An Network Measurement Architecture for Adaptive Applications

by

Mark Richard Stemm

B.S. (Carnegie Mellon University) 1994
M.S. (University of California, Berkeley) 1996

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:

Professor Randy H. Katz, Chair
Professor Steven R. McCanne
Professor Eric Brewer
Professor George Shanthikumar

1999

The dissertation of *Mark Richard Stemm* is approved:

Chair Date

Date

Date

Date

University of California at Berkeley

1999

An Network Measurement Architecture for Adaptive Applications

Abstract

An Network Measurement Architecture for Adaptive Applications

by

Mark Richard Stemm
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Randy H. Katz, Chair

In today's Internet, the characteristics of the network path between a pair of Internet hosts can span several orders of magnitude. Some hosts may communicate over high bandwidth, low latency, uncongested paths, while others communicate over much lower quality paths. Applications can cope with these differences by *adapting* to network changes: for example, choosing alternate representations of objects or streams or downloading objects from alternate locations. For applications to adapt most effectively, however, they must discover the condition of the network path before communicating with distant hosts in order to make appropriate adaptation decisions. Unfortunately, the ability to determine the quality of network paths is missing in today's suite of Internet services, and applications have no way to make informed adaptation decisions.

To address this limitation, we have developed a network measurement architecture called SPAND (**S**hared **PA**ssive **N**etwork Performance **D**iscovery) that enables a new class of adaptive networked applications. In SPAND, applications make passive application-specific measurements of the network and store the results of the measurements in a per-domain centralized repository of network performance information. Other applications retrieve this information from the repository–thereby leveraging the shared experiences of all hosts in a domain–and use it to predict future performance. Through SPAND, applications make more informed decisions about adaptation choices as they communicate with distant Internet hosts.

In this thesis, we describe the design, implementation, and evaluation of the SPAND architecture. We describe and justify the design choices we make in SPAND and show the strengths and limitations of the architecture when compared to alternate design choices. We describe how SPAND is flexible and extensible: applications define their own types of performance measurements and averaging algorithms by providing an Active Messages-like interface between SPAND clients and SPAND's repository of network performance information. We show how *measurement noise*, the variation associated with the measurement of a particular network performance statistic, affects the granularity of application-level adaptation decisions. We also categorize and quantify measurement noise into three components: *network noise* (variations in the state of the network over small times scales), *sharing noise* (variations between the observed performance of nearby clients), and *temporal noise* (variations in performance over longer time scales).

To illustrate these concepts, we then present two realizations of the architecture that measure network performance for different types of data transport: a generic bulk transfer data transport that measures TCP-specific performance and a HTTP-specific data transport that more closely measures the way in which web clients use multiple parallel TCP connections to complete web page transfers. Measurements of the bulk-transfer realization of SPAND show that SPAND works well at providing relevant, accurate responses to clients: in the steady state, SPAND can respond to 95% of performance queries with predictions, and 70% of the time, these predictions are within a factor of two of actual performance (a discrepancy equal to the network noise inherent in the state of the network).

To validate the SPAND architecture, we built and evaluated two specific adaptive networked applications: SpandConneg, a suite of applications that use HTTP Content Negotiation to reduce client and server-side network bottlenecks, and LookingGlass, a WWW mirror selection tool. By using SpandConneg, web clients can fix download times by matching content fidelity to network conditions, and web servers can handle large numbers of clients by reducing document quality under periods of heavy load. LookingGlass presents a complete solution to the problem of replicating web content at multiple web sites, addressing the problems of transparently notifying web clients of mirrored content, providing a mechanism to disseminate mirror information in a distributed way between mirror locations, and providing algorithms for choosing mirror locations that take actual network performance into account.

Measurements of these applications show that SPAND dramatically improves the performance of adaptive networked applications. SpandConneg works well at both the client and server side of the network. Web clients that use SPAND to trade off document quality for download time can reduce the frequency of excessive user-visible (i.e. more than 30 seconds) download times from 35% to less than 10%, and reduce the median download time from 16 to 6 seconds. Web servers that use SPAND to handle an unexpected burst of clients can increase their throughput by as much as 450%. LookingGlass performs well despite the challenges in meeting the dual goals of collecting passive network performance measurements while maintaining good client repose times. LookingGlass's application-level measurements and server selection algorithms lead to faster (i.e., factor of 40) web page downloads than alternate techniques such as relying on geographic location or routing metrics to make server selection decisions. More than 90% of the time, our technique allows clients to download mirrored web objects within 40% of the fastest possible download time.

Dissertation Committee Chair

To Mom and Dad,
Mike and DC,
You made this possible.

Contents

List of Figures

List of Tables

Acknowledgements

The thing I've been told about a Ph.D. thesis is that the only parts of the thesis that 98% of people read are the Abstract, Introduction chapter, and Acknowledgments, and from my own thesis reading experience, this statement is pretty much true (grin). Knowing this, one would think that I would not forget anyone, but nevertheless, I'm sure I accidentally left some of the 98% of you out. For anyone who I neglected to thank individually below, I sincerely apologize and thank you for your contributions. I've tried to order the following paragraphs chronologically and not in any order of importance, so don't take it personally if you're mentioned near the end (grin again!).

It's impossible to completely describe how important my family was in a single paragraph, so I won't even try. Needless to say, they were the most important influence in my life, development, and choice of career path. My parents and siblings were a never-ending source of support and encouragement at every phase of my education. They always told me that anything was possible and nothing was impossible. From when they bought me my first Coleco Adam to when they dropped me off at Donner Hall at Carnegie Mellon, I knew that they would always be there for me if I needed them. Mom, Dad, Mike, and D.C., thanks for everything.

My Computer Science career by no means began in 1994 when I entered graduate school. My deepest thanks to Emil Biga who first showed me that Computer Science is more than playing games on an Apple IIGS and Mathematics is the science of abstraction and not just solving formulas. It took me almost 10 years to realize why he taught Mathematics the way he did, and now that I do, I realize that he's way ahead of his time.

My Senior Research project and interactions with Tom Mitchell during my Senior year at Carnegie Mellon were the driving force behind my decision to enter graduate school. When I asked him about the advantages of UC Berkeley over other graduate schools or a job at IBM, he told me that the advantage of graduate school, and especially UC Berkeley, was that I would learn more from my fellow graduate students than from any job or class, and he was right.

I also had the privilege of working with the Digital Mapping Laboratory for almost three years while at Carnegie Mellon. In addition to being my primary source of income for rent and ramen noodles, I learned a tremendous amount about the research process and what it was like to work as part of a large research project. Although it seems like most of the foosball madness happened after I left, I look back at my time there as three years well spent.

A good advisor makes or breaks one's graduate school experience, and Randy Katz has been the best advisor that I could have hoped for. He gave me guidance when I was a young graduate student and gave me freedom as I progressed through the learning and research process. My weekly meetings with him were never wasted, whether we were talking about my research, career choices, or neighborhoods to live in San Francisco. He did an amazing job at running the Daedalus project during the three core years of my stay at UC Berkeley, and was an excellent instructor for the classes I took from him. Most amazingly, he did all of this while being Chair of the EECS Department, with all of the additional work that entailed. No matter how close it was to the conference deadline before we gave him a draft of a submission, he always read it carefully and gave constructive criticism. At one of the Daedalus retreats, we defined a "Randy" as a unit of productivity analogous to c as a unit of velocity. It was possible to get close to being as productive as Randy, but impossible to reach it exactly. Thanks, Randy, you made graduate school worth it.

Srini Seshan should probably be considered my second advisor. He was a co-author of practically every paper I wrote at Berkeley as well as the source of the idea behind this thesis. From my first days in 443 Soda when I had no idea of how to write PCMCIA device drivers to this spring when we were working on the final conference paper for SPAND, he was always someone I could turn to to brainstorm, discuss results, help out with trace collection, or help write software. Without Srini, this thesis wouldn't exist.

I spent the last three summers at IBM Research not only with Srini, but with a great group of other researchers. Arvind Krishna, Dilip Kandlur, Pravin Bhagwat, David Maltz, Rick Han, Richard LeMaire, and Erich Nahum were the source of many great brainstorming sessions and lunchtime conversations about the computer industry and computer science research. It made IBM not just a good place to work, but a great place to work.

I also worked closely with Hari Balakrishnan and Venkat Padmanabhan for many of the papers I wrote at Berkeley. I learned as much (or maybe more) about computer networking from my many discussions with them than I did in my graduate classes. They represented the high standard to meet when it came to being a computer science researcher, and I hope I came close to it.

My past and present officemates in 443 Soda–Todd Hodes, Elan Amir, Srini Seshan, Hari Balakrishnan, Venkat Padmanabhan, Steve Czerwinski, and Ben Zhao–were always a good source of stimulating (and sometimes heated) conversation about the computer industry, politics, TCP dynamics, the latest Sigcomm papers, or anything else we could think of to distract us from work. After looking back at those heated afternoon conversations, it's a surprise I got anything done at all!

One of the best things about the Ph.D. thesis process at Berkeley is that candidates receive feedback at the beginning of the thesis proposal phase through a qualifying exam instead of after the thesis is complete through a thesis defense. Eric Brewer, Steve McCanne, and George Shanthikumar provided a lot of useful feedback on the thesis proposal that helped define the thesis, identify the interesting problems to be solved, and the way to solve them.

I spent my three core graduate school years and wrote my Master's Thesis as a part of the Daedalus and GloMop projects. The members of these projects were (in alphabetical order): Elan Amir, Hari Balakrishnan, Eric Brewer, Yatin Chawathe, Armando Fox, Steve Gribble, Tom Henderson, Todd Hodes, Daniel Jiang, Randy Katz, Giao Nguyen, Venkat Padmanabhan, Srinivasan Seshan, Brian Shiratsuki, Keith Sklower, Helen Wang, and Tao Ye. Being a part of this stellar team of researchers was one of the most rewarding parts of being a graduate student.

The biannual Daedalus/GloMop retreats at Lake Tahoe were a wonderful way to define our research and present our work to the external research community. The list of participants is too long to mention here, so I'll give a global thanks to everyone who came to our retreats. You gave useful feedback on our work and provided useful perspectives from the external research and industrial communities.

Classes and writing papers are part of the graduate school process, but by no means the only part. We have to get computer equipment for our experiments, keep the equipment running, register for classes, and handle the other bureaucratic hassles that are a part of being a graduate student. Fortunately, I hardly ever had to worry about these things, because Kathyrn Crabtree, Bob Miller, Terry Lessard-Smith, Kieth Sklower, and Brian Shiratsuki did an amazing job of hiding the details from me and allowing me to focus on my work. Thanks to all of you, you allowed me to graduate at least one year earlier than I otherwise would have.

Although I did work a lot at Berkeley, I had a lot of fun too, mostly due to a great group of fellow graduate student friends and housemates. Thanks to Eric Vigoda, Sean Hallgren, Paul Gauthier, Todd Hodes, Daishi Harada, and everyone else at the Santa Barbara House for all of the great "home office" discussions and generally making our house a great place to live in. In addition, thanks to all of the members of Alexis–Ketan Meyer-Patel, Micah Adler, John Byers, Jeff Forbes, David Palmer, David Blackston, John Hauser, David Bacher, Mike Dahlin, Eric Vigoda, Todd Hodes, Randy Keller, Ashu Rege, Tim Callahan, Daishi Harada, Susan Lee, Paul Gauthier, Matt Welsh, David Simpson, Rich Fromm, Andrew Swan, Noah Treuhaft, and Ngeci Bowman–for lots of memorable nights at Jupiter, Barclay's, The Mallard, and other watering holes in the East Bay. I'll never look at a Winnebago in quite the same way again.

And last, but by no means least, thanks to my girlfriend Becca for putting up with me during the last six months of thesis writing. She was understanding, patient, and supportive when I was too busy to spend time with her, and my best friend and companion at all times, (although the hide-a-bed is a little squeaky). Becca, thanks for being there, and I hope we're together for a lot longer.

And of course, somebody paid for me while I was here. My research was primarily supported by DARPA under contract DAAB07-95-C-D154 and an IBM Fellowship. It was also supported by grants or equipment from Hughes Aircraft, Metricom, Intel, IBM, and the California MICRO program.

Elan Amir told me when he was writing his thesis that the Acknowledgments were the most fun to write, and now that I've done it, I agree with him. It's the only opportunity in my years at Berkeley to thank all of the people who helped make my thesis happen and my life here more fun, and I'm glad I had the chance to write it all down in one place. Thank you all.

Mark Stemm
June 1999

Chapter 1

Introduction

1.1 The Problem: Enabling Adaptive Applications

1.1.1 The Explosive Growth of the Internet

The Internet is exploding as a communications medium. In the five years since the introduction of the World Wide Web in 1994, the number of Internet hosts has grown from 2 million to over 40 million and the number of Internet users has grown from 3 million to 165 million [56] [82] [59]. The Internet has grown faster in these five years than the radio and television markets in their first five years of widespread use [82]. The radio market took 38 years to reach 50 million listeners. The television market took 13 years to reach 50 million users. Once introduced to the widespread public, the Internet reached 50 million users in four years.

With this unprecedented rapid growth, it is worthwhile to examine the reasons why the Internet has grown faster and been adopted earlier by users than other communication technologies. Although it might seem at first glance to be just luck, this is not the case. There are deliberate design principles behind the Internet protocol architecture that have helped contribute to its rapid growth. The two most important of these are that the Internet architecture is *simple*, and that the Internet architecture is *decentralized*.

The Simplicity of the Internet Architecture

One important reason behind the success of the Internet is not in the services it provides but those it does *not* provide. The Internet architecture was designed to be simple. It provides only host addressing and packet routing services. It leaves other services such as naming, flow control, congestion control, and resource allocation up to the endpoints of the network. The minimum requirements that a host must fulfill to be connected to the Internet are extremely simple, which in turn allows for tremendous flexibility in how hosts can be connected to each other through the Internet [15]. A supercomputer with a 45 MBit/sec Internet connection can communicate with a Personal Digital Assistant (PDA) connected to the Internet via a 6 KBit/sec link.

The Decentralized Internet Architecture

A second important reason behind the success of the Internet is its decentralized network architecture. The original goal behind this principle was survivability. The Internet was decentralized to avoid single points of failure. Independent regions of the network can operate autonomously. This has an unanticipated but important secondary benefit. A decentralized network architecture enables rapid expansion of the network infrastructure. Entire autonomous networks can be built up independently from the Internet for later interconnection with it.

1.1.2 Heterogeneity in the Internet

Although these design principles have contributed to the rapid growth of the Internet, each of them introduces significant challenges. These come in the form of heterogeneity that Internet Hosts must overcome to effectively communicate with each other. This heterogeneity comes in two forms: *network heterogeneity* and *provider heterogeneity*.

Because the Internet architecture provides little more than packet routing facilities, its protocols specify the means by which Internet hosts are connected but not the *quality* of that connectivity. There is no automatic mechanism to determine the characteristics of a remote host's Internet connection. For example, it is difficult for the Supercomputer connected over the 45 MBit/sec link described above to determine that it is communicating with the PDA over a 6 KBit/sec link. This illustrates an example of *network heterogeneity*.

In addition, the decentralized network architecture behind the Internet implies that a packet may traverse a number of different networks, each operated by different Internet Service Providers (ISP's). The rapid growth of the Internet means that each of these Internet Service Providers is scrambling to build up their network infrastructure to keep up with demand. The combination of these two factors means that a packet traversing the Internet is likely to traverse networks run by different Service Providers with different levels of maturity and robustness. We call this *provider heterogeneity*.

The prevalence of network and provider heterogeneity makes it very difficult for applications to determine the network characteristics of the path between two Internet hosts. The quality of connectivity from one network path to another may span several orders of magnitude. For example, the available bandwidth along a network path could range from kilobits to megabits per second, and the latency could range from microseconds to seconds. A network path could be relatively uncongested with almost no packet loss, or it could be very congested with almost complete packet loss. As a result, networked applications may experience order of magnitude changes in Internet performance as they communicate from one host to another.

1.1.3 Coping with Heterogeneity through Adaptation

Heterogeneity is fundamental to the Internet's architecture, and is likely to increase, not decrease, over time with the introduction of new networked devices such as Personal Digital Assistants (PDAs), set-top "Internet Terminals", or embedded networked devices and new network technologies such as wide-area wireless access or consumer broadband access. In addition, decentralized control is inherent in the Internet, and there will

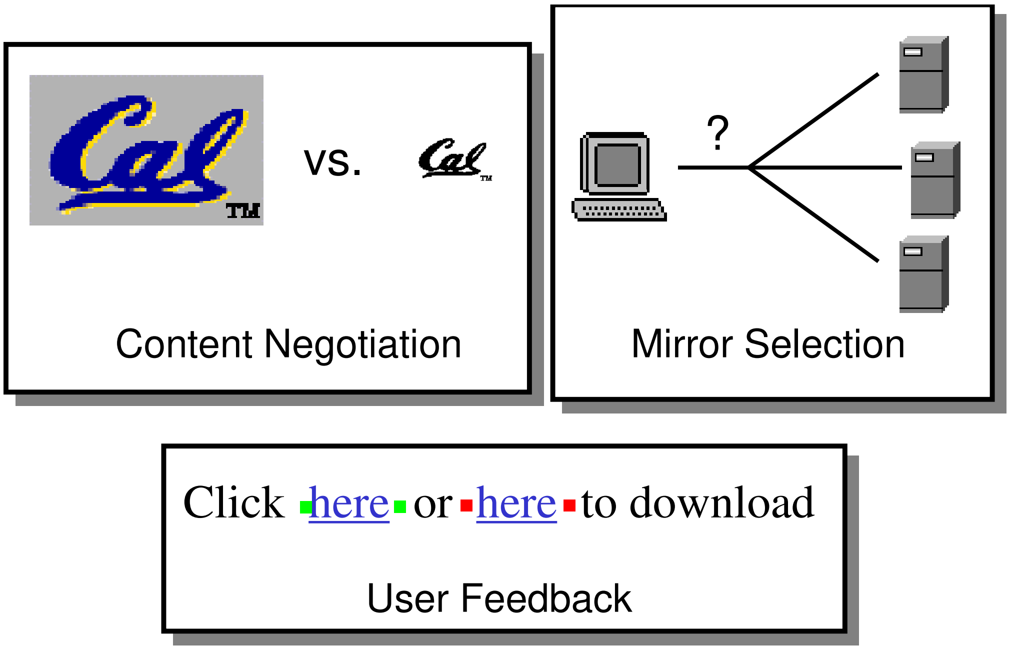

Figure 1.1: Examples of Application Level Adaptation.

be different Internet Service Providers with networks of differing quality for the foreseeable future.

One way for applications to cope with changes in network performance that arise with network and provider heterogeneity is to *adapt* by measuring the state of the network and changing their behavior in response to measurements. The idea of adaptation in response to network changes is not new. In fact, adaptation at the network layer has been a part of the Internet Protocols from their inception. TCP, the transport protocol used by Internet hosts, adapts the amount of data it sends into the network in response to packet losses and measurements of round trip time. Clients using RTCP [69] to participate in a multicast session change the rate at which they send Receiver Reports in response to the number of participants in the session.

In addition to adaptation at the lower layers of the network stack, there are also adaptation techniques that can be applied at the application layer to cope with changes in network performance. Examples of these techniques are illustrated in (Figure 1.1):

- Client applications presented with a choice of servers that replicate the same service can choose the one that offers the highest-quality client $\longleftrightarrow$ server network path. A specific example of this is Harvest [13] [14] [22], a WWW caching system where a cache selects one of a number of "peer" caches from which to retrieve a web object. Another example is mirrored web and FTP sites such as the Internet Movie Database `www.imdb.com` or C–Net's `www.download.com`, which provide multiple locations to retrieve the same files but do not provide a way to intelligently choose between them.

- Applications that are presented with data types that span a range of content fidelities can choose a particular representation to balance quality against end-to-end download time. For example, WWW clients can use a transcoding proxy (e.g. Transend [28]) to change the quality of web objects to match available bandwidth. A client receiving a multimedia stream can use a real-time transcoding service (e.g. Video Gateway [3],[2]) to change the data rate of a multimedia stream depending on network characteristics.

- Applications can expose the state of the network to the user, providing feedback that indicates the expected performance to a distant site. For example, web browsers could insert an informative icon next to a hyperlink indicating the expected transfer time for the object referred to by the hyperlink. Clients using search engines could post-process the query results and re-score the documents based on the expected time to download a page.

However, these examples of adaptation do not completely solve the problems of network and provider heterogeneity. They provide only bare mechanisms for adaptation and do not specify the policy that uses the mechanism to improve application-level performance. The policy could in general be based on several factors, but the most obvious (and effective) policy uses current network performance information, for example, by using estimates of network performance to determine the appropriate amount of data transcoding.

As a result, these policies need a way to measure the network performance between local applications and distant hosts. Unfortunately, this ability is missing from today's suite of Internet services. *Applications need a way to measure the state of the network path between themselves and a particular distant Internet host to drive their adaptation decisions.* Solving this problem and proving its effectiveness for building adaptive applications is the goal of our thesis.

1.2 Current Solutions and their Limitations

Previous attempts at creating network measurement services to drive application adaptation policies have focused primarily on *isolated*, *active*, *network-level* measurements of network statistics. By isolated measurements, we mean that applications individually make measurements of the state of the network and do not share them with other hosts. By active measurements, we mean that applications introduce probe traffic into the network to measure it. By network-level measurements, we mean that applications measure network-level statistics such as hop count, latency, available bandwidth, etc. as approximations to actual application-level performance such as perceived web page download time.

However, the design choices used in past approaches have limitations. Isolated measurements from a single host prevent a client from using the past information of nearby clients to predict future performance. Recent studies [5] [65] have shown that network performance from a client to a server is often stable for many minutes and very similar to the performance observed by other nearby clients, so there are potential benefits of sharing information between hosts. In Chapter 3, we show examples where using shared rather than isolated information increases the likelihood that previously collected network characteristics are "valid" by as much as 500%.

Active measurements require the introduction of extraneous traffic into the network. Clearly, an approach that determines the same information with a minimum of unnecessary traffic is more desirable. We show in Section 3.4.2 that this unnecessary traffic can quickly grow to become a non-negligible part of the traffic reaching busy web servers, reducing their efficiency and sometimes their scalability.

A limitation of network-level measurements is that although metrics such as hop count, latency, or peak bandwidth may correlate with application-level performance, they are not guaranteed to always reflect exactly the metric in which local hosts are interested. For example, a host may be many network hops away but still perform better in terms of actual web page download time than a host that is one or two hops away. A distant multimedia server may have excellent network connectivity, but if the server is down or overloaded, an application will still observe poor performance in the form of high packet loss rates or inability to view media clips.

Our solution to the network measurement problem addresses these limitations, as we describe in the next section.

1.3 Thesis Contributions

The primary goal of this dissertation is to solve the problem of efficient network measurement for use in creating adaptive networked applications. We do this by developing a network measurement service and then validating the service by creating applications that use the service and measuring their performance. In particular, the specific contributions we make in this thesis are the following:

- **SPAND**, a network performance measurement service that can be utilized by Internet hosts in a local domain.
- **SpandConneg**, a HTTP content negotiation application that uses SPAND to drive the choice of data representation at web servers and web clients.
- **LookingGlass**, a web mirror selection application that uses SPAND to drive the choice of web server to contact.

We describe each of these contributions in more detail below.

1.3.1 SPAND, a Network Performance Measurement Service

The first contribution in this thesis is a network measurement service called SPAND (**S**hared **PA**ssive **N**etwork **P**erformance **D**iscovery). SPAND acts as a shared per-domain repository of network performance information that can be used by all hosts in a domain. Individual clients in a domain make end-to-end, application specific measurements of network performance. These clients then place the results of these measurements in the SPAND repository. Later, other clients can make queries of the network performance information in the repository to estimate the current network performance between the local domain and a particular distant host. This allows a group of hosts to work collectively to obtain timely and accurate network performance information without introducing unnecessary network traffic.

The SPAND design reflects several important design decisions that contrast with those of previous systems:

- In SPAND, measurements are *shared.* Hosts explicitly share measurements by placing them in a centralized per-domain repository.

By using shared measurements, hosts enjoy increased likelihood of having up-to-date performance information about a distant host, because it takes advantage of the collective knowledge collected by all hosts in a domain. This can be especially useful when applications are relatively short-lived and do not keep the information they measure on stable storage. SPAND provides a centralized persistent repository of performance information that can be updated and accessed by network clients with similar network connectivity.

- In SPAND, measurements are *passive.* Instead of introducing traffic into the network in the form of probe packets or simulated connections, we rely only on the traffic that applications generate as they communicate with other Internet hosts.

The advantage of passive measurements over active probing is that the system automatically tunes itself. Hosts in a distant domain that interact often with local hosts generate more traffic, and as a result, receive more network measurements. Distant hosts that are unpopular receive less samples. In contrast, active probing requires potentially complex mechanisms to direct probes towards those hosts that are visited most frequently. Passive measurements by definition do not introduce probe traffic into the network and as a result, do not suffer from the limitations of active probing described above.

- In SPAND, measurements are *application specific.* Instead of measuring network-level statistics such as routing metrics, latencies, or link bandwidths in our system, we rely whenever possible on application-level measurements such as response time or web page download time to drive adaptation decisions.

The limitations of network-level measurements as estimates of actual application-level performance were qualitatively described earlier. In Section 3.4.3, we present one quantitative example of the difference between network-level and application-level measurements. We measure packet loss rates using `ping`, a network-level measurement tool, and RealMedia, a streaming media application, and show that there are significant differences between the loss statistics reported by each method.

Although our design choices have significant advantages over alternate design choices, the use of shared, passive measurements also presents challenges. One such challenge is coping with *measurement noise*, the difference between predicted and actual performance. In this thesis, we show how measurement noise affects the granularity of application level decisions. We separate measurement noise into three categories: *network noise*, the variation that is inherent in the network, *sharing noise*, the variation that results from inappropriately sharing performance information between hosts, and *temporal noise*, the

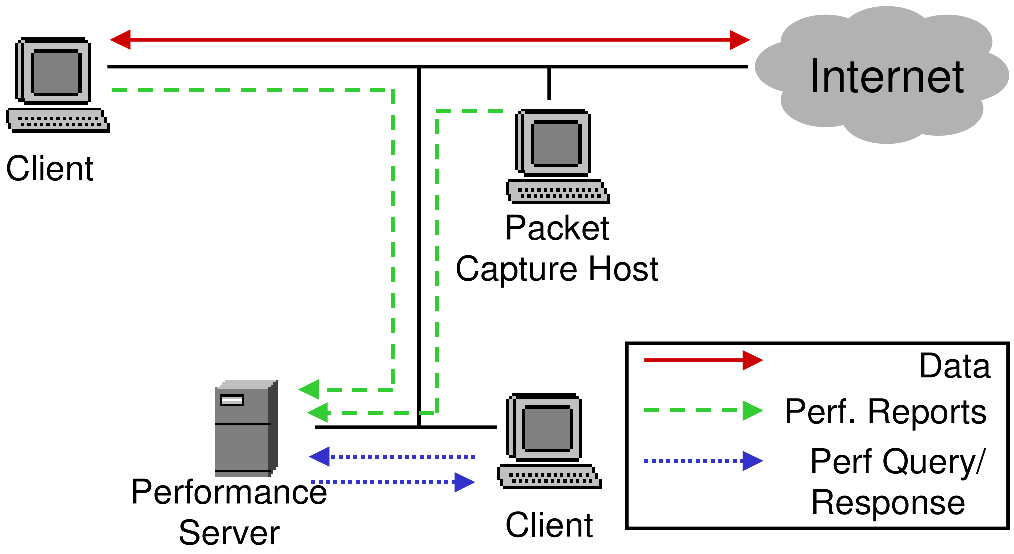

Figure 1.2: Components of SPAND.

variation that results from using past out-of-date information to predict current performance. In Section 3.5.1, we present measurements of performance for actual clients and show that of these three components of measurement noise, network noise is by far the largest contributor. Network noise can increase the variation of measurements by as much as a factor of two. Sharing noise and temporal noise only slightly increase (less than 10%) the variation over the amount introduced by network noise.

In Chapter 3, we describe in more detail the advantages and challenges of using shared, passive measurements of network performance.

