strong evidence of a 10-second periodicity. We find a number of other traces to `ucl` with the same spacing between delay peaks, suggesting that this is an ongoing phenomenon.

We observed other traces with apparent 5-second and 30-second periodicities in delay spikes, involving different hosts, indicating that the phenomenon is not confined to only `ucl`. On the other hand, we did not find strong evidence above of global periodic delay variation among the highly-synchronized North American sites. Thus, we conclude that the phenomenon is definitely present, but, if widespread, at least not globally synchronized.

16.3 Timing compression

Packet timing *compression* occurs when a flight of packets are sent over an interval ΔT_s but arrive at the receiver over an interval ΔT_r, with $\Delta T_r < \Delta T_s$. To first order, compression should not occur, since the main mechanism at work in the network for altering the spacing between packets is queueing, which in general *expands* flights of packets, as later ones have to wait behind the transmission of earlier ones (§ 14.2). However, compression can occur if a flight of packets is at some point *held up* by the network, such that transmission of the first packet stalls and the later packets have time to catch up.

Zhang et al. predicted from theory and simulation that acks could be compressed ("ack compression") if a flight arrived at a busy router (one with a significant queue), and if no intervening packets arrived between the different acks [ZSC91]. As the acks queue behind one another, the potentially large spacing between them due to self-clocking (§ 9.2.5) and ack-every-other policies (§ 11.6) would then be lost when the acks were later transmitted back-to-back upon reaching the front of the queue.

This situation corresponds to a *draining* queue: a router that was busy when the first ack arrived (and hence could not service it before the others arrived), and yet new arrivals from other traffic sources are sporadic. If instead new arrivals were steady, then they would occupy slots in the queue between the acks in the flight, and their spacing would be (roughly) preserved, rather than compressed.

Mogul subsequently analyzed a trace of Internet traffic and confirmed (among other phenomena) the presence of ack compression [Mo92]. His definition of ack compression is somewhat complex, involving significant deviations from the median inter-ack spacing, since he had to infer endpoint behavior from an observation point inside the network (a vantage point problem, per § 10.4). But he clearly detected the presence of ack compression. He found that compression was correlated with packet loss but considerably more rare. His study was limited, however, to a single 5-hour traffic trace.

Since we can readily compute from our data both ΔT_s and ΔT_r for any flight of packets, we can use a simpler definition of compression than employed by Mogul. In this section we characterize three different types of compression: ack compression (§ 16.3.1), data packet compression (§ 16.3.2), and receiver compression (§ 16.3.3). We show that all three types of compression occur within the Internet, though each is limited in its effects.

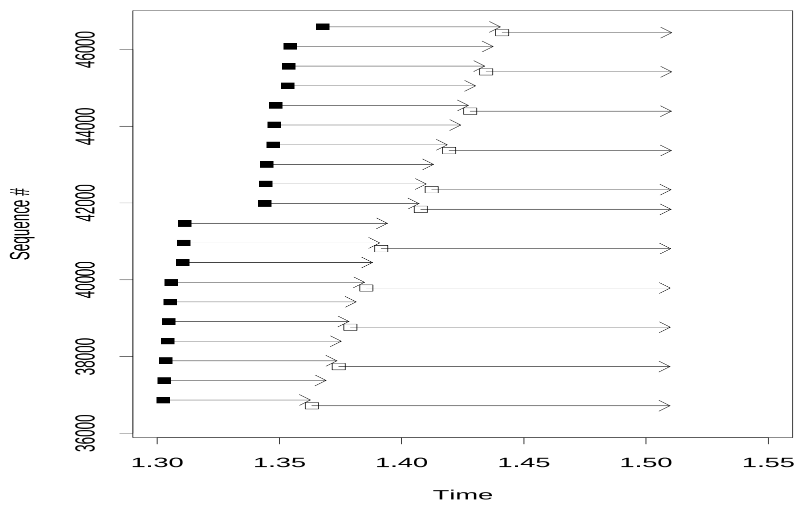

Figure 16.18: Paired sequence plot showing ack compression

16.3.1 Ack compression

If ack compression is frequent, it presents two problems. First, as acks arrive they advance TCP's sliding window and "clock out" new data packets at the rate reflected by their arrival (§ 9.2.5). For compressed acks, this means that the data packets go out *faster* than previously, which can result in network stress. Second, sender-based measurement techniques such as SBPP (§ 14.3) can misinterpret compressed acks as reflecting greater bandwidth than truly available. On the other hand, some researchers argue that occasional ack compression is beneficial since it provides an opportunity for self-clocking to discover newly-available bandwidth.

To detect ack compression, for each group of at least 3 acks we compute:

$$\xi = \frac{\Delta T_r + C_r}{\Delta T_s - C_s}, \tag{16.2}$$

where C_r and C_s are the receiver and sender's clock resolutions. Using Eqn 16.2 results in ξ being a conservative estimate, since by adding C_r in the numerator but subtracting C_s in the denominator, we tend to inflate ξ.

We consider a group of acks compressed if $\xi < 0.75$. We term such a group a *compression event*. In $\mathcal{N}_1$, 50% of the connections experienced at least one compression event, and in $\mathcal{N}_2$, 60% did. In both, the mean number of events per connection was around 2, and 1% of the connections experienced 15 or more. Almost all compression events are small, however, with only 5% spanning more than five acks. Figure 16.18 shows a paired sequence plot of one of the larger events, in which eleven acks were compressed. The solid squares indicate when the data packets were sent, and the arrows stemming from them point to their arrival times at the receiver. The corresponding acks (offset downward a bit, for legibility) are shown with hollow squares. The arrows from these squares all stop at virtually the same point in time, $T = 1.51$, indicating that, even though the acks were sent over an interval of 77 msec, they arrived all together, about 760 μsec apart—compressed by a factor of 100.

We also note that a significant minority (10–25%) of the compression events occurred for dup acks. These are sent with less spacing between them than regular acks sent by ack-every-other policies, so it takes less timing perturbation to compress them. Compressed dup acks are only slightly more likely to occur in a large burst than compressed regular acks. In $\mathcal{N}_2$, overall 5.1% of the compression events consisted of six or more acks: 4.8% of the regular-ack compression events, and 6.0

Finally, we classify compression events as "major" if the compression results in the acks arriving at the data sender with a spacing less than that corresponding to the bottleneck bandwidth; otherwise, we term the event "minor." Major events are significant because they reflect a breakdown in self-clocking—namely, the sender will transmit in response to them at a rate exceeding the bottleneck bandwidth—and they also make sender-based bottleneck estimation difficult, since, unless detected, they will lead to overestimates.

Let ρ_B^+ be the upper bound on the estimated bottleneck bandwidth, per Eqn 14.12. If a flight of k packets arrives during an interval ΔT_r, and they together acknowledge a total of b bytes, then we consider the flight to reflect a major compression event if:

$$\frac{b}{\Delta T_r} > \rho_B^+.$$

We apply this test to each ack compression event detected by `tcpanaly`, except we omit the final ack of the event. The reason for this omission is that `tcpanaly` finds compression events by constructing groups of acks for which $\xi < 0.75$, and sometimes the final ack of the group is relatively uncompressed compared to the others (i.e., it raised ξ from a small value to a value near 0.75). Consequently, we omit this final ack to avoid skewing the assessment of "major" events by our methodology for grouping acks into events.

We find that in both $\mathcal{N}_1$ and $\mathcal{N}_2$, about 75% of the compression events are major. This figure only slightly diminishes if we confine our analysis to compressed "regular" acks, eliminating compressed dup acks.

Of the major compression events, 80% reflect acks arriving at a rate corresponding to more than twice ρ_B^+. Thus, when compression occurs, it is usually large enough to result in a significant overestimate of the bottleneck bandwidth.

From these findings, we conclude that ack compression definitely occurs in the Internet, but rarely enough as to not pose a significant problem by corrupting self-clocking or causing excessive burstiness. That it occurs for more than half the connections, however, and that most of these are "major," indicates that a sender-based measurement scheme *must* employ filtering to remove extreme values from its bottleneck estimates, as otherwise it is very likely to overestimate the bottleneck bandwidth, with perhaps disastrous consequences.

16.3.2 Data packet timing compression

For data packet timing compression, our concerns are different. Sometimes a flight of data packets is sent at a high rate due to a sudden advance in the receiver's offered window. Normally these flights are spread out by the bottleneck and arrive at the receiver with a distance Q_b between each packet (§ 14.2). If after the bottleneck their timing is compressed, then use of Eqn 16.2 will *not*

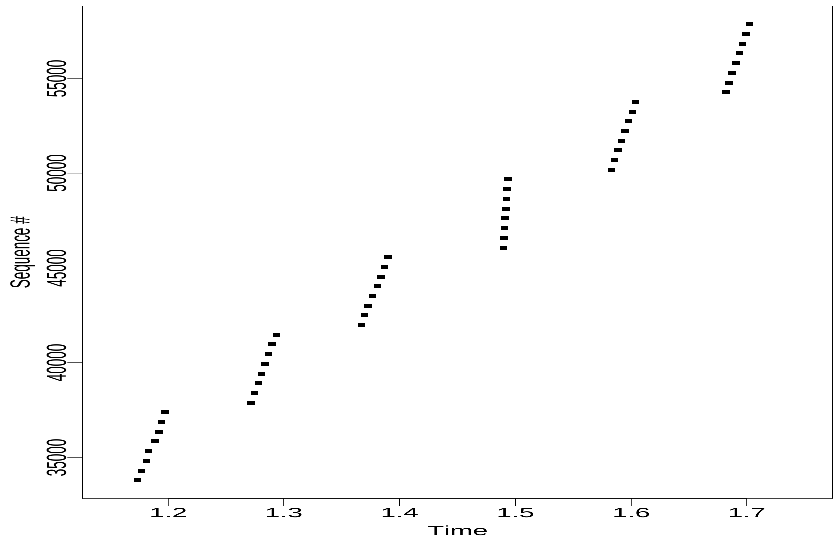

Figure 16.19: Data packet timing compression

detect this fact unless they are compressed to a greater degree than their sending rate. Figure 16.19 illustrates this concern: the flights of data packets arrived at the receiver at 170 Kbyte/sec (T1 rate), except for the central flight, which arrived at Ethernet speed. However, it was also sent at Ethernet speed, so, for it, $\xi \approx 1$.

Consequently, we consider a group of data packets as "compressed" if they arrive at greater than twice the upper bound on the estimated bottleneck bandwidth, ρ_B^+. We only consider groups of at least four data packets, as these, coupled with ack-every-other policies, have the potential to then elicit a pair of acks reflecting the compressed timing, leading to bogus self-clocking.

These compression events are more rare than ack compression, occurring in only 3% of the $\mathcal{N}_1$ traces and 7% of those in $\mathcal{N}_2$. We were interested in whether some paths might be plagued by repeated compression events due to either peculiar router architectures or network dynamics. Only 25–30% of the traces with an event had more than one, and 3% had more than five, suggesting that such phenomena are rare. But those connections with multiple events are dominated by a few host pairs, indicating that some paths are indeed prone to timing compression. Figure 16.20 shows an example. Here, the bottleneck rate is T1, which corresponds closely with the flatter slopes in the plot.

Thus, it appears that data packet timing compression is rare enough not to present a significant problem. That it does occur, though, again highlights the necessity for outlier-filtering when conducting timing measurements.[7]

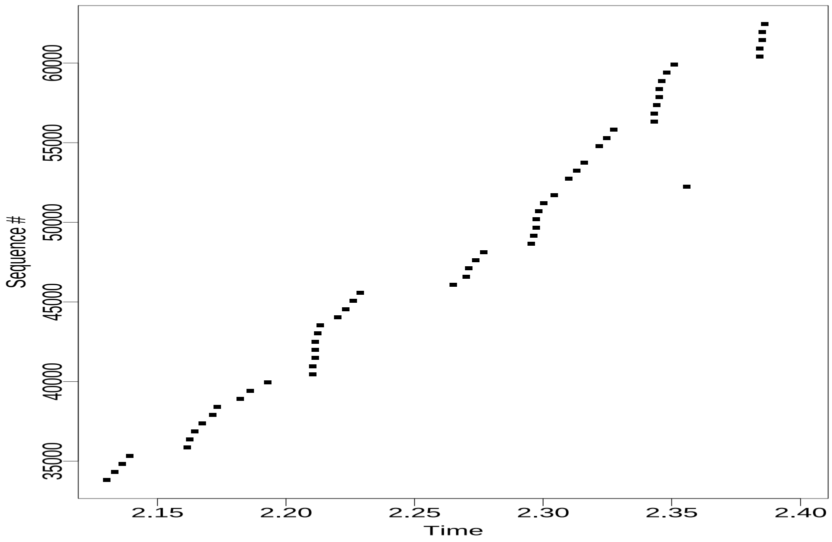

Figure 16.20: Rampant data packet timing compression

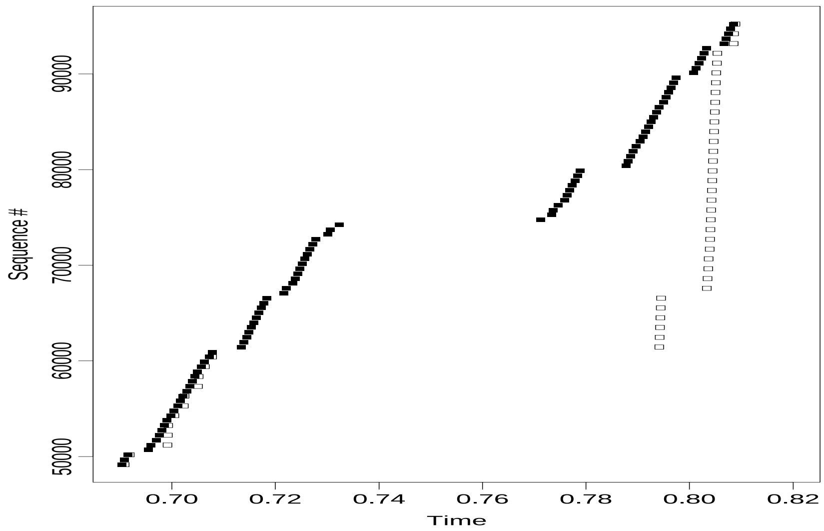

Figure 16.21: Receiver sequence plot showing major receiver compression

16.3.3 Receiver compression

A third type of timing compression occurs when the receiver delays in generating acks in response to incoming data packets, and then generates a whole series of acks at one time. The timing of these acks appears compressed to the sender, though *not* for reasons of network dynamics, but instead due to lulls at its remote peer. Figure 16.21 shows the most striking example in our traces, in which the `lbl` receiver compressed 25 of its acks, sending them over a 2 msec interval instead of over the 83 msec interval corresponding to the data packets they acknowledged. (Slightly earlier, the receiver also compressed 6 other acks, as seen in the figure.)