Components of SPAND

Figure 1.2 shows in more detail the components of SPAND. SPAND is comprised of *Client Applications*, *Performance Servers*, and *Packet Capture Hosts*. Client Applications communicate with distant network hosts and passively measure their performance as they perform this communication. When the communication session terminates, the application constructs a *Performance Report* that summarize the observed performance to distant hosts. It then sends the performance report to a per-domain Performance Server. Like Domain Name System (DNS) servers, the location of a performance server is configured statically at the client. Applications that use the same performance report format can take advantage of the shared experiences of all Clients that communicate with the same performance server.

It may be difficult to immediately upgrade all clients to generate their own performance reports, so as an aid in deployment, we also use a specialized client called a *Packet Capture Host*. A Packet Capture Host snoops on local network traffic and makes measurements on behalf of clients that do not make measurements on their own. It uses heuristics to reconstruct application-level transactions from the sequence of packets.

After creating performance reports, Client Applications and Packet Capture Hosts send them to a *Performance Server* which maintains a centralized database of reports. The performance server acts as the repository for network performance information and the per-domain authority to contact to answer questions about network performance. Client Applications send *Performance Requests* to the performance server requesting the observed performance to a distant host. The performance server responds with an application-specific *Performance Response* that indicates the typical performance seen by clients.

Although the format of performance requests and performance responses could be arbitrary, many applications use the network in approximately the same way. To standardize this, we have developed two types of performance report/response combinations that closely map the way in which many applications use the network. The first class of performance reports describes performance for a Generic Bulk Transfer Application that uses TCP. This is a network-level measurement, in that it does not capture application-level dynamics and only measures performance at the transport level. These performance reports measure TCP-specific statistics such as available bandwidth and round trip time. In addition to these statistics, we also define a new TCP-specific metric called *Time To Completion* that captures the effects of TCP's window growth algorithms on end-to-end response time. Our measurement of this metric is completely *nonparametric*; it does not use an analytical model to describe TCP dynamics. Instead, our measurement of Time To Completion only uses the actual performance of past connections to make measurements.

The second class of performance reports measures network performance for HTTP transfers. This metric is an application-specific metric in that it captures application-level dynamics that are not available to a network-level statistic. In particular, it captures the way a HTTP transfer uses a single TCP connection for multiple HTTP transfers as well as the way multiple HTTP objects comprise a single web page.

To test the performance of our network measurement service, we present application-independent (i.e., TCP-specific) experiments designed to measure how well SPAND does at accurately measuring network performance for a collection of local clients. These experiments measure the likelihood that a given performance request can be serviced by our system and the difference between SPAND's prediction of network performance and the actual performance observed by clients. These experiments show that SPAND does well on both accounts: SPAND's performance server quickly accumulates enough performance reports to service more than 95% of the performance requests presented by clients, and the performance responses to these requests are usually within a factor of 2 of actual performance.

A more complete description of the SPAND architecture and application-independent results are given in Chapter 4.

Once we created the network measurement service, we validate it by creating applications that use the network measurement service and examining how effectively the applications can use SPAND to improve their adaptation decisions.

1.3.2 SpandConneg, a Content Negotiation Application

The second contribution of our thesis is SpandConneg, a suite of web applications that use a mechanism in HTTP for Transparent Content Negotiation along with SPAND's network performance measurements to improve application level performance. SpandConneg uses this mechanism in two ways:

- To reduce excessively long download times at web clients, and
- to allow web servers to handle an unexpected burst of requests from a large number of clients.

We describe this process in more detail below.

HTTP's Transparent Content Negotiation mechanism allows web clients and web servers to choose among *variants* of the same web object. In general, these variants can be different data types, sizes, languages, or of differing quality. For example, a text file may be available in several languages, an image may be available in several sizes, or a paper may be available as a postscript document or an HTML page. Transparent Content Negotiation is a mechanism that allows a client and server to select the most appropriate variant for a particular client, based on the characteristics, capabilities, and current network and load conditions at the client or server.

SpandConneg utilizes content negotiation both at the client side of the network and the server side of the network. In SpandConneg, we store multiple representations of image objects at web servers and allow web clients to choose one of the representations based on current network conditions. At the client side of the network, web clients can use this mechanism to trade between web object quality and a reduced response time. For example, web clients with low latency and high bandwidth connectivity to a web server can request full-color, high resolution versions of image objects. A web client with lower quality connectivity can request black and white, low resolution versions of objects. This allows web clients to trade web object fidelity for reduced object download time.

We also allow web servers to prune proactively the choice of alternate representations for individual web objects. For example, a web server can choose to hide some of the alternate representations from web clients when performing content negotiation in response to client load. For example, a lowly loaded server may serve out full-size versions of web objects. If the server is suddenly swamped with a large number of clients, it can hide full-size variants of image objects and force web clients to choose between smaller representations. As a result, the web server serves out smaller web objects and reduces the load on the connection between the web server and the rest of the Internet. By carefully manipulating the choice of possible negotiable variants in response to network load and making a large fraction of the web server traffic come from negotiable content, a web server can gracefully handle a larger number of web clients than it would without content negotiation.

Effectiveness of SpandConneg

To evaluate the effectiveness of client side negotiation, we used a policy that attempts to limit all web page downloads to less than ten seconds. Ten seconds is a heuristic, chosen as a reasonable upper bound on the amount of time that web users are willing to wait for web pages to download. Before a web client downloaded a page, it would consult SPAND to determine the estimated download time for the page. If the estimate indicated that the page would take more than ten seconds to download, then the appropriately

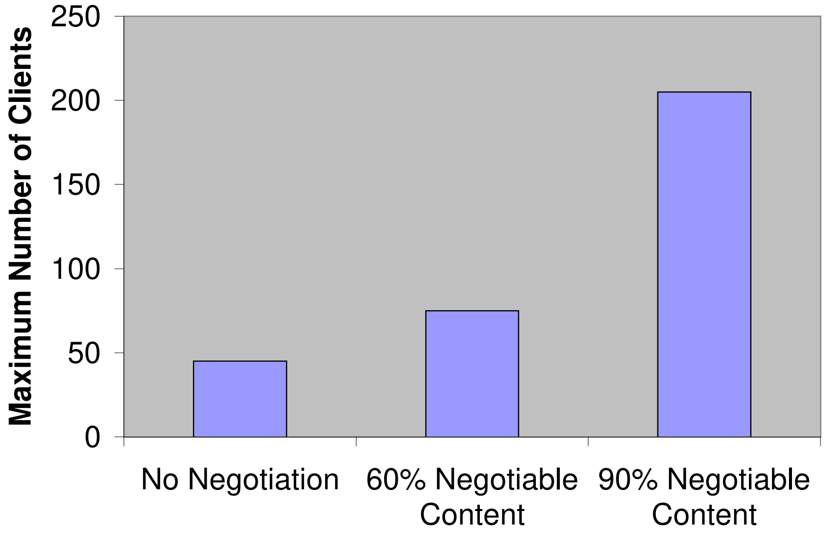

Figure 1.3: Maximum number of clients supported with Surge workload and varying amounts of Content Negotiation.

reduced-size version of the page was downloaded instead. Using this process, 60% of the time, SpandConneg was able to reduce download times to less than ten seconds. If no content negotiation were used, the likelihood of a download time of less than ten seconds drops to 35%. Our system also reduces the likelihood of extremely long download times. When clients perform content negotiation, only 10% of the time are clients forced to wait more than 30 seconds to download a web page. On the other hand, when clients do not use content negotiation, they must wait more than 30 seconds more than 35% of the time. A more complete analysis of the client side implementation is presented in Section 5.5.3.

To evaluate the effectiveness of server side negotiation, we stressed a widely used and available web server, Apache, with artificially generated client load from Surge [6], a web request generator. We limited the link between the clients and server to 1.5 MBits/sec and examined how many clients the web server could support with and without content negotiation. We also measured the peak server throughput (measured in web operations per second) with and without the use of content negotiation. More details on the experimental setup are in Section 5.6.2.

Figures 1.3 and 1.4 show quantitatively the potential benefits of this process. We see that careful use of content negotiation allows a web server to increase their client population and throughput (measured in objects served per second) by as much as 450%.

1.3.3 LookingGlass, a Server Selection Service

The third contribution of our thesis is LookingGlass, a web mirror selection tool that takes advantage of SPAND's network performance measurements. LookingGlass is designed to solve the problem of efficient dissemination of mirrored web content. This problem can be broken down into three subproblems:

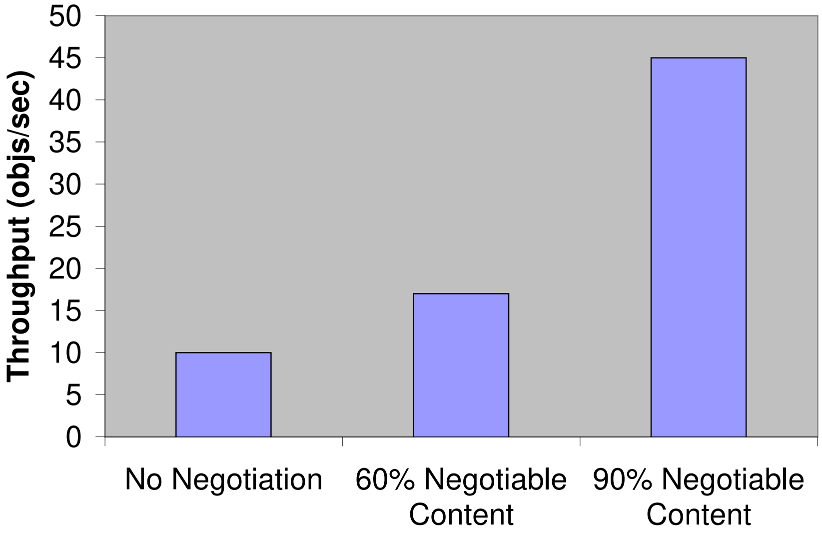

Figure 1.4: Maximum web server throughput with Surge workload and varying amounts of Content Negotiation.

- *Mechanisms for Mirror Advertisement:* There must be a way for web servers to advertise to web clients where the mirrored content is located.
- *Metrics for Mirror Ranking:* A web client must decide how to rank the mirror locations in terms of their quality of connectivity to local clients.
- *Mirror Selection Algorithm:* Given a ranking, there must be an algorithm for web clients to select the most appropriate mirror from which to download the object.

Existing solutions to these problems are solved in manual, ad-hoc ways. Mirrors are usually advertised from a maintained list of hyperlinks on a HTML page. The administrator of the site must update this list as mirror sites are added and deleted. The user then ranks the mirror locations based on minimal information such as guesses about geographic location or hints on the HTML Page. The user then manually selects one of the links on the web page. This may lead to hotspots or other inefficient load balancing between the mirror locations.

LookingGlass more effectively solves the above subproblems in dissemination of mirrored content. It reuses the HTTP Transparent Content Negotiation framework to transparently inform web client Browsers of possible mirror locations for web objects. The user is now no longer involved in selecting a mirror. In addition, LookingGlass uses a UUCP-like algorithm to efficiently disseminate mirror information between web mirrors. The primary site administrator no longer has to manually create a web page that lists the mirror locations. To rank the mirror locations, web client Browsers use SPAND's network performance information to rank the mirror locations. This eliminates dependence on crude hints and instead ranks the sites on the actual past performance of clients. Web clients use randomization when choosing a mirror to efficiently spread request load across the mirror locations and to keep SPAND's passive performance measurements up-to-date.

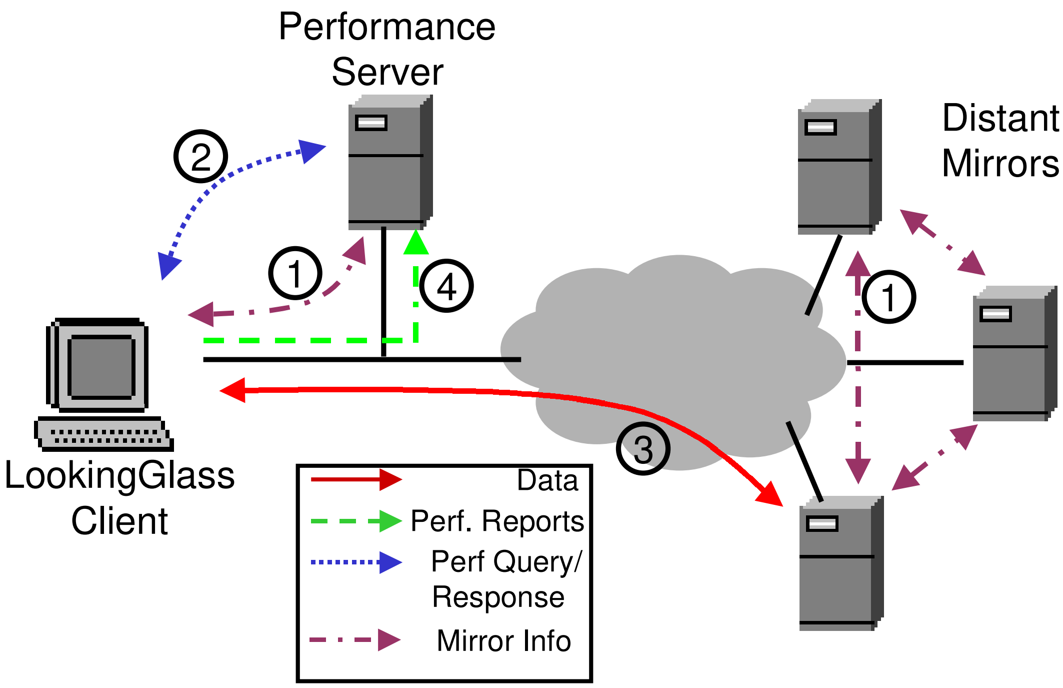

Figure 1.5: Example download using LookingGlass.

Figure 1.5 shows an example web object download using LookingGlass. At the server side, the collection of mirrors exchange lists of mirror locations for each web object. At the client side, the process begins when the web client starts to fetch a web object from one of the mirror locations. The web mirror responds with a list of alternate mirror locations for the requested objects. Using the list of alternate locations, the client-side LookingGlass component queries the SPAND network measurement service to obtain estimates of the download time for each of the mirror locations. Using these estimates, the client-side component randomly selects a mirror based on SPAND's performance estimates. After the transfer has completed, the client-side component sends performance reports to the performance server that indicate the response time for the primary location and (if used) the backup location.

This description does not include all of the details in the entire selection process. A more complete description is in Section 6.3.4.

To evaluate the effectiveness of LookingGlass, we presented a web client with several different locations for mirrored web content and examined which mirror location the client selected. We then measured the download time for the selected mirror location. As a basis for comparison, we also measured the download time for all of the other mirror locations and used the shortest time as a baseline "optimal" choice. We compared LookingGlass's policy of using actual performance information to drive mirror selection against several alternate policies for mirror selection:

- Using geographic location to approximate good performance. In this approach, the client chooses a mirror that is closest geographically.

- Choosing the mirror that is was the least number of network hops away from the client.
- Randomly selecting a different mirror for each round.
- Always choosing the primary mirror of the content.

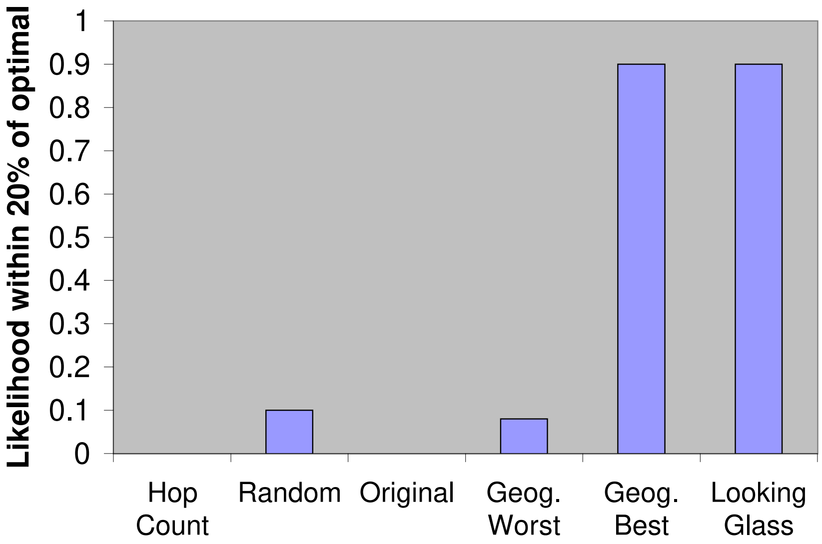

Figure 1.6: Likelihood of being within 20% of optimal for various server selection policies.

Figures 1.6 and 1.7 show the results of this comparison. They show the fraction of time the download time for a particular mirror was within 20% and within a factor of 2 of the download time for the optimal mirror, respectively. We see that LookingGlass does as well as or better than every other policy we considered. In addition, it does a near-optimal job at finding the best mirror. The download time is almost always close to the optimal download time. A more complete description of the analysis of LookingGlass is in Chapter 6.

Together, these three contributions–SPAND, SpandConneg, and LookingGlass–present a complete solution to enabling adaptive applications through a generalized network measurement service.

1.4 Dissertation Outline

The rest of this dissertation is organized as follows. In Chapter 2, we describe related work in network measurement tools and adaptive applications in more detail. We also point out the differences between the design choices made by this previous work and the design choices made in SPAND. In Chapter 3, we discuss in more detail the design choices made in our system. We also quantitatively show the potential benefits of using Shared, Passive, Application Specific network measurements over Isolated, Active, Network Specific measurements. We also describe and quantify the challenges that arise from using Shared

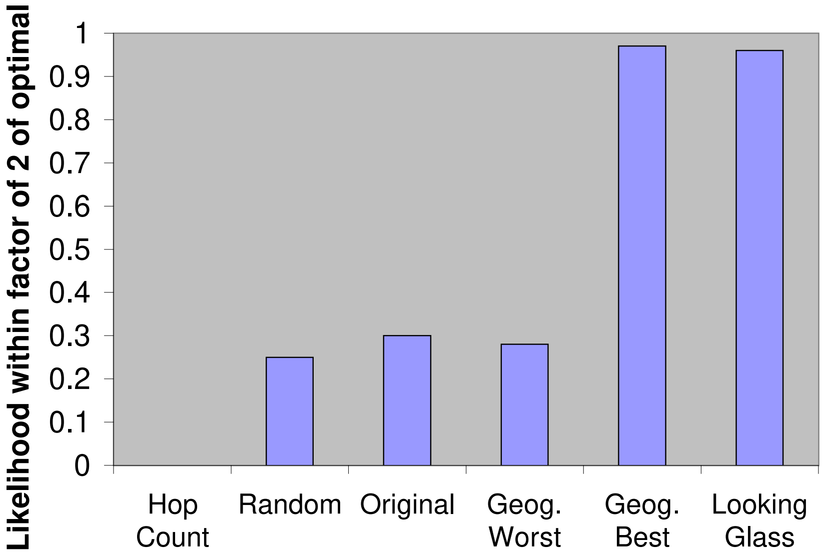

Figure 1.7: Likelihood of being within factor of 2 of optimal for various server selection policies.

Passive measurements by quantifying the amount of *measurement noise* they introduce into our performance statistics.

In Chapter 4, we present the core SPAND architecture. We describe the components of SPAND and how they communicate. We also describe the types of network measurements we make in SPAND and the implementation details of making these measurements. We also present application-independent results that show how well SPAND does at presenting clients with a relevant, accurate view of network performance. We find that SPAND responds to 95% of performance queries with meaningful responses, and that SPAND's performance responses are usually within a factor of two of actual observed performance (a discrepancy due to inherent network noise).

In Chapter 5, we describe SpandConneg, the Content Negotiation application of SPAND. We show how SpandConneg utilizes the SPAND measurement service and HTTP's Content Negotiation mechanism. We also quantify the potential benefits of using content negotiation to improve client-side as well as server-side HTTP performance. We find that web clients that use SPAND can reduce the frequency of excessive download times from 35% to 10%, and web servers that use SPAND can increase their throughput by a factor of four.

In Chapter 6, we present LookingGlass, the Mirror Selection application of S-PAND. We describe how LookingGlass uses SPAND to drive its adaptation decisions and quantify the benefits of using network performance information to make intelligent mirror selection choices. More than 90% of the time, our technique allows clients to download mirrored web objects within 40% of the fastest possible download time. Finally, in Chapter 7, we conclude and present possible directions for future work in enhancing the performance of SPAND as well as ideas for new applications of SPAND.

Chapter 2

Related Work

The two major contributions of our work are a wide-area network measurement service and a collection of adaptive networked applications that use this service to drive their adaptation policies. Each of these contributions builds upon the lessons learned by existing systems that have similar goals. In this chapter, we describe related work in the areas of wide-area network performance measurement and adaptive networked applications. In Section 2.1, we discuss existing tools and architectures for network performance measurement. Then in Section 2.2, we describe the network measurement mechanisms and policies used by adaptive networked applications. Finally, in Section 2.3, we conclude by comparing the techniques used in previous work with the techniques used in our architecture.

2.1 Wide-area Network Measurement Tools

Previous work in wide-area network measurement has almost exclusively focused on *active* probing, where tools measure the state of the network by injecting probe packets into the network. The objective of these probes is to measure the latency, packet loss probability, bandwidth, or other characteristics along the network path from one host to another. There are too many tools to mention here, and the list of tools is constantly growing. A more comprehensive and up-to-date listing of network measurement tools can be found at the CAIDA web site [17]. In the following sections, we present representative examples of these tools, grouping them by the statistics they measure and describing how they measure them.

2.1.1 Latency and Packet Loss Probability

Many network measurement tools send periodic round-trip echo probes to a distant host who then responds back to the recipient. These probes are usually in the form of ICMP Echo packets, and are called NetDyn probes in [11] [10], Netnow probes in [55], and Fping [29] probes in Imeter [70].

These tools typically report the round trip time (how long it takes for the probes to travel to the distant host and back) and the packet loss probability (what fraction of the probes are dropped along the network path to and from the distant host) statistics for the network path from the local host to the distant host. If a significant fraction of the probes are dropped or the round trip time for the probes is excessive, this indicates that the network path between the two hosts is congested.

MINC [16] expands on this technique by using multicast traffic to a group of recipients rather than unicast traffic to a single host. The sender sends probe traffic into the network and examines the packet loss and delay statistics at each recipient. In conjunction with knowledge of the topology of the multicast tree from the source to the recipients, this allows the sender to determine latency and packet loss characteristics of specific internal links in the network.

2.1.2 Peak Bandwidth along a path

Other network measurement tools send groups of back-to-back packets into the network and examine the spacing between the packets when they return to the sender. These are called bprobes in [19] and bing probes in [9]. Bprobes are used in VitalSign's NetMedic [78] product.

As pointed out in earlier work on TCP dynamics [40], the back-to-back packets are separated at the bottleneck link either by the link itself or by competing traffic at the link. This spacing is preserved on higher-bandwidth links. Out of many measurements, one pair of packets is likely to traverse the bottleneck link without competing traffic and the spacing is due only to the characteristics of the link. By making many probes and measuring the minimum spacing between the packets when they return, a sender can estimate the raw bandwidth of the bottleneck link.

Packet Bunch Mode (PBM) [65] extends this technique by analyzing various sized groups of packets inserted into the network back-to-back. This allows PBM to handle multi-channel links (i.e., ISDN connections, multi-link Point-to-Point Protocol (PPP) links, etc.) as well as improve the accuracy of the resulting measurements. Pathchar [41] combines this technique with the hop-by-hop features of traceroute [74] to measure the link bandwidth and latency of each hop along the path from one host to another.

2.1.3 Available Bandwidth along a path

Although the minimum spacing between packet pairs can be used to estimate the raw bottleneck link bandwidth, assumptions about the queuing behavior in the network are necessary before using the average spacing between packets to estimate the available bandwidth at the bottleneck link. If routers in the network implement fair queuing, the packets will be evenly separated at the bottleneck link by traffic from other flows and the packet spacing can be used to estimate a particular flow's "fair share" of the bottleneck link's bandwidth [44] [43]. Unfortunately, few routers today implement per-flow fair queuing, so this assumption is often wrong in practice.

Other tools measure the available bandwidth along the path between two hosts by sending simulated connections into the network. Cprobe [19] sends a short sequence of ICMP echo packets from one host to another as a simulated connection (with minimal flow control and no congestion control). By assuming that "almost-fair" queuing occurs over the short sequence of packets, cprobe provides an estimate for the available bandwidth along the path from one host to another. Combined with information from bprobes, cprobes can estimate the amount of competing traffic along the bottleneck link. TReno [47] also uses ICMP echo packets as a simulated connection, but uses flow control and congestion control algorithms equivalent to that used by TCP.

Other approaches measure available bandwidth by making end-to-end measurements between applications. Lawrence Berkeley Lab's Network Probe Daemon (NPD) [65] provides a mechanism to remotely invoke traceroute or bulk transfer probes from one network host to another. This tool has been used in work on measurement of routing [63] and TCP [64] dynamics.

2.1.4 Application-level Response Time

Other tools measure application-level metrics such as response time rather than network-level metrics. Keynote [45] and Servicemetrics [71] are commercial web server measurement services that periodically fetch pages from a predefined list of web servers. Subscribers, usually the web server administrators, pay for access to the results of the fetches. Timeit [76] is a free software tool that also measures application-level web response times.

2.1.5 Internet Measurement Architectures

Other efforts build upon the tools described above to provide a more generalized measurement architecture. Paxson et al [66] propose a generalized measurement infrastructure that builds upon per-domain Network Probe Daemons. The goal of the Host Proximity Service (HOPS)/Internet Distance Maps (IDMAPS) [30] [31] [73] is to provide a global "distance map" between Internet domains. The definition of "distance" is designed to be general, but current instantiations of them rely on routing metrics or round trip time. Per-domain *Tracers* initiate probes to distant Tracers to measure domain-to-domain distances. Tracers then exchange this information with distant Tracers to build a global distance map.

Work at UCSD uses a similar architecture [80]. The Network Weather Service uses periodic probes between a number of distributed servers to determine the available bandwidth to and CPU load on each server. It is somewhat extensible in that a client can use one of a number of different averaging algorithms to calculate an "average" throughput or CPU load statistic.

In the next section, we describe how adaptive applications use these and other network measurement techniques to drive their adaptation decisions.

2.2 Adaptive Applications

Many networked applications are adaptive, making application-level decisions based on current network and endpoint conditions. These applications usually fall into one of the following categories: *Server Selection Applications* that choose a endpoint to communicate with based on current conditions, and *Content Negotiation Applications* that choose a data representation to use based on current conditions. Within these classifications, however, the actual performance metrics they use to drive their adaptation decisions vary widely from application to application.

In this section, we discuss several example Server Selection and Content Negotiation applications and describe the mechanisms they use for adaptation and the performance metrics they use to drive their adaptation decisions.

2.2.1 Server Selection Applications

In Server Selection Applications, a client is presented with a list of servers that replicate the same content or provide the same service. The client must then choose one of the servers based on whatever network or endpoint information that it has available to it. The mechanisms by which the client is presented with the list of servers and chooses one of them varies from application to application, as well as the policy that the client uses to choose one of the mirrors.

Server Selection at Web Servers

Many web server installations consist of a cluster of individual machines and perform some amount of load balancing between them. This is a transparent form of Server Selection, in that the client is unaware or uninvolved in the selection process.

One simple way to perform load balancing between individual machines is to map the Domain Name System (DNS) name of the web server to multiple IP addresses. The DNS server for the web server domain performs load balancing by returning one of the IP addresses in response to a DNS query. Usually, this is done in a round-robin fashion [1] [42], but more sophisticated techniques take server load into account [53] [67].

Although this technique is usually used to perform load balancing between server machines that are in close proximity to each other, it is occasionally used to perform server-side load balancing for distributed web servers. For example, DNS Round Robin was used to distribute load between web servers in Japan and the United States for the 1998 Olympics web site [21]. Although the web servers were not geographically close to each other, it is not a true example of wide-area server selection because this system did not take the wide-area performance along the path from clients to servers into account. In this case, the servers in Japan and the United States were connected by a private high-speed network that effectively eliminated wide area considerations.

Server Selection at Web Clients

Other systems perform server selection at the client side of the network instead of at the server side. This approach has the advantage of offloading the network measurement and load balancing processes from an already busy sever complex to the client side of the network. The disadvantage of a client-side approach is that it is no longer transparent: applications at the client side of the network, either web clients or per-domain web proxies, must make network measurements and load balancing decisions. The major difference between the systems described below are the mechanisms used to choose between mirrors and the measurement techniques used to determine the "best" server to contact.