Since receiver compression is an endpoint effect, its presence tells us nothing about the dynamics of the connection's Internet path. However, receiver compression remains quite interesting because it is an additional noise element that any sender-only measurement scheme must contend with. It also leads to the same consequences as true ack compression, namely a break-down of a connection's self-clocking.

To assess receiver compression, we compute:

$$\xi' = \frac{\Delta T_a + C_r}{\Delta T_d - C_r},$$

where ΔT_a is the spacing between the generated acks, ΔT_d is the spacing between the arriving data packets (the ones that led to the acks), and C_r again is the receiver's clock resolution. As in Eqn 16.2, the addition of C_r in the numerator and subtraction in the denominator makes ξ' conservative. We consider $\xi' < 0.75$ as indicating a receiver compression event. Note that our earlier analysis of ack compression uses ΔT_a as the original spacing of a flight of acks, and then checks whether that was compressed while the packets were in flight. Consequently, that analysis does *not* confuse ack compression with receiver compression: the earlier ack compression analysis only evaluates compression due to network behavior.

We include delayed acks in our analysis, as these affect self-clocking. Sender-based measurement techniques can generally detect delayed acks.[8] In both $\mathcal{N}_1$ and $\mathcal{N}_2$, we find that about 10% of the connections included a receiver compression event of at least three acks. Of these, about three-quarters experienced only one receiver compression event, and, in $\mathcal{N}_1$, none experienced more than four, though, in $\mathcal{N}_2$, the upper limit was 15. Almost all events were only 3 acks in size (95% in $\mathcal{N}_1$, 80% in $\mathcal{N}_2$).

While these statistics indicate that receiver compression is fairly rare, and even less often significant, we must note that, because receiver compression is an *endpoint* effect, these statistics are *not* necessarily representative of its frequency in the Internet as a whole. In particular, we find that just a few sites cause the majority of the receiver compression events in our study, so we have no way of telling whether other sites would tend overall towards more receiver compression or less.

Given this caveat, we note that we find receiver compression, like other forms of timing compression, to be fairly rare. In particular, in our datasets it appears more rare than ack compression, so, if this is a representative finding, then sender-based assessment of ack compression caused by network dynamics will not be terribly skewed by the presence of receiver compression.

[7] It also has a measurement benefit: from the arrival rate of the compressed packets, we can estimate the downstream bottleneck rate.

[8] Using the rule that an ack for less than two full segments was presumably delayed. This heuristic could fail in the future, if TCPs begin to ack every packet, which they might do to accelerate the slow-start process.

If the sender-based measurement employs filtering to remove outliers, as it needs to do anyway to deal with ack compression, then receiver compression does not make the measurement significantly harder.

16.4 Queueing analysis

In this section we develop a rough estimate of the time scales over which queueing occurs. If we take care to eliminate suspect clocks (Chapter 12), reordered packets (§ 13.1), compressed packets (§ 16.3), and traces exhibiting TTL shifts (which indicate routing changes, per § 7.7), then we argue that the remaining measured OTT variation is mostly due to queueing. Hence, we can estimate queueing time scales by analyzing time scales of OTT variations.

For a given time scale, τ, we compute the queueing variation on that time scale as follows. First, we partition the packets sent by a TCP into intervals of length τ. For each interval, let n_l and n_r be the number of successfully-arriving packets in the left and right halves of the interval. If either is zero, or if $n_l < \frac{1}{4}n_r$, or vice versa, then we reject the interval as containing too few measurements or too much imbalance between the halves.

Otherwise, let m_l and m_r be the median OTTs of the two halves. We then define the interval's queueing variation as $|m_l - m_r|$. Thus, we quantify the variation as how much the OTTs changed over a time scale of τ, but, by computing the change only as the difference between two intervals of length $\frac{1}{2}\tau$, we include in the variation *only* changes that occurred on the time scale of τ. Changes that occurred on smaller time scales will in general all occur within either the left or right half, and the *median* of the half will not reflect the smaller-time-scale change. Changes occurring on larger time scales will not in general result in variation between the two halves, and so likewise will not enter into the computation.

By using medians, we attempt to reduce the effects of occasionally very large OTTs. We found that means and standard deviations can often be unduly skewed by a small set of large OTTs.

The question remains how to summarize the interval changes. We investigate two different summaries. In the first, we define ΔQ_τ as the median of $|m_l - m_r|$ over all such intervals. Thus, ΔQ_τ reflects the "average" variation we observe in packet delays over a time scale of τ. By using medians, this estimate again is robust in the presence of noise due to non-queueing effects, or transient queueing spikes. In addition, we compute $Q_\tau^{\max}$, the maximum observed difference across any two halves of an interval of length τ. ΔQ_τ thus summarizes *sustained* variation on the time scale τ, while $Q_\tau^{\max}$ summarizes *bursts* of variation on the time scale τ.

We now analyze ΔQ_τ and $Q_\tau^{\max}$ for different values of τ, confining ourselves to variations in ack OTTs, as these are not clouded by self-interference and adaptive transmission rate effects (§ 15.2). The question we wish to address is: are there particular τ's on which most queueing variation occurs? This question is particularly interesting because of its potential implications for engineering transport protocols. For example, if the dominant τ is less than a connection's RTT, then it is pointless for the connection to try to adapt to queueing fluctuations, since it cannot acquire feedback quickly enough to do so. Or if, for example, the dominant τ is on the order of 1 sec, then that constant helps us determine the related constants—such as the α's for EWMA estimators—governing how a transport connection should update its RTT estimate in order to compute its retransmission timeout.

For each connection, we range through $2^4, 2^5, \ldots, 2^{16}$ msec to find $\widehat{\tau}$, the value of τ for

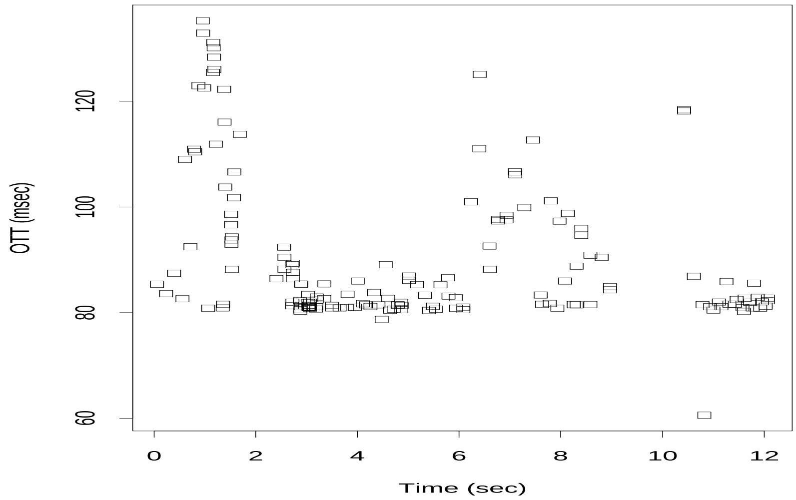

Figure 16.22: Ack OTT plot for a connection with $\widehat{\tau} = 4$ sec for ΔQ_τ

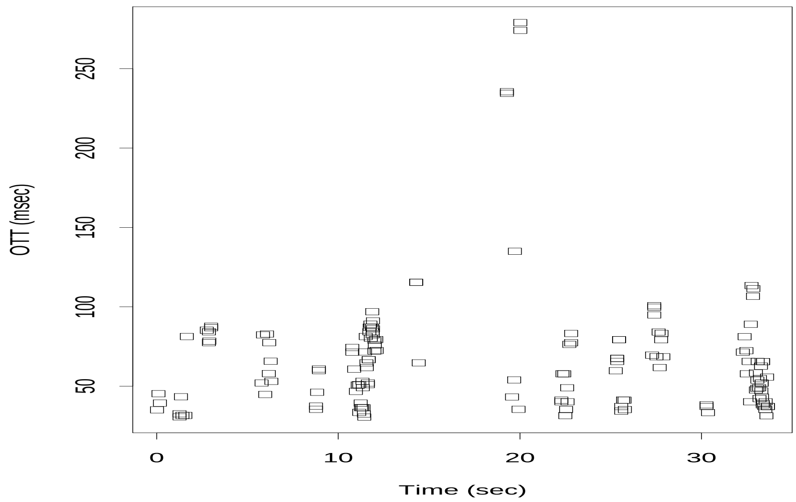

Figure 16.23: Ack OTT plot for a connection with $\widehat{\tau} = 1$ sec for $Q_\tau^{\max}$

which ΔQ_τ or $Q_\tau^{\max}$ is greatest. $\widehat{\tau}$ reflects the time scale for which the connection experienced the greatest OTT variation, where the variation is *sustained* if computed for ΔQ_τ, and *momentary* if computed for $Q_\tau^{\max}$. Figure 16.22 shows a plot of the ack OTTs for a connection with $\widehat{\tau} = 4$ sec for ΔQ_τ, indicating maximum sustained variation occurs on 4-second time scales. Figure 16.23 shows a connection with $\widehat{\tau} = 1$ sec for $Q_\tau^{\max}$, which emphasizes the large increase in delay at $T = 20$ sec. For the first connection, the maximal $Q_\tau^{\max}$ occurs for $\widehat{\tau} = 64$ msec, corresponding to the sharp spike just after $T = 1$ sec. For the second, the maximal ΔQ_τ occurs for $\widehat{\tau} = 4$ sec, due to the sustained variation on 4-second time scales (for this connection, other time scales have large ΔQ_τ, too, but the largest is for $\tau = 4$ sec). Clearly, the time scales of maximum *sustained* burstiness versus those of maximum *peak* burstiness can differ considerably.

Before looking at the range in $\widehat{\tau}$'s for our measurements, a natural calibration question is what sort of $\widehat{\tau}$'s we find for synthetic variations. We investigated this question by simulating 10,000 independent and identically distributed (i.i.d.) OTT variations. Each variation was simulated as a random variable drawn from an exponential distribution with $\lambda = 1$,[9] corresponding to an OTT variation computed for one unit of time (the equivalent of 2^4 msec for the preceding discussion). For 100 simulation runs, $\widehat{\tau}$ was always ≤ 2 units of time for ΔQ_τ, and ≤ 4 units of time for $Q_\tau^{\max}$. Thus, we see that $\widehat{\tau}$ correctly indicates that the variation is confined to small time scales. If we simulate i.i.d. Pareto variations with $\alpha = 1.01$ (so, infinite variance and, just barely, finite mean), we still find $\widehat{\tau}$ confined to small time scales, never exceeding 4 units of time. Again, this is what we would expect, because the fundamental time scale of change is one time unit, since the variations are independent.

Figure 16.24 shows the normalized proportion of the connections in $\mathcal{N}_1$ and $\mathcal{N}_2$ exhibiting different values of $\widehat{\tau}$ for ΔQ_τ. Normalization is done by dividing the number of connections that exhibited $\widehat{\tau}$ by the number that had durations at least as long as $\widehat{\tau}$, so that the prevalence of short connections does not skew the distribution. For both datasets, time scales of 128–2048 msec primarily dominate. This range, though, spans more than an order of magnitude, and also exceeds typical RTT values. Furthermore, while less prevalent, $\widehat{\tau}$ values all the way up to 65 sec remain common, with $\mathcal{N}_1$ having a strong peak at 65 sec.[10]

Consequently, the figure indicates that *sustained Internet delay variations occur primarily on time scales of 0.1-2 sec, but extend out quite frequently to much larger time scales.*

Figure 16.25 shows the same figure but for $Q_\tau^{\max}$. Here we see that basically the same time scales dominate variation peaks, ranging from 128 to 1024 msec. Smaller time scales clearly contribute, however, and so do larger time scales up to about 4 sec, with $\mathcal{N}_1$ exhibiting a trend towards still larger time scales, while $\mathcal{N}_2$ does not. We interpret the figure as indicating that *peak Internet delay variations also occur primarily on time scales of 0.1-1 sec, but they too extend to larger time scales, and quite often to smaller time scales*. Consequently, it appears clear that there is no single time scale of "burstiness," which accords with the recent "self-similar" models of network traffic [LTWW94], though, as a rule of thumb, most variation occurs on time scales of a quarter-second to a half-second, a bit above usual connection round-trip times. Thus, it appears that transport connections *can* feasibly adapt to queueing changes, but to do so they must act quickly, within a few RTTs, or else it will often be too late.

[9]The results are independent of λ, however, since λ only determines the size of the identically-distributed variations, but not the time scales of the variations among them.

[10]Manual inspection of traces with $\widehat{\tau} = 65$ sec indicates that they do indeed exhibit their maximum variation on that time scale, addressing the concern that perhaps the peaks were due to some other effect, and hence spurious.