Carter et al. at Boston University [18] use cprobes and bprobes to classify the connectivity of a group of candidate mirror sites. They also contrast the use of bprobes and cprobes against alternate metrics for mirror selection such as hop count, showing that hop count is a poor predictor of actual performance. IPV6's Anycast [35] [62] service provides a mechanism that directs a client's packets to any one of a number of hosts that represent the same IP addresses. This service uses routing metrics as the criteria for server selection. Cisco's DistributedDirector [23] product relies on measurements from Director Response Protocol (DRP) servers to perform efficient wide area server selection. The DRP servers collect Border Gateway Protocol (BGP) and Interior Gateway Protocol (IGP) routing table metrics between distributed servers and clients. When a client connects to a server, DistributedDirector contacts the DRP server for each replica site to retrieve the information about the distance between the replica site and the client. Work at the university of Colorado [33] focuses on ways to use topology and hop-count probes to locate and select the closest server using hop count and latency metrics. The primary focus of this work was measuring the number of hosts contacted before a replica of a given service was found.

Harvest [13] [14] [22] uses round-trip latency to identify the best peer cache from which to retrieve a web page. Requests are initiated to each peer cache, and the first to begin responding with a positive answer is selected and the other connections are closed. Other proposals [34] rely on geographic location for selecting the best cache location when push-caching web documents.

Work at Georgia Tech [24] details an implementation of application-level anycasting [8] in the context of wide-area replicated server selection. This approach uses per-domain application-level probing clients that make periodic probes to replicated servers in conjunction with server pushes of load information. Clients can consult the probing client to determine the current connectivity to distant hosts.

2.2.2 Content Negotiation Applications

In Content Negotiation Applications, applications have a choice of data representation to use when communicating with distant hosts. The goal of these applications is usually to keep response time or performance fixed at the expense of content fidelity. Particular applications differ in the mechanisms they use to present these different data representations and the policies they use to choose one representation over another.

HTTP's Transparent Content Negotiation mechanism [38] provides a way for web servers to present multiple representations of web objects and a way for web clients to choose between them. However, this provides only a mechanism and leaves the policy of which representation to choose completely unspecified. Odyssey [58] [57] focuses on a file-system oriented mechanism for content negotiation, providing multiple data representations at the file system level and an API that allows applications to specify adaptation policies. Adaptation is based on isolated passive measurements of per-path bandwidth and latency.

In Transend [28] [26], web clients can use a transcoding proxy [28]) to change the quality of web objects to match available bandwidth. A fixed number of representations are available for transcodable web objects such as images, and clients manually choose one of the representations to receive from the web proxy (an example of a static policy). In this application, there is no feedback loop from the client to the proxy to drive dynamic adaptation.

A similar system was developed at Columbia University [83]. In this system, clients could insert customized content filters into proxies on the other side of a low-bandwidth links. Example filters did lossless (gzip) as well as lossy (multimedia frame dropping) compression of data on a per-connection basis.

A client receiving a multimedia stream can use a real-time transcoding service such as a Video Gateway [3][2] to change the data rate of a multimedia stream depending on network characteristics. Again, the representation is manually chosen by the client, and there is no feedback loop between the client and the transcoding service.

RealMedia [68] player applications have a limited form of content negotiation functionality. A multimedia clip at a RealMedia server can be available at one of a number of bitrates. While downloading and playing a streaming media clip, the RealMedia player measures the current packet loss rate. If this loss rate is high due to limitations of the player, network path, or RealMedia server, it instructs the RealMedia server to switch mid-stream to a lower bitrate representation.

In the next section, we compare these past efforts in network measurement tools and adaptive networked applications with our work.

2.3 Comparison of Related Work to SPAND

Tables 2.1 and 2.2 summarize the previous work in the areas of network measurement tools and adaptive networked applications. In this section, we compare this related work with our system along several axes that we feel are important design decisions to make when designing network probing systems.

2.3.1 What Metric is Measured?

One factor to consider is the actual metric that is measured. Some approaches measure packet loss statistics, round trip time, peak bandwidth, routing metrics, or geographic location and use them as approximations to application-level response time. In contrast to these approaches, SPAND allows applications to directly measure application-level statistics such as response time and use these measurements to drive adaptation decisions.

This difference is important because these network-level statistics are often poor estimates of application-level completion time. Latency and packet loss measurements are usually used only to determine if the path to a given host is extremely congested. Peak bandwidth measurements only give an upper bound on the performance that a client will see. Past work has shown that hop count and geographic location are poorly correlated with available bandwidth [18] [51]. In addition, in Chapter 6, we present results for LookingGlass, a server selection tool that uses SPAND's measurements to choose one of a number of servers that replicate the same content. We find that metrics such as hop count and geographic location are poor predictors of good performance as compared to application-level response time.

By measuring application-level response time, we avoid these problems and ensure that applications measure (and receive information about) the characteristics they are most interested in.

System	What metric is measured	How metric is measured	Additional traffic introduced	Where deployed	Flow/ congestion control?
NetDyn/ NetNow/ Fping probes	Per-path latency, packet loss prob.	Artificially generated	Significant (≈10K)	Each client	No
MINC	Per-link latency, packet loss prob.	Artificially generated	Significant (≈10K)	Group of clients	No
Bprobe	Peak bottleneck bandwidth	Artificially generated	Little (≈1K)	Each client	Only 2 packets
Bing,Packet Bunch Mode	Peak bottleneck bandwidth	Artificially generated	Significant (≈10K)	Each client	No
Pathchar	Hop-by-hop peak bandwidth, latency	Artificially generated	Significant (>10K)	Each client	No
Packet Pair	Available bottleneck bandwidth	Artificially generated	Little (≈1K)	Each client	Only 2 packets
Cprobe	Available bottleneck bandwidth	Artificially generated	Significant (≈10K)	Each client	No
Treno	Available bottleneck bandwidth	Artificially generated	Significant (≈10K)	Each client	Yes
Network Probe Daemon	Available bandwidth, hop-by-hop route taken	Artificially generated from application	Significant (>10K)	Client and server domain(s)	Yes
Keynote, Servicemetrics	Application-level response time	Non-live from application	Significant (>10K)	Internal Network	Yes, uses TCP
Timeit	Application-level response time	Non-live from application	Significant (>10K)	Each Client	Yes, uses TCP
HOPS/ IDMAPS	Global per-path latency and packet loss prob	Artificially generated from application	Significant (≈10K)	Client and server domain(s)	No
Network Weather Service	Application-level response time, server load	Non-live from application	Significant (≈10K)	Client and server domain(s)	Yes, uses TCP
SPAND	Application-level stats e.g. response time	Live from application	Little (perf reports and queries)	Each client and client-side domain	Whatever application does

Table 2.1: Summary of Network Probing Tools and SPAND

System	What metric is used for adaptation	How metric is measured	Additional traffic introduced	Where deployed
Round Robin DNS	None	Static	None (static)	Server domain
Load Balancing DNS	Server load	Live from application	Little (Load queries and responses)	Server domain
IPV6 Anycast	Routing metric	Live from application	Little (routing data and queries)	Internal Network
Cisco Distributed Director	Routing metric	Live from application	Little (routing queries and responses)	Client domain and internal network
Univ of Colorado	Routing Metrics	Live from application	Significant, (if clients measure) little (if anycasting used)	Each client and internal network
Harvest	End-to-end latency	Live from application	Little ($\approx$1K)	Internal network
Harvard	Geographic location	Static	None (static)	Server side, internal network
Georgia Tech	Application-level response time	Non-live from application	Significant (>10K)	Each client and client-side domain
HTTP Content Negotiation	Unspecified	Unspecified	Unspecified	Client and server side
Odyssey	Per-path bandwidth, latency	Live from application	None (all at client)	Each client
TranSend/ Columbia/ Video Gateway	None	Static	None (static)	Client-side domain
RealMedia	Packet loss prob	Live from application	None (all at client)	Each client
SPAND	Application-level stats e.g. response time	Live from Application	reports and queries)	Each Client and Client-side domain

Table 2.2: Summary of Adaptive Applications and SPAND

2.3.2 How is the Metric Measured?

Another important consideration to consider is how the system makes its measurements of the network. Even if a system measures an application-level metric, it may do so in a way that is different than the way in which an application would measure the same metric. For each system described above, we placed it in one of the following categories that describe how close the system's measurement of a metric is to an actual application's measurement of the same metric. Moving down the list, each category becomes progressively closer to live measurement by actual applications.

- *Static*: The approach relies on static information that is not dynamically measured at all.
- *Artificially Generated*: The technique uses artificially generated traffic that may be treated differently by the routers or endpoints in the network (for example, ICMP packets).
- *Artificially Generated from Application*: Application-to-application traffic is used to measure performance, but the application is not one that is actually used by clients (for example, TCP sink or chargen ports).
- *Non-live from application*: These systems use a "typical" workload from actual applications. This workload must be determined in advance.
- *Live from application*: Actual application level performance statistics as they are experienced by clients are used to measure performance.

By examining the summary of previous work, we see that most of it uses artificially generated traffic for network measurements. In particular, many network measurement tools use ICMP traffic to measure packet loss and round trip time statistics. The advantage of this approach is that it eases deployment: most hosts respond to ICMP Echo requests, so by using ICMP traffic, no changes are required to distant hosts. However, ICMP traffic is sometimes treated differently by routers in the network than application-level traffic. We show a detailed example of this in Section 3.4.3, where ICMP traffic is blocked by a firewall in the network, leading to incorrect loss rate measurements.

A few systems use artificially generated traffic from an application. This does not suffer from the above limitations, but the disadvantage of this approach is that it does not measure application-level bottlenecks that may limit actual application-level performance. For example, the connectivity to a distant domain may be good, but if a web server is down or overloaded, an application will still see poor performance.

Some systems use artificial workloads to drive application-level "robots". This captures application-level bottlenecks, but works well only if the workload closely tracks real applications' usage of the network. For example, if clients visit sites that are not included in the artificial workload, those clients will not have any network measurement information to use in making adaptation decisions.

Some systems use live measurements of hop count or server load metrics to drive adaptation decisions. Although using live measurements is the best possible way to measure these metrics, hop count metrics correlate poorly with application-level response time, and server load does not take wide-area performance into account at all.

Of all the systems described above, only Odyssey, Harvest, RealMedia, and SPAND use live application measurements of meaningful statistics to drive adaptation decisions.

2.3.3 How Much Traffic is Introduced?

Another factor to consider is the amount of additional traffic that must be introduced to make and report network measurements. This is important because the additional traffic introduced to perform measurements is not directly used by any application, and would significantly decrease the goodput of the network if all hosts independently probed the network before each connection.

From looking at the summary of previous work, we see that many tools introduce significant amounts of probe traffic into the network. For example, pathchar sends at least tens of kilobytes of probe traffic per hop, and a cprobe sends 6 kilobytes of traffic per probe. This amount of probe traffic is a significant fraction (approximately 20%) of the mean transfer size for many web connections ([4] [5] [32]) as well as a large portion of the mean transfer size for many web sessions. If clients individually used these tools to probe the network, they would quickly overwhelm web servers with probe traffic. We present a more quantitative example of this in Section 3.4.2, showing that active probing can limit the scalability of a distributed web server system.

In contrast, SPAND introduces a minimal amount (approximately 100 bytes per connection) of additional traffic into the network in the form of small performance reports and queries. In addition, all of this traffic is confined to the local domain and none of it traverses the wide-area network.

2.3.4 Where is the Service Implemented?

Another design choice to consider is where the system must be deployed. A measurement service that is deployed only at the endpoints of the network is easier to maintain and deploy than one that requires support from the internal network infrastructure. In addition, a system that is deployed only at the client side of the network is easier to deploy than a system that relies on client and server side components. From the summary, we see that many systems depend on server side or internal network support. Some approaches such as the Network Probe Daemon depend on the deployment of probing hosts at both endpoints of the network. Others systems such as Cisco's DistributedDirector depend on the ability to listen to internal routing metrics, which can present significant administrative challenges in a heterogeneous network where backbone networks are administered by different entities than endpoint networks. Systems such as Keynote and Servicemetrics are in practice deployed in the internal network to make accurate measurements only of the path from the internal network to a particular web server.

SPAND, however, does not depend on any support outside the local client's domain. To deploy our system, a domain must simply install a performance server (and optionally a packet capture host) and configure applications to use these services. This makes it easy to gradually deploy this service in a heterogeneous, distributively managed network like the Internet.

2.3.5 Does the Approach use Flow or Congestion Control?

Another consideration is whether or not the network probing algorithm uses flow and congestion control while making measurements. This is important for two reasons:

- Approaches that do not use such mechanisms are less likely to accurately reflect actual application performance, because many reliable transport protocols today (i.e., TCP) do implement flow and congestion control.
- If many Internet hosts use tools without flow or congestion control, routers and endpoints may quickly become swamped with probe traffic, leading to persistent congestion.

Most network probing algorithms do not exploit flow or congestion control. Relatively harmless examples include the NetDyn/NetNow/Fping tools. These send small ICMP packets at a rate of once per second regardless of congestion. This is intentional: usually these tools measure packet loss as an indicator of congestion. A more harmful example is cprobe, which deliberately sends kilobytes of data into the network at a rate faster than the bottleneck link of the network. If every client used cprobe to probe the network before initiating a transfer, the likelihood of congestion would be significantly increased.

In contrast, SPAND makes passive measurements, relying on application to application traffic for network measurements. As a result, any lack of flow or congestion control is due to applications and not our system.

2.4 Summary

In this chapter, we have presented a summary of past efforts in network measurement tools and adaptive networked applications. Previous work in individual network measurement tools has measured network-level characteristics such as packet loss rate, round trip time, and peak and available bandwidth, both on a per-path and per-hop basis. Other tools make application-level measurements that capture the performance of the entire end-to-end, client-to-server system. Other efforts have focused on using these tools to deploy global network measurement infrastructures.

Previous work in adaptive applications has focused primarily on developing the mechanisms for adaptation, with less attention to the policies that drive these mechanisms. These mechanisms can be divided into server selection mechanisms, where applications choose between a number of hosts that replicate the same content or service, and content negotiation mechanisms, where applications choose a data representation to use based on current network conditions.

We also compared the design choices made by these past efforts with the design choices made in our system. From this comparison, we found that the significant shortcomings of many existing network performance discovery and adaptive applications are:

- *Isolated Measurement*: Using measurements from a single host to characterize the state of the network.

- *Active Measurement*: The introduction of traffic into the network in order to measure it.

- *Lack of Application-level Measurement*: The use of metrics such as hop count, latency, and geographic location as imprecise estimates of actual end-to-end application performance.

We discuss these shortcomings further in the next chapter by examining in detail the design choices made by SPAND (*Shared, Passive, Application-specific* measurements) as well as the advantages and disadvantages of these design choices over alternate design choices.

Chapter 3

Methodology

3.1 Introduction

In this chapter, we discuss the network model and assumptions that underlie our work, and how this leads to the three major design choices that we make in SPAND: *shared*, *passive*, *application-specific* measurements. In Section 3.4, we discuss both qualitatively and quantitatively the advantages of these choices over alternate design choices. To show the advantage of shared measurements, we analyze access patterns of web clients from a client-side packet trace and find that sharing performance measurements can significantly increase (from less than 50% to over 80%) the likelihood of having relevant performance information. To demonstrate the advantage of passive measurements, we present an example where the use of active measurements decreases the throughput of a mirrored web server complex by as much as 100%. To illustrate the advantage of application-level measurements, we contrast simultaneous network and application level measurements of packet loss statistics and present an example where network level tools consistently overestimate the likelihood of packet loss due to a lack of knowledge about application level behavior.

We then discuss the challenges that arise from the use of shared, passive, application specific measurements. These manifest themselves by adding *measurement noise* which limits the granularity of application adaptation choices. We divide measurement noise into three components: *network noise*, the variation that is inherent in the network, *sharing noise*, additional inaccuracy that results from inappropriately sharing performance information between hosts, and *temporal noise*, the result of using past out-of-date information to predict current performance. In Section 3.5.1, we present experiments designed to quantify the contribution of each component to measurement noise. We find that network noise is by far the largest contributor to measurement noise and limits the granularity of adaptation decisions to coarse grained (order of magnitude) decisions.

Finally, we conclude the chapter by discussing which classes of applications benefit most from using SPAND and which which will not.

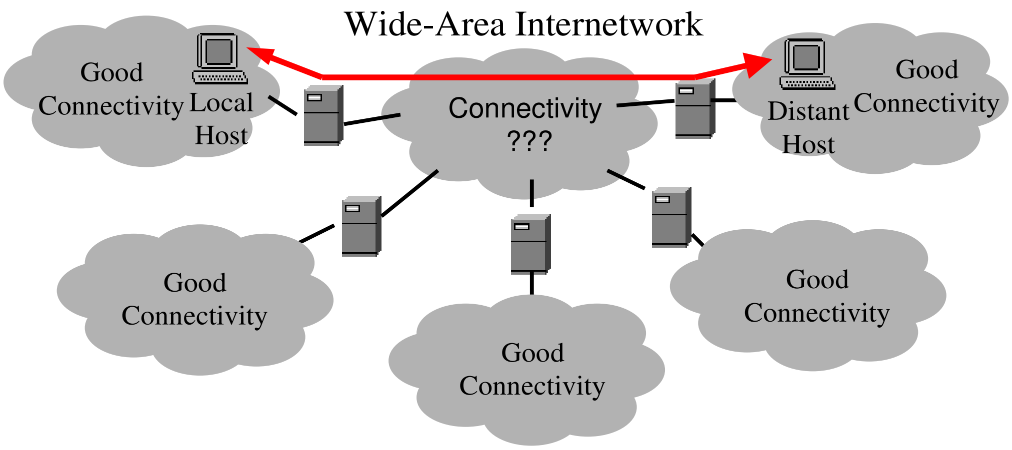

Figure 3.1: Network Model behind SPAND. Local Hosts in well connected domain communicate with distant hosts through an Internetwork, the properties of which are unknown.

3.2 Network Model and Terminology

In this section, we present the network model that is behind our work and the terminology that will be used throughout the rest of the thesis.

Our network model is summarized in Figure 3.1. The network is abstracted into *domains* of relatively high-bandwidth, low-latency, uncongested connectivity, each connected by a wide-area internetwork. The properties of the network path between these domains are unknown, but are usually an order of magnitude lower quality (i.e., lower bandwidth, higher packet loss rate) than intra-domain paths. In addition, the quality of a wide-area path can vary widely from path to path. A domain is the basic unit of performance sharing in SPAND. All hosts in a domain cooperate to share information about the state of wide area paths between domains.

Although the above description defines what a domain is, it does not specify exactly what collection of Internet hosts comprise a domain. For a group of hosts to be in the same domain, the following requirements must be met:

- The hosts in the domain should have the same connectivity to other distant Internet hosts. This could be because they share the same bottleneck link to the rest of the Internet (for example, their wide-area connection to the Internet).
- Hosts should be under control of the same administrative entity. The reason for this is that the collection of hosts share performance information by placing it in a per-domain repository, as described further in Chapter 4. With the hosts under control of the same administrative entity, this eases the process of administering the repository.

Given these two requirements, we can more formally state that a domain is comprised of the largest collection of hosts that meet these two criteria. For example, at UC Berkeley there is a domain consisting of all hosts under the administrative control of UC

Berkeley that connect to the Internet via local area networks (LANs). Note that this domain explicitly excludes Berkeley hosts that connect to the Internet via dial-up modem connections, because the two groups of hosts do not share the same bottleneck link.

3.2.1 Techniques for Grouping Together Distant Hosts

Hosts in a local domain are explicitly grouped together by placing their network performance information in the same repository. This is described in more detail in Section 4.1. However, there is no mechanism by which local hosts can determine the domain to which a particular distant host belongs. We considered using several heuristics to group together distant hosts:

- Relying on measurements of network topology such as traceroute probes to determine the topology of distant domains.
- Relying on the structure of addresses to group distant hosts. For example, assume that two hosts with IP addresses differing only in the last octet are on the same subnet and as a result have similar connectivity.
- Performing Domain Name System (DNS) lookups from IP addresses to domain names and using domains to group together distant hosts.

Each of these heuristics has limitations. The first approach requires building a topology map of the entire Internet from the viewpoint of the local domain, and clearly introduces a significant load on the network. The second reduces this problem to some degree by relying on the structure of IP addresses instead of traceroute probes. However, both approaches only determine topology (i.e., hosts A and B are k network hops away from each other) but do not determine similar connectivity (i.e., hosts A and B share the same bottleneck link). The third approach assumes that hosts in the same DNS domain will have similar connectivity. This is often not the case. For example, using this technique, modem hosts and LAN hosts in UC Berkeley would be grouped together into a single domain, even though they have very different connectivity.

Because of these limitations, we chose to consider distant hosts independently and not perform any grouping of distant hosts into distant domains. This conservative technique limits the degree of performance information sharing to local hosts in a domain, but is guaranteed to avoid mistakenly grouping together hosts with differing connectivity.

In summary, under this network model, the goal of SPAND is to measure the application-level network performance along the path from hosts in a local domain and a particular host in a distant domain and make that performance information available to applications in the local domain.

3.2.2 Application Classes

The network model described above specifies how the organization of the network defines the set of clients that share performance information, but not how application-level behavior further limits the set of applications that can effectively share performance information. This is important to consider because not all applications use the network in the same way, and as such, there is a need to separate applications into *classes* of equivalent functionality.

As an example of this, consider the various abstractions that a TCP connection provides (flow control, congestion control, and reliability) and the ways in which different applications utilize these features of TCP connections. Some applications (such as web browsers and FTP programs) use TCP connections for bulk transfers and depend on the reliability, flow control, and congestion control abstractions that TCP connections provide. Applications such as telnet primarily use TCP connections for reliability and not for flow or congestion control. Other applications such as RealAudio in TCP mode use TCP connections for different reasons such as the ability to traverse firewalls.

The network performance metrics reported by different applications may vary widely depending on the application. For example, available bandwidth metrics from telnet applications are not likely to be meaningful to bulk transfer applications. Streaming media applications typically send data at a fixed rate, so they have little need of an available bandwidth metric, other to determine that the rate they plan to use can be supported by the network.

With this in mind, we define an *Application Class* to be all applications that use the network in a similar way with respect to the performance they observe, and as a result, can meaningfully share network performance information. Just as network location groups a client into a sharing domain, the choice of application groups a client into a particular application class that best describes their desired performance. In Chapter 4.3, we present several sample application classes.

With this in mind, we can restate the goal of SPAND as: *for applications in same application class*, to measure the application-level network performance along the path from clients in a local domain and a particular host in a distant domain and make that performance information available to applications in the local domain.

In the next section, we show how the network model and terminology described above leads to the design choices we make in SPAND.

3.3 Design Choices in SPAND

Our architecture incorporates several important design choices that offer significant advantages over alternate approaches:

- Our measurements are *shared*. Hosts cooperate by placing their individual measurements of network performance in a centralized per-domain repository.

The decision to use shared measurements follows directly from the network model. If two hosts in a local domain have high-quality connectivity to each other and lower-quality connectivity to some distant host, they can share performance information because it is likely that they share the same bottleneck link with respect to the distant host.

An alternate design choice rather than a shared measurement approach is for each host to independently measure and store the characteristics of the network. Many of the network measurement systems described later in this section use this approach. The advantage of shared measurements is that they increase the likelihood that a host will have up-to-date performance information about a distant host, because it can leverage the collective knowledge of all hosts in an domain. This can be especially useful when applications are relatively short-lived and do not keep the information they measure on stable storage. In SPAND, we provide a centralized persistent repository of performance information that can be used by all hosts in an domain.

- Our measurements are *passive*. Instead of introducing traffic into the network in the form of probe packets or simulated connections, we rely only on the traffic that applications generate as they communicate with other Internet hosts.

We decided to use passive measurements instead of active probing because this allows us automatically to measure performance for popular distant hosts. Distant hosts that interact often with local hosts will generate more traffic, and as a result, receive more network measurements. Those that are unpopular will receive less samples. If we were to use active probing instead, we would have to determine how often to probe distant hosts and make sure that we were probing hosts that were actually being visited.

Another advantage of using passive measurements over active probing is that by making passive measurements, clients do not introduce artificial probe traffic into the network. This probe traffic can sometimes total tens of kilobytes, as in the case of Cprobes [18], Packet Bunch Mode [65], or Pathchar [41]. This traffic is not directly used by any application and would significantly decrease the goodput of the network if all hosts independently probed the network before each connection. We present an example of this in Section 3.4.2.

- Our measurements are *application specific*. Instead of measuring network-level statistics such as routing metrics, latencies, or link bandwidths in our system, we rely whenever possible on application-level measurements (i.e., response time, web page download time) to drive application decisions. Each application independently defines its own measurement of network performance and tags its information with the type of application that measured it.

Many existing systems rely on imperfect metrics to make application-level decisions. Systems such as IP Anycast [62], Cisco's DistributedDirector [23], and work at the University of Colorado [33] rely on routing metrics for server selection. Harvest [13] [14] depends on round trip time to choose the best peer WWW cache. Work at Harvard [34] relies on geographic location to disseminate web documents. Other frameworks are more general but current instantiations of them rely on routing metrics [31] or round trip time [73]. Some approaches, such as Packet Pair [44] [43], measure available bandwidth but make assumptions about the queuing discipline in the network.

Although these metrics may correlate with application-level performance, they are not guaranteed to always reflect exactly the metric in which local hosts are interested. For example, a host may be many network hops away but still perform better than a host that is one or two hops away. A distant RealMedia server may have excellent network connectivity, but if the RealMedia server is down or overloaded, an application will still observe poor performance.

Even in cases where "robots" initiate application-level transfers such as work from Georgia Tech [8] [24] or UCSD [80], care must be taken to assure that application-level transfers accurately reflect the usage patterns of actual clients. The transfers they initiate may be too long or too short, they may download a different set of web pages than actual applications, or the spacing between transfers may not reflect an actual user's "think time".

By making passive application-level measurements, we track actual communication patterns and as a result automatically make measurements for the hosts that are accessed most often and that most closely matches actual application level performance.

- Our system is *extensible*. Our architecture makes it very easy to integrate new applications and ways to measure the network into SPAND.

Because our system utilizes application-specific measurements whenever possible, we cannot predict in advance all possible applications that may wish to use SPAND to measure and store network performance information. Rather than providing only a limited set of network performance metrics such as bandwidth, latency, packet loss probability, etc, we also present a more database-style API to applications where application classes can specify their own network performance report formats. This does not mean, however, that we should not measure and store basic performance metrics such as bandwidth and round trip time. Applications that do not want to measure their own performance can rely on these basic metrics to make application-level decisions.

Many of the systems described in Chapter 2 use some, but not all, of the same design principles as in SPAND. For example, many systems share network performance information by designating a single host to make active probes on behalf of a number of clients. Others passively listen to routing table exchanges to determine routing metric (network-level) information. However, of the systems described above, only SPAND uses all of the design principles that are important in an architecture for supporting adaptive applications.

In the next two sections, we present more quantitatively the advantages of using shared, passive, application-specific measurements over isolated, active, network-level measurements and the challenges that arise from using shared, passive, application-specific measurements.

3.4 Advantages of Shared, Passive, Application Specific Measurements

In this section, we discuss in more detail the advantages of shared, passive, application specific measurement over isolated, active, network-level measurements. We do this by comparing each design choice with its alternative in turn and showing quantitative results that highlight the benefits of using shared, passive, application specific measurements over isolated, active, network-level measurements.

3.4.1 Advantages of Shared Measurements

As described in Section 3.1, if two hosts are in the same domain, they share the same bottleneck link with respect to a distant host. As a result, the network-level measurement characteristics to this host are likely to be similar [5]. Using shared rather than isolated measurements allows clients to increase the likelihood of having up-to-date network performance information because they can combine their independent measurements of performance together into a larger, more comprehensive collection of information about a distant host.

To quantify the effectiveness of sharing measurements across a domain, we examined Internet usage patterns by analyzing client-side web traces and extracted the time at which individual web clients contacted individual web servers. From these, we compared the "freshness" of network performance information for a particular web server under two scenarios:

- When clients rely only on their own measurements for information about a distant host, and
- when clients rely on the collective measurements of the group of clients for information about a distant host.