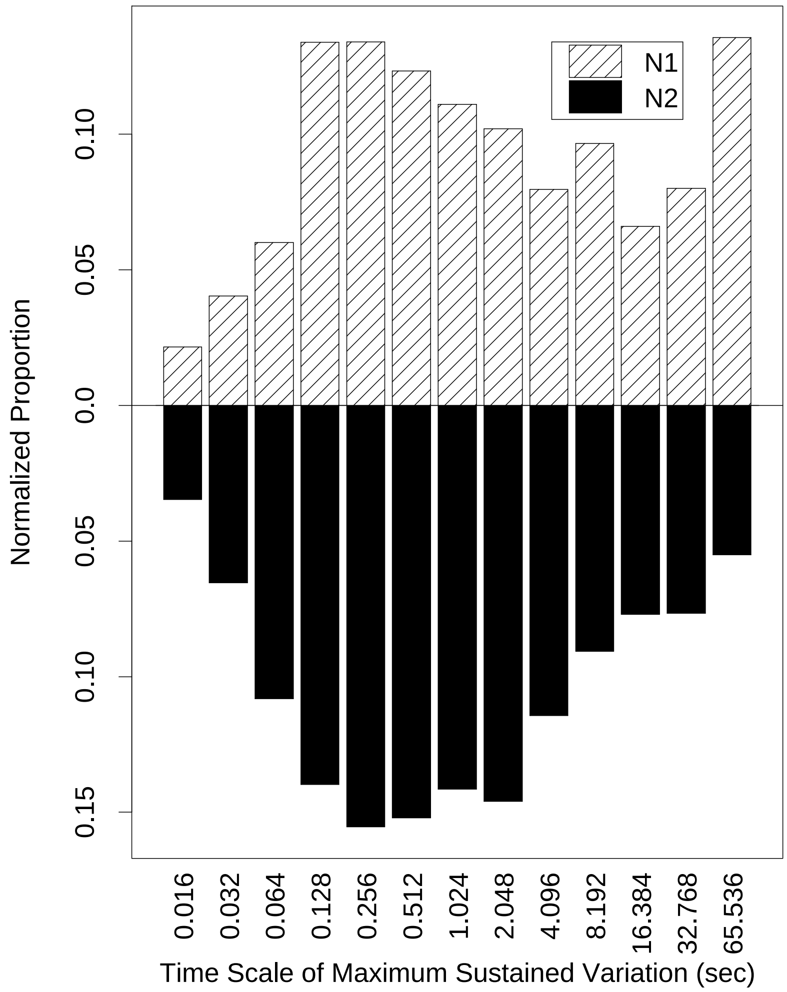

Figure 16.24: Proportion (normalized) of connections with given timescale of maximum sustained delay variation ($\hat{\tau}$)

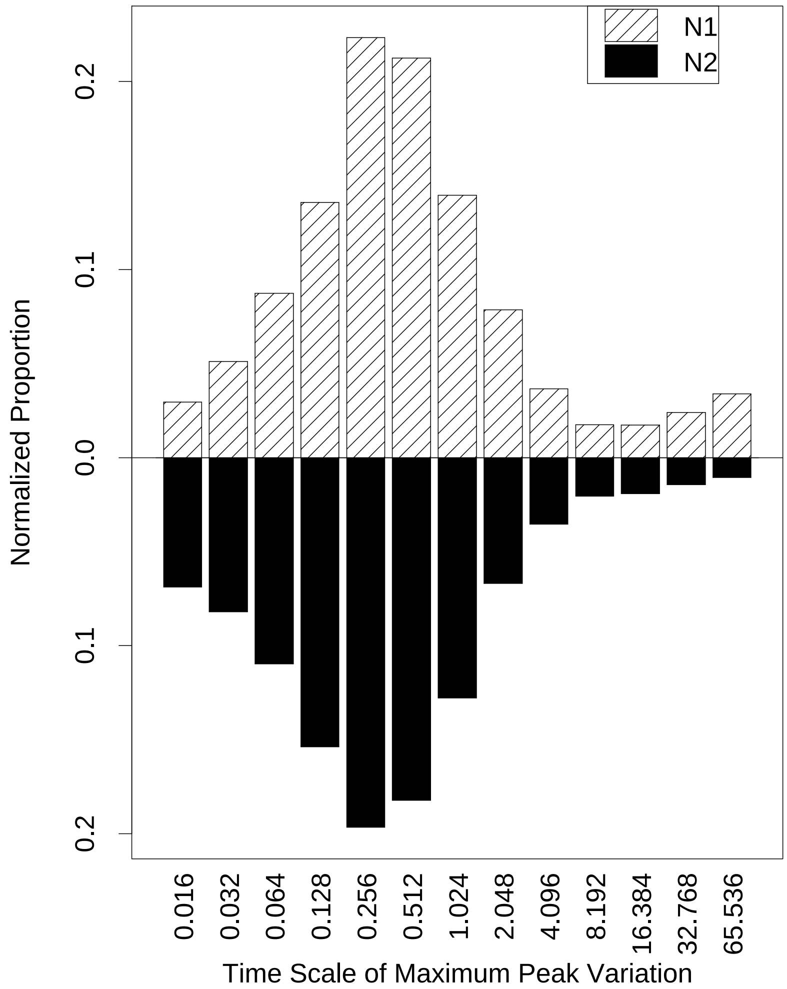

Figure 16.25: Proportion (normalized) of connections with given timescale of maximum peak delay variation ($\widehat{\tau}$)

16.5 Available bandwidth

The last aspect of delay variation we look at is an interpretation of how it reflects the *available bandwidth*. In a packet-switched network, available bandwidth is a somewhat elusive notion. The amount of bandwidth a connection might fruitfully use varies with time, as other cross-traffic connections come and go. From § 16.4 we know that significant OTT fluctuations often occur on time scales of 100-1000 msec, and for the upper end of this range (which actually extends appreciably to much larger time scales), no doubt most of the fluctuations are due to connections beginning or ending, rather than flights of packets within single connections beginning or ending.

Two existing approaches for estimating available bandwidth are `cprobe` [CC96b] and `Treno` [MM96]. `cprobe` works in conjunction with `bprobe` [CC96a], which we discussed in § 14.2. To estimate available bandwidth along a network path, `cprobe` first uses `bprobe` to estimate the bottleneck bandwidth along the path. `cprobe` then transmits four groups of probes, each probe consisting of 10 ICMP echo packets (as with `bprobe`). The echo packets are sent at a rate exceeding that of the estimated bottleneck bandwidth, to make sure they attempt to fully utilize the bottleneck. `cprobe` then computes from the timing of the ICMP echo replies the achieved throughput, and considers the ratio between this and the bottleneck bandwidth to be the *utilization* (similar to the value β which we define in Eqn 16.4 below), which indicates how much of the bottleneck bandwidth was actually available.

`cprobe` has three limitations that we attempt to address. The first is that it requires sending a fairly large flight of packets at a rate known to exceed what the network path can support, so `cprobe` can be viewed as fairly *stressful* to a network path. The second is that, because its probes use ICMP echo packets, which elicit same-sized replies, the achieved throughput the probes achieve will reflect the *minimum* of the available bandwidth along the forward and reverse paths. As we have seen that many path properties are asymmetric, it would not be surprising to find that available bandwidth is, too, and thus, for a unidirectional connection, `cprobe` might produce too pessimistic an estimate. The third limitation is that the pattern in which the probe packets are sent differs from that in which a TCP sender will transmit its data packets. We have seen in § 15.2 that, because TCP adapts its transmission rate to the presence of packet loss along the forward path, network conditions observed by TCP data packets can differ significantly from those observed by TCP ack packets. Thus, we suspect that available bandwidth estimates produces by `cprobe` might not closely reflect the throughput that a TCP would actually achieve.

This second point is addressed by the developers of the `Treno` utility [MM96]. `Treno` also uses ICMP echo packets to probe network paths, but it sends them using an algorithm equivalent to that used by TCP congestion control (§ 9.2). In addition, `Treno` can probe hop-by-hop available bandwidth by using increasing TTL (time-to-live) values in the IP headers of the echo packets it sends, just as does `traceroute` (§ 4.2.1). When doing so, it receives in response from each hop (except the last) not a full-sized echo reply, but a short ICMP Time Exceeded message. Thus, even if the available bandwidth along the return path is less than that along the forward path, `Treno` will still primarily observe the forward-path available bandwidth, just as would a TCP connection that receives only data-less acks in response to its data packets.

The main drawback of `Treno` is that it is a *stressful* technique. It estimates how fast a TCP could transfer data over a given network path by seeing how fast it itself can transfer data over the path, using a standard-conformant, but well-tuned, implementation of the TCP congestion control algorithm.

Ideally, we would like to estimate available TCP bandwidth *without* fully stressing the network path to do so. We do not achieve this goal in our present work. Instead, in this section we analyze our TCP transfer data both to characterize available bandwidths in the Internet, and to explore how we might perhaps in the future develop a non-stressful available-bandwidth estimation technique, based on fine-scale analysis of TCP packet timings. For this technique, the hope is that by carefully scrutinizing the delays of individual TCP packets, we might form a good estimate of the bandwidth available along the path they were sent, without requiring that we send the packets at a rate that saturates the path for any lengthy period of time.

We proceed as follows. First, we need to define what we mean by available bandwidth. We might argue that, if we know that a connection is competing with k other connections, then its fair share of the network resources is $\frac{1}{k+1}$. In particular, the connection's fair share of the bottleneck bandwidth, ρ_B, is $\frac{\rho_B}{k+1}$. These simple notions, however, quickly run into difficulties. First, during a connection's lifetime, competing connections come and go, so there is no single value to assign to k. Second, the competing connections do *not* in general compete along the entire end-to-end path, but only for a portion of it, so there may in fact be a great number of competing connections, but each competing for different resources. Finally, "fairness" itself is an elusive notion: it might well be the case that, for policy reasons (such as who is paying for what), or due to different traffic types, the simple each-gets-an-equal-share division of the resources is deemed inappropriate. (See [Fl91] for further discussion of the difficulty of defining a single notion of fairness.)

With these considerations in mind, we now strive to develop a notion of "equivalent competing connections," in order to talk in general terms about available resources. To do so, we attempt to characterize the network resources available to a connection as a fraction of the total resources in use. The term we will use to capture this notion is "available bandwidth." Here we presume that connections push on the network to extract as much resource from it as they can—TCP's slow start does exactly this.[11] Therefore, if a connection pushes on the network and we observe that it consumed m units of resources, and we can determine that other connections consumed n units of the same resources, then we will consider the available bandwidth as $\frac{m}{m+n}$; or, equivalently, that, over its lifetime, the connection competed with the equivalent of $\frac{n}{m}$ other connections like itself.

We will use as our unit of resource the amounts of buffer space and transmission time the connection consumed at the bottleneck link. In § 15.2 we developed a notion of data packet i's "load," λ_i, meaning how much delay it incurs due to queueing at the bottleneck behind its predecessors, plus its own bottleneck transmission time, ϕ_i, which is directly determined by the packet's size and the bottleneck bandwidth. Let

$$\psi_i = \lambda_i - \phi_i, \tag{16.3}$$

namely, just the amount of a packet's delay that is due to queueing behind its predecessors.

Let γ_i denote the difference between packet i's measured OTT and the minimum observed OTT (for full-sized packets). If the network path is completely unloaded except for the connection's load itself (no competing traffic), then we should have $\psi_i = \gamma_i$, i.e., all of i's delay variation is due to queueing behind its predecessors. More generally, define

$$\beta = \frac{\sum_i (\psi_i + \phi_i)}{\sum_j (\gamma_j + \phi_j)}. \tag{16.4}$$

[11] We do not, however, presume that the *measurement technique* for estimating how much bandwidth is available must also do so.

β then reflects the proportion of the packet's delay due to the connection's own loading of the network. If $\beta \approx 1$, then all of the delay variation is due to the connection's own queueing load on the network, while, if $\beta \approx 0$, then the connection's load is *insignificant* compared to that of other traffic in the network.

More generally, $\sum_i (\psi_i + \phi_i)$ reflects the resources consumed by the connection, while

$$\sum_j (\gamma_j + \phi_j) - \sum_i (\psi_i + \phi_i) = \sum_j \gamma_j - \sum_i \psi_i$$

reflects the resources consumed by the competing connections.

Note that including the ϕ_i terms in Eqn 16.4 is important: they reflect the basic bottleneck transmission cost. Without them, a connection that does not load the bottleneck link (perhaps because its transmission perfectly matches the bottleneck rate) will exhibit

$$\sum_i \psi_i = 0.$$

In this case, any slight variation in its OTTs, i.e.,

$$\sum_j \gamma_j = \epsilon > 0,$$

will result in $\beta = 0$. But in this limiting case we want our evaluation to indicate that almost all the resource was available (as indicated by $\sum_j \gamma_j$ being small), and this is exactly the limiting behavior of Eqn 16.4.

Thus, β captures the proportion of the total resources that were consumed by the connection itself, and we interpret β as reflecting the *available bandwidth*. Values of β close to 1 mean that the entire bottleneck bandwidth was available, and values close to 0 mean that almost none of it was actually available.

Note that we can have $\beta \approx 1$ even if the connection does not consume all of the network path's capacity. All that is required is that, to the degree that the connection did attempt to consume network resources, they were readily available. This observation provides the basis for hoping that we might be able to use β to estimate available bandwidth without fully stressing the network path.

We can gauge how well β truly reflects available bandwidth by computing the coefficient of correlation between β and the connection's overall throughput (normalized by dividing by the bottleneck bandwidth). For $\mathcal{N}_1$, this is 0.44, while, for $\mathcal{N}_2$, it rises to 0.55. We conjecture that the difference is due to the use of bigger windows in $\mathcal{N}_2$ (§ 9.3), which lead to more opportunities for fast retransmission. Any time a connection times out, its overall throughput becomes greatly diluted by the lengthy timeout lull.

Thus, the correlations, particularly for $\mathcal{N}_2$, indicate that β is indeed a solid predictor of a connection's likely overall performance. It is not a perfect predictor, however, nor would we expect it to be: a TCP connection's overall throughput is affected by the number of retransmissions it incurs, whether any of these are timeout retransmissions, the receiver's offered window, the sender's internal window (§ 11.3.2), how the TCP manages the congestion window, and the acking policy used by its remote peer, which determines how fast the slow-start sequence increases the window (§ 11.6.1).

Figure 16.26 and Figure 16.27 show the density of β for $\mathcal{N}_1$ and $\mathcal{N}_2$. Values less than zero and greater than one, which can result from erroneous estimates of ρ_B, have been adjusted

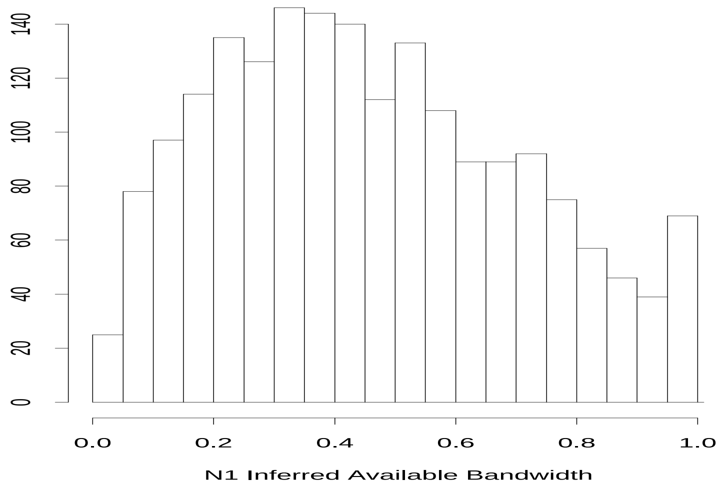

Figure 16.26: Distribution of $\mathcal{N}_1$ inferred available bandwidth (β)

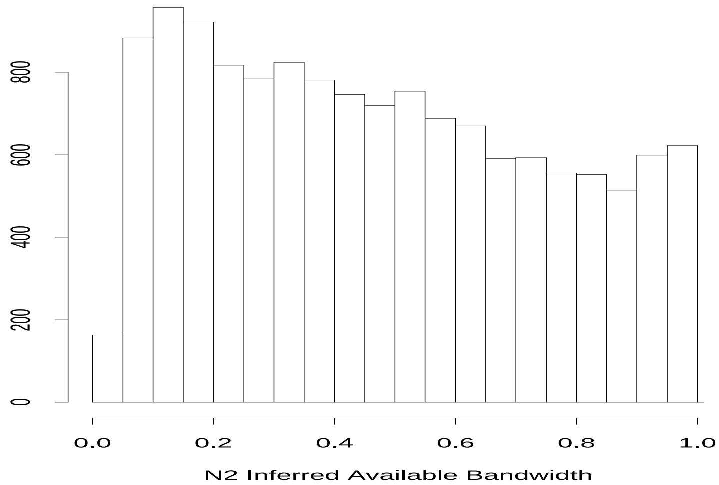

Figure 16.27: Distribution of $\mathcal{N}_2$ inferred available bandwidth (β)

to zero and one, respectively.[12] Clearly, Internet connections encounter a broad range of available bandwidths, ranging from very little to almost all. $\mathcal{N}_1$'s main mode lies at 0.30-0.35, corresponding to about two equivalent competing connections, while for $\mathcal{N}_2$ this shifts considerably downward, to about 0.10-0.15, or eight equivalent competing connections. The overall decrease in β between $\mathcal{N}_1$ and $\mathcal{N}_2$ is clear, though the $\mathcal{N}_2$ density diminishes less quickly than that of $\mathcal{N}_1$, indicating that for it, especially, the range of available bandwidth was indeed very broad. Unfortunately, it is difficult from these statistics to make a definitive statement about how available bandwidth changed over the course of 1995, because the use of bigger windows (§ 9.3) in $\mathcal{N}_2$ means that the notion of "equivalent connection" is different between the two datasets. It is not clear how we could adjust for this difference in order to directly compare the two.

Both densities exhibit two "edge" effects: a greatly diminished density at 0.0-0.05, and a second mode at 0.95–1.0. The first most likely reflects the measurement *bias* our experiment suffers from due to the limited lifetimes of each connection (§ 9.3): those connections for which very little bandwidth was available often did not finish within the allotted ten minutes, and thus do not figure into the measured distribution of β.

The second mode at 0.95–1.0 at first appears to indicate that sometimes a network path is completely quiescent, and packets sail along it without any cross traffic perturbing them. This, however, turns out to only sometimes be the case. Closer inspection of those connections with $\beta \approx 1$ reveals that many are connections with low bottleneck bandwidths. These connections very often are able to completely fill the bottleneck link, because, even if the network can provide only a few non-bottleneck resources to the connection, these still suffice to drive the bottleneck at capacity. That is, the connection requires only modest resources available elsewhere to saturate the bottleneck link and achieve the maximum possible end-to-end performance. We summarize this effect as: *If you only want to go slowly, the network often can provide enough resources for doing so.*

Figure 16.28 and Figure 16.29 show the same densities if we restrict the analysis to connections with $\rho_B \geq 100$ Kbyte/sec. We see that, for $\mathcal{N}_1$, doing so completely eliminates the secondary "all bandwidth available" peak, though, for $\mathcal{N}_2$, it only slightly diminishes it. The difference again appears due to the use of bigger windows in $\mathcal{N}_2$. Figure 16.30 shows the $\mathcal{N}_2$ densities if we restrict ourselves to $\rho_B \geq 250$ Kbyte/sec. Doing so eliminates the T1- and E1-limited connections, which with the bigger windows the $\mathcal{N}_2$ connections could often fill to capacity, much as the $\mathcal{N}_1$ connection could for the slower bottleneck links. Now, the second peak has disappeared, indicating that, at these speeds, the connections could no longer often utilize the entire bottleneck bandwidth.[13] We see that, overall, *as path bandwidths increase, proportionally less bandwidth is available to connections using the path.* This observation is not too surprising: higher bandwidths naturally attract higher traffic loads.

Our observations so far have been based on the load, λ_i, and the bottleneck transmission time, ϕ_i, per Eqn 16.3. Both are computed using the *central* bottleneck bandwidth estimate, ρ_B. The PBM algorithm, however, produces upper and lower *bounds* on the estimate, too, denoted by ρ_B^+ and ρ_B^- (Eqn 14.12). We can accordingly define λ_i^- (ϕ_i^-) and λ_i^+ (ϕ_i^+), based on the upper

[12] We do not discard these connections because sometimes only a slight error in ρ_B will lead to an "out of range" estimate for β, if the connection occurred at a time during which very little or almost all of the bandwidth was available. This point will be developed in more depth shortly.

[13] The depression at 0.0-0.05 has grown, too, a change likely due to the fact that, for high-bandwidth paths, a TCP connection can transfer 100 Kbyte in 10 minutes even in the face of many competing connections, so the measurement bias discussed earlier does not apply to such a large degree.

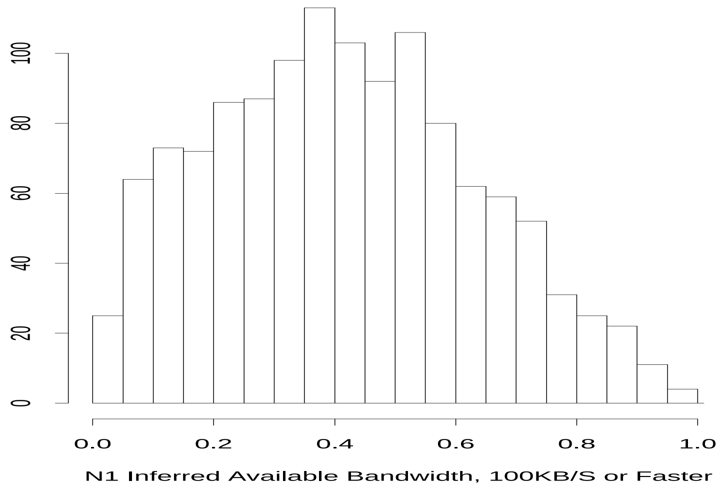

Figure 16.28: Distribution of $\mathcal{N}_1$ inferred available bandwidth (β) for connections with bottleneck rates exceeding 100 Kbyte/sec

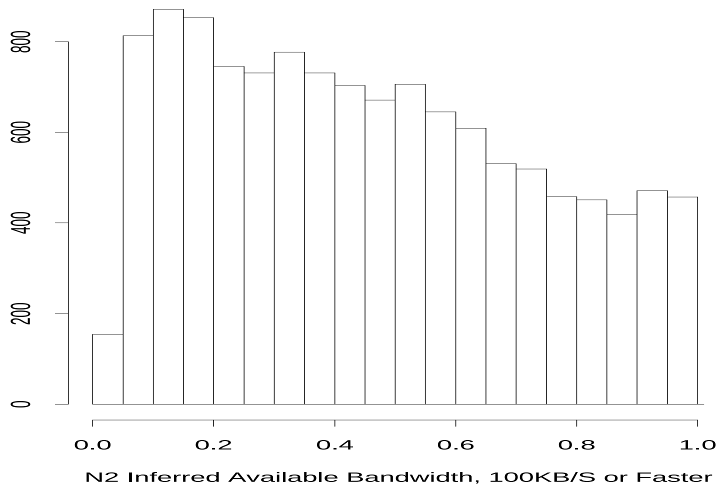

Figure 16.29: Distribution of $\mathcal{N}_2$ inferred available bandwidth (β) for connections with bottleneck rates exceeding 100 Kbyte/sec

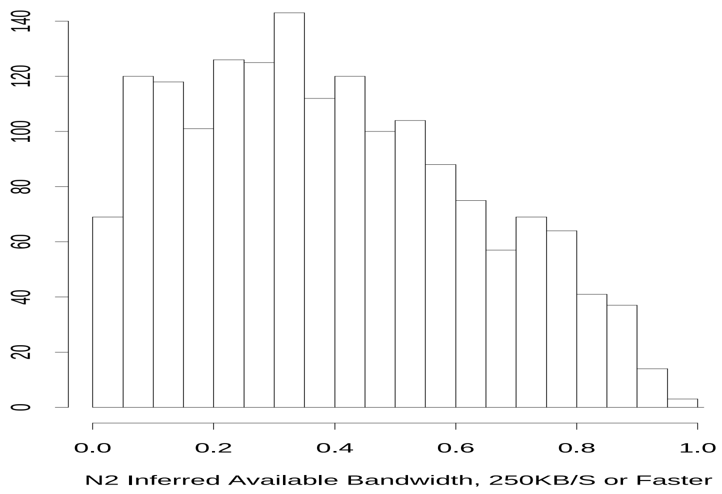

Figure 16.30: Distribution of $\mathcal{N}_2$ inferred available bandwidth (β) for connections with bottleneck rates exceeding 250 Kbyte/sec

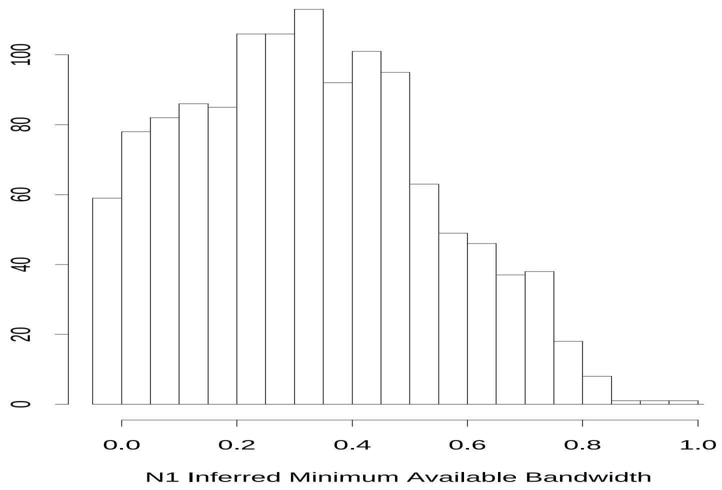

Figure 16.31: Distribution of $\mathcal{N}_1$ minimum inferred available bandwidth (β) for connections with bottleneck rates exceeding 100 Kbyte/sec

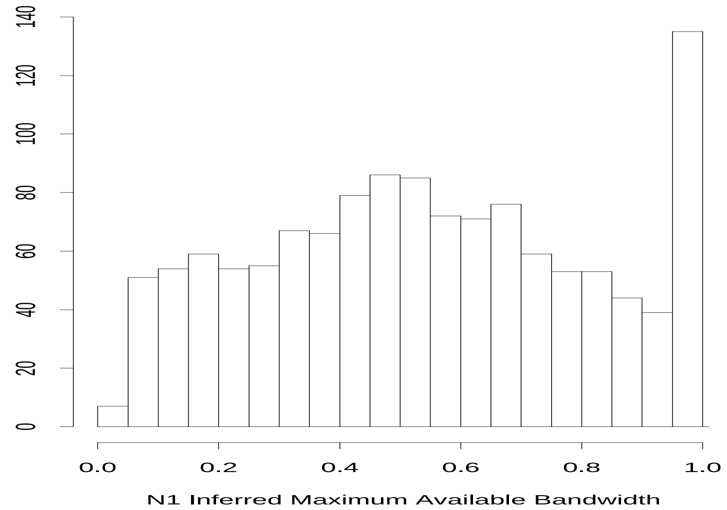

Figure 16.32: Distribution of $\mathcal{N}_1$ maximum inferred available bandwidth (β) for connections with bottleneck rates exceeding 100 Kbyte/sec

and lower bounds, respectively, and from them compute β^- and β^+, lower and upper bounds of the available bandwidth. Figure 16.31 and Figure 16.32 show the densities of β^- and β^+ for the connections in the $\mathcal{N}_1$ dataset with $\rho_B \geq 100$ Kbyte/sec.

The density of β^- fairly closely matches that of β given in Figure 16.28, but shifted about 0.05 to the left, except for the upper regime, which is shifted by about 0.15 to the left. The density of β^+, however, shows roughly the same shape shifted about 0.1 to the right, except for a striking spike at $\beta \approx 1$. This spike is telling: three-quarters of it is for $\beta^+ > 1$, which is an unphysical situation, namely, that the connection's load on the path exceeds the total variation observed on the path. Thus, the spike indicates that ρ_B^-, from which β^+ is derived, is *erroneously too low*. Because it is too low, the corresponding loads, λ_i^+, are too high. Furthermore, the loads can rapidly become *much* too high, due to self-clocking: if the connection is indeed transmitting at exactly the bottleneck rate, which self-clocking will promote in the absence of significant cross-traffic, then each packet's load will be zero, or perhaps will correspond to one additional packet at the bottleneck link if the receiver uses ack-every-other (so the window advances by two packets at a time). In this case, a slightly low estimate of ρ_B^- will result in a determination that the load continually builds up, since the bad estimate will imply that packets are being sent at a rate exceeding the bottleneck's capacity, and hence the queue at the bottleneck grows and grows (per Figure 16.13).

Consequently, we should not trust the variation between β and β^+ as reflecting the true error-bar range in β's density; but that between β and β^- does not suffer from this problem. Based on the latter, then, we conclude that the error in our estimates of β is about ± 0.1, with somewhat lower errors for small values of β, and somewhat higher errors for larger values. This level of error is not large enough to alter any of the conclusions drawn above.

As we might expect, we find that β is inversely correlated with data packet loss rate. For both $\mathcal{N}_1$ and $\mathcal{N}_2$, for connections with $\rho_B \geq 100$ Kbyte/sec, the coefficient of correlation between

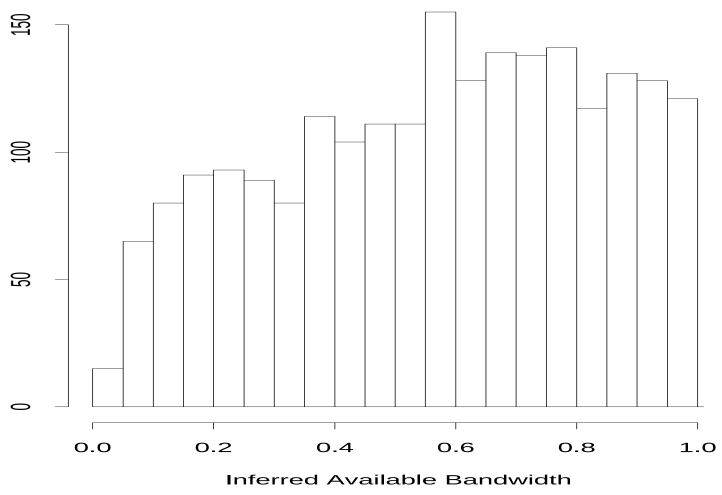

Figure 16.33: Distribution of $\mathcal{N}_2$ inferred available bandwidth (β) for U.S. connections

β and the loss rate is -0.36. This provides us with a solid connection between delay variation and packet loss, which agrees with the widely held assumption that most packet loss in the Internet is due to congestion (which will first lead to delay variations as queues build up). The connection is not overwhelmingly strong, however, which we would also expect, because delay variation need not lead to packet loss if the congested element contains sufficient buffer space to absorb the variation.

That β and loss rate are negatively correlated suggests that we might find significant regional variation in β, much as we did for loss rates in § 15.1. Indeed, we do. Figure 16.33 shows β for connections with $\rho_B \geq 100$ Kbyte/sec and with both sender and receiver sited in the United States. Figure 16.34 shows the same for sender and receiver both sited in Europe. Clearly, European sites suffer from much lower β's than their U.S. counterparts, with the mean (and median) European β at 40%, while for the U.S. connections, it is just under 60%.