More formally, for a single web server, we represent the list of contact times from a single client (or a shared collection of clients) as a sequence (t_1, t_2, ..., t_n). If the difference between t_{i+1} and t_i is small (less than ten seconds), we merge the events into a single web browsing "session." Clearly, the first arrival is never up-to-date due to a lack of any past information. In addition, we assume that the time between significant network changes is a fixed value D. Under this assumption, if $t_{i+1} - t_i > D$, then the client does not have up-to-date information on performance characteristics. If $t_{i+1} - t_i < D$, then the information is up-to-date. As mentioned previously [5] [65], a conservative value for D is on the order of tens of minutes.

To quantify the benefits of sharing, we chose different values for D and used the above model to count the number of times network performance information was up-to-date and was out-of-date under each scenario (sharing vs. isolated) and then compared the results.

Figure 3.2 shows the results of this analysis for a particular client-side trace consisting of 404780 connections from approximately 600 modem users over an 80 hour time period [77]. This packet trace was collected by placing a machine on the local area network between the modem users and the rest of the Internet. After the trace was collected, it was post-processed to determine the distant host that was accessed and the time at which the distant host was accessed.

In the figure, the x-axis represents the time D (in seconds) between network changes, and the y-axis represents the fraction of time that network performance information was up-to-date, using the techniques described in the previous section to calculate up-to-date and out-of-date arrivals. There are two curves in the figure: the upper curve represents the fraction of time that network performance information was out-of-date if no sharing between clients is performed, and the lower curve represents the number of probes

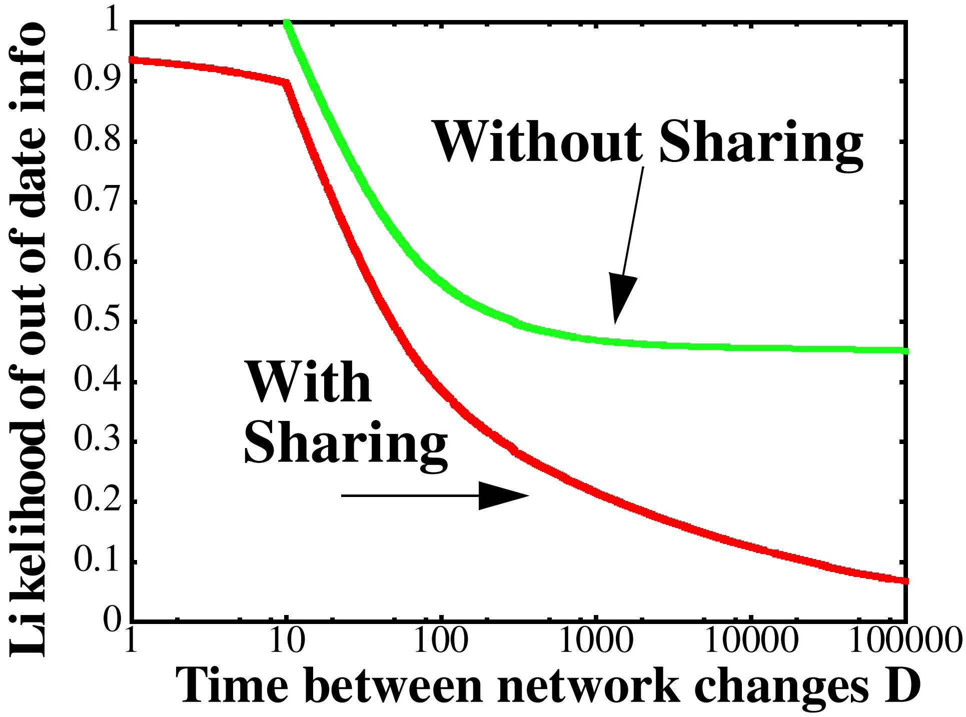

Figure 3.2: The benefit of sharing. Figure shows the likelihood of up-to-date information as a function of the time between network state changes.

that are necessary if clients share information between them. The upper curve begins at $D = 10$ seconds because of the "sessionizing" of individual connections described previously. We see that the likelihood of out-of-date information is consistently lower when clients share network information as compared to relying only on individual information.

In addition, for lengthier time scales, we see that the likelihood of out-of-date information plateaus in the case of individual measurements, while the likelihood of out-of-date information continues to decrease in the case of shared measurements. This is because individual clients may visit a particular web site once over a course of several minutes and not visit it again for hours or longer. When clients do not share information, the results of these measurements are lost because that client will not visit that web server again within the time scale D. When clients share information, however, a client that visits a server that was recently visited by a different nearby client can take advantage of previous information.

In addition to comparing the relative benefits of sharing performance information, we can also choose appropriate values for D and quantify the likelihood of up-to-date performance information in typical scenarios. When we do this, we see that even a relatively small collection of hosts can obtain timely network information when sharing information. If we assume that network conditions change approximately every 15 minutes (D=900), indicated by other studies [65] [5] as an acceptable rate of change for network performance, then the passive measurements collected from this relatively small collection of 600 hosts will be accurate approximately 78% of the time. For larger collections of hosts (such as domain-wide passive measurements) and slightly less conservative values of D, the availability of timely information will be even greater.

In summary, sharing network performance measurements between a group of similarly connected clients can significantly increase the likelihood of up-to-date measurements for groups of clients. For conservative values for the rate of change of the network, this likelihood can be as much as 78% of the time.

3.4.2 Advantages of Passive Measurements

The obvious advantage of passive measurements over active probing is that probes introduce network traffic that is not directly used by any application. This decreases network goodput. The degree to which this will be a problem depends on how often client applications probe the network and the ratio of probe traffic to application-level traffic. Regardless of this ratio, however, there are situations where probe traffic can be become "concentrated" at certain hosts in the network, thereby reducing their application-level throughput. We present an example of this concentration effect here.

For example, consider the scenario of web mirror sites that replicate the same content. In an active probing system, a client must first contact each of the mirror sites to determine which mirror is the "best." This slows down servers with probe-only traffic and limits their scalability. The following thought experiment shows why. Consider a web server with a variable number of mirror sites. Assume that each mirror site is connected to the Internet via a 45 MBit/second T3 link and that the mean transfer size is 100 KBytes and the mean probe size is 6 KBytes. These are optimistic estimates; most web transfers are shorter than 100 KBytes and many of the network probing algorithms discussed in Chapter 2 introduce more than 6 KBytes.

From a network perspective, an estimate of the number of requests per second that the collection of mirrors can support is the aggregate bandwidth of their Internet links divided by the sum of the average web transfer size and any associated probe traffic for the transfer. Under this model, the throughput of a web server complex with k mirrors is $k \times (45 \times 1024^2) \times .125/(100 \times 1024 + (k-1) \times 6 \times 1024$ in the case with network probes and $k \times (45 \times 1024^2) \times .125/(100 \times 1024)$, both measured in requests per second.

Figure 3.3 shows the number of requests per second that such a system can support as a function of the number of mirror sites for two systems: one without probe traffic and one with probe traffic. We see that from a network perspective, the system without probe traffic scales perfectly with the number of mirrors. For the system with probe traffic, however, for each web request that is handled by a single mirror, a network probe must be sent to all of the other mirrors. On the server side, this means that for each web request a particular mirror site handles, it must also handle a probe request from clients being serviced at every other mirror location. As the number of mirrors increases, the ratio of probe to application-level traffic increases and the number of requests served per second becomes limited by the additional probe traffic.

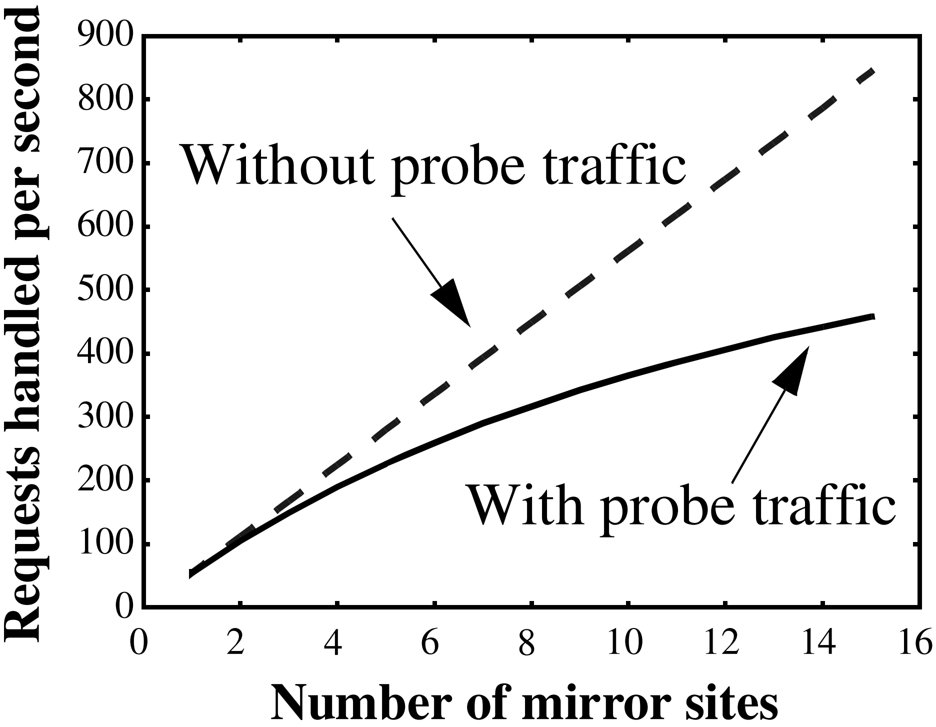

Figure 3.3: The effect of probe traffic on scalability. Figure shows requests/second that mirrors can serve as a function of the number of mirror sites.

3.4.3 Advantages of Application-Specific Measurements

The disadvantage of using network-level measurements is that the statistics they measure may not correlate well with application level measurements of the same statistic. To quantify the degree to which network- and application-level measurements may differ, we compared packet loss and failure rate statistics reported by RealMedia 5.0 applications with packet loss and failure rate statistics reported by ping, a network-level measurement tool. This experiment was conducted for a RealMedia client in the San Francisco Bay Area and a number of RealMedia servers spread throughout the Internet.

RealMedia and Ping Background

RealMedia is a streaming media application that uses a TCP connection to request an audio or video clip. This clip is sent from the server to the client using either a TCP connection or a stream of UDP packets. (The experiments below use the UDP-based transport). The client application implements a limited degree of reliability by maintaining a playout buffer and requesting individual packets that are lost. The loss is considered recovered if the retransmitted packet arrives before the gap travels through the playout buffer. RealMedia clients can be configured to report performance statistics to a RealMedia server at the end of the clip. These statistics include the total number of packets in the stream, the number of packets not received (lost) at the client as well as the number of packets received too late or too early to be used by the client.

Ping is an application that sends a sequence of ICMP echo request packets to a specified distant host. The distant host responds to each echo request packet with an echo response packet. Ping keeps track of which ICMP packets were lost between the local host and distant host and measures the round trip time between sending the echo request packet to the distant host and receiving the echo response packet from the distant host for all echo response packets that are received.

Experimental Setup

For each server, we ran the RealMedia client and ping program in parallel to measure the application- and network-level loss statistics along the path from the Bay Area client to that server. Each server was probed approximately 48 times (every 30 minutes) over a 24 hour interval. The duration of each probe was approximately 90 seconds.

To measure the application level performance, we used `tcpdump` [49], a network-level packet capture tool, to capture the packets sent by the RealMedia client. From the packet trace, we extracted the statistics that were reported by the RealMedia client to the RealMedia server. If the clip was not downloaded at all, we considered it a failure. Otherwise, we calculated the packet loss ratio as $(Lost + Early + Late)/(Total)$, where $Lost$ is the number of packets lost, $Early$ is the number of packets received too early to be of use, $Late$ is the number of packets received too late to be of use, and $Total$ is the total number of packets that should have been received. All of these totals are in the feedback sent back to the RealMedia server.

To measure network-level packet loss statistics and failure rates, we used the ping program to send ICMP echo request packets to the node that hosted the RealMedia server. Ping reports the fraction of ICMP echo reply packets that are received. If the packet loss rate was 100% or if ping could not look up the host name of the RealMedia server, we considered that a failure.

Once we had collected the two sets of measurements for each distant server, we compared the two sets of measurements both qualitatively and quantitatively. Differences between the measured statistics of the two methods indicate a difference in the way that each technique measures the state of the network.

To qualitatively compare the two sets of measurements, we graphed the cumulative distribution function of the application-level loss ratios and network-level loss ratios for a particular host on the same graph and compared the two.

Figure 3.4 shows the results for one of the server hosts (`audioraarc004.audionet.com`). These results are representative of the type of results observed for all hosts. From Figure 3.4, we see that the loss probability measured using ICMP packets is consistently greater than the loss probability measured by RealMedia applications. For example, the RealMedia loss rate was less than 2% more than 95% of the time, whereas the ping loss rate was less than 2% only 70% of the time. We describe why this occurs below.

In addition to this qualitative comparison, we also quantitatively compared the packet loss statistics and failure probabilities for each method. To compare packet loss statistics, we calculated the likelihood that the average network-level loss probability for a given host is the same as the application-level loss probability for the same host. We assume

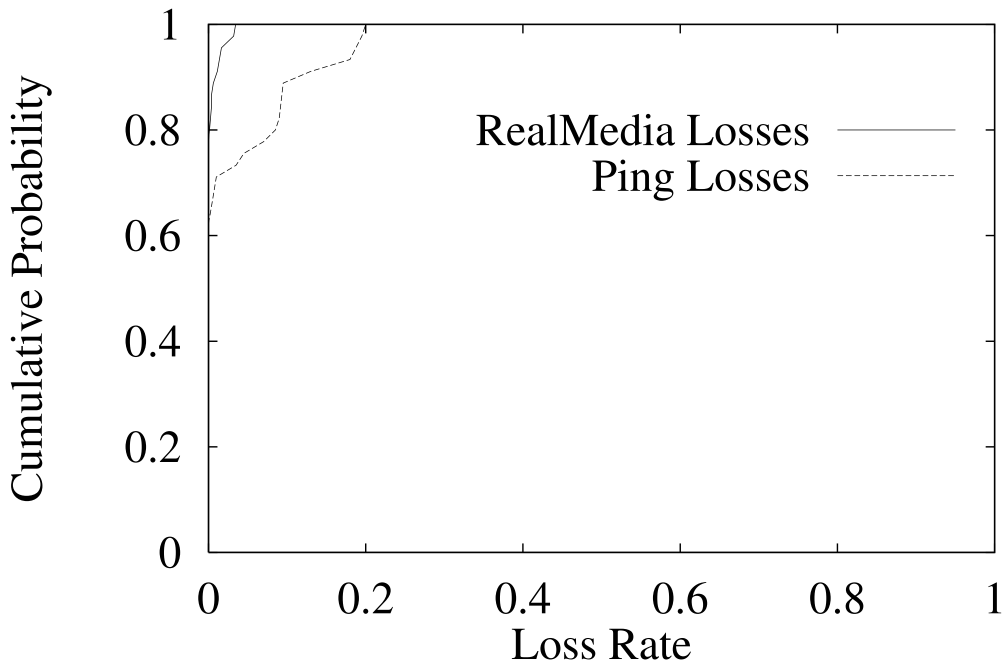

Figure 3.4: ICMP vs. RealMedia Loss Statistics for audioraarc004.audionet.com

that $X_1 ... X_n$ are samples of network-level loss statistics following a normal distribution with mean μ_x and standard deviation σ^2, and $Y_1 ... Y_n$ are samples of application-level loss statistics also following a normal distribution with mean μ_y and standard deviation σ^2. We test the hypothesis: $H_0 : \mu_x = \mu_y$ using the test statistic $t = \frac{\bar{X} - \bar{Y}}{s_{\bar{X} - \bar{Y}}}$ and calculate the p-value for the statistic. The p-value indicates the likelihood that an observed difference this large or greater would occur by chance if the two distributions have the same mean.

To compare the failure statistics, we use the same process but instead use $\hat{X}_1 ... \hat{X}_n$ and $\hat{Y}_1 ... \hat{Y}_n$ where $\hat{X}_i = 1$ if $X_i = 1$ and 0 otherwise.

Table 3.4.3 shows the results of this comparison for each host. It shows the empirical mean and standard deviation of loss and failure probabilities and for each type of measurement for each host as well as the p-value for the two hypotheses. Figures 3.5 and 3.6 also show a graphical comparison of the packet loss and failure statistics, respectively. We see from the table that there is little doubt that the two methods measure different loss rates. If the hypothesis were true, the observed differences between the two methods would occur by chance only from .1% to 8% of the time, excluding the host `rto.rbn.com`.

There are several reasons why the two types of measurements are different. First of all, the RealMedia application implements a limited degree of reliability, so packets that are initially lost can be sometimes be retransmitted in time to be used by the application. If a ICMP packet is lost, there is no recovery mechanism. In addition, ICMP traffic is often handled differently than application-level traffic both by end hosts in the network as well as interior routers in the network. Firewalls near end hosts often block ICMP traffic from

Host	Network-level Loss Ratio $(\bar{X}, S_{\bar{X}})$ Failure Ratio $(\hat{\bar{X}}, S_{\hat{\bar{X}}})$ (In Percent)	Application-level Loss Ratio $(\bar{Y}, S_{\bar{Y}})$ Failure Ratio $(\bar{Y}, S_{\bar{Y}})$ (In Percent)	p-value for hypothesis Loss Ratio $\mu_X = \mu_Y$ Failure Ratio $\mu_{\hat{X}} = \mu_{\hat{Y}}$ (In Percent)
audioraarc004. audionet.com	(3.42,6.10) (4.26,20.40)	(0.287,0.766) (4.26,20.40)	.093 100
www.necnews.com	(3.58,11.57) (4.26,20.40)	(0.0269,0.101) (6.38,24.71)	4.47 65.0
g2-cw-wae03-rbn.com	(2.27,6.31) (2.13,14.59)	(0.11,0.23) (4.26,20.40)	2.41 56.22
real.cnn.com	N/A (100,0)	(0.0042,0.0165) (8.51,28.21)	N/A 0
rto.rbn.com	(3.36,13.10) (6.38,24.71)	(9.61,61.51) (12.77,33.73)	51.25 29.8
rv.foxnews.com	(6.16,10.87) (8.51,28.21)	(2.71,6.21) (14.89,35.99)	8.2 34.1
www.itn.co.uk	(2.72,7.71) (6.38,24.71)	(0.0087,0.058) (6.38,24.71)	2.21 100
www2. cummingsvideo.com	(1.72,2.38) (4.26,20.40)	(0.56,1.63) (8.51,28.21)	.939 40.41

Table 3.1: Comparison of Application-level and Network-level loss statistics

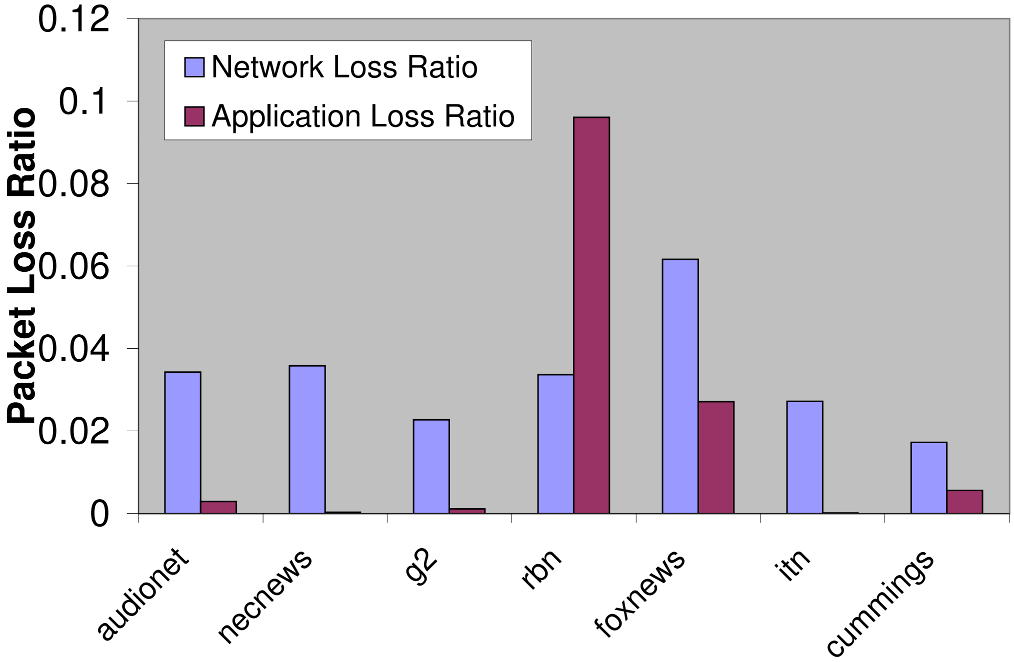

Figure 3.5: Graphical comparison of network and application level loss ratios.

entering an administrative domain, and routers sometimes block or delay transit ICMP traffic to prevent denial of service attacks such as Smurf [20] attacks. A Smurf attack is an IP-level denial of service attack where a sender sends forged ICMP echo request packets with the broadcast address of a distant subnet as the source address. Each forged ICMP packet results in a flood of traffic at the distant subnet. We see this ICMP blocking effect in the case of `real.cnn.com`, where ping always reported a loss ratio of 1.

It is less clear that the two methods measure different failure probabilities. The likelihood that the observed differences are due to chance instead of due to actual differences in measurement range from 30% to 100%, mostly due to the relatively large standard deviation in the failure statistics. The lack of a difference is not very surprising, however. The ping experiments still had to perform a host name lookup to determine the IP address, and if the distant site was unreachable, the lookup would fail and as a result the ping experiment would also fail.

In summary, these experiments show that there can sometimes be significant differences between the network statistics measured by applications and the statistics measured by network-level tools, and application-level measurements should be used whenever possible to drive application-level decisions.

In the next section, we describe the challenges that arise from the use of shared, passive measurements.

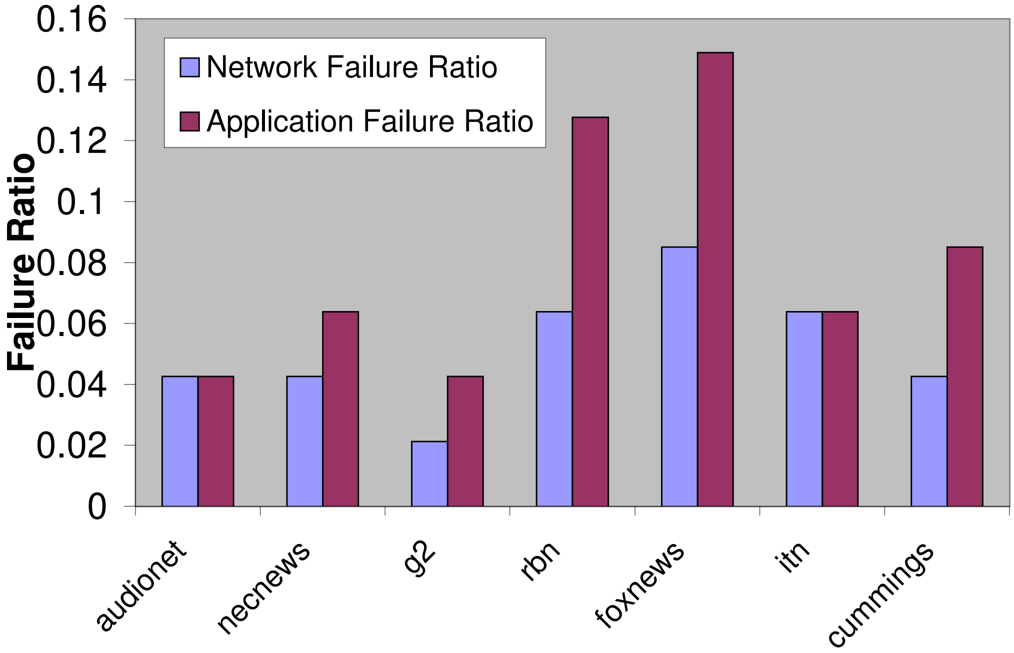

Figure 3.6: Graphical comparison of network and application level failure ratios.

3.5 Challenges of Shared, Passive Measurements

In this section, we describe the problems of using shared, passive, measurements instead of isolated, active measurements and show that these problems do not limit their utility in measuring network performance.

Using shared, passive, application specific measurements provide significant challenges. For example, by sharing information across hosts in a domain, SPAND assumes that all hosts observe similar application performance. This assumption may not always be true. Modem users usually observe much lower performance to distant hosts than local area network users in the same domain. In addition, hosts may observe significantly different application performance due to configuration, operating system or application differences. If these hosts are incorrectly clustered together, SPAND may provide misleading performance estimates. In addition, since SPAND relies on passive measurements, it may have no information or very old information about the performance of an application.

3.5.1 Measurement Noise and How it Affects Application-level Decisions

The basic question that arises from these challenges is whether a shared, passive system can provide *accurate* performance predictions. All of the statistics measured in SPAND have a certain amount of *noise* associated with them. This can be divided into several categories:

- *Network noise* is inherent in the state of the network. If a single client made back-to-back measurements of the state of the network, it would see some degree of variability in the statistics they measured.

- *Sharing noise* results from inappropriate sharing of network measurements between hosts and applications. This noise could be due to clients who have different connectivity to a distant host, clients with different network stack implementations, clients running slightly different versions of the same application, or other differences between clients.

- *Temporal noise* comes from the use of out-of-date network measurements that no longer reflect the true state of the network. For example, a measurement made at 3:00 AM on a Saturday most likely does not reflect the state of the network at 4:00 PM on a Wednesday.

The total amount of noise from network, sharing, and temporal sources affects the standard deviation of the statistics we measure and, as a result, affects the granularity of the application-level decisions that can be made. For example, consider a streaming media application that attempts to choose between 2 sources at bitrates of 300 KBit/sec and 100 KBit/sec. If our system reports an available bandwidth statistic of 150 $\pm$ 50 KBits/sec, then the application can be confident that the network cannot support the 300 KBit/sec stream. The 100 KBit/sec stream is the more appropriate. However, if our system reports an available bandwidth statistic of 150 $\pm$ 200 KBits/sec, then the application cannot confidently state that choosing one representation over another will lead to better application-level performance.

SPAND's use of shared, passive measurements assumes that sharing and temporal noise are minor factors as compared to network noise. To verify these assumptions, we performed some experiments designed to measure the degree to which network, sharing, and temporal noise affects the variation of actual network performance measurements. These experiments are not comprehensive, and the amount of noise for each category will depend on the population of clients and the state of the network between those clients and the hosts they visit. There will always be situations where one type of noise dominates. The attempt here is to give a rough indication of how much noise is likely to come from network, sharing, and temporal sources.

3.5.2 Case Study: One Client-Server Pair

In our first experiment, to understand more closely the dynamics of network characteristics to a distant host, we performed a controlled set of network measurements between a single web client at UC Berkeley and a single web server at IBM Watson in New York. Although focusing on a single client and server is clearly not representative of the variety of connectivity and access patterns that exist between Internet hosts, it allows us to focus on the contribution of Network and Temporal noise to the variance in network characteristics that could occur between a pair of well-connected Internet hosts separated by a large number of Internet hops. In the next section, we present results from a larger number of clients that are less in depth but are more representative of actual network performance.