The last aspect of available bandwidth we investigate is how it evolves over time. To do so, we group connections with the same source and destination hosts together, after eliminating any with $\rho_B < 100$ Kbyte/sec. For successive connections c and c' in each group, we compute the pair $\langle \Delta T_c, |\Delta\beta_c| \rangle$, where ΔT_c is the time between c and c', and $|\Delta\beta_c|$ is the magnitude of the difference between the computed β's for each connection.

After constructing these pairs, we sort them on ΔT_c and then compute $|\Delta\beta_c|$ smoothed using an exponentially-weighted moving average with $\alpha = 0.01$ and an initial value of 0. Figure 16.35 shows the resulting smoothed evolution for the $\mathcal{N}_2$ dataset. (The $\mathcal{N}_1$ dataset exhibits a similar evolution.) We see that $|\Delta\beta_c|$ almost immediately rises to about 0.12, which is somewhat higher than the error range we estimated for β above, but not greatly higher.[14] This level is sus-

[14]The exponential smoothing, along with starting the averaging with an value of 0, limits how rapidly the plot can reach this level. This is what creates the plotting artifact of what appears to be a rapid climb, falsely suggesting that $|\Delta\beta_c|$ is significantly smaller for very low inter-connection times. A more sound interpretation is that even for very low inter-connection times, we will usually find $|\Delta\beta_c|$ already quite close to 0.12.

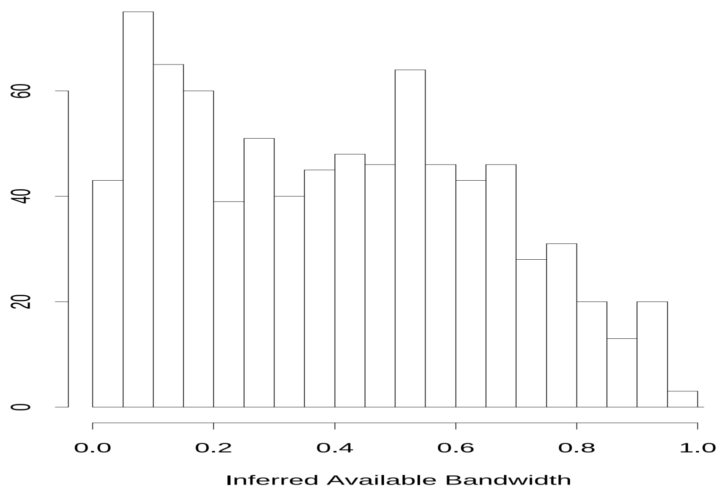

Figure 16.34: Distribution of $\mathcal{N}_2$ inferred available bandwidth (β) for European connections

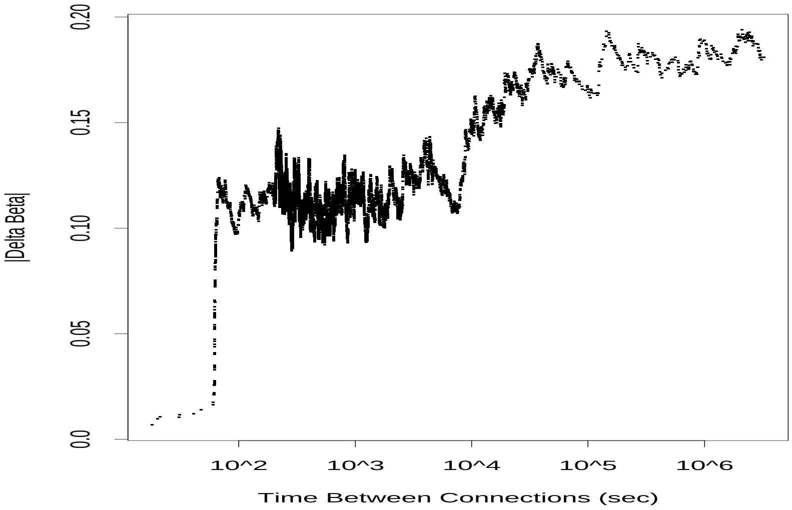

Figure 16.35: Evolution of difference between inferred available bandwidth (β) for successive connections

tained for a number of hours, after which it increases markedly, by about 50%. The transition no doubt coincides with the diurnal cycle we noted in § 15.1: the network is much more congested during working hours than during off hours. Since the predictive power is, qualitatively, fairly good for time scales of several hours, we conclude that transport connections can fruitfully cache information regarding a path's available bandwidth for use in subsequent connections.

Chapter 17

Summary

We endeavored in this work to characterize a number of aspects of end-to-end Internet dynamics in general, meaningful ways. The Internet's great diversity makes this undertaking immensely challenging.

At the heart of our study lies the NPD measurement framework, in which a number of sites around the Internet run a specialized daemon that provides measurement services to authenticated users. The key scaling property of this framework is that, for N participating sites, it can probe $O(N^2)$ Internet paths. This scaling enabled us to probe over 1,000 Internet paths, due to the participation of 37 sites. Consequently, the data for our analysis is more than an order of magnitude richer than that available for previous end-to-end studies, and a serious argument can be made that we can indeed extrapolate our findings to conclusions about Internet paths in general.

17.1 The routing study

In Part I, we used the NPD framework to study the dynamics of end-to-end routing in the Internet, using two experimental runs, one at the end of 1994 and one at the end of 1995. The results were discussed in Chapter 2; here, we briefly summarize them.

We began by characterizing routing pathologies, as we must first identify anomalies before proceeding to analysis of more typical behavior, lest they skew our results. We cataloged a number of pathologies, including loops, outages, and flutter. Furthermore, the prevalence of pathologies significantly increased between the 1994 dataset and the 1995 dataset, indicating that routing degraded over the course of 1995.

We next analyzed routing stability, first developing a distinction between two orthogonal types of stability, routing "prevalence" and routing "persistence." We found that most Internet paths are heavily dominated by a single dominant route, but that the length of time over which routes persist varies greatly, from seconds to many days.

We finished our look at routing with an assessment of routing symmetry. While asymmetries have little direct impact on end-to-end performance, they introduce significant measurement problems, because they cloud the accuracy of the easiest form of measurement, "sender-only" measurement, in which no receiver cooperation is required. We found that about half of all Internet routes exhibited a major asymmetry, in which at least one city differed between the route from A to B versus that from B to A.

17.2 The packet dynamics study

The goal of Part II of our study was to use the NPD framework to measure end-to-end Internet packet dynamics. We recorded over 20,000 TCP transfers at both sender and receiver, again in two experimental runs. Faced with such a large volume of data, we adopted the strategy of developing an analysis tool, `tcpanaly`, for automating the "micro-analysis" of individual connections.

Our goal was to develop meaningful characterizations of end-to-end packet delays. To do so required a great deal of preparatory work, to assure that the analysis rested upon sound measurements.

17.2.1 Measurement calibration and TCP behavior

We first needed to devise techniques for calibrating the measurement data, to assure that we did not misinterpret measurement artifacts for *bona fide* networking effects. The measurement process could fail in two basic ways: by misrecording which packets traversed the network, and by misrecording the times at which they appeared. We found that packet filters can: fail to record packets; record packets more than once; truncate the beginning or end of trace files; and rearrange the sequencing of packets. Accordingly, we developed tests so that `tcpanaly` can detect these events. We further developed the important notion of a packet filter's "vantage point," meaning where in the network path it observed the traffic. A filter's vantage point can introduce ambiguities in the apparent chain of cause-and-effect, which can only be removed with considerable care.

Hand-in-hand with calibrating the integrity of the traffic traces comes the problem of identifying the exact behavior of the TCP implementations used by the sending and receiving hosts. Often, the only way to accurately gauge the integrity of a traffic trace is by knowing in intimate detail how the TCPs participating in the connection behave and respond. Apparent deviations from this behavior then indicate a likely lack of integrity in the traffic trace, if the behavior has indeed been correctly characterized.

`tcpanaly` holds promise as a valuable tool for analyzing TCP behavior, useful both in its own right for diagnosing performance and congestion problems, and also as a way to account for the separate effects on a connection's dynamics of the behavior of the TCP endpoints versus that of the connection's Internet path. In the course of its development, we found a wide range of TCP behaviors, some of which have major, negative performance and stability implications for the associated TCPs. The most serious problems include excessive retransmissions and failures to correctly diminish the transmission rate during periods of congestion. Indeed, if some of these TCPs were ubiquitous in the Internet, the network would quite simply cease to function, due to "congestion collapse."

In the process of this analysis, we observed that the TCPs with the most serious problems were the only two in our study written independently from the "BSD-derived" implementations that directly benefited from much of the fundamental TCP research. To investigate this observation, we analyzed three additional implementations, finding a mid-level performance problem in one, a major performance problem in another (but one possibly due to use of a specific network interface card), and severe performance and stability problems in the third. Thus, our findings strongly argue that implementing TCP correctly is exceptionally difficult. Given that Internet stability *relies* on TCP correctness, it therefore behooves the Internet community to take energetic steps towards providing analysis tools and reference implementations to aid the efforts of implementors.

17.2.2 Timing calibration

Armed with the ability to detect inaccurate packet traces and to distinguish between TCP-induced effects and networking effects, we next turned to the difficult problem of calibrating the packet timings. The effort continued to be driven by the ultimate goal of analyzing end-to-end packet delays. To do so requires comparing pairs of unsynchronized clocks, namely those used by the tracing programs at the sender and receiver. We developed algorithms for (1) estimating clock resolution, (2) synchronizing clocks *post facto*, (3) detecting clock adjustments, and (4) detecting and removing relative clock skew. This last is particularly important because, if undetected, relative clock skew leads to variations in apparent packet delays quite similar to those of genuine networking effects. We found that it is fairly common for a pair of clocks to exhibit discernible relative skew. We also found that the fact that two clocks agree quite closely does *not* eliminate the possibility that the clocks suffer from problems such as adjustments and relative skew.

17.2.3 Network pathologies

With our measurements fully calibrated, we could then turn to analyzing packet dynamics. We began by characterizing packet-forwarding pathologies: out-of-order delivery, packet replication, and packet corruption. We found that the frequency with which packets arrive in a different order than sent varies enormously among Internet paths. While reordering often occurs in conjunction with the route "flutter" pathology, we also observed numerous instances in which it occurred in the absence of flutter, and some instances in which massive reordering events occurred due to "pauses" in router forwarding. Finally, the possibility of reordering limits how quickly a TCP sender can infer a packet loss using the "fast retransmission" mechanism. We investigated whether, based on our data, this mechanism could be altered to retransmit more efficiently. We found that we could only do so if we required changes at both the TCP sender and receiver. Consequently, we might as well instead change the sender and receiver to use the more sophisticated TCP "selective acknowledgement" extension, now being standardized [MMFR96].

We found that the curious phenomenon of packet replication—the network delivering a single packet more than once—does indeed occur, but it is exceptionally rare. On the other hand, our analysis of packet corruption suggests that, overall, about 1 Internet data packet in 5,000 arrives with data different than what was originally sent. This rate is high enough that, given TCP's 16-bit checksum, about one packet in 300,000,000 will be accepted with undetected errors. The Internet carries many more packets than this each day.

17.2.4 Estimating bottleneck bandwidth

We next turned to the problem of identifying a network path's *bottleneck bandwidth*. We needed to do so before analyzing packet loss and delay because the bottleneck bandwidth determines what we call the "self-interference time constant," Q_b. Two data packets of size b sent less than an interval Q_b apart must necessarily queue at the bottleneck element of the network path. Thus, knowledge of Q_b enables us to determine which of our measurement probes were perforce correlated. It further plays a major role in assessing packet loss, because we want to distinguish between the loss of data packets that we know had to queue behind their predecessors ("self-interference"), versus those lost even though they did not have to queue on account of the connection's own loading of the network path.

We discussed how the main existing technique for estimating bottleneck bandwidth, "packet pair," could produce incorrect estimates. These can occur in the presence of: excessive noise; packet reordering; changes in the bottleneck bandwidth; or network paths in which the bottleneck is comprised of multiple, separate channels or links. This last case is particularly interesting, because it leads to erroneously large bottleneck estimates even if the network is completely quiescent. The problem lies in the fundamental assumption made by packet pair that packets must queue behind one another at the bottleneck and be served by it one at a time. For a multi-channel or multi-link bottleneck, however, this assumption does not in fact apply, and a pair of packets can traverse the bottleneck without it altering the spacing between them.

These observations motivated us to devise a robust algorithm for estimating bottleneck bandwidth, based on "packet bunch modes" (PBM). By focussing on identifying multiple modes in the distribution of the estimated bottleneck bandwidth, PBM can accommodate errors introduced by noise, as well as detecting changes in bottleneck bandwidth and the presence of multi-channel links. By using receiver-based measurement, it also can cope with packet reordering, and with the possibility of asymmetries in the bottleneck bandwidths along the two directions of a network path.

We calibrated PBM by testing whether we could associate known, common link speeds with its estimates. We found that we could almost always do so. Once we had faith in PBM's accuracy, we could then test other estimation methods against PBM to see how well they perform. We found that receiver-based packet pair performs almost as well, if we can tolerate failing to detect shifts in bottleneck bandwidth or multi-channel links, both of which prove rare. Sender-based packet pair, however, does not perform nearly as well, due to the additional noise incurred by measuring timings that reflect the traversal of packets in both of a path's directions. Finally, we find that about 20% of the time, a path's two directions have *asymmetric* bottleneck bandwidths, but that, along a single direction, the bottleneck generally remains constant over lengthy periods of time.

One drawback with PBM is that it is ad hoc to an unsatisfying degree. It uses a considerable number of heuristics that can only be defended on the basis that they appear to work well in practice. We found this acceptable (if regrettable), because for our study bottleneck bandwidth estimation was fundamentally only a stepping stone to the later analysis, and not an end in itself. We hope, however, that the basic ideas underlying PBM—searching for multiple modes and interpreting the ways they overlap in terms of bottleneck changes and multi-channel paths—might be revisited in the future, in an attempt to develop them in a more systematic fashion.

17.2.5 Packet loss

We now could turn to analyzing patterns of packet loss in the Internet. We found that over the course of 1995, packet loss rates *nearly doubled*, indicating a marked degradation in service. However, these rates required further inspection to understand their implications. We first developed the notion of the network having two general states, "quiescent," corresponding to periods of no loss, and "busy," corresponding to periods in which connections observe at least one loss. The proportion of quiescent connections did not change appreciably during 1995; instead, the loss rate increases were due to higher levels of loss during busy periods.

We also distinguished between three different types of lost packets: "loaded" data packets, meaning those that necessarily had to wait at the bottleneck behind one or more of their predecessors; "unloaded" data packets, meaning those that did not have to queue behind predecessors, unless cross traffic arrived and delayed their predecessors; and acknowledgements.

We found that loaded packets are much more likely to suffer high loss rates than unloaded packets, which is not surprising, since they encounter not only the ambient network load but that of their predecessors; and that acks are more likely to be lost than unloaded packets (or even loaded packets, for high loss rates). We interpret these findings as reflecting the fundamental difference between data packets being sent at a rate that *adapts* in an effort to diminish packet loss, and acks being sent at a rate that does *not* adapt to the rate at which acks are lost. This finding highlights how the loss rates observed by a TCP connection's data packets *differ* from the unconditional loss rates along the path they traverse.

The last comparison between data packet and ack loss rates we made was to determine the degree of correlation between the two rates for a single connection. We found that the two are nearly uncorrelated, indicating that this fundamental property of a network path is *asymmetric*.

We next found that different major regions of the Internet—the United States, Europe, and connections from one to the other—experienced very different loss rates. Then, after showing that loss rates follow the well-known diurnal cycle reflecting working hours and off-work hours, we analyzed variations in the time of day during which our measurement apparatus succeeded in executing a measurement. For North American sites, these successes were uniformly spread over the 24 hours of each day. For European sites, though, the frequency of successes dipped to low points in patterns that closely matched the loss-rate cycle, indicating that our European measurements suffered from a discernible *bias* towards underestimating loss rates.

Another question we investigated was whether packet loss events are well-modeled as independent, since this assumption is sometimes made when theorizing about network behavior. We found that loss events are instead strongly correlated. Furthermore, the duration of loss "outages" exhibits infinite variance, which accords with a recent model of how individual connection behavior can give rise to "self-similar" aggregate traffic behavior [WTSW95].

We then looked at the question of *where* packets are lost along an Internet path. In particular, whether they are lost before or after the bottleneck element. From careful analysis of timing information we can sometimes distinguish between these two. We found that, while most losses occur at or before the bottleneck, a significant minority (roughly 25%) occur after.

We next evaluated how packet loss rates evolve over time, with an eye towards gauging the efficacy of caching packet loss statistics associated with a path in order to predict future path performance. We found that a path's state, in terms of "quiescent" or "busy," is a good predictor of its future state for many hours, but a path's observed loss rate is *not* a good predictor of its future loss rate.

We then investigated how efficiently TCP implementations retransmit. We found that, for some implementations, the large majority of their retransmissions are unnecessary. Fixing these implementations and deploying the SACK extension would eliminate nearly all of the unnecessary retransmissions.

17.2.6 Packet delay

We finished our study with an analysis of end-to-end packet transit delays. We found that both round-trip times (RTTs) and one-way transit times (OTTs) exhibit great "peak-to-peak" variation. OTT variations for the most part are asymmetric. The only clear correlation occurs between the order-of-magnitude (logarithm) variation in the two directions. On the other hand, OTT variation is clearly correlated with packet loss rates, as we would expect. We further found that OTT variation is not a good predictor of future OTT variation, in accord with the finding that packet loss rates are not good predictors of future loss rates.

We then turned to an assessment of packet *timing compression*, in which a group of packets arrives at their receiver more closely spaced than when they were sent. We identify three types of compression: ack compression, data packet compression, and receiver compression. Each requires somewhat different assessment considerations. Overall, none of the three types occur frequently enough to pose a significant problem in terms of network performance and stability. Their presence does, however, complicate path measurement efforts, which must use judicious filtering to avoid mistaking compression events for different network effects, such as a temporary increase in bottleneck bandwidth.

We next investigated the *time scales* over which queueing occurs, by determining on which time scales we observed the maximum sustained and peak OTT variations. We found that both occur most frequently on time scales of about 100–1000 msec, though, as with many Internet phenomena, we also found a wide range of behavior beyond this region. (In particular, we sometimes found maximal queueing occurring on much longer time scales.)

The last aspect of packet delay we analyzed was the degree to which it reflects *available bandwidth*. We did this by studying the ratio between the delay a packet incurred due to its connection's own loading of the network path, versus the total delay it incurred. This ratio correlates well with the overall throughput achieved by a connection. However, we also showed that the accuracy of the ratio is diminished by the presence of errors in estimating the bottleneck bandwidth.

We observed a distinct decrease in available bandwidth over the course of 1995, though we also observed significant regional variation, with U.S. sites enjoying considerably more available bandwidth than European sites. Finally, we investigated how available bandwidth evolves over time. We found that a connection's available bandwidth is a fairly good predictor of future available bandwidth out to time scales of hours.

17.3 Future research

There are three general areas of future work suggested by our research. First, our original goal when proposing the research was to use end-to-end measurements to drive the development of new algorithms for how transport protocols can adapt to changing network conditions. We had to abandon this goal once the scope of analyzing the measurements themselves became apparent, but clearly an important potential benefit of end-to-end characterization such as we have undertaken is to better optimize how connections use the network.

Closely related to developing such new algorithms is the question of *fast* estimation of Internet path behavior. The algorithms we developed for calibrating network clocks (Chapter 12), estimating bottleneck bandwidth (Chapter 14), and assessing queueing time scales and available bandwidth (Chapter 16) all in their present form analyze entire connection traces. Yet, transport connections clearly need to make decisions based on path properties quickly, and cannot afford the luxury of analyzing the fate of several hundred packets. Our work, though, can play a key role in developing fast estimation techniques, because the algorithms we developed can then be used to calibrate the faster algorithms.

Finally, the NPD framework serves well to address the issue of capturing reasonably representative samples of a cross-section of Internet path behavior. Another important form of Internet heterogeneity, however, is how Internet traffic changes *with time*. Only longitudinal studies can address such "temporal" heterogeneity. We have attempted to touch on this issue by capturing two datasets spaced a year apart. Clearly, though, we need longer-term studies to develop solid conclusions about traffic trends. We believe this goal can be met in conjunction with the development of an Internet "measurement infrastructure," that is, large-scale deployment of NPD-like measurement platforms. We do not claim that the NPD framework can simply be scaled up to serve as this infrastructure; indeed, the problem of an infrastructure that *can* scale to the full Internet is the key research problem for the infrastructure. But, if accomplished, such an infrastructure could serve, through the accumulated archives of its measurements, as the basis for longitudinal studies; and, even more significantly, as a mechanism for assessing and improving the overall health of the network.

17.4 Themes of the work

Several themes emerge from our study:

- The N^2 scaling property of our measurement framework serves to measure a sufficiently diverse set of Internet paths that we might plausibly interpret the resulting analysis as accurately reflecting general Internet behavior.
- To cope with such large-scaled measurements requires attention to calibration using self-consistency checks; robust statistics to avoid skewing by outliers; and automated "micro-analysis," such as that performed by `tcpanaly`, that we might see the forest as well as the trees.
- With due diligence to remove packet filter errors and TCP effects, TCP-based measurement provides a viable means for assessing end-to-end packet dynamics.
- We find wide ranges of behavior, so we must exercise great caution in regarding any aspect of packet dynamics as "typical."
- Some common assumptions such as in-order packet delivery, FIFO bottleneck queueing, independent loss events, single congestion time scales, and path symmetries are all violated, sometimes frequently.
- The combination of path asymmetries and reverse-path noise renders sender-only measurement techniques markedly inferior to those that include receiver cooperation.

This last point argues that, when the measurement of interest concerns a unidirectional path—be it for measurement-based adaptive transport techniques such as TCP Vegas [BOP94], or general Internet performance metrics such as those in development by the IPPM effort [A+96, Pa96a]—the extra complications incurred by coordinating the sender and receiver are worth the effort.

Finally, we believe an important aspect of this work is how it might contribute towards developing a "measurement infrastructure" for the Internet: one that proves ubiquitous, informative, and sound.

Bibliography

[A+96] G. Almes et al., "Framework for IP Provider Metrics," Internet draft, *ftp://ftp.isi.edu/internet-drafts/draft-ietf-bmwg-ippm-framework-00.txt*, Nov. 1996.

[AW96] M. Arlitt and C. Williamson, "Web Server Workload Characterization: The Search for Invariants," *Proceedings of SIGMETRICS '96*, Philadelphia, May 23-26, 1996.

[Aw90] B. Awerbuch, "Shortest Paths and Loop-Free Routing in dynamic networks (Extended Abstract)," *Proceedings of SIGCOMM '90*, pp. 177-187, September 1990.

[Ba95] F. Baker, Ed., "Requirements for IP Version 4 Routers," RFC 1812, DDN Network Information Center, June 1995.

[Ba94] A. Banerjea, *Fault Management for Realtime Networks*, Ph.D. thesis, University of California, Berkeley, 1994.

[BDG95] C. Baransel, W. Dobosiewicz, and P. Gburzynski, "Routing in Multihop Packet Switching Networks: Gb/s Challenge," *IEEE Network*, 9(3), pp. 38-61, May/June 1995.

[BCW88] R. Becker, J. Chambers, and A. Wilks, *The New S Language*, Wadsworth & Brooks/Cole, 1988.

[Be95] S. Bellovin, "Using the Domain Name System for System Break-ins," *Proceedings of the 5th USENIX UNIX Security Symposium*, Salt Lake City, June 1995.

[BCLF+] T. Berners-Lee et al., "The World-Wide Web," *Communications of the ACM*, 37(8), pp. 76-82, August 1994.

[Be82] D. Bertsekas, "Dynamic Behavior of Shortest Path Routing Algorithms for Communication Networks," *IEEE Transactions on Automatic Control*, AC-27, pp. 60-74, February 1982.

[BM92] I. Bilinskis and A. Mikelsons, *Randomized Signal Processing*, Prentice Hall International, 1992.

[Bi95] P.G. Bilse, private communication, October 16, 1995.

[Bo93] J-C. Bolot, "End-to-End Packet Delay and Loss Behavior in the Internet," *Proceedings of SIGCOMM '93*, pp. 289-298, September 1993.

[BCG95] J-C. Bolot, H. Crépin, and A.V. Garcia, “Analysis of Audio Packet Loss in the Internet,” *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, Durham, New Hampshire, April 1995.

[BBJ92] D. Borman, R. Braden and V. Jacobson, “TCP Extensions for High Performance,” RFC 1323, Network Information Center, SRI International, Menlo Park, CA, May 1992.

[BJ88] R. Braden and V. Jacobson, “TCP extensions for long-delay paths,” RFC 1072, Network Information Center, SRI International, Menlo Park, CA, October 1988.

[Br89] R. Braden, Ed., “Requirements for Internet Hosts—Communication Layers,” RFC 1122, Network Information Center, SRI International, Menlo Park, CA, October 1989.

[Br94] R. Braden, “T/TCP — TCP Extensions for Transactions: Functional Specification,” RFC 1644, DDN Network Information Center, July 1994.

[BCS94] R. Braden, D. Clark, and S. Shenker, “Integrated Services in the Internet Architecture: an Overview,” RFC 1633, DDN Network Information Center, June 1994.

[BOP94] L. Brakmo, S. O'Malley, and L. Peterson, “TCP Vegas: New Techniques for Congestion Detection and Avoidance,” *Proceedings of SIGCOMM '94*, pp. 24-35, September 1994.

[BP95a] L. Brakmo and L. Peterson, “TCP Vegas: End to End Congestion Avoidance on a Global Internet,” *IEEE JSAC*, 13(8), pp. 1465-1480, October 1995.

[BP95b] L. Brakmo and L. Peterson, “Performance Problems in BSD4.4 TCP,” *Computer Communication Review*, 25(5), pp. 69-84, October 1995.

[BE90] L. Breslau and D. Estrin, “Design of Inter-Administrative Domain Routing Protocols,” *Proceedings of SIGCOMM '90*, pp. 231-241, September 1990.

[BMR97] N. Brownlee, C. Mills, and G. Ruth, “Traffic Flow Measurement: Architecture,” RFC 2063, DDN Network Information Center, January 1997.

[CC96a] R. Carter and M. Crovella, “Measuring Bottleneck Link Speed in Packet-Switched Networks,” Technical Report BU-CS-96-006, Computer Science Department, Boston University, March 1996.

[CC96b] R. Carter and M. Crovella, “Dynamic Server Selection using Bandwidth Probing in Wide-Area Networks,” Technical Report BU-CS-96-007, Computer Science Department, Boston University, March 1996.

[Cha95] C. Chatfield, “Model uncertainty, data mining and statistical inference,” *Journal of the Royal Statistical Society* **A**, 158:419–466, 1995.

[Che95] E. Chen, “Symmetric routing in a Multi-Provider Internet,” North American Network Operators' Group, May 1995 Meeting Notes, S. Barber, Ed., *http://www.academ.com/nanog/may1995/symmetric.html*.

[CB94] W. Cheswick and S. Bellovin, *Firewalls and Internet Security: Repelling the Wily Hacker,* Addison-Wesley, 1994.

[Ch93] B. Chinoy, “Dynamics of Internet Routing Information,” *Proceedings of SIGCOMM '93*, pp. 45-52, September 1993.

[CBP94] K. Claffy, H-W. Braun and G. Polyzos, “Tracking Long-Term Growth of the NSFNET,” *Communications of the ACM*, 37(8), pp. 34-45, Aug. 1994.

[CBP95] K. Claffy, H-W. Braun and G. Polyzos, “A Parameterizable Methodology for Internet Traffic Flow Profiling,” *IEEE JSAC*, 13(8), pp. 1481-1494 October 1995.

[CPB93a] K. Claffy, G. Polyzos and H-W. Braun, “Measurement Considerations for Assessing Unidirectional Latencies,” *Internetworking: Research and Experience*, 4 (3), pp. 121-132, September 1993.

[CPB93b] K. Claffy, G. Polyzos, and H-W. Braun, “Traffic Characteristics of the T1 NSFNET Backbone,” *Proceedings of INFOCOM '93*, San Francisco, March, 1993.

[Cl82] D. Clark, “Window and Acknowledgement Strategy in TCP,” RFC 813, Network Information Center, SRI International, Menlo Park, CA, July 1982.

[Cl88] D. Clark, “The Design Philosophy of the DARPA Internet Protocols,” *Proceedings of SIGCOMM '88*, pp. 106-114, August 1988.

[CJRS89] D. Clark, V. Jacobson, J. Romkey, and H. Salwen, “An Analysis of TCP Processing Overhead,” *IEEE Communications*, pp. 23-29, June 1989.

[CSZ92] D. Clark, S. Shenker, and L. Zhang, “Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism,” *Proceedings of SIGCOMM '92*, pp. 14-26, August 1992.

[Co90] S. Cohn, “Arpanet Routing,” in *Fault-Tolerant Distributed Computing*, B. Simons and A. Spector, editors, Springer-Verlag, 1990.