In the first experiment, for a 5 hour daytime period (from 9AM PDT to 2:00 PM PDT), a web client at UC Berkeley periodically downloaded an image object from a

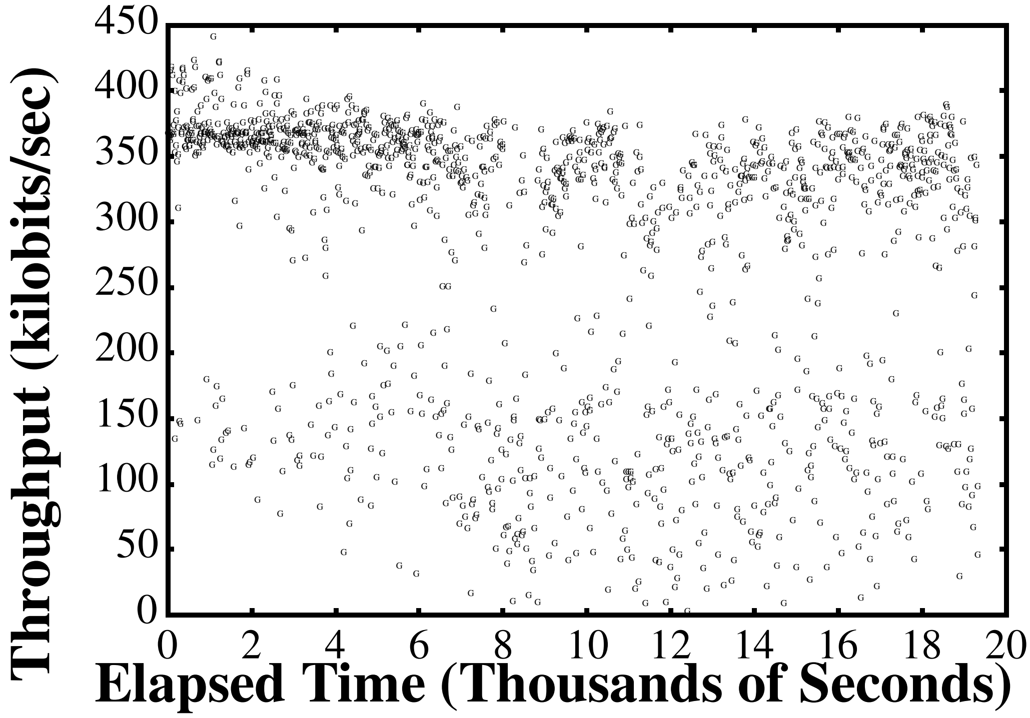

Figure 3.7: Scatter plot of throughput from IBM to UC Berkeley over a 5 hour period.

web server running at IBM Watson. For each individual transfer, we recorded the time of the transfer as well as the throughput (number of bytes divided by elapsed time) for the transfer. We then examined the dynamics of the throughput measurements over the time period.

Figure 3.7 shows the raw throughput measurements as a function of time over the 5 hour period. We see that in the first 30 minutes of the trace, one group of measurements is clustered around 350 kilobits/sec (presumably the available bandwidth on the path between UC Berkeley and IBM). A smaller group of measurements has lower throughputs, at 200 kilobits/sec and lower. This second group of connections presumably experiences one or more packet losses, leading to a reduced throughput.

This clustering is more clearly shown in Figure 3.8, which plots the cumulative distribution function (CDF) of throughputs during the first 30 minutes of the trace. One large mode is centered around 300 KBits/sec, with a smaller, more dispersed mode around 200 KBits/sec.

As the day progresses, two things change. The available bandwidth decreases over the day, and a larger fraction of transfers experience one or more packet losses, leading to a greater variance in throughput.

This effect is shown in Figure 3.9, the cumulative distribution function of throughputs for a 30 minute period in the afternoon. More samples are clustered around lower throughput values and there is more variation in the available bandwidth. However, there is still a noticeable separation between the two groups of throughput measurements. This

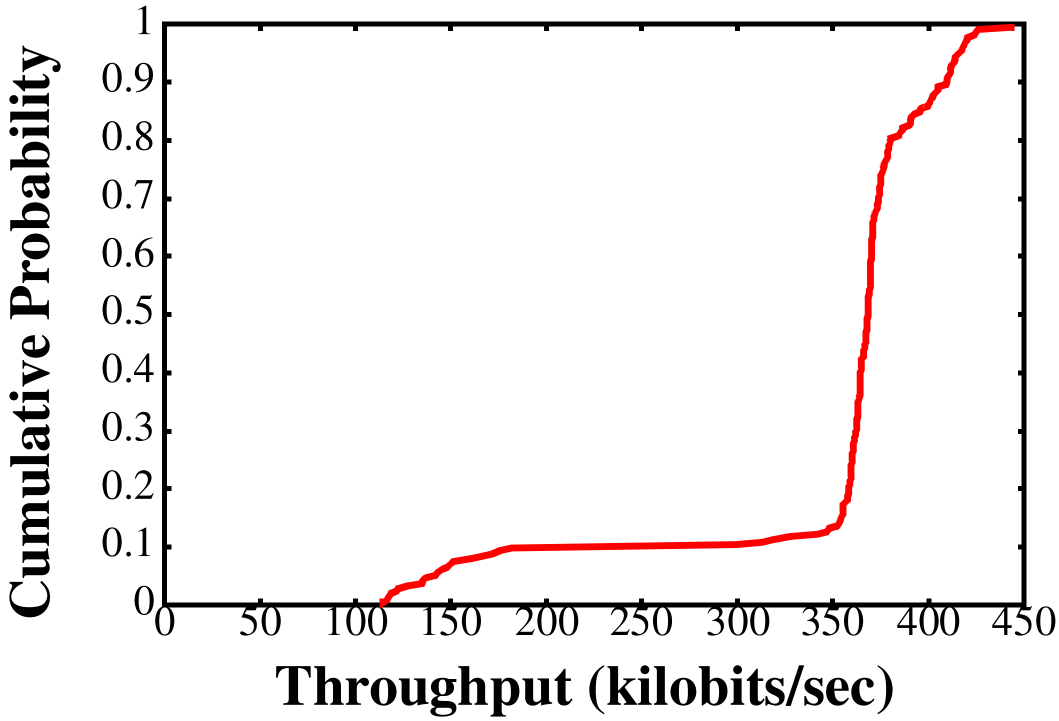

Figure 3.8: CDF of throughput from IBM to UC Berkeley: initial 30 minutes.

distribution of performance suggests that although the distribution of throughputs changes as the day progresses, a system like SPAND could still provide meaningful performance predictions. Even when aggregating all the different performance measurements for the entire 5 hour period, approximately 65% of the throughput samples are within a factor of 2 of the median throughput.

In summary, for this client-server pair, we see that network noise can lead to variations of as much as a factor of 2 in measured performance. This limits the granularity of application-level decisions to rather coarse grained (but in many cases, still useful) decisions.

3.5.3 Using Trace Analysis to Measure Noise

Although the above experiment allowed us to focus on the dynamics of a single client and server, it is not necessarily representative of the typical performance for Internet hosts. To give a rough indication of the amount of network noise that can occur for a larger group of clients, we analyzed a packet trace collected at the gateway between the UC Berkeley Electrical Engineering and Computer Science Department and the rest of the Internet. When post-processing the trace, we only considered TCP connections that transferred at least 1 kilobyte and only considered client-server pairs that had at least 30 connections between them. This resulted in a collection of approximately 700,000 TCP connections with a TCP sender in the Internet and a local TCP receiver at UC Berkeley starting on June 14, 1999 at 12 midnight and ending on June 14, 1999 at 4:00 PM.

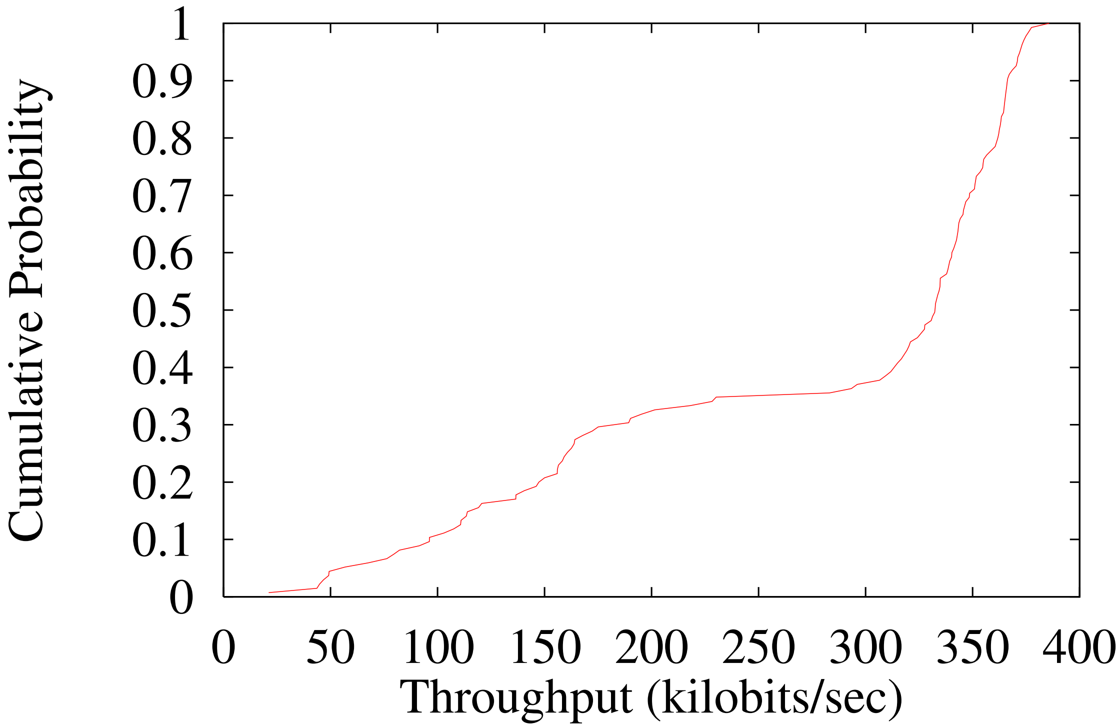

Figure 3.9: CDF of throughput from IBM to UC Berkeley: Afternoon 30 minute period.

We measured the duration of each TCP connection as well as the number of bytes transferred in the connection and divided these two measurements to obtain a generic "available bandwidth" network performance metric for each connection.

To measure the variation in performance, we measured the fraction of time t the performance for a given (client, server) pair was more than a factor F away from the median performance for that pair. If t is close to 1, many measurements are more than a factor of F away from the median performance, indicating a great deal of variation. If t is close to 0, most measurements are less than a factor of F away from median performance, indicating relatively stable performance. This unitless quantity allows us to compare variation in performance for client-server pairs that may potentially have different available bandwidth measurements.

To quantify network, sharing, and temporal noise, we performed the following analyses with the trace. We first measured the amount of network noise in the trace by examining the variation in performance for individual (Berkeley client, server) pairs over a relatively small period of 30 minutes from 1:00 PM to 1:30 PM. We then assumed that the clients would share performance information and again measured the variation in performance, this time for (group of Berkeley clients, server) pairs over the 30 minute period. The difference in variation between these two cases allows us to measure the sharing noise introduced by sharing performance information between clients. We then repeated the experiment again by considering longer and longer time scales starting at 12 AM and continuing until 4:00 PM. Examining the variation for longer time scales allows us to measure

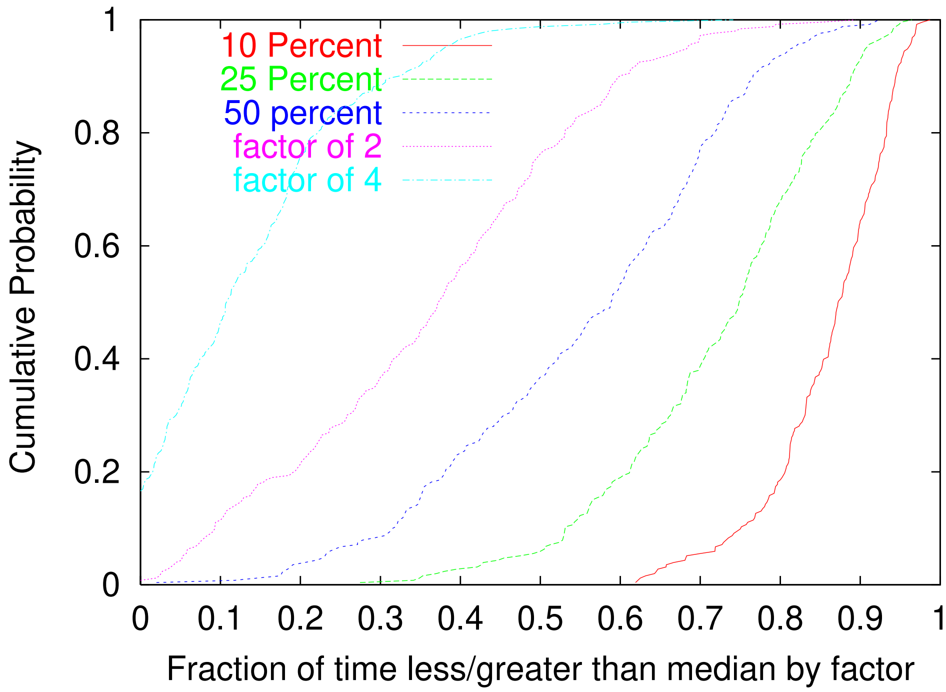

Figure 3.10: Quantifying network noise. Figure shows likelihood of being more F away from median performance for a given client-server pair.

the amount of temporal noise introduced by using potentially stale past performance information to indicate current performance.

Contribution of Network Noise

Figure 3.10 shows the results of the first analysis by showing aggregate measurements in variation for a variety of factors F. The x axis shows the fraction of time that the actual performance for a given (client, server) pair was more than a factor F away from the median performance observed for that pair of hosts, and the y axis shows cumulative probability. There are 5 curves on the graph, one for factors of (from left to right): 4, 2, 1.5, 1.25, and 1.1, respectively. For example, from examining the rightmost curve (10%), we see that most performance reports are more than 10% away from the median performance. In contrast, for the leftmost curve, on average, only 10% of performance reports are more than a factor of four away from the median performance. As expected, as the factor F increases, the acceptable region around the median measurement also increases, and the CDF curve shifts to the left.

We see from the figure that the available bandwidth for individual connections is almost always more than 10% away from the median bandwidth for a client-server pair, between 80% and 90% of the time. Even for large factors of F such as 400%, a measurable number of throughput samples are outside the acceptable region, as much as 40%.

This reinforces the results of Section 3.5.2, showing that network noise limits the

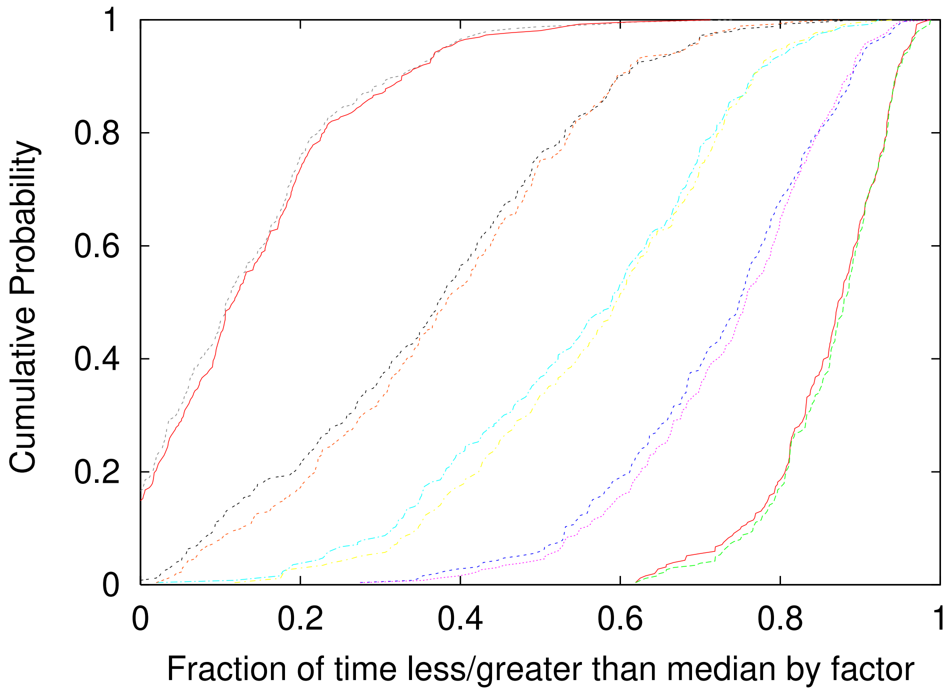

Figure 3.11: Quantifying sharing noise. Figure shows likelihood of being more than F away from median performance for a given (group of clients,server) pair.

granularity of application-level adaptations to relatively coarse-grained decisions that differentiate between order of magnitude changes in performance.

Contribution of Sharing Noise

Figure 3.11 shows the results of the second analysis. In addition to the curves in Figure 3.10, the figure shows the degree of variation when local clients share performance information and comparisons are made on (group of clients, server) pairs instead of (client, server) pairs. The order of the curves from left to right is the same as in Figure 3.10.

We see that for each factor F, there is only a small increase in variation (a shift in the curve to the right) from sharing performance information between clients. This means that sharing noise is relatively unimportant when compared to network noise, and the benefits of sharing (increased availability of network performance information) outweigh the additional variation in performance.

Contribution of Temporal Noise

Figure 3.12 shows the results of the third analysis. We chose $F = 2$, allowed clients to share performance information, and then examined increasing time windows from 30 minutes before 4:00 PM (3:30-4:00 PM) up to 16 hours before 4:00 PM (12:00 AM-4:00 PM). The figure does not include curves for start times before 6:00 AM to aid in legibility.

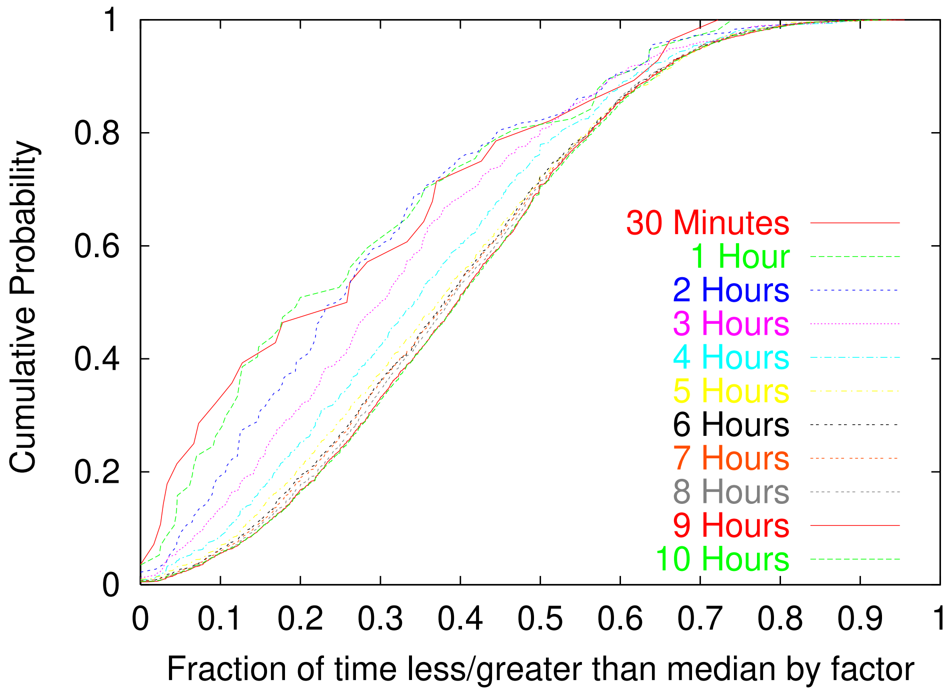

Figure 3.12: Quantifying temporal noise. Figure shows likelihood of being more than a factor of 2 away from median performance for a given (group of clients, server) pair for increasing time scales.

We see that there is little difference between the 1 hour and 30 minute curves, indicating that time windows of up to an hour have little effect on variation. For time windows between 2 and 6 hours, there is a measurable increase in variation, indicating that past measurements from hours ago are no longer completely representative of actual performance. For time periods greater than 6 hours, however, there is little increase in variation in performance, indicating that there is a limit in the effect of temporal noise on overall variation.

From these measurements, we can see that measurements from as long as one hour ago are likely not to affect the variation in performance imposed by the network. Over longer time scales, however, temporal noise can have an effect (but not as much as the effect of network noise) on the granularity of application-level decisions.

In the next section, we discuss the types of applications that can best take advantage of SPAND.

3.6 Applications Best Suited for SPAND

Although a goal of our architecture is to be extensible and support a wide variety of applications, it is important to discuss applications that would benefit more from using a framework other than SPAND. We first describe applications where SPAND works well.

SPAND works best for applications that meet the following criteria:

- Applications that do not initially have enough network performance information to make a good adaptation decision.
- Applications that, once they make an adaptation decision, can not easily re-adapt before the end of a communication.
- Short-lived applications that do not naturally have a way to disseminate performance information among each other in a distributed fashion.

These criteria are somewhat abstract and are more easily illustrated using the context of the following three example applications:

- WWW browsing, where clients perform adaptation by selecting alternate representations of WWW pages,
- Bulk transfer applications, where a client selects one of a number of mirror sites to retrieve a single large object, and
- Streaming multimedia applications such as RLM [50], where clients select the number of layers they wish to receive from a multicast source.

The WWW browsing application meets all of these. A client typically contacts one of a large number of web servers and downloads a small number of its pages. This makes it unlikely that the client will have enough information to select an appropriate representation to download. There is limited benefit in choosing an alternate representation for subsequent pages if the original choice was inappropriate.

The bulk transfer application meets some, but not all, of these criteria. It is difficult for a single client to determine which mirror will result in the best performance. The only way to do this would be to download the content from all of them simultaneously. It may or may not be easy to re-adapt once a decision has been made, however, depending on whether or not mechanisms exist for retrieving sub-ranges of documents.

For both of the above applications, it is difficult to share performance information in a distributed manner, as the client applications may be short-lived and cannot participate in a distributed sharing algorithm.

In contrast, however, the streaming multimedia application meets few of these criteria. Although a client may not have much information about the initial number of layers to receive, it can easily re-evaluate this decision based on network conditions, adding or dropping layers as necessary. Because sessions are relatively long-lived, it is easy for members of the group to disseminate performance information in a distributed manner, by using a separate multicast group to share the results of join experiments.

3.6.1 Application-level Adaptation vs. Network Diagnosis

Many previous efforts have focused on the development of network measurement tools to measure performance metrics such as packet loss rate, latency, and available bandwidth. These tools can be used to measure the quality of Internet Service Providers (ISPs)

[55] [39] [70] [47] [66] [45], to identify and diagnose network problems [41] [78] [60] [25], and debug and improve network protocols and implementations [11] [10] [65] [19]. This list is by no means complete–a more comprehensive listing of these tools can be found at the CAIDA web site [17].

Although SPAND and network diagnosis tools both require wide-area network performance measurements, the goals of our system are orthogonal to theirs. SPAND's primary focus is to support applications that must adapt to the state of the network, including any flaws, limitations, or problems that may arise, while the goal of network diagnosis systems is to identify and correct these problems. Both types of network measurement tools are necessary to have a complete solution to managing the problems that may arise in today's distributely-managed heterogeneous Internet.

3.7 Summary

In this chapter, we have discussed the assumptions and methodology that lay the foundation for our thesis. We discussed the network model behind our work: *domains* of high quality connectivity connected by an internetwork with unknown properties. We then described the three major design decisions in SPAND: the use of *Shared*, *Passive*, *Application Specific* measurements.

We compared these design decisions with alternate design choices both qualitatively and quantitatively, showing that in many cases, shared, passive, application specific measurements have significant advantages over isolated, active, network-level measurements. To quantify the benefits of shared measurements, we analyzed a large client-side packet trace and showed that the use of shared measurements increases the likelihood of relevant network performance information from 50% to 80%. To quantify the benefits of passive measurements, we present one example where the use of passive measurements doubles the throughput of a mirrored web server system. To quantify the benefits of application-specific measurements, we measured one network performance statistic (packet loss rate) both at the network and application level and showed that application-level behavior leads to significant differences–in some cases, as much as 400%–between application and network measurements of the same statistic.

We also identified the challenges that result from the use of shared, passive, application specific measurements, in particular, the amount of *noise* they add to measurements of network performance. Quantitative measurements of network, sharing, and temporal noise show that network noise (the inherent variability in the network over short time scales) is the most significant contributor to variance in network measurements and limits the granularity of application-level decisions to order of magnitude differences. In contrast, sharing and temporal noise make relatively small (i.e., less than 25%) contributions to overall variance.

We also discussed the types of applications that are likely to benefit most from SPAND, showing that the best candidates for our system are applications that have limited mechanisms for adaptation but lack agility, the ability to adapt quickly, and policies that effectively use adaptation mechanisms to improve application-level performance.

In the next chapter, we build on these assumptions and design decisions and present the core SPAND architecture and implementation. The architecture builds on the concept of domains of connectivity and the design principle of shared measurements by introducing the concept of a centralized per-domain *Performance Server* that acts as the repository for network performance information. The architecture builds on the concept of application-specific measurements by introducing an extensible messaging model between SPAND Clients and the performance server. The architecture builds on the concept of passive measurements by defining a *Packet Capture Host* who makes network performance reports on behalf of unmodified clients by reconstructing application-level behavior from individual packets. We also present measurements of the core SPAND architecture that quantify the benefits of our design decisions.

Chapter 4

The SPAND Architecture

In this chapter, we build upon the design choices described in Chapter 3 and present the core SPAND Architecture. We describe the components of the architecture and how they communicate with each other. We also describe how the architecture is *extensible*, facilitating the development of new applications. We then describe how the architecture is realized to measure performance for two types of data transport: a generic bulk transfer data transport and a HTTP specific data transport that more closely matches the performance observed by actual web clients.

Finally, we present application-independent performance results that evaluate how well SPAND performs in returning *accurate* and *relevant* measurements of network performance for a group of clients.

4.1 Components of SPAND

In Chapter 3, we presented the concept of sharing domains consisting of clients of equivalent connectivity using applications of the same application class. In this section, we examine in more detail the makeup of a domain and describe the components of SPAND, how they work, and how they communicate with each other.

Figure 4.1 shows a diagram of the components of SPAND. SPAND is comprised of *Client Applications*, *Performance Servers*, and *Packet Capture Hosts*.

4.1.1 Client Applications

Client applications measure the state of the network while communicating with distant network hosts. When this communication is finished, clients create *Performance Reports* that summarize the observed performance to distant hosts and send these reports to performance servers. These reports are application-specific, meaning that the statistics they capture are those in which the application is most interested. For example, instead of reporting available bandwidth or round trip time, a web client reports the amount of time taken to retrieve a web object, even if that object spans multiple TCP connections (as in the case of HTML pages with associated in-line images).

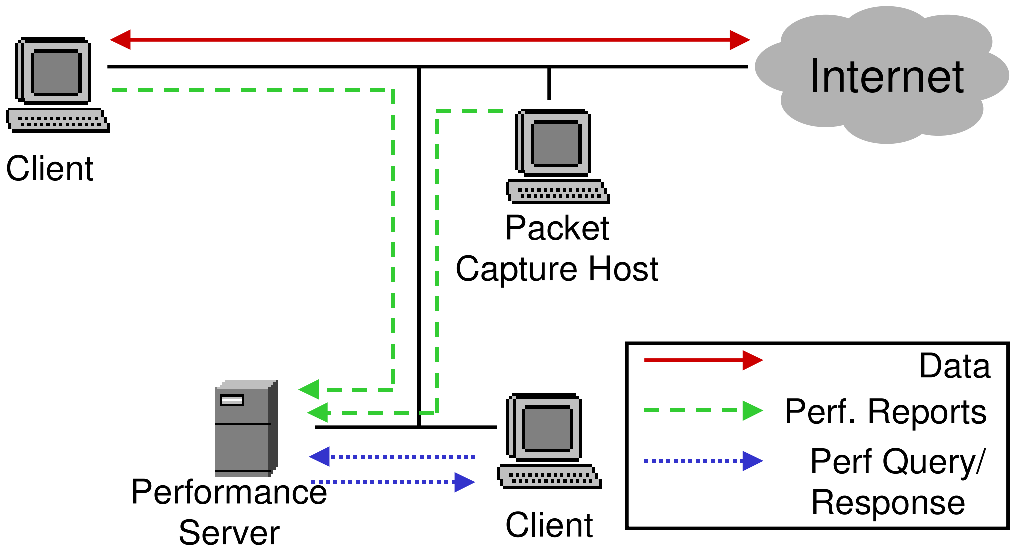

Figure 4.1: Components of SPAND.

4.1.2 Packet Capture Host

Our system works best when a large population of client applications are modified to generate performance reports, because it allows these clients to benefit from the shared experiences of a large pool of users. However, it may be difficult to immediately upgrade all client applications to versions that generate performance reports. To quickly capture performance from a large number of end clients, a *Packet Capture Host* can be deployed that uses a tool such as BPF [49] to observe all packet transferred to and from these clients. The packet capture host determines the network performance from the packet trace and sends reports to the performance server on behalf of the clients. This allows a large number of performance reports to be collected while end clients are slowly upgraded. The weakness of this approach is that a number of heuristics must be employed to recreate application-level information that is available at the end client but not at the packet capture host. Sections 4.3.1 and 4.3.2 describe these heuristics in more detail.