[CL94] D. Comer and J. Lin, “Probing TCP Implementations,” *Proceedings of the 1994 Summer USENIX Conference*, Boston, MA.

[Co91-95] A. Cooper, Ed., “Internet Monthly Reports,” *http://www.isi.edu:80/in-notes/imr/*.

[CB96] M. Crovella and A. Bestavros, “Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes,” *Proceedings of SIGMETRICS '96*, Philadelphia, May 23-26, 1996.

[CW91] J. Crowcroft and I. Wakeman, “Traffic Analysis of some UK-US Academic Network Data,” *Proceedings of INET '91*, Copenhagen, June 1991.

[DS86] R. B. D'Agostino and M. A. Stephens, editors, *Goodness-of-Fit Techniques*, Marcel Dekker, Inc., 1986.

[DHS93] P. Danzig, R. Hall, and M. Schwartz, “A Case for Caching File Objects Inside Internetworks,” *Proceedings of SIGCOMM '93*, San Francisco, September 1993.

[DJCME92] P. Danzig, S. Jamin, R. Cáceres, D. Mitzel, and D. Estrin, "An Empirical Workload Model for Driving Wide-area TCP/IP Network Simulations," *Internetworking: Research and Experience*, 3(1), pp. 1-26, 1992.

[DOK92] P. Danzig, K. Obraczka, and A. Kumar, "An Analysis of Wide-Area Name Server Traffic," *Proceedings of SIGCOMM '92*, Baltimore, Aug. 1992.

[DJM97] S. Dawson, F. Jahanian, and T. Mitton, "Experiments on Six Commercial TCP Implementations Using a Software Fault Injection Tool," to appear in *Software: Practice & Experience*.

[DC90] S. Deering and D. Cheriton, "Multicast Routing in Datagram Internetworks and Extended LANs," *ACM Transactions on Computer Systems*, 8(2), pp. 85-110, May 1990.

[DEFJLW94] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C-G. Liu, and L. Wei, "An Architecture for Wide-Area Multicast Routing," *Proceedings of SIGCOMM '94*, pp. 126-135, September 1994.

[DB95] L. Delgrossi and L. Berger, Ed., "Internet Stream Protocol Version 2 (ST2), Protocol Specification — Version ST2+," RFC 1819, DDN Network Information Center, August 1995.

[DISA95] Defense Information Systems Agency DDN Program Office, *ftp://nic.ddn.mil/netinfo/asn.txt*; August 1995.

[Do95] S. Doran, "Route Flapping," with notes by Stan Barber, *http://www.merit.edu/routing.arbiter/NANOG/2.95.NANOG.notes/route-flapping.html.*

[DMT96] R. Durst, G. Miller and E. Travis, "TCP Extensions for Space Communications," *Proceedings of MOBICOM '96*, pp. 15-26, November 1996.

[El96] R. Elz, private communication, January 12, 1996.

[ERH92] D. Estrin, Y. Rekhter and S. Hotz, "Scalable Inter-Domain Routing Architecture," *Proceedings of SIGCOMM '92*, pp. 40-52, August 1992.

[EHS92] D. Ewing, R. Hall, and M. Schwartz, "A Measurement Study of Internet File Transfer Traffic," *Report CU-CS-571-92*, Department of Computer Science, University of Colorado, Boulder, 1992.

[FF96] K. Fall and S. Floyd, "Simulation-based Comparisons of Tahoe, Reno, and SACK TCP," *Computer Communication Review*, 26(3), pp. 5-21, July 1996.

[Fe90] D. Ferrari, "Client Requirements for Real-Time Communication Services," *IEEE Communications*, pp. 65-72, November 1990.

[FBZ94] D. Ferrari, A. Banerjea and H. Zhang, "Network support for multimedia: A discussion of the Tenet approach," *Computer Networks and ISDN Systems*, 26(10), pp. 1267-1280, July 1994.

[FGV95] D. Ferrari, A. Gupta and G. Ventre, "Distributed Advance Reservation of Real-Time Connections," *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, Durham, New Hampshire, April 1995.

[Fl91] S. Floyd, "Connections with Multiple Congested Gateways in Packet-Switched Networks, Part 1: One-way Traffic," *Computer Communication Review*, 21(5), pp. 30-47, October 1991.

[FJ92] S. Floyd and V. Jacobson, "On Traffic Phase Effects in Packet-switched Gateways," *Internetworking: Research and Experience*, 3(3), pp. 115-156, September 1992.

[FJ93] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *IEEE/ACM Transactions on Networking*, 1(4), pp. 397-413, August 1993.

[FJ94] S. Floyd and V. Jacobson, "The Synchronization of Periodic Routing Messages," *IEEE/ACM Transactions on Networking*, 2(2), pp. 122-136, April 1994.

[FL91] H. Fowler and W. Leland, "Local Area Network Traffic Characteristics, with Implications for Broadband Network Congestion Management," *IEEE JSAC*, 9(7), pp. 1139-1149, September 1991.

[FJ70] E. Fuchs and P. E. Jackson, "Estimates of Distributions of Random Variables for Certain Computer Communications Traffic Models," *Communications of the ACM*, 13(12), pp. 752-757, December 1970.

[GR95] A. Gupta and K. Rothermel, "Fault handling for multi-party real-time communication," Technical Report TR-95-059, International Computer Science Institute, University of California, Berkeley, October 1995.

[G90] R. Gusella, "A Measurement Study of Diskless Workstation Traffic on an Ethernet," *IEEE Transactions on Communications*, 38(9), pp. 1557-1568, September 1990.

[GK97] E. Gustafsson and G. Karlsson, "A Literature Survey on Traffic Dispersion," *IEEE Network*, 11(2), pp. 28-36, March/April 1997.

[HK89] S. Hares and D. Katz, "Administrative Domains and Routing Domains: A Model for Routing in the Internet," RFC 1136, Network Information Center, SRI International, Menlo Park, CA, December, 1989.

[HSF85] K. Harrenstien, M. Stahl, and E. Feinler, "NICNAME/WHOIS," RFC 954, Network Information Center, SRI International, Menlo Park, CA, 1985.

[He90] S. Heimlich, "Traffic Characterization of the NSFNET National Backbone," Proceedings of the 1990 Winter USENIX Conference, Washington, D.C.

[HMT83] D. Hoaglin, F. Mosteller, and J. Tukey, Ed., "Understanding Robust and Exploratory Data Analysis," John Wiley & Sons, 1983.

[Ho96] J. Hoe, "Improving the Start-up Behavior of a Congestion Control Scheme for TCP," *Proceedings of SIGCOMM '96*, pp. 270-280, August 1996.

[Hu95] C. Huitema, *Routing in the Internet*, Prentice Hall PTR, 1995.

[HP91] N. Hutchinson and L. Peterson, "The x-kernel: An architecture for implementing network protocols," *IEEE Transactions on Software Engineering*, 17(1), pp. 64-76, January 1991.

[Ja88] V. Jacobson, "Congestion Avoidance and Control," *Proceedings of SIGCOMM '88*, pp. 314-329, August 1988.

[Jac89] V. Jacobson, *traceroute*, *ftp://ftp.ee.lbl.gov/traceroute.tar.Z*, 1989.

[JLM89] V. Jacobson, C. Leres, and S. McCanne, `tcpdump`, available via anonymous ftp to ftp.ee.lbl.gov, June 1989.

[Jac90] V. Jacobson, "Compressing TCP/IP headers for low-speed serial links," RFC 1144, Network Information Center, SRI International, Menlo Park, CA, February 1990.

[JLM97] F. Jahanian, C. Labovitz, and G. Malan, "Internet Routing Instability," to appear in *Proceedings of SIGCOMM '97*, September 1997.

[Jai89] R. Jain, "A Delay-Based Approach for Congestion Avoidance in Interconnected Heterogeneous Computer Networks," *Computer Communication Review*, 19(5), pp. 56-71, October 1989.

[Jai90] R. Jain, "Performance Analysis of FDDI Token Ring Networks: Effect of Parameters and Guidelines for Setting TTRT," *Proceedings of SIGCOMM '90*, pp. 264-275, September 1990.

[JR86] R. Jain and S. Routhier, "Packet Trains — Measurements and a New Model for Computer Network Traffic," *IEEE JSAC*, 4(6), pp. 986-995, September, 1986.

[KP87] P. Karn and C. Partridge. "Estimating round-trip times in reliable transport protocols," *Proceedings of SIGCOMM '87*, August 1987.

[Ke91] S. Keshav, "A Control-Theoretic Approach to Flow Control," *Proceedings of SIGCOMM '91*, pp. 3-15, September 1991.

[KZ89] A. Khanna and J. Zinky, "The Revised ARPANET Routing Metric," *Proceedings of SIGCOMM '89*, pp. 45-56, September 1989.

[Kl76] L. Kleinrock, "Queueing Systems, Volume II: Computer Applications," John Wiley & Sons, 1976.

[LTWW94] W. Leland, M. Taqqu, W. Willinger, and D. Wilson, "On the Self-Similar Nature of Ethernet Traffic (Extended Version)," *IEEE/ACM Transactions on Networking*, 2(1), pp. 1-15, February 1994.

[Lid96] K. Lidl, private communication, January 3, 1996.

[Lin96] T. Lindgreen, private communication, January 12, 1996.

[Li89] M. Little, “Goals and Functional Requirements for Inter-Autonomous System Routing,” RFC 1126, Network Information Center, SRI International, Menlo Park, CA, October, 1989.

[LMG95] D. Long, A. Muir and R. Golding, “A longitudinal survey of Internet host reliability,” Technical Report UCSC-CRL-95-16, University of California, Santa Cruz, 1995.

[Lo95] M. Lottor, *ftp://nic.merit.edu/nsfnet/statistics*; October 1995.

[Lo97] M. Lottor, *http://www.nw.com/zone/WWW/top.html*; February 1997.

[MMFR96] M. Mathis, J. Mahdavi, S. Floyd and A. Romanow, “TCP Selective Acknowledgment Options,” RFC 2018, DDN Network Information Center, Oct. 1995.

[MM96] M. Mathis and J. Mahdavi, “Diagnosing Internet Congestion with a Transport Layer Performance Tool,” *Proceedings of INET '96*, Montreal, June 1996.

[MJ93] S. McCanne and V. Jacobson, “The BSD Packet Filter: A New Architecture for User-level Packet Capture,” *Proceedings of the 1993 Winter USENIX Conference*, San Diego, CA.

[MLJ94] S. McCanne, C. Leres and V. Jacobson, `libpcap`, available via anonymous ftp to ftp.ee.lbl.gov, 1994.

[MFR78] J. McQuillan, G. Falk and I. Richer, “A Review of the Development and Performance of the ARPANET Routing Algorithm,” *IEEE Transactions on Communications*, 26(12), pp. 1802-1811, December 1978.

[MRR80] J. McQuillan, I. Richer and E. Rosen, “The New Routing Algorithm for the ARPANET,” *IEEE Transactions on Communications*, 28(5), pp. 711-719, May 1980.

[Me95a] Merit Network, Inc., *ftp://nic.merit.edu/nsfnet/statistics/history.nets*; May 1995.

[Me95b] Merit Network, Inc., “The Routing Arbiter” home page. *http://www.merit.edu/routing.arbiter/RA/* describes the project as a whole, and *http://nic.merit.edu/routing.arbiter/RA/statistics/flap.html* gives “route flap” statistics.

[Mi83] D. Mills, “Internet Delay Experiments,” RFC 889, Network Information Center, SRI International, Menlo Park, CA, 1983.

[Mi92a] D. Mills, “Network Time Protocol (Version 3): Specification, Implementation and Analysis,” RFC 1305, Network Information Center, SRI International, Menlo Park, CA, March 1992.

[Mi92b] D. Mills, “Modelling and Analysis of Computer Network Clocks,” Technical Report 92-5-2, Electrical Engineering Department, University of Delaware, May 1992.

[MD88] P. Mockapetris and K. Dunlap, “Development of the Domain Name System,” *Proceedings of SIGCOMM '88*, pp. 123-133, August 1988.

[Mo92] J. Mogul, “Observing TCP Dynamics in Real Networks,” *Proceedings of SIGCOMM '92*, pp. 305-317, August 1992.

[Mo95] J. Moy, "Link-State Routing," in [St95].

[Mu94] A. Mukherjee, "On the Dynamics and Significance of Low Frequency Components of Internet Load," *Internetworking: Research and Experience*, Vol. 5, pp. 163-205, December 1994.

[My95] C. Myers, private communication, October 16, 1995.

[Na84] J. Nagle, "Congestion Control in IP/TCP Internetworks," RFC 896, Network Information Center, SRI International, Menlo Park, CA, January 1984.

[Na87] J. Nagle, "On Packet Switches with Infinite Storage," *IEEE Transactions on Communications*, 35(4), pp. 435-438, 1987.

[PaFe94] C. Parris and D. Ferrari, "A Dynamic Connection Management Scheme for Guaranteed Performance Services in Packet-Switching Integrated Services Networks," *Proceedings of INFOCOM '94*, Toronto, June 1994.

[PHS95] C. Partridge, J. Hughes, and J. Stone, "Performance of Checksums and CRCs over Real Data," *Proceedings of SIGCOMM '95*, Cambridge, Massachusetts, August 1995.

[Pa94a] V. Paxson, "Empirically-Derived Analytic Models of Wide-Area TCP Connections," *IEEE/ACM Transactions on Networking*, 2(4), pp. 316-336, August 1994.

[Pa94b] V. Paxson, "Growth Trends in Wide-Area TCP Connections," *IEEE Network*, 8(4), pp. 8-17, July/August 1994.

[PF95] V. Paxson and S. Floyd, "Wide-Area Traffic: The Failure of Poisson Modeling," *IEEE/ACM Transactions on Networking*, 3(3), pp. 226-244, June 1995.

[Pa96a] V. Paxson, "Towards a Framework for Defining Internet Performance Metrics," *Proceedings of INET '96*, Montreal, June 1996.

[Pa96b] V. Paxson, "End-to-End Routing Behavior in the Internet," *Proceedings of SIGCOMM '96*, pp. 25-38, August 1996.

[PV88] R. Perlman and G. Varghese, "Pitfalls in the Design of Distributed Routing Algorithms," *Proceedings of SIGCOMM '88*, pp. 43-54, August 1988.

[Pe91] R. Perlman, "A comparison between two routing protocols: OSPF and IS-IS," *IEEE Network*, 5(5), pp. 18-24, September 1991.

[Pe92] R. Perlman, *Interconnections: Bridges and Routers,* Addison-Wesley, 1992.

[Po80] J. Postel, "User Datagram Protocol," RFC 768, Network Information Center, SRI International, Menlo Park, CA, August 1980.