4.1.3 Performance Server

After creating performance reports, Client Applications and Packet Capture Hosts send them to *Performance Servers*, who store them in a centralized repository. Client applications also send *Performance Requests* to a performance server to ask for the observed performance to a distant host. The performance server responds with a *Performance Response* that indicates the typical performance seen by clients. Again, the format of all of these messages is completely application-specific.

The actual functions used to add performance reports to the repository and respond to performance requests are provided by the Application and named in the messages sent to the performance server. This is described in more detail in Section 4.2.

The performance server also handles the lower-level details of managing client

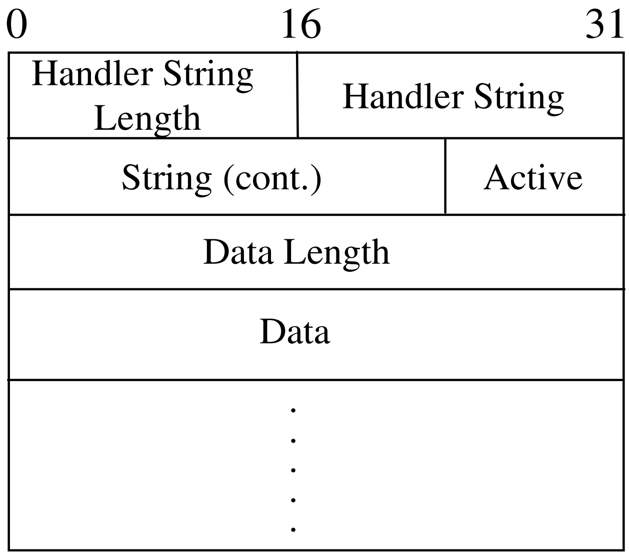

Figure 4.2: SPAND Message Format

sockets, managing synchronized access to the repository, spawning and reaping threads, and logging.

4.2 Messages Between SPAND Components

All messages between the components of our system use a format similar to Active Messages [79]. A SPAND message contains a *Handler String*, an active flag, a data length, and a message-specific payload, as shown in Figure 4.2.

If the active flag is 1, the Handler String indicates the name of the function that should be used to process the message. If the active flag is 0, then the Handler String indicates the type of the message. Performance reports and performance requests from Applications and packet capture hosts are typically active messages, whereas performance responses from the performance server are typically not active.

To incorporate a new application into SPAND, an application writer implements active message handlers and integrates them into the performance server. The performance server is currently written in Java, so this process simply means placing the Java class files at a location where the Java Virtual Machine's class loader can find them. This code may or not be "mobile": we discuss the implications of this in Section 4.2.3.

The use of active messages provides a uniform interface between applications and the performance server. Applications send active messages to the performance server to add performance reports to the repository and to perform performance lookups. Application writers do not have to handle the details of concurrent access to the repository or lower-level socket communications. The performance server simply receives messages and invokes their handlers without knowing what the handler does. This allows for maximum extensibility and ease of use in adding new application types to SPAND.

We chose to write the performance server in Java because of Java's ability to dynamically loading Java class files from a string, its language-level thread abstractions, and its inherent garbage collection facilities. However, the performance server could be written in any language: the use of Java simply made this process easier.

4.2.1 Basic Message Types

Although applications can create arbitrary handlers for SPAND messages, most applications will provide at least the following basic handlers:

- *AddPerf*: Add the following performance report to the repository. The data portion of the message contains the Report.
- *GetPerf*: Process the following performance request. The data portion of the message contains the performance request.
- *GetSumm*: Return a summary of the repository for this application type. Usually, this includes the number of reports collected for each key (e.g. address, URL, etc.) in the repository.
- *GetRaw*: Return the raw performance reports for the specified key (e.g. address, URL, etc.). The data portion of the message contains the key.

By implementing these handlers, applications can provide basic report addition and processing functionality.

4.2.2 Environment for Active Message Handlers

All active message handlers run in an uniform execution environment. The generic format of a message handler is the following:

```
Msg execute(Msg m, Hashtable reportDB);
```

A handler takes two arguments: the original message sent to the performance server and the repository of performance reports.

The report repository is organized as a hash table of repositories. The key in the top-level hash table is the application type as a string, and the value is an application-specific repository. When a handler is executed, the handler accesses the application-specific repository, unmarshals the message and processes the data portion of the message.

When a handler is done executing, it returns a `Msg` object. The performance server invokes the object's marshal method to serialize the object and sends it back to the application who sent the message.

4.2.3 Extensible SPAND vs. Active SPAND

Although SPAND is extensible, it is not a goal of our system that arbitrary clients be able to upload active message handlers in real time, like the model of Active Messages [75]. Although nothing in our architecture prevents the dynamic uploading of code, we do not address the resource sharing, safety, and security issues that are necessary for a complete implementation of mobile code.

4.2.4 Averaging Algorithms to Obtain Typical Performance

GetPerf message handlers take as input the repository of reports for a particular distant host and must determine the "typical" performance seen by clients to that host. This process incorporates two parts:

- Identifying the *relevant* performance reports to use when calculating typical performance.
- Choosing the *averaging algorithm* to perform on the relevant performance reports to obtain typical performance.

Because our system is extensible, applications are free to make their own choices in averaging algorithms. In this section, we describe the choices we made in devising averaging algorithms.

To identify the relevant set of performance reports, we use the results learned in Section 3.5.3. When a performance request is received at time t, we considered all performance reports in the past from t for a duration D. A typical value used for D is 3 hours. If the number of reports in the time period $(t - D, t)$ is more than a threshold value H, we considered the H most recent performance reports when generating a performance response. If the number of performance reports collected in the time period is less than H, we use all Reports falling in the time period when generating a performance response. A typical value for H is 20. 20 was empirically chosen as the smallest number of performance reports that led to an accurate performance response. This algorithm allows us to use the most recent performance reports for frequently contacted distant hosts and provides a cut-off for less frequently visited hosts, reducing the effect of temporal noise.

Once we identified the relevant set of performance reports, we used the value of the median performance report to represent typical performance. For example, for TCP performance, we calculate an "available bandwidth" metric for each performance report by dividing the number of bytes transferred in the report by the duration of the performance report. We then take the median of these measurements and use it to report typical performance. Using the median has the advantage that it is not affected by outlier values and does not make assumptions about the underlying distribution of the statistic measured.

4.3 Realizations of the SPAND Architecture

In this section, we describe realizations of the SPAND architecture for two specific application classes. The first realization measures performance for a generic bulk transfer application that uses TCP as the data transport. The second realization measures performance for a HTTP-specific data transport. The emphasis in this section is not on which metrics we chose for each type of data transport, but on *how* we measured these metrics in our system.

4.3.1 Bulk Transfer Application

The goal of the bulk transfer realization is to measure performance for an application that uses a single TCP connection to send data from one host to another. We assume that the transmission of data is limited only by the speed of the network and not the speed of the sender or receiver of data. Our realization measures the following characteristics of the data transfer: the *available bandwidth* for the connection, the *round trip time* for the connection, and the *time to completion* for a specific transfer size of B bytes.

Although this realization does not measure true end-to-end application level performance, our goal is that the statistics it measures be useful to a wide variety of applications that use TCP connections for bulk data transport.

Available Bandwidth

The available bandwidth metric is designed to estimate the long-term bandwidth that a one-way transfer using a single TCP connection will receive, similar to the Bulk Transfer Capacity metric defined by the IETF IPPM Working Group [46]. We measure available bandwidth as the total length of a transfer divided by the total time of the transfer, including the initial SYN exchange but not including the FIN exchange.

Our metric differs from the one proposed by the IPPM Working Group in that it is not tied specifically to any particular reference implementation. Because we use passive measurements, the measurements may come from a number of different TCP implementations. Differences between implementations will reflect themselves as sharing noise. In Section 3.5.3, we showed that the contribution of sharing noise to overall variation is rather small.

Round Trip Time

The round trip time metric is designed to measure the round trip time between hosts in a local domain and a specific host in a distant domain. The metric we measure is smoothed RTT (SRTT), using an algorithm similar to the one used by TCP [81]. Individual round trip time samples are measured by measuring the time from when a TCP packet is sent out to when the acknowledgment for that packet is received. The individual samples are then exponentially smoothed to obtain a SRTT metric.

Time to Completion

The two statistics mentioned above are useful for applications that do not know the connection transfer size in advance. For applications that do know a connection's transfer size in advance, we provide a third metric, *time to completion*. We define time to completion as the time T needed to transfer B bytes using a single TCP connection, including connection set-up time. This metric may produce a different result than simply dividing the number of bytes B by the available bandwidth metric above, especially in the case of connections with short transfer sizes where round-trip times have a significant impact on performance.

We measure the time to completion by finding the median time at which past transfers have completed sending B bytes. This process is shown in Figure 4.3. We start by combining sequence number plots for a large number of connections, resulting in a single scatter plot of sequence number vs. time. To aid in legibility, Figure 4.3 only shows those

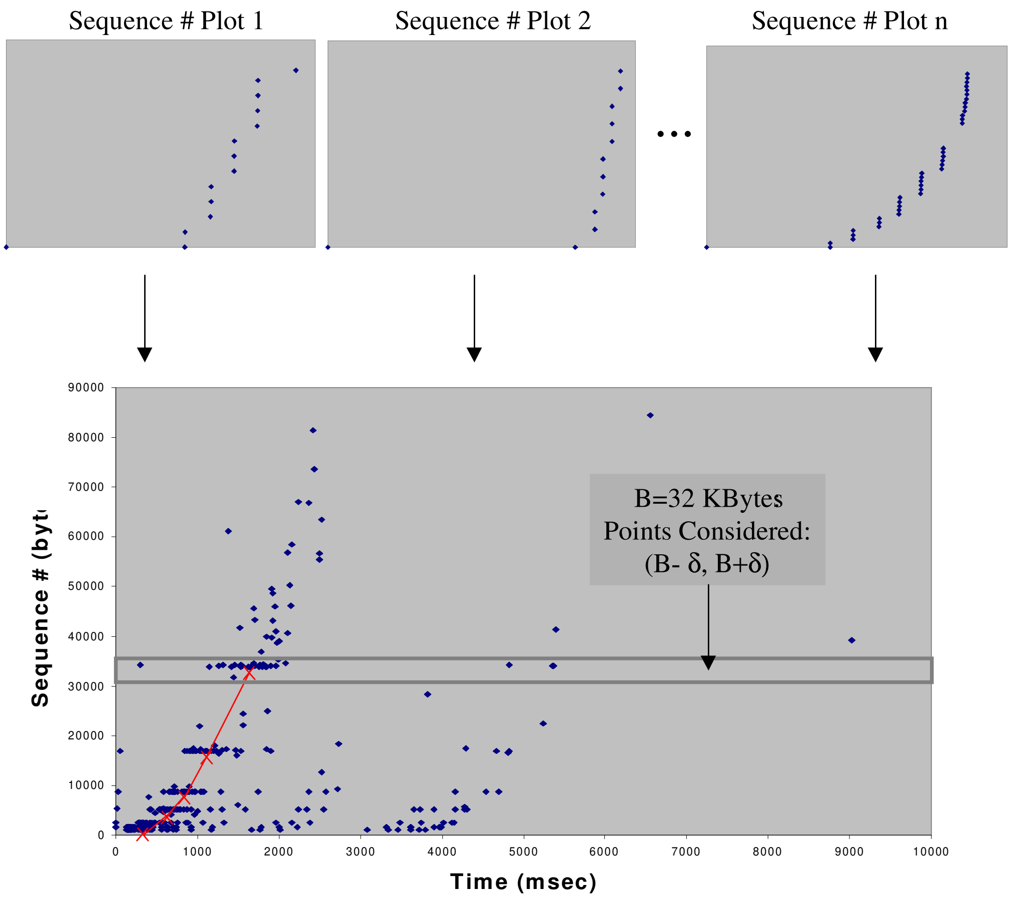

Figure 4.3: Graphical example of time-to-completion metric.

points where a the number of bytes transferred had just surpassed a power of 2 (512, 1024, 2048, ...). On this scatter plot, we consider all points slightly above and below target number of bytes B, $(B-\delta, B+\delta)$. δ is dynamically chosen as the smallest δ for which at least 20 points fall between $(B-\delta, B+\delta)$. 20 was empirically chosen as the smallest number of samples that led to an accurate median measurement. If δ is greater than 5 kilobytes, then we give up, reporting the statistic as not measurable. If δ is smaller than 5 kilobytes, then we take the median time value for all points within the region as the result T.

From looking at the graph, we see that the metric does a good job of following the "center of mass" for this set of connections. We also see that in the 64 KByte case, our algorithm gives up (i.e. reports no metric) because there are not enough data samples to make a good estimate.

It is important to mention that this metric is *non-parametric*: we do not develop an analytic model for the way a TCP connection should behave (such as the ones presented in [61] or [48]), and use formulas to predict the time to completion. We rely on the observed dynamics of actual TCP connections to determine the time to completion. By relying on actual dynamics, we do not worry about whether or not the models accurately describe how TCP connections behave. For example, the model presented in [61] does not model the slow-start or fast recovery phases of TCP, and the paper's comparison of predicted and actual performance (where performance is defined as number of packets sent during a connection) assumes transfer lengths of 100 seconds or one hour, both significantly longer than typical transfer lengths for WWW Applications. Even with these assumptions, predicted performance differs from actual performance by a minimum of 10% to a maximum of 200%, with a median of approximately 50%. By using a non-parametric model, we do not suffer from the differences in performance predicted by a model and actual performance observed by clients.

Packet Capture Host Policies

Because the packet capture host is not located at end clients, it does not have perfect information about the way in which applications use TCP connections. This can lead to inaccurate measurements of network characteristics such as bandwidth. For example, if a web browser uses persistent or keep-alive connections to make many HTTP requests over a single TCP connection, then simply measuring the observed bandwidth over the TCP connection will include the gaps between HTTP requests in the total time of the connection, leading to a reduction in reported bandwidth. To account for this effect, we modified the packet capture host to use heuristics to detect these idle periods in connections. When the packet capture host detects an idle period, it makes two reports: one for the part of the connection before the idle period, and another for the part of the connection after the idle period.

In addition, we added heuristics at the packet capture host to make accurate round trip time measurements. Because the packet capture host is located on the path between a local host and a distant host, it cannot exactly measure the round trip time as seen by the local host. Instead, it makes separate round trip time measurements for the path from the local host to the packet capture host (using the HTTP response) and for the path from the packet capture host to the distant host (using the HTTP request). This process is shown in Figure 4.4.

In the case of local web clients downloading content from distant web servers, the packet capture host uses the web request data to make RTT measurements from the packet capture host to the distant host, and the web response data to make RTT measurements from the packet capture host to the local host. The measurements for each component of the path are individually smoothed and then added together to result in a single SRTT measurement.

4.3.2 HTTP Statistics

The goal of the HTTP realization is to measure performance for applications that use HTTP (on top of TCP) for data transport. These applications differ from applications that simply use TCP for bulk data transport in the following ways:

- Web browsers use HTTP to download web pages that usually consist of an HTML document with the addition of in-line images. These objects are usually retrieved using

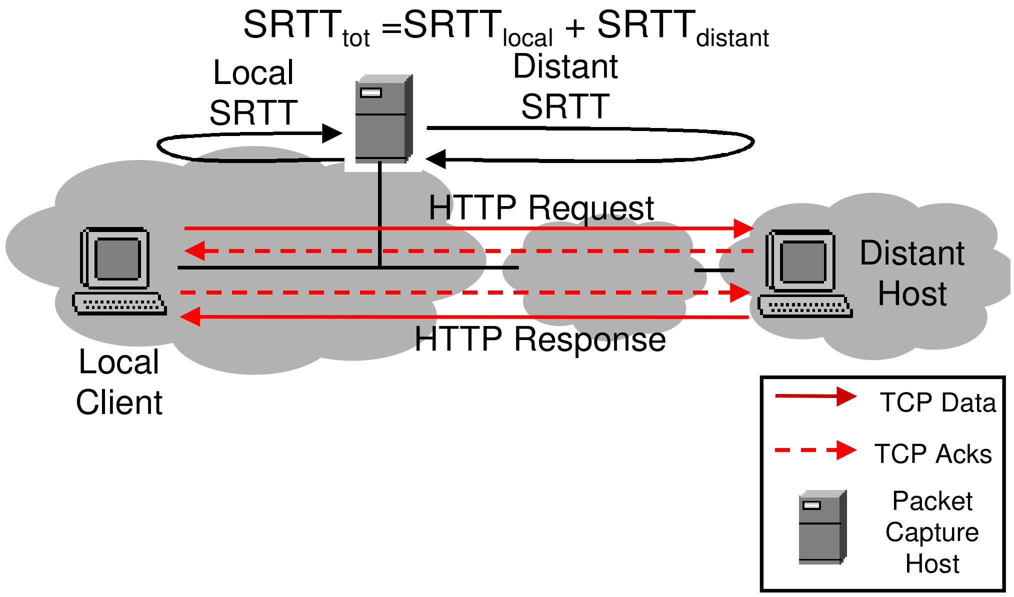

Figure 4.4: How SRTT is measured at the packet capture host.

multiple TCP connections. As a result, metrics that only report the performance of individual TCP connections are not as useful to applications as metrics that report full-page download times.

- Many web pages are produced as the output of server-side execution of Common Gateway Interface (CGI) programs. This process incorporates server-side execution time that is not effectively reported in a TCP bandwidth or round trip time metrics.

Web Object Download Time

To overcome these limitations, we measure and report *web object download time* as the primary metric for HTTP applications. We define the web object download time as starting when the application does a Domain Name System lookup for the host mentioned in an URL and ending when the application receives the last byte of content for the web object. For HTML objects, this also includes the time to download in-line images that are a part of the web page. This also includes any server execution time as the result of CGI programs.

Packet Capture Host Policies

Because the packet capture host is not implemented at web clients, it cannot exactly measure the web object time as defined above. It must reconstruct this metric by examining the HTTP headers and responses in TCP connections. This can be difficult: a single TCP connection may be used for a single HTTP transaction (for HTTP/1.0) or multiple HTTP transactions (for HTTP/1.1). In addition, multiple HTTP transactions may comprise a web object, as in the case of HTML pages with in-line images.

The packet capture host handles the first two problems by recreating the ordered byte stream from individual TCP packets and explicitly parsing the HTTP headers in the byte stream. It examines the *Content-Length:* field in HTTP Responses and measures the web object download time as starting at the start of the SYN exchange of the connection and ending when the last byte of an object has been transferred over that connection. For subsequent HTTP requests using the same connection, we start the download time using the first byte of the HTTP request instead of the beginning of the SYN exchange. Note that this means if web clients use request pipelining to initiate a batch of HTTP requests at once, the reported response time for subsequent (second, third, etc.) objects in a batch request will have longer-than-expected download times, as later responses must wait for earlier ones to be sent to the client.

In the case of HTML pages with in-line images, the packet capture host does not have perfect information about exactly which web objects comprise a single web Page. There is nothing in the HTTP protocol that explicitly links together HTML pages and in-line objects that are included in them. One alternative is to explicitly parse the HTML pages to find references to in-line images. However, this would significantly increase the amount of computation at the packet capture host and decrease the number of HTTP streams it could handle. We wanted to find a balance between accurately reporting statistics for full HTML pages and handling a large number of clients.

Fortunately, there is implicit information that can often be used as a heuristic to connect HTML Pages and in-line images. Many web clients include a `Referer` field in HTTP requests that indicate the object from which the current request was referred. The inclusion of this field is not mandatory, and some web clients such as JDK/1.1 and Pointcast do not include this header. However, the majority of HTTP requests are made by browsers that provide this field, so it is a useful heuristic that the packet capture host can use to link up HTML pages and in-line objects.

Another challenge from using the `Referer` field is that the field is a backward reference–the in-line objects point backward to the HTML pages. As a result, it is difficult to know when all the in-line objects for a HTML page have been transferred. The packet capture host handles this by using a two-pass approach. The first pass links together HTML pages and in-line objects using the `Referer` field, and the second pass makes performance reports using the information gained from the first pass. We wait long enough between the two passes to assure that most web pages have been completely transferred by the time the second pass starts.

4.4 SPAND Applications Using TCP/HTTP Metrics

Using these realizations of the SPAND architecture, we developed several adaptive applications that use network measurements to improve application-level performance. Two of these applications, SpandConneg and LookingGlass, are described in detail in Chapters 5 and 6.

We have developed some additional applications using Muffin [52], an extensible HTTP proxy that allows the addition of customized filters for filtering HTTP request, response, and content streams. In particular, we added a content filter to Muffin that inserts informative icons in HTML documents that indicate the expected download time for hyper-linked objects (e.g. red icons for slow downloads, green icons for fast downloads). The filter uses SPAND to decide which icon to place beside the hyperlink. We have also written several diagnostic tools that allow users to browse the contents of the performance server's repository and examine the distribution of the metrics measured by SPAND.

4.5 Application-independent Performance Results

In this section, we present performance results that show how well SPAND performs at providing accurate performance responses to clients. We do this in an application-independent way by using the TCP realization, which captures transport-level measurements. In Chapters 5 and 6, we show more application dependent results, showing how well SPAND works in the context of two specific applications.

There are several important metrics by which we can measure the accuracy of the SPAND system:

1. How long does it take before the system can service the bulk of performance requests?
2. In the steady-state, what percentage of performance requests does the system service?
3. How accurate are the performance predictions?

To test the performance of our system, we deployed a packet capture host at the connection between IBM Research and its Internet service provider. Hosts within IBM communicate with the Internet through the use of a SOCKSv4 firewall [72]. This firewall forces all internal hosts to use TCP and to initiate transfers (i.e. servers can not be inside the firewall). The packet capture host monitored all traffic between the SOCKS firewall at IBM Research and servers outside IBM's internal domain. The measurements we present here are from a 3 hour long weekday period. During this period, 62,781 performance reports were generated by the packet capture host for 3,008 external hosts. At the end of this period, the performance server maintained a repository of approximately 60 megabytes of performance reports.

Figure 4.5 shows the cumulative number of reports generated and hosts reported about as a function of time. We see that about 10 reports are generated per second, which results in a network overhead of approximately 5 kilobits per second. We also see that while initially a large number of reports are about a relatively small number of hosts (the upward curve and leveling off of the curve), as time progresses, a significant number of new hosts are reported about as time progresses. This indicates that the "working set" of hosts includes a set of hosts who are reported about a small number of times.

This finding is reinforced in Figure 4.6, which shows a histogram of the number of reports received for each host over the trace. We see that a large majority of hosts receive only a few reports, while a small fraction of hosts receive most of the reports. The mean number of reports received per host was 23.67 and the median number of reports received per host was 7.

To test the accuracy of the system, we had to generate a sequence of performance reports and performance requests to test the system. Since there are no applications running

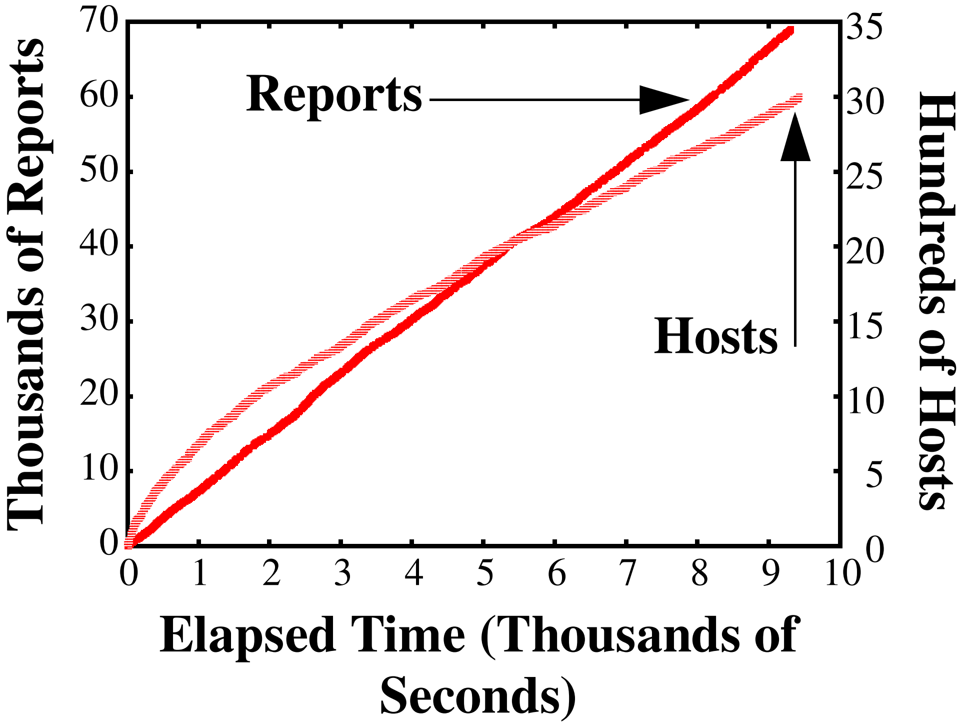

Figure 4.5: Cumulative number of reports generated and hosts reported about as a function of time.

at IBM that currently use the SPAND system, we assumed that each client host would make a single performance request to the performance server for a distant host before connecting to that host, and a single performance report to the performance server after completing a connection. In actual practice, applications using SPAND would probably request the performance for many hosts and then make a connection to only one of them.

The performance of the SPAND system on this workload is summarized in Figures 4.7 and 4.8. When a performance server is first started, it has no information about prior network performance and cannot respond to many of the requests made to it. As the server begins to receive performance reports, it is able to respond to a greater percentage of requests. Determining the exact "warmup" time before the performance server can service most requests is important. Figure 4.7 shows the probability that a performance request can be serviced by the performance server as a function of the number of reports since the "cold start" time. We say that a request can be serviced if there is at least one previously collected performance report for that host in the performance server's repository. As we can see from the graph, the performance server is able to service 70% of the requests within the first 300 reports (less than 1 minute), and the performance server reaches a steady-state service rate of 95% at around 10,000 reports (approximately 20 minutes). This indicates that when a performance server is first brought up, there is enough locality in client access patterns that it can quickly service the bulk of the performance requests sent to it.

To measure the accuracy of performance responses, for each connection we com-

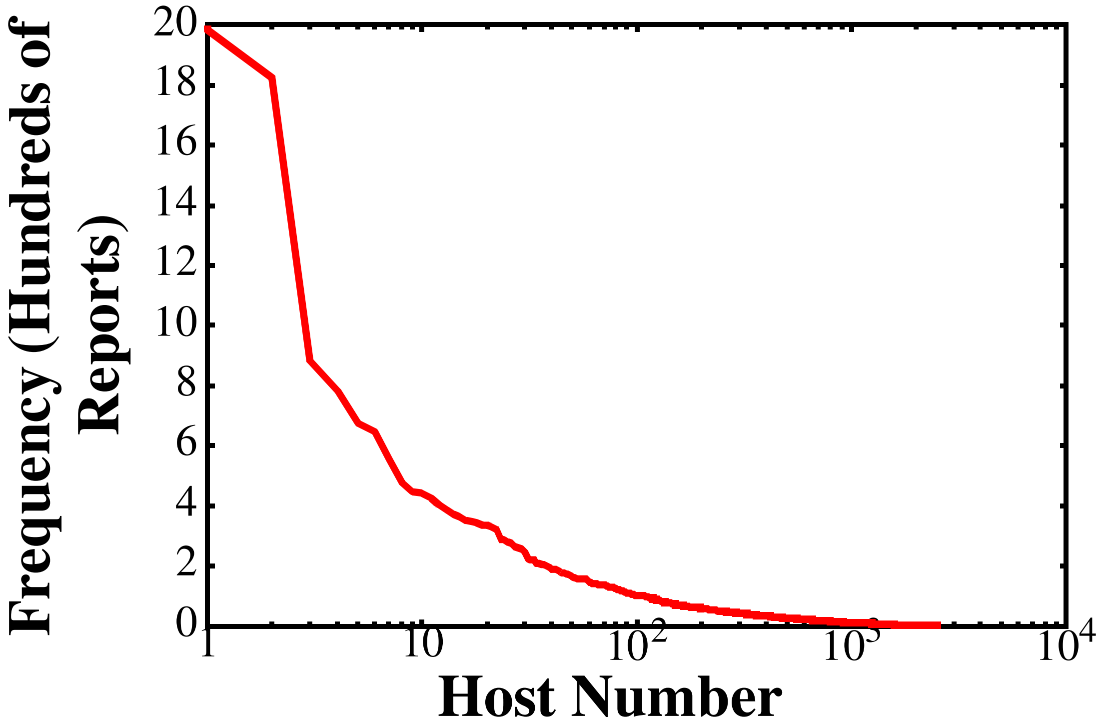

Figure 4.6: Histogram of number of performance reports received per host. The X axis is on a log scale.

puted the ratio of the throughput returned by the performance server for that connection's host with the throughput actually reported by the packet capture host for that connection. Figure 4.8 plots the cumulative distribution function of these ratios. The X axis is plotted on a log scale to equally show ratios that are less than and greater than one. There are two curves on the graph. One shows the CDF of ratios where the performance server does not use any of the heuristics described in Section 4.3.1 to eliminate idle periods in connections, and the other shows the CDF of ratios where the performance server does employ these heuristics.

Table 4.5 shows the probability that a performance response is within a factor of 2 and 4 of the actual observed throughput. We see that performance responses are often close to the actual observed throughput. Obviously, different applications will have different

System	% within a factor of 2	% within a factor of 4
Base System	59.08%	84.05%
Base System + App Heuristics	68.84%	90.18%

Table 4.1: Accuracy of Performance Responses

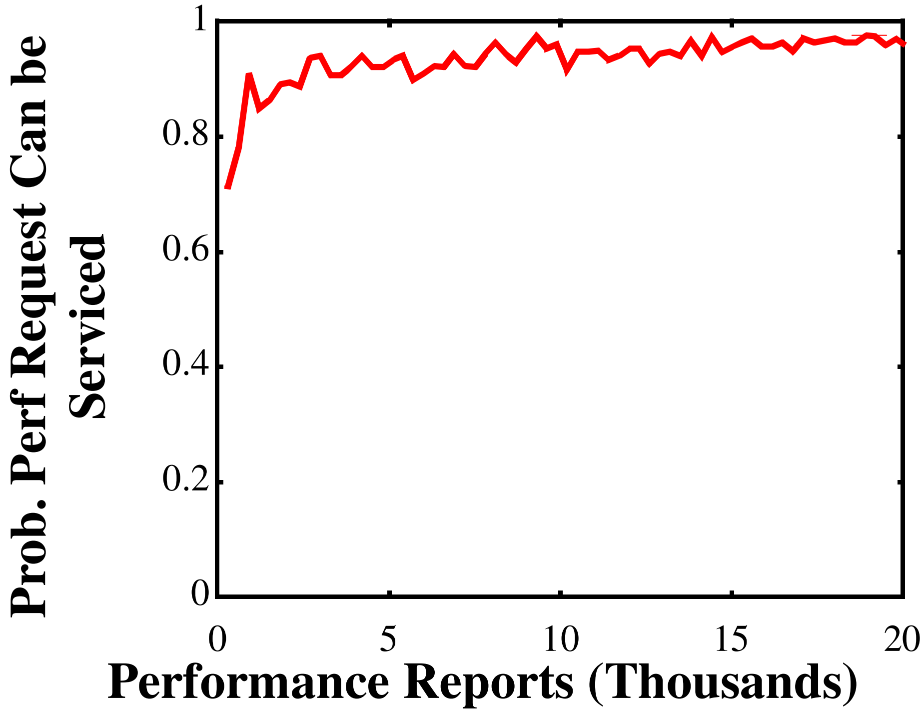

Figure 4.7: Probability that a performance request can be serviced as a function of the number of performance reports.

requirements as to the error that they can tolerate. Factors of 2 and 4 are shown only as representative data points. We see that SPAND does a good job of predicting performance, and as shown in Section 3.5.3, inherent network noise is the largest contributor to the gap between predicted and actual performance.

4.6 Taking Advantage of Daily Cycles to Improve Performance

From examining the above experiments, we see that one of the limitations of our system is that a large number of infrequently visited distant hosts only have a small number of performance reports collected for them. In some cases, this prevents us from generating accurate responses to performance queries for these hosts. Although these hosts only make a small contribution to the total number of performance requests, we wanted to investigate changes to improve the number of performance reports available for these infrequently accessed hosts.

With this in mind, we experimented with improvements to our system to increase the number of performance reports available for infrequently accessed distant hosts by taking advantage of daily cycles in network usage. For example, network measurements made at 9:00 AM are not likely to reflect performance at 3:00 PM on the same day, but measure-

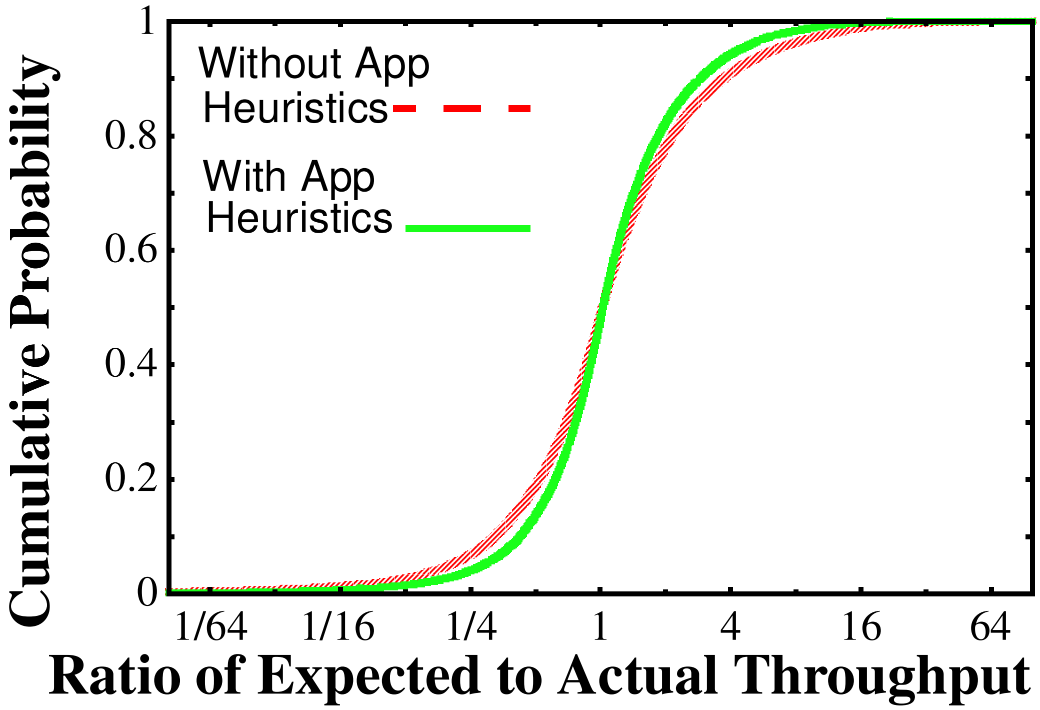

Figure 4.8: CDF of ratio of expected throughput (as generated by the performance server) to actual throughput (as reported by the client). The X axis is on a log scale.

ments made at 3:00 PM the previous day are. By using past measurements of performance from previous days in addition to current measurements, we can increase the number of performance reports available for these hosts and increase the accuracy of performance responses. Unfortunately, we found that the benefits of these improvements *are not worth the costs*, as we describe below.

4.6.1 Methodology

Before we modified our system to take advantage of daily cycles, we performed some experiments designed to quantify the potential benefits of using past performance reports. We must trade off the benefits of using past performance from days ago in terms of increasing the number of relevant performance reports for uncommonly accessed distant hosts with the costs of increased repository size (the collection of performance reports now spans days or weeks instead of hours) and possible additional temporal noise (defined in Section 3.5.1). If we find that using information from days ago only modestly increases the number of performance reports for uncommonly accessed distant hosts, the benefits are not worth the costs.

To show the potential effectiveness of using daily cycles to improve the number of relevant performance reports, we examined a long-running client side network trace. This trace consisted of a longer portion of the trace described in Section 3.4.1, in particular, approximately 9.2 million HTTP requests from 8000 unique clients over an 18 day period. To examine the effectiveness of our algorithm, we calculated the number of performance reports generated for each distant host under two scenarios: a baseline case where we do not take advantage of daily cycles in finding the set of relevant performance reports, and a second case where we take advantage of daily cycles in finding the set of relevant performance reports.

To measure the performance for baseline case, we considered only the weekday days of the trace (12 days) and divided the 12 day trace into 3 hour sections. For each 3 hour section, we calculated the number of performance reports reported for each distant host during that 3 hour period. For each distant host, we averaged together the number of performance reports across all of the 3 hour sections to obtain an average number of performance reports available for each distant host over small time scales.

To measure the performance for the second case, we again divided the 12 day trace into 3 hour sections, but unified the 12 day trace into a single day by combining similar 3 hour sections together into a single unit. For example, we combined together the 12:00 Noon-3:00 PM time period for each of the 12 days together and treated them as if they had all occurred on the same day. We then averaged together the number of performance reports across all of the 3 hour sections as before to obtain an average number of relevant performance reports over multi-day time scales.

4.6.2 Results

Figure 4.9 shows the results of this analysis. We see that taking advantage of past information over multi-day time scales improves the number of performance reports available for distant hosts, but only modestly. When only current-day information is used, 82% of distant hosts have less than 20 performance reports available for them. When multi-day information is used, this drops to 65%. This small improvement must be balanced against then costs of maintaining a much larger repository and an increase in temporal noise. Because this improvement is rather modest and the costs are significant, We concluded that taking advantage of Daily Cycles to improve the number of performance reports available for infrequently accessed distant hosts is not worth the costs.

4.7 Summary

In this chapter, we presented the core SPAND architecture. We described the components of the architecture (*Client Applications*, *Performance Servers*, and *Packet Capture Hosts*), and described how these components communicate (via active messages). We then described how the architecture is realized to measure performance for a generic bulk transfer transport and a HTTP specific transport, focusing on the challenges of making these measurements from a packet capture host. We then presented application-independent measurements of SPAND designed to show how well SPAND performs at providing meaningful network performance information to a group of clients. We saw that SPAND can quickly service over 95% of the Performance Queries presented to it, and that SPAND's Performance Responses are usually within a factor of two of actual observed performance. The

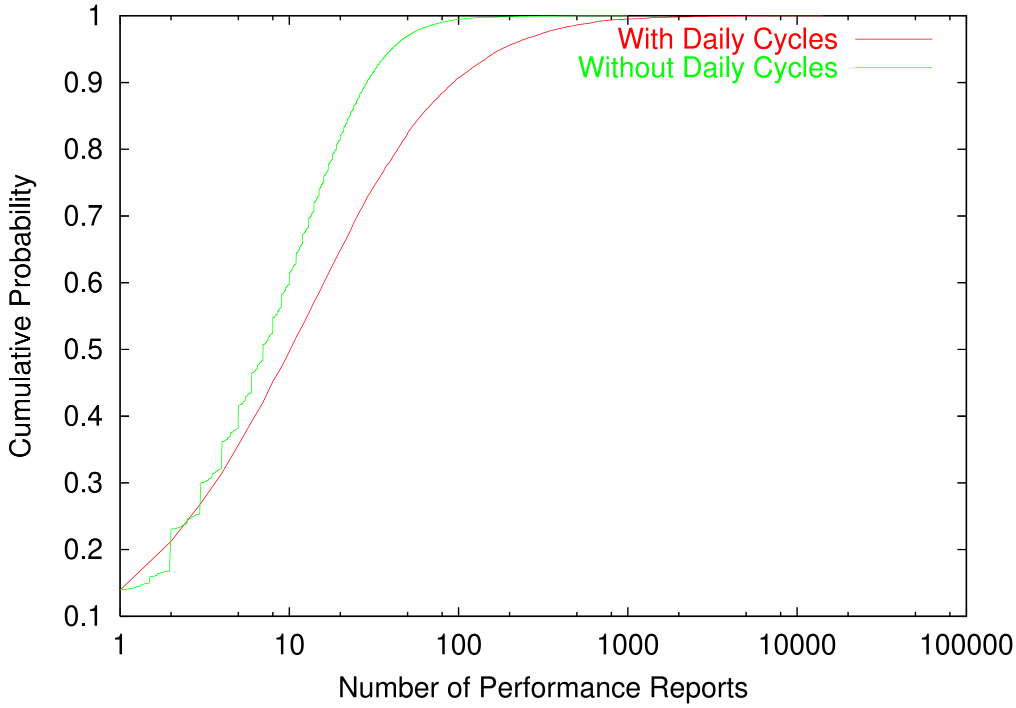

Figure 4.9: Distribution of number of performance reports for a given distant host when daily cycles are and are not taken into account. The X axis is on a log scale.

discrepancy can be attributed to the amount of *network noise*, the inherent variability in the state of the network.

In the next Chapter, we present our first application of SPAND, SpandConneg. SpandConneg utilizes SPAND to intelligently trade off document quality for improved application level performance.

Chapter 5

SPAND and HTTP Content Negotiation

In this chapter, we present a solution to the problem of intelligently choosing between alternate representations of Web Objects in response to network and server conditions. *SpandConneg* is a suite of applications that uses HTTP Content Negotiation to adapt to network bottlenecks both at web clients and web servers. SpandConneg uses SPAND's network measurement service to drive its adaptation policies. We begin by introducing the problem that SpandConneg solves and motivating why our solution solves this problem.

5.1 Background and Motivation

The Hypertext Transport Protocol (HTTP) is the application-to-application transport protocol for World Wide Web (WWW) applications. The response times that clients experience while using HTTP are related to the available bandwidth along the network path from a client to a particular server, the number of other clients that are also contacting the same server, and the size of the pages being transferred. Designers of a web site usually make assumptions about the number and type of clients that will visit them and design their site appropriately. For example, if the typical client is expected to have high available bandwidth, the site may consist of large, full color images or streaming multimedia. If the typical client is expected to have low available bandwidth, then the site may consist of smaller low color depth images without streaming multimedia. In addition, web site designers allocate bandwidth to the Internet in proportion to the number of clients that are expected to visit the site.

Unfortunately, these assumptions about a client's available bandwidth and the total client request load are often wrong. If the available bandwidth to a client is lower than the estimate determined by the web site creators, then the client will have to wait an excessive amount of time to download a particular web object. Unfortunately, due to the heterogeneity of today's Internet, there is no such thing as a "typical" client, and some clients visiting a web site will have longer download times than those predicted by the site designers. In addition, when a web server site is swamped with an unexpected burst of requests from a large number of clients, the available bandwidth to each client is limited.

The response times of clients increases as the connection from the server to the Internet becomes a bottleneck. WWW clients and servers need a mechanism to adapt WWW content to match dynamic changes in available bandwidth.

To overcome this problem of bandwidth heterogeneity, the IETF HTTP Working Group and the World Wide Web Consortium have defined mechanisms in HTTP for Transparent Content Negotiation. This allows a client and server to negotiate features of a web object including, most importantly, the content fidelity (and as a result, the object size). For example, clients with higher available bandwidth can request large full-color versions of images, and clients with lower available bandwidth can request smaller black and white versions of images.

The idea of changing object fidelity to match network characteristics is not unique to HTTP. Related work has experimented with different ways to achieve the same goal by maintaining or generating multiple representations of web content [58] [28] [27], sometimes in ad-hoc manual ways. One interesting example is what happens at some popular web server sites when they are swamped with an unexpected number of clients. For example, the administrators of the CNN web site www.cnn.com turn off advertisements (their primary source of income) during periods of heavy client traffic in an attempt to reduce server load. Using HTTP Content Negotiation, clients and servers can modify content fidelity to obtain acceptable response times. Clients may initiate a change in content fidelity to obtain a fixed response time at the cost of content quality. Servers may initiate change in content fidelity to reduce the total bandwidth leaving a server complex as the number of client requests per second increases.

In this chapter, we examine the effectiveness of HTTP content negotiation at reducing the actual response times of WWW clients and handling variable request loads on WWW servers. In particular, we answer the following specific questions:

- Is the first bottleneck of a web server complex likely to be its computing resources or its network connection to the Internet?
- How often do mismatches between expected and actual available bandwidth lead to excessively long (i.e. more than 30 seconds) transfer times for web clients today?
- Client-side initiated negotiation requires some estimate of available bandwidth and other network characteristics to work. How often is this estimate accurate, and do clients obtain acceptable response times using these estimates?
- What is the overhead of using content negotiation in a real web server implementation?
- For server-side negotiation, can outgoing bandwidth stay relatively constant as client load is added? What is the increase in web server throughput by using content negotiation?

To examine the effectiveness of client-initiated content negotiation, we perform trace analysis of actual client traffic and use this trace as a workload to drive our implementation. To examine the effectiveness of server-initiated content negotiation, we implement HTTP Content Negotiation in Apache, a commonly used web server, and stress the implementation by generating requests from Surge, a web client request generator.

The rest of this chapter is organized as follows. In Section 5.2, we discuss the problems of long client-side response times and server-side bandwidth shortages. Section 5.3 provides details about the IETF Content Negotiation mechanism and our implementation of it. Section 5.5 analyzes the effectiveness of client-initiated content negotiation at reducing web page download times. Section 5.6 describes the benefits of server-initiated content negotiation in handling additional request load, and finally, in Section 5.7, we summarize.

5.2 Motivating the Problem

In this section, we quantify the effects of long variable response times at web clients and bandwidth bottlenecks at web servers. We do this by measuring typical client response times and characterizing the behavior of servers under heavy load.

5.2.1 Long Response Times at Web Clients

To understand the response times observed by web clients, we used a network level packet trace to capture the behavior and dynamics of web clients at IBM research. The IBM research clients consisted of mainly workstations and PCs connected via ethernet or token ring local area networks to the external network. All clients were at most a few network hops from IBM's connection to its Internet service provider. From the packet trace, we extracted individual web page transfers and recorded the *response time*, how long it took for the clients to download web pages from web servers in the Internet. The response time for a page was measured by determining the difference in time from when the web page was requested to when the last web object embedded in the page was transferred. Because the response time is a function of the size of the web page, we also recorded the available bandwidth (the size of the page divided by the time taken to transfer the page) for each web page transfer.

Figure 5.1 plots the CDF of response times for these page transfers. We see that most of the transfers completed in a reasonable amount of time. 90% of the transfers completed in less than 5 seconds. However, some clients were forced to wait excessively long before receiving web pages. 5% of the time, clients waited more than 20 seconds before receiving a complete web page.

Figure 5.2 shows the CDF of the average bandwidth of these transfers. Again, we see that most transfers had acceptable performance. Approximately 80% of the transfers had an effective bandwidth of more than 200 KBits/sec. However, some transfers had much lower effective bandwidths. Approximately 45% of the time, transfers had effective bandwidths of less than 50 KBits/sec.

These measurements show us that web clients occasionally experience excessively long response times. Since the IBM community was very homogeneous and well connected, this wide variation can be attributed to the performance of the distant web server and the available bandwidth along the wide-area network path to the server and not the characteristics of the local connectivity to individual clients.

It is also important to remember that given the location and makeup of the client population, these are very conservative results. This was a relatively homogeneous pop-

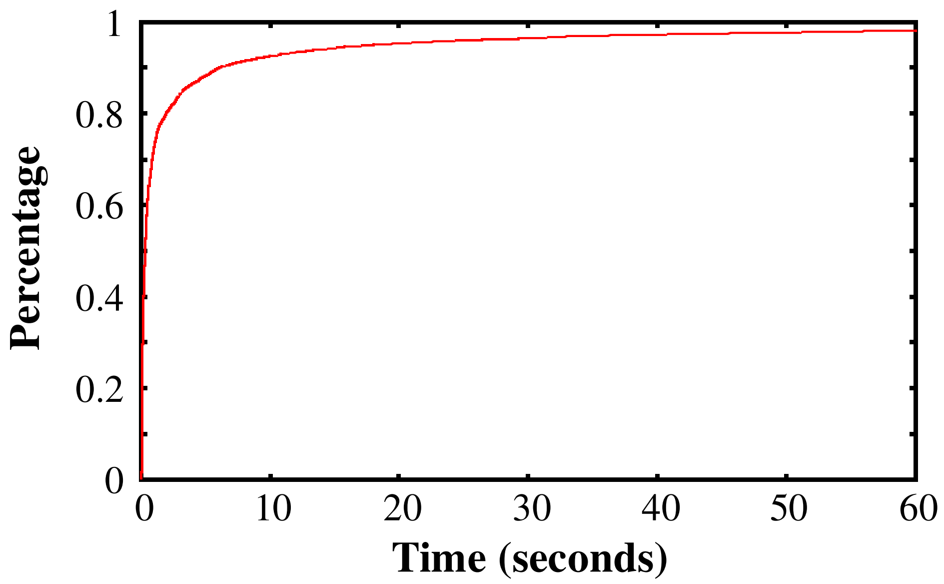

Figure 5.1: CDF of measured transmission times for pages retrieved by clients at IBM Research from servers in the Internet

ulation of web clients connected to the Internet via high-bandwidth local area networks, and IBM is a corporation that can pay for high quality Internet connectivity. Other client populations may not be as lucky, and for these groups of clients, the above problems will only be aggravated.

5.2.2 Bottlenecks at Web Servers

Every system has a bottleneck component whose performance limits the overall performance of the system. For web servers, the bottleneck could be one of three components: The CPU running the web server software, the disk holding the web content, or the network connection between the web server and the web client. In this section, we present results that show that the network connection is likely to be the first bottleneck of the overall web server.

To show this, we performed several calculations designed to quantify how much traffic web servers can generate in the absence of bottleneck constraints. If this amount of traffic is greater than the typical connectivity of web server sites to the Internet, then we can be sure that the first bottleneck that a web server is likely to face is their connection to the Internet.

We did this by examining recently published reports for the SpecWeb96 benchmark (available at `http://www.spec.org/osg/web96/results`) for web servers offered by a variety of companies. SpecWeb96 is a web server benchmark that reports the maximum number of web operations (web objects served) per second by an individual web server. Each company obtained these results by connecting a single web server to a collection of web clients via a high speed network connection such as an ATM network. The clients

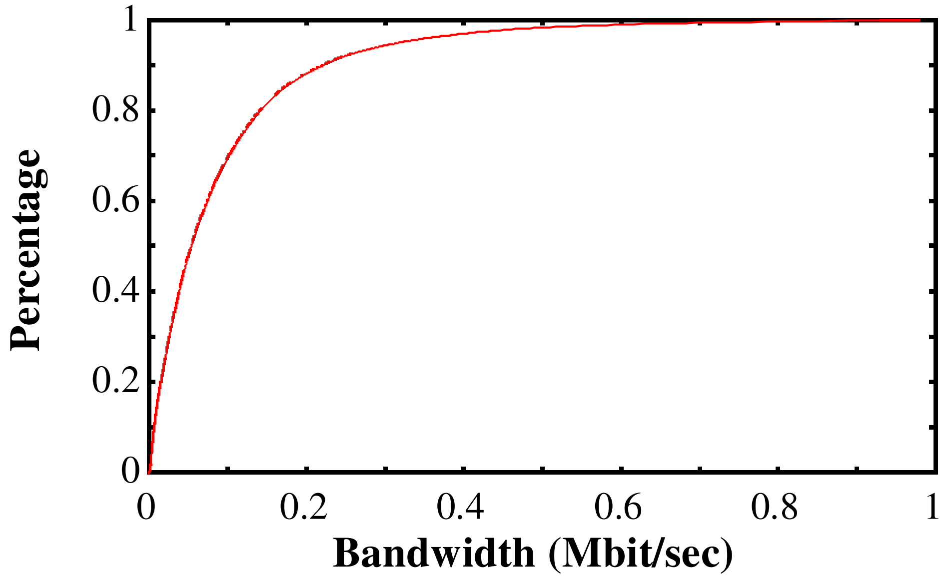

Figure 5.2: CDF of measured bandwidth for for transfers between clients at IBM Research and servers in the Internet

then requested objects from the server according to the guidelines in the benchmark. The company measured the number of web requests handled by the web server and reported this throughput as the benchmark result.

For the fastest results reported by each company, we multiplied the benchmark throughput of operations/second by the average size of a document in the benchmark (approximately 16 kilobytes). This calculation allows us to reconstruct the load on the link between the web clients and web server, resulting in an average bandwidth requirement for the server.

The results of this calculation are shown in Figure 5.3. We see that the bandwidth varies from 10 to 100 Megabytes per second, which is equivalent to the capacity of approximately two to 20 T3 lines. This means that a single web server machine could saturate from two to 20 T3 lines by serving out documents that are comparable to the ones in the SpecWeb benchmark. By using multiple machines and relatively simple load balancing techniques, this traffic could be easily increased by a factor of 10 or more.

As few web server sites are connected to the Internet with 20 T3 lines, this is a clear indication that for a well-designed web server, the bandwidth from the server complex to the rest of the Internet is likely to become the first bottleneck of the system.

From these measurements we see that bandwidth at the server and in the network is the likely cause of variations in response times observed by clients. This indicates that by controlling the size of transferred pages, and as a result the quality, we can control the response times observed by the clients and the load on the server's network connections. In the next section, we describe how HTTP Content Negotiation can be used to alleviate these problems.

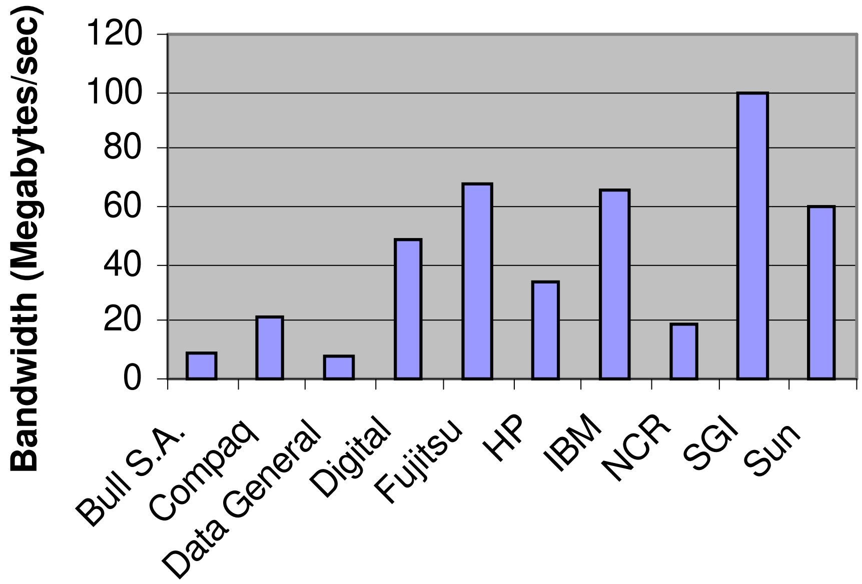

Figure 5.3: Traffic generated by different servers under the SpecWeb benchmark.

5.3 How Content Negotiation Works

This section describes the details of the IETF Transparent Content Negotiation Protocol and our implementation of the protocol. A more complete description is provided in [37].

5.4 IETF Transparent Content Negotiation

Sometimes web objects are available in alternate representations. For example, a text file may be available in several languages, an image may be available in several sizes, or a paper may be available as a postscript document or an HTML page. Transparent Content Negotiation is a mechanism that allows a client and server to select the most appropriate variant for a particular client. The word "Transparent" is used because both the client and server are aware of the available alternate representations. HTTP also supports Nontransparent Content Negotiation, where only the server is aware of the available representations. Typically, clients and servers can negotiate on the following features of an object:

- Mime Type
- Language
- Character Set
- Encoding
- Length

Negotiation can also be done on other arbitrarily defined feature tags (for example, screen size, color depth, etc.). Separate work is being done on a "Feature Tag Registry" [36] that allows developers to define Feature Tags in a unified framework similar to that used for MIME types [12].