[Po81a] J. Postel, "Internet Protocol," RFC 791, Network Information Center, SRI International, Menlo Park, CA, September 1981.

[Po81b] J. Postel, "Internet Control Message Protocol," RFC 792, Network Information Center, SRI International, Menlo Park, CA, September 1981.

[Po81c] J. Postel, "Transmission Control Protocol," RFC 793, Network Information Center, SRI International, Menlo Park, CA, September 1981.

[PFTV86] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling, *Numerical Recipes*, Cambridge University Press, 1986.

[RJ90] K. Ramakrishnan and R. Jain, "A Binary Feedback Scheme for Congestion Avoidance in Computer Networks," *ACM Transactions on Computer Systems*, 8(2), pp. 158-181, May 1990.

[RY94] K. Ramakrishnan and H. Yang, "The Ethernet Capture Effect: Analysis and Solution," *Proceedings of IEEE 19th Conference on Local Computer Networks*, October 1994.

[Re89] J. Rekhter, "EGP and Policy Based Routing in the New NSFNET Backbone," RFC 1092, Network Information Center, SRI International, Menlo Park, CA, February 1989.

[RC92] Y. Rekhter and B. Chinoy, "Injecting Inter-autonomous System Routes into Intra-autonomous System Routing: a Performance Analysis," *Internetworking: Research and Experience*, Vol. 3, pp. 189-202, 1992.

[Re95] Y. Rekhter, "Inter-Domain Routing: EGP, BGP, and IDRP," in [St95].

[RL95] Y. Rekhter and T. Li, "A Border Gateway Protocol 4 (BGP-4)," RFC 1771, DDN Network Information Center, March 1995.

[RG95] Y. Rekhter and P. Gross, "Application of the Border Gateway Protocol in the Internet," RFC 1772, DDN Network Information Center, March 1995.

[Ri95] J. Rice, *Mathematical Statistics and Data Analysis*, 2nd edition, Duxbury Press, 1995.

[Ri92] R. Rivest, "The MD5 Message-Digest Algorithm," RFC 1321, DDN Network Information Center, April 1992.

[Ro82] E. Rosen, "Exterior Gateway Protocol (EGP)," RFC 827, Network Information Center, SRI International, Menlo Park, CA, October 1982.

[Ro83] S. Ross, *Stochastic Processes*, John Wiley & Sons, 1983.

[SRC84] J. Saltzer, D. Reed and D. Clark, "End-To-End Arguments in System Design," *ACM Transactions on Computer Systems*, 2(4), pp. 277-288, November 1984.

[SAGJ93] D. Sanghi, A.K. Agrawal, Ó. Gudmundsson, and B.N. Jain, "Experimental Assessment of End-to-end Behavior on Internet," *Proceedings of INFOCOM '93*, San Francisco, March, 1993.

[Sc77] M. Schwartz, *Computer Communication Network Design and Analysis*, Prentice Hall, 1977.

[SS80] M. Schwartz and T. Stern, “Routing Techniques Used in Computer Communication Networks,” *IEEE Transactions on Communications*, 28(4), pp. 539-552, April 1980.

[SFANC93] D. Sidhu, T. Fu, S. Abdallah, R. Nair, and R. Coltun, “Open Shortest Path First (OSPF) Routing Protocol Simulation,” *Proceedings of SIGCOMM '93*, pp. 53-62, September 1993.

[St95] M. Steenstrup, editor, *Routing in Communications Networks*, Prentice-Hall, 1995.

[St94] W.R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*, Addison-Wesley, 1994.

[St96] W.R. Stevens, *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols*, Addison-Wesley, 1996.

[St97] W.R. Stevens, “TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms,” RFC 2001, DDN Network Information Center, January 1997.

[Ta96] A.S. Tanenbaum, *Computer Networks,* 3rd edition, Prentice Hall, 1996.

[Tr95a] P. Traina, “Experience with the BGP-4 Protocol,” RFC 1773, DDN Network Information Center, March 1995.

[Tr95b] P. Traina, editor, “BGP-4 Protocol Analysis,” RFC 1774, DDN Network Information Center, March 1995.

[Va95] Y. Vardi, “Network Tomography I: Estimating Source-Destination Traffic Intensities from Link Data (Fixed Routing),” under revision for publication in *Journal of the American Statistical Association*, 1995.

[WLC92] I. Wakeman, D. Lewis, and J. Crowcroft, “Traffic Analysis of Trans-Atlantic Traffic,” *Proceedings of INET '92*, Kyoto, Japan, 1992.

[WC91] Z. Wang and J. Crowcroft, “A New Congestion Control Scheme: Slow Start and Search (Tri-S),” *Computer Communication Review*, 21(1), pp. 32-43, January 1991.

[WC92] Z. Wang and J. Crowcroft, “Eliminating Periodic Packet Losses in the 4.3-Tahoe BSD TCP Congestion Control Algorithm,” *Computer Communication Review*, 22(2), pp. 9-16, April 1992.

[WTSW95] W. Willinger, M. Taqqu, R. Sherman, and D. Wilson, “Self-Similarity Through High-Variability: Statistical Analysis of Ethernet LAN Traffic at the Source Level,” *Proceedings of SIGCOMM '95*, pp. 100-113, Cambridge, MA, September 1995.

[WP97] W. Willinger and V. Paxson, “Discussion of `Heavy Tail Modeling and Teletraffic Data' by S.R. Resnick,” to appear in *Annals of Statistics*, 1997.

[WPT97] W. Willinger, V. Paxson and M. Taqqu, “Self-Similarity and Heavy Tails: Structural Modeling of Network Traffic,” in *A Practical Guide To Heavy Tails: Statistical Techniques for Analysing Heavy Tailed Distributions*, to be published by Birkhauser, 1997.

[Wo82] R. Wolff, “Poisson Arrivals See Time Averages,” *Operations Research*, 30(2), pp. 223-231, 1982.

[WS95] G. Wright and W. Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*, Addison-Wesley, 1995.

[ZG-LA91] W. Zaumen and J.J. Garcia-Luna Aceves, “Dynamics of Distributed Shortest-Path Routing Algorithms,” *Proceedings of SIGCOMM '91*, pp. 31-42, September 1991.

[ZG-LA92] W. Zaumen and J.J. Garcia-Luna Aceves, “Dynamics of Link-state and Loop-free Distance-vector Routing Algorithms,” *Internetworking: Research and Experience*, Vol. 3, pp. 161-188, 1992.

[Zh86] L. Zhang. “Why TCP timers don't work well,” *Proceedings of SIGCOMM '86*, August 1986.

[ZSC91] L. Zhang, S. Shenker, and D. Clark, “Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic,” *Proceedings of SIGCOMM '91*, pp. 133-147, September 1991.

[ZDESZ93] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala, “RSVP: A New Resource ReSerVation Protocol,” *IEEE Network*, 7(5), pp. 8-18, September 1993.

Appendix A

The Network Probe Daemon

NPD (Network Probe Daemon) is a framework for probing paths through the Internet by tracing the routes corresponding to the paths, and by sending TCP packets along the paths and tracing the arrivals of both the packets and their acknowledgements. NPD consists of a daemon (`npd`) that services authenticated requests for tracing and generating probes, and a control program (`npd_control`), which is run only at the site conducting the probe experiments.

The following sections discuss the daemon's operation (§ A.1) and the steps taken to address security concerns (§ A.2).

A.1 Daemon operation

A site participates in the network probe experiment by running the network probe daemon `npd` on a Unix workstation connected to the Internet. The workstation does not need any special location in the network topology (e.g., it does not need to be located on the wide-area gateway network).

The `npd` process is run by Internet services daemon `inetd` whenever a connection appears for the "npd" service (TCP port 7504, by default). This means that installing the daemon requires editing */etc/services* to add the "npd" service, and */etc/inetd.conf* to add the service with the given port number.

Once running, `npd` responds to the following requests:

`trace-route` *X*

Run the `traceroute` utility [Jac89] to measure the path to host *X* and send back the results.

`begin-trace` *X Y*

Begin tracing "discard" or `npd`-to-`npd` packets and their acknowledgements between hosts *X* and *Y*.

`terminate-trace`

Stop the trace and send back the results.

`sink` *s*

Accept a connection on the "npd" port, using a socket receive buffer of *s* bytes, and read from it until the connection is closed.

`source` *X p n s*

Send *n* bytes to the discard or "npd" port (as indicated by *p* being "discard" or "npd") of host *X*, using a socket send buffer of *s* bytes.

`npd` sources and sinks always use a local TCP port of 7505 (that they both do has security benefits, as discussed in § A.2 below). If the bytes are sent to the "discard" port, then no remote `npd` need run; the `inetd` process on the remote machine will instead handle discarding the data packets itself.

`restart-log`

Mail the current log to a preconfigured address and, upon success, clear it.

`self-test`

Perform a self-test and report the results.

`quit` Terminate the connection.

On some operating systems, the packet filter cannot capture traffic generated by the same host that is running the filter. In particular, Sun workstations using SunOS and the stock "NIT" (Network Interface Tap) interface do not capture their own outbound traffic. Because SunOS is quite popular, it was necessary to accommodate this deficiency. For the `traceroute` experiment it makes no difference, but for the packet dynamics (*probe*) experiment it is crucial that the TCP traffic comprising the probe be recorded at both endpoints. NPD can thus be configured at a site to run on two workstations, a *source/sink* host that sources or sinks TCP probes, and a *trace* host that runs `traceroute` or `tcpdump`, depending on the experiment. For a given site A, we refer here to these machines as A_s (source) and A_t (trace) respectively. For many sites, $A_s = A_t$, as summarized in Table XIV.

To conduct a `traceroute` experiment measuring the route from site A to site B, the NPD master program (`npd_control`) connects to the `npd` daemon at host A_t and (after authentication) issues:

```
trace-route B
quit
```

and reads back the `traceroute` output, if successful. To conduct a *probe* experiment of b bytes between A and B, using send and receive buffer sizes of s and r, `npd_control` executes the following steps (assuming each preceding step is successful):

1. Send the request `begin-trace` A B to A_t and B_t, and wait for them to indicate they are ready.

2. Send the request `sink` r to B_s and wait for it to indicate it is ready.

3. Send the request `source` B `npd` b r to A_s.

4. Wait for A_s and B_s to indicate they have finished sourcing/sinking the data stream.
5. Wait two more seconds, to allow any packets still traveling inside the network to arrive at the endpoints.
6. Send the request `terminate-trace` to A_t and B_t.
7. Receive the trace and error files from A_t and B_t.
8. Send the request `quit` to A_s and B_s, and to A_t and B_t if different.

A.2 Security issues

Allowing a program to originate and trace network traffic at an Internet site naturally raises important security issues. To this end, we took a number of steps to make NPD secure:

- A host attempting to make NPD requests must first authenticate itself, as explained below.
- `npd` does not need to be installed with any privilege, other than being able to exec `tcpdump` and `traceroute`. A site can also configure it so it can only run a special, restricted version of `tcpdump` (`rtcpdump`; see below).
- `npd` is hardwired to only be able to trace TCP "discard" traffic, or traffic between two `npd`'s. This is done by constructing a `tcpdump` filter of

  ```
  (RESTRICTION) and (XXX)
  ```

  whenever `npd` is asked to trace traffic using the filter `XXX`, where `RESTRICTION` is:

  ```
  (tcp port 9) or (tcp src port 7505 and tcp dst port 7505)
  ```

  i.e., only allow traffic involving either the TCP `discard` port, or both an `npd` sender and receiver. (TCP port 7505 is the well-known port used by `npd` for sourcing and sinking traffic; see § A.1.)
- `npd` logs all of its connections and activity. If writing to the log fails, or if `npd` cannot lock the log for exclusive access, `npd` exits.
- The log file can only be reset if `npd` first succeeds in mailing the previous log to a preset Internet mail address. Sites can configure this address to include a local address.
- The only files created by `npd` (other than the log file) are temporary files created using the Unix `tmpfile(3)` library routine, which are guaranteed to disappear when `npd` exits, and also to be unreadable by other local processes.
- When executed, `npd` forks a child process that sleeps for a fixed amount of time (10 minutes). When the child process wakes up, it kills its parent process. This mechanism acts as a crude "fail-safe." Normally, after `npd` successfully completes its requests, it kills the child process prior to exiting itself. But if for any reason `npd` fails to do so (for example, if the network connection between `npd` and `npd_control` is lost), the fail-safe guarantees that `npd` will at some point cease consuming resources on the host.

A.2.1 Using `rtcpdump` instead of `tcpdump`

The NPD sources include `rtcpdump`, a version of `tcpdump` that is restricted to capturing TCP discard packets (or `npd`-to-`npd` packets, as described above). `rtcpdump` can only capture live, restricted packets (it cannot read existing trace files), and only writes to *stdout*, which is under the full control of `npd`.

Thus, a site can safely give `rtcpdump` "setgid" or "setuid" privilege to the Unix "group id" or "user id" necessary for packet capture on the tracing host, without needing to give the tracing group-id or user-id to `npd` itself.

`rtcpdump` terminates whenever its *stdin* is closed, which happens automatically when `npd` exits.

A.2.2 NPD authentication

An important aspect of NPD security is the use of fairly strong authentication to restrict use of `npd` at a site to only authorized remote sites. `npd` authenticates a remote site in the following manner:

1. The IP address of the remote host must translate to a hostname that in turn translates back to the given IP address. To illicitly pass this test, an attacker must subvert a Domain Name System (DNS) name server [MD88] (which, unfortunately, is possible [Be95]).

2. As part of the authentication procedure, the host must identify itself using a DNS hostname. The host's claimed identity must then translate to the host's IP address. Like the previous step, this step requires that an attacker subvert a DNS name server.

3. The host's claimed identity must appear in `npd`'s directory of secret keys. For an attacker to pass this test, they must successfully subvert a DNS name server authoritative for one of sites appearing in the directory of secret keys; more difficult than the subversions above, but still possible.

4. `npd` challenges the remote host to prove its identity by sending it a random bit-string. The remote site must successfully xor this bit-string with the secret key and send to `npd` the MD5 checksum [Ri92] of the result. `npd` then verifies that the result matches its own local computation of what the checksum should be. If so, then the remote site is presumed to know the secret key and is authenticated.

   For an attacker to successfully pass this test essentially requires that they know the secret key, since MD5 checksums take on $2^{128} \approx 10^{38}$ possible values. Since the secret key never crosses the network,[1] to acquire the secret key requires either subverting the `npd_control` site or the `npd` site, or computing the key by observing previous authentication exchanges as they crossed the network. This latter attack is believed infeasible due to the presumed non-invertibility of MD5 [Ri92].

[1] Except when distributing the NPD sources to a remote site; or if `npd` retrieves the key using NFS.