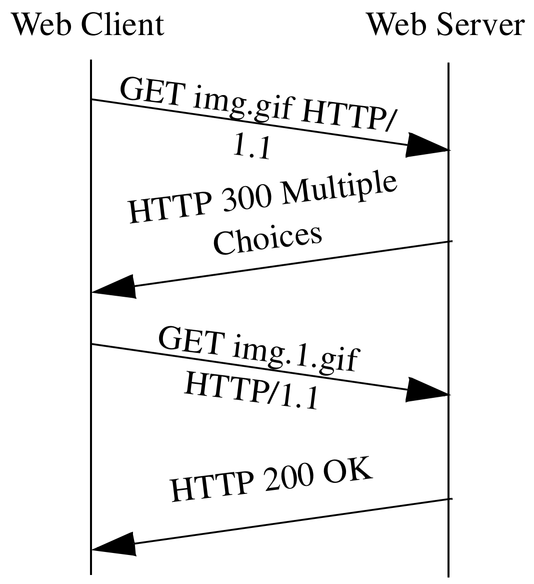

Figure 5.4: Sample transaction using transparent content negotiation

Figure 5.4 shows a typical transaction using Transparent Content Negotiation. A web server responds to a request for a web object with a "HTTP 300 Multiple Choices" response that lists the variants and their features. Figure 5.5 shows the format of a Multiple Choices response.

```
HTTP 300 Multiple Choices:
Date: Tue, 11 Jun 1996 20:02:21 GMT
    TCN: list
    Alternates:
{"img.1.gif" 1.0 {type image/gif} {length 4000}},
{"img.2.gif" 0.75 {type image/gif} {length 3000}},
{"img.3.gif" 0.5 {type image/gif} {length 2000}}
```

Figure 5.5: Format of a multiple choices response

The web client uses this response to select one of the variants and downloads that variant from the web server. There are also mechanisms that allow a client to execute a predefined variant selection algorithm on the web server (for example, to automatically choose the smallest acceptable object), to avoid the extra round trip associated with sending the Multiple Choices response.

In SpandConneg, we only consider negotiating between static discrete representations of image objects with different sizes. This allows a client to trade off image size, quality, and color depth for downloading speed and a server to trade off image quality for greater throughput under periods of high load.

5.4.1 Content Negotiation in Apache

In this section, we describe how content negotiation is implemented in Apache and how we modified it for our purposes. The version of Apache we used only implements Nontransparent Content Negotiation. For our purposes, this was acceptable, because we used the Apache implementation to measure server-side negotiation where the client does not take part in the variant selection process.

Apache has two mechanisms to enable negotiation of web content: *Type Maps* and *MultiViews Searches*. A Type Map is a file that explicitly lists the variants to be negotiated and their characteristics (e.g., size, quality, language, mime type). When a web client requests a Type Map, the web server uses the Type Map to find the characteristics of the variants and selects the variant that is most appropriate for that particular client. Using MultiViews, a web server does an implicit filename pattern search and chooses among the results. For example, if a web client requests a file "img.gif" and no such file exists on the server, the server looks for files with names "img.*" and internally constructs a Type Map using those files. If the web server happens to find a Type Map file while doing this wildcard search, then that Type Map is used instead. Figure 5.6 shows examples of negotiation using Type Maps and MultiViews Searches.

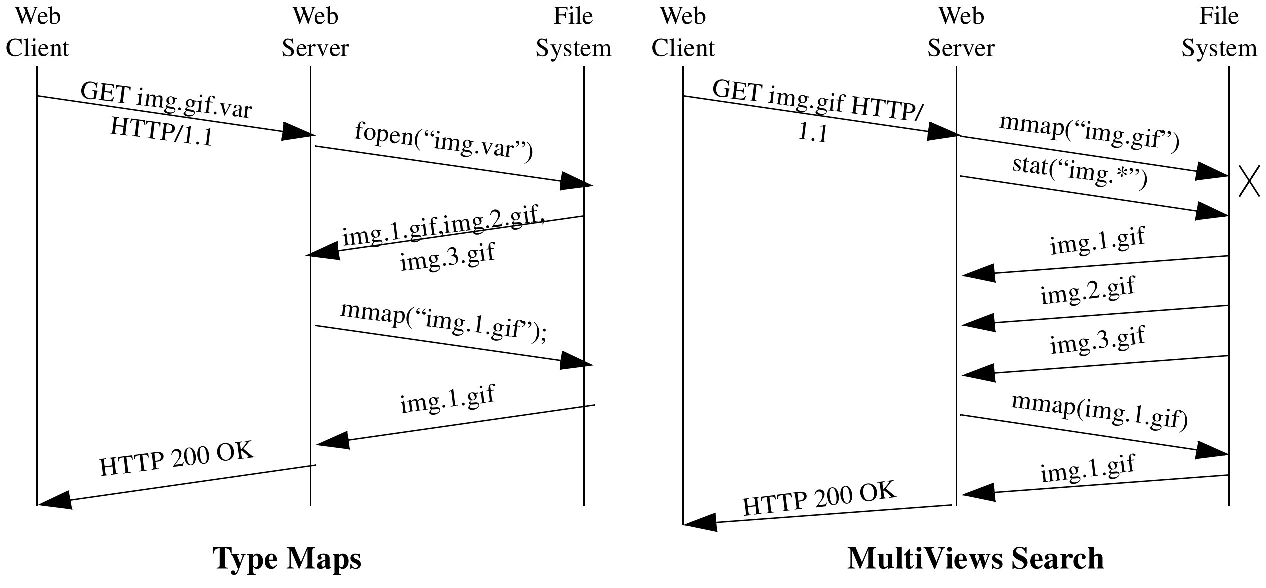

Figure 5.6: Apache mechanisms for retrieving negotiated documents

We used Apache's MultiViews mechanism for content negotiation. We chose to use this rather than Type Maps because it allows us to avoid changing HTML documents. For example, if we used Type Maps, we would have to modify HTML documents to change all hyperlinks to images and embedded image references to point to the Type Map instead of the original images. To make a object negotiable, we changed the name of the stored object to a different one. When a client requested that negotiable object, it forced a MultiViews search. To speed up the MultiViews process, we modified the web server to explicitly look for a Type Map file before doing the filename pattern search. This avoided a linear search through the files in a directory.

5.4.2 Generating Alternate Representations

An important part of a content negotiation implementation is a program that generates the alternate representations among which the client and server choose. To address this, we wrote a web content crawler that scans the files on a web server and generates alternate representations. As previously mentioned, we only consider negotiation of image files. To generate alternate representations for GIF and JPEG image files, we used transcoding programs written by the GloMop group at UC Berkeley for their TranSend WWW proxy [28]. We converted all image files to JPEGs, and then changed the JPEG quality parameter to produce a particular alternate representation.

Our web content crawler generated 6 alternate representations for each image file, each of a different size. Each alternate was 70% of the size of the next larger alternate, leading to possible alternates that are 70%, 49%, 34.3%, 24%, 16.8%, and 11.8% of the size of the original image. We chose a multiplicative decrease in size rather than a linear decrease in size to keep the relative size between representations the same. For example, if we had chosen representations that were 20%, 40%, 60%, and 80% of the size of the original image, the relative size of the first alternate to the full size image is 80%, whereas the relative size of the smallest two alternates is 50%.

5.5 Using Content Negotiation and SPAND at clients

As shown in Section 5.2.1, the transfer time for retrieving web pages from different servers varies greatly. This is undesirable since users would like consistent response times for viewing similar information. This variation is primarily a result of differences in available bandwidth to the server site, speed of the server and size of the requested pages. Since there is no easy way to modify available bandwidth or the speed of the server, we must resort to modifying the size of web pages to provide unchanging response times. In this section, we explore the effectiveness of our algorithms to provide this consistent performance.

5.5.1 Algorithm

The basic technique for providing constant response times consists of the following steps. The client obtains a list of equivalent alternate versions of the page. The client then retrieves a performance estimate for the server. Based on this performance prediction and the size of the variants, the client estimates the transfer time for the different versions of the web page and chooses the one that most closely matches the user's requested response time.

For the client to estimate the transfer time for a web page, it needs to know the combined size of the base HTML page and its embedded objects. To do this, we add a feature tag called "full-page length" to the base HTML page. Alternate versions of the HTML document have different quality embedded images and different total page sizes. The client obtains a full list of the variants and total sizes via the Transparent Content Negotiation mechanism.

5.5.2 Experimental Methodology

To test the effectiveness of our client-side implementation, we used a combination of trace analysis and implementation for our experiments. We began by collecting a client-side network level packet trace. This trace was played back as a workload into our client side content negotiation implementation to determine the representation that our system would choose based on current network conditions. The trace was then used again to determine how well that choice would actually have worked at reducing client-side download times. We describe this process in more detail below.

A machine at the connection between IBM Research and the its Internet service provider collected the packet trace. This machine used `tcpdump` to record all packets sent from or to the HTTP port (port 80). The trace was collected over a 4 hour period on a Sunday. It contained a total of 8618 web page retrievals and 23719 web object transfers. From the packet trace, we used techniques described in Section 4.3.2 to determine the web object transfers in the individual trace, which web object transfers comprised a single web page transfer, and the duration of each web page transfer.

Once we post-processed the traces to determine what web pages were transferred by the collection of clients and how long it took them to transfer the pages, we used this sequence of transfers as a workload for our content negotiation implementation. The constraint placed on our implementation was to retrieve the highest quality page possible subject to a maximum download time constraint of 10 seconds. In particular, for each web page transfer in the trace, our client performed the following steps:

- Contact the distant web server to get the list of alternate web page representations and their sizes.
- Contact the SPAND performance server to get an estimate of how long it would take for the client to download each of these representations.
- Choose the largest representation whose transfer would complete in less than 10 seconds. Note that this could be the original representation, meaning that no negotiation was necessary.

To obtain an estimate for how long it would take for the client to download a particular representation, we used SPAND's web object Download Time Metric, asking for the performance for the original web page as well as the alternate representations for that page. If no information was available for a particular web object, we fell back to using the TCP Time To Completion Metric. In particular, if a particular representation were B bytes in size, we queried the performance server, asking it how long it would take to retrieve B bytes from the host named in the URL.

Once a representation of a page was chosen, we examined the traces to determine how long it would have actually taken to transfer that representation of the page. For each web object, the size of the object was adjusted by the appropriate factor to determine the actual transfer size of the object. The `tcpdump` trace was then examined to determine how long it would take for that representation of the object to be transferred. For example, assume that a particular inline image has a full size of 100 KBytes, but our client actually chose a representation that was only 70 KBytes in size. We examined the packet trace to see when 70% of the original full-page transfer completed, and used the elapsed time for 70% of the transfer as an estimate of how long it would take the smaller representation to complete. We estimated the total transfer time to be from the beginning of the HTML Object fetch to the end of the fetch of the last scaled down web object for that page.

5.5.3 Results

We evaluated the client-side system by answering two questions:

- How often does our system correctly identify that no content negotiation is necessary?
- When our system does determine that content negotiation is necessary, how well does our system perform at reducing client-side download times to acceptable levels?

We answer these questions below.

Identifying When Negotiation is not Necessary

The collected trace contained a total of 8618 page transfers over the four hour period. Of these, performance estimates were available for 3854. If the trace had been collected for a longer period, a larger percentage of the transfers would have had estimates. Of these transfers, the SPAND predictions for 3424 of the transfers indicated that their transfers would complete in less than 10 seconds, and as a result, no negotiation was necessary. To see if our predictions were correct, we examined the actual time taken to download the web page and examined what fraction of them took less than ten seconds to download.

The actual time to transmit these pages is shown in Figure 5.7. The X axis shows the download time for the web page in seconds, and the Y axis shows cumulative probability. We see that over 95% of these requests completed in well under the target time of 10 sec. meaning that our system often determines correctly that no negotiation was necessary.

Reducing Client Side Download Times

The remaining 430 transfers were expected to take more than 10 seconds and smaller versions of the pages were requested. Of these transfers, we wanted to see how often the transfers of reduced-size web pages completed in less than 10 seconds.

Figure 5.8 shows the results of this analysis. The X axis shows the page download time in seconds, and the Y axis shows cumulative probability. There are two curves on

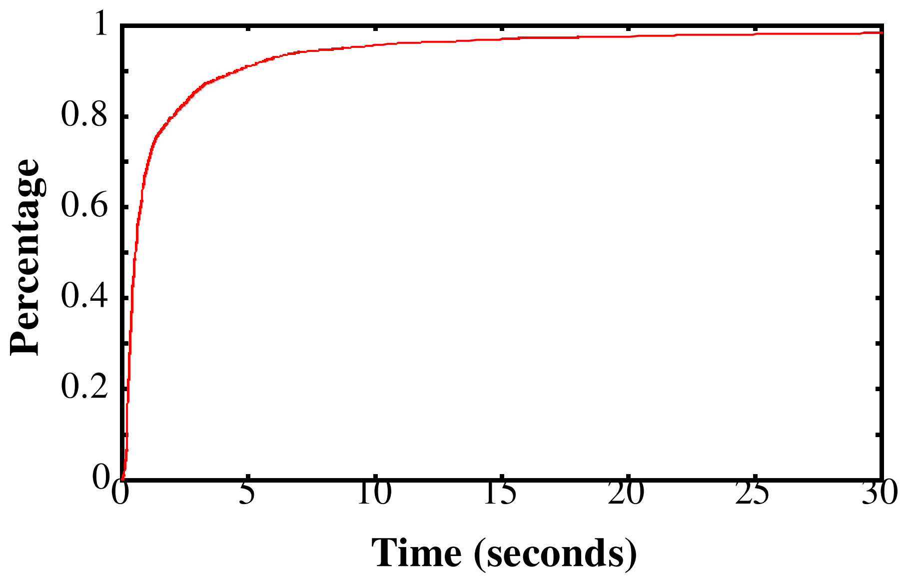

Figure 5.7: CDF of measured transmission times for pages that had performance estimates but did not require retrieval of an alternate version.

the graph. The upper curve shows the CDF of transmission times when clients download reduced size versions of the pages. The lower curve shows the actual time taken to download full-size representations of the pages, showing the performance that clients would have if they did not use content negotiation at all.

We can see that using content negotiation reduces the time it takes to download pages. The median time taken for the full page transfers was about 16 sec, whereas the median negotiated page transfer only took 6 sec. More importantly, it shrinks the tail of the distribution and reduces the likelihood that clients will experience extremely long download times. When clients perform content negotiation, we see that only 10% of the time, clients wait more than 30 seconds to download a web page. On the other hand, when clients do not use content negotiation, they must wait more than 30 seconds more than 35% of the time.

However, our system only does an acceptable job at meeting the constraint of reducing download times to less than ten seconds. Using content negotiation, approximately 60% of the downloads completed in under 10 seconds. This shows the effect of *network noise* which we described in Section 3.5.1. Even though we made network measurements from a group of similarly connected clients over a short period of time, these estimates were often a factor of 2 away from actual network performance, due to inherent variation in characteristics such as round trip time and available bandwidth that change from minute to minute. Although SPAND does a fairly good job at predicting performance, the performance responses given by SPAND still have error associated with them and are sometimes wrong, leading to errors in the estimated size reduction required to download a page within the target time. This means that network-aware applications that wish to make hard guarantees about application-level performance will have to be conservative in interpreting the network

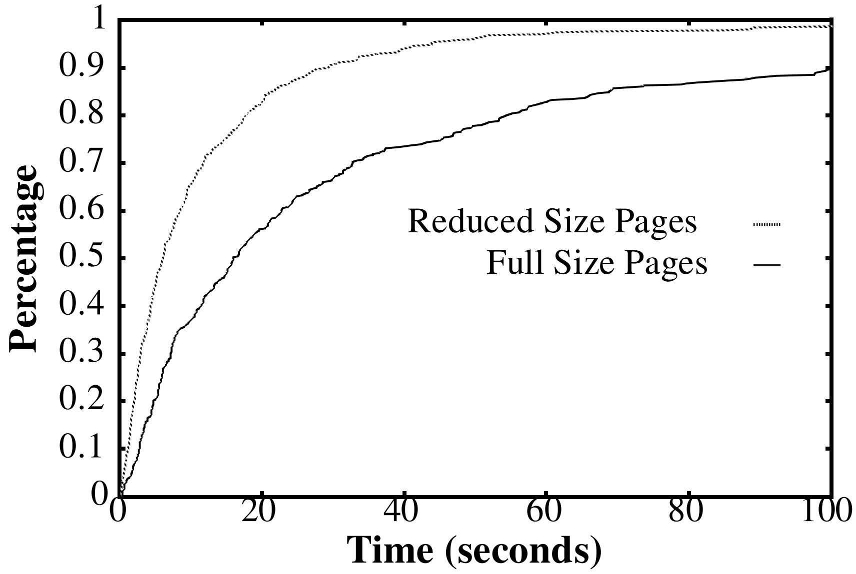

Figure 5.8: CDF of transmission times for pages that had performance estimates and required an alternate version. Retrieval times for the original page and the alternate version are shown.

measurement statistics they receive from SPAND, for example, by subtracting one or more standard deviations from network metrics such as available bandwidth before using them to drive application level decisions.

In any case, we see that network-aware content negotiation can be effectively used to reduce long download times at web clients by making informed tradeoffs between content fidelity and response time.

5.6 Using Content Negotiation and SPAND at Servers

In this section, we quantify the benefits of server-initiated content negotiation in managing bandwidth requirements of busy web servers. This allows a web server to handle additional load in cases where the connection between the web server and the Internet is the bottleneck link.

5.6.1 Algorithm

Our implementation consists of two parts: adding mechanisms to the web server to change the size and quality of web objects served (and therefore the bandwidth leaving the server), and a separate policy program that watches and manages the bandwidth leaving the server.

To provide a mechanism for changing the quality of web objects, we added a configurable *Maximum Size* option to the variant selection algorithm. The Maximum Size is a fraction between 0 and 1 which indicates the largest allowable variant that is selected by

Apache. For example, if the Maximum Size fraction is 0.5, only variants that are less than one-half the size of the original object are considered when performing content negotiation. By modifying the Maximum Size fraction, we can proactively control the size of the objects being delivered by the web server, and therefore the total bandwidth leaving the server machine.

The policy program is a separate application that watches the total bandwidth leaving the web server and reduces the quality of objects served by the web server if the total bandwidth becomes too great. Using `tcpdump`, we examine packet headers to measure the total number of bytes transmitted between the web server and the external network. The policy program reads this statistic every fifteen seconds to obtain an estimate of the bandwidth of the outgoing link from the web server. This statistic is then exponentially smoothed resulting in an average bandwidth `sbw`. The goal of the policy program was to keep `sbw` between a high water mark that was 90% of the limit bandwidth and a low water mark that was 50% of the limit bandwidth. If `sbw` rose above the high water mark, the web server was (gracefully [1]) restarted by the policy program with a lower Maximum Size fraction. If `sbw` fell below the low water mark, the web server was restarted with a higher Maximum Size fraction. Whenever the web server was restarted, a minimum of two minutes passed before changing the Maximum Size fraction again, to allow enough time for the changes to take effect.

5.6.2 Experimental Methodology

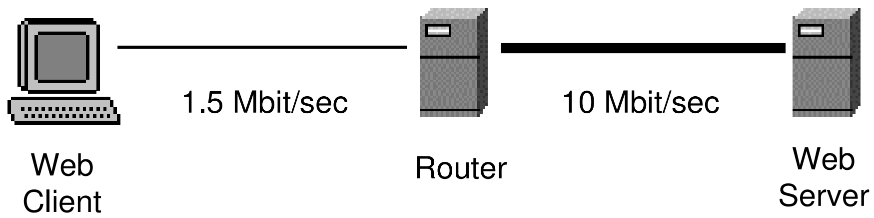

Figure 5.9: Topology for server side experiments

Our experimental setup consisted of a web client machine connected to a web server machine through a router (Figure 5.9). The web client and server machines were 200 Mhz Pentium Pro PCs running Linux 2.0.30, and the router was a 133 Mhz PC running BSD/OS 3.0. The bandwidth of the link between the client and router was limited to 1.5 MBits/sec, using a network emulator driver written by Venkata Padmanabhan. The web client was running Surge, a HTTP request generator written at Boston University [6]. Surge is an artificial HTTP workload generator that uses empirical measurements of client behavior to accurately recreate typical HTTP requests. The web server was running Apache 1.3b5. The web server document pool consisted of 200 objects with sizes from 198 bytes to 700 KBytes with a distribution in sizes described in [7]. Characteristics of the request pattern (in particular, document popularity, temporal locality of requests, embedded document count

[1] Apache has a mechanism that allows the web server to be gracefully restarted by spawning new worker processes while allowing existing work processes to die after handling any pending requests.

of HTML pages, and length of a client's active and inactive periods) all follow representative distributions reported in a background paper on Surge [7]. We made approximately 75% of the documents negotiable by creating Type Map files for them and alternate representations of the documents. With this partition of negotiable and non-negotiable documents and the access pattern requested by Surge, 60% of the bytes transferred were from negotiable documents and 68% of the documents transferred were negotiable documents, both agreeing with characteristics from client-side traces [32].

To stress the web server, we started with an initial fixed client population requesting objects from the web server. Every 800 seconds, additional clients were added to the client population by restarting the Surge process on the client machine. This continued until a maximum number of simultaneous clients were requesting objects from the web server. We measured the traffic on the link between the client and router and the throughput of the server as a function of time under four scenarios:

- An unmodified Apache server with an unconstrained network link.
- An Apache server modified for content negotiation with an unconstrained network link.
- An unmodified Apache server with a network link constrained to 1.5 MBits/sec.
- An Apache server modified for content negotiation with a network link constrained to 1.5 MBits/sec.

Comparing the first two scenarios allows us to quantify the overhead of our content negotiation implementation. Comparing the second two scenarios allows us to quantify the benefits of content negotiation in increasing throughput as a large number of clients access the web server through a bottleneck link which is at or close to 100% utilization. For the first two scenarios, we started with an initial client population of 5 clients and added 25 clients every 800 seconds until we reached a maximum of 530 clients. For the second two scenarios, we started with an initial client population of 5 clients and added 10 clients every 800 seconds until we reached a maximum of 95 clients.

5.6.3 Results

Overhead of Content Negotiation

Figure 5.10 shows the overhead of the content negotiation process. The curve shows smoothed throughput (measured in web operations per second) as a function of time for the duration of the experiment. As a convenience, the X axis of each figure also shows when the number of clients was increased to a larger number. The Y axis is the smoothed throughput of the web server measured in connections per second. The periodic dips in the graph are due to the time it takes to restart the Surge process with a greater number of clients.

There are two curves on the graph. One is for an unmodified Apache server, and the other is for Apache server modified to use MultiViews to serve web objects. We see that in the early part of the trace (which corresponds to a small number of clients accessing

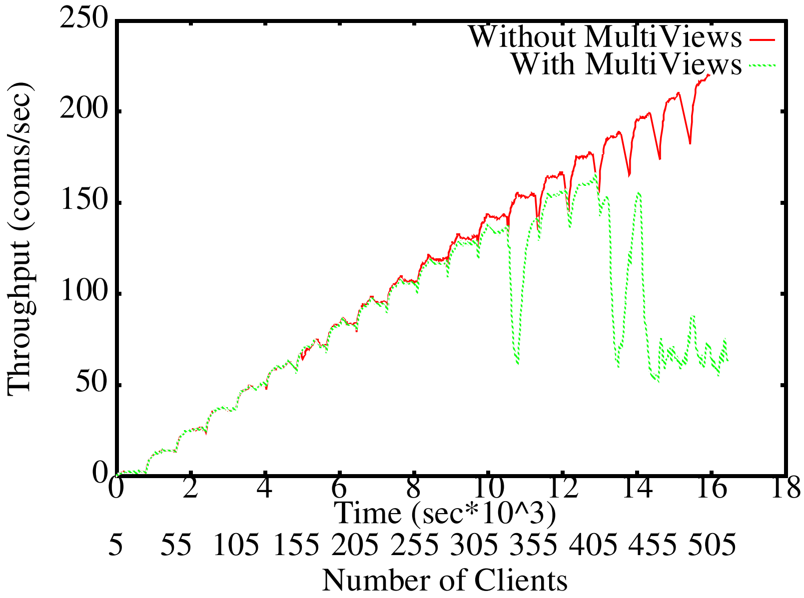

Figure 5.10: Unconstrained Apache throughput with and without MultiViews

the server), there is virtually no difference between the two curves, indicating that there is little overhead in using MultiViews to serve objects. As more clients are added, however, the MultiViews version of Apache performs significantly worse than the version without MultiViews, and actually performs worse than MultiViews Apache under lower load.

From examining the Apache log files, we found that this degraded performance is due to a limit on the number of open files in the system. The MultiViews version of Apache must open additional files (in particular, the Type Map files), and under heavy load, there are not enough file descriptors available to process all client requests, leading to a large number of aborted transactions. We feel that this problem is largely due to the way in which Apache handles MultiViews searches and could be overcome if more effort were put into optimizing the MultiViews process, for example, by caching the contents of Type Map files in memory instead of accessing them from the file system. If this were done, then there would be little or no overhead incurred by using Content Negotiation on a server.

Benefits of Server Side Content Negotiation

Figure 5.11 shows the smoothed link bandwidth `sbw` on the connection between Apache and the router as a function of time in the case without any content negotiation. As a convenience, the X axis of the figure also shows when the number of clients was increased to a larger number. Also included in Figure 5.11 are the 90% and 50% values of the limit bandwidth (1.5 MBits/sec).

We see from the figure that the constrained link becomes the bottleneck of the system after the client population grows to approximately 35 clients. The smoothed bandwidth curve quickly grows to the limit bandwidth and stays there. After this point, traffic is dropped at the gateway to the constrained link and clients are either turned away or queued

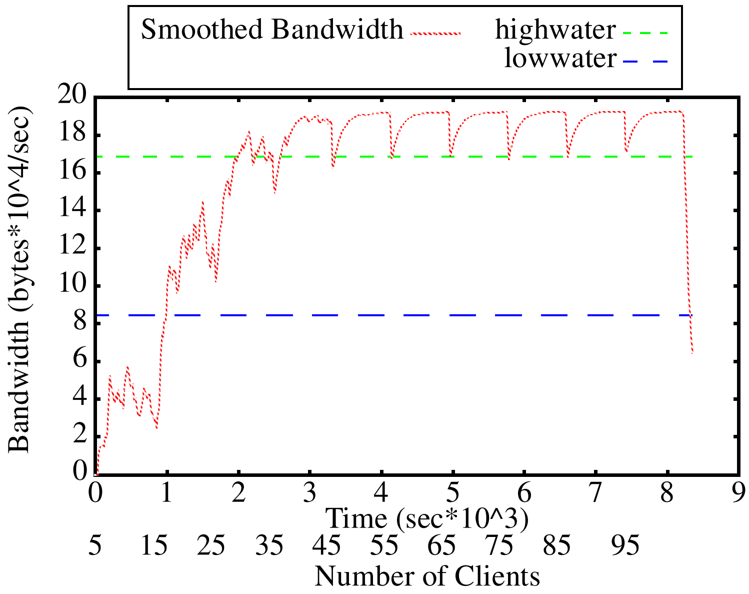

Figure 5.11: Bandwidth leaving Apache as a function of number of clients without content negotiation

up at the web server. We can more directly see this effect in Figure 5.12, which shows the web server throughput as a function of time. The throughput is limited to approximately 10 connections per second at a population of approximately 45 clients.

Figure 5.13 shows the smoothed link bandwidth `sbw` on the connection between Apache and the router as a function of time (and number of clients) in the case with content negotiation. Also included are the high and low limit bandwidth and the changes in the Maximum Size fraction as a function of time. The bandwidth still approaches the maximum link bandwidth, but because the server can reduce the size of objects being served, it approaches the maximum more slowly. This leads to a greater peak throughput of approximately 17 connections/second with a client population of 75 clients, as shown in Figure 5.13.

Although using content negotiation does increase the peak throughput of the system, the increase is rather modest (a 50% increase). This is mostly because only 60% of the bytes come from negotiable documents. The bandwidth requirement of the non-negotiable documents is unaffected by content negotiation and becomes a larger fraction of the total bytes transferred as negotiated documents become smaller.

The choice of 60% was from analyzing a client-side web trace and as a result captures the average characteristics of a large number of web servers as compared to the specific characteristics of a particular web server. Some web servers may serve out a higher or lower fraction of negotiable bytes than 60%. For example, a sampling of the pages at the New York Times web site on March 28, 1998 indicates that approximately 90% of its bytes are negotiable. In this case, the benefits of server-side content negotiation will be more significant in handling additional client load.

To quantify this, we repeated our experiment choosing a different set 'of non-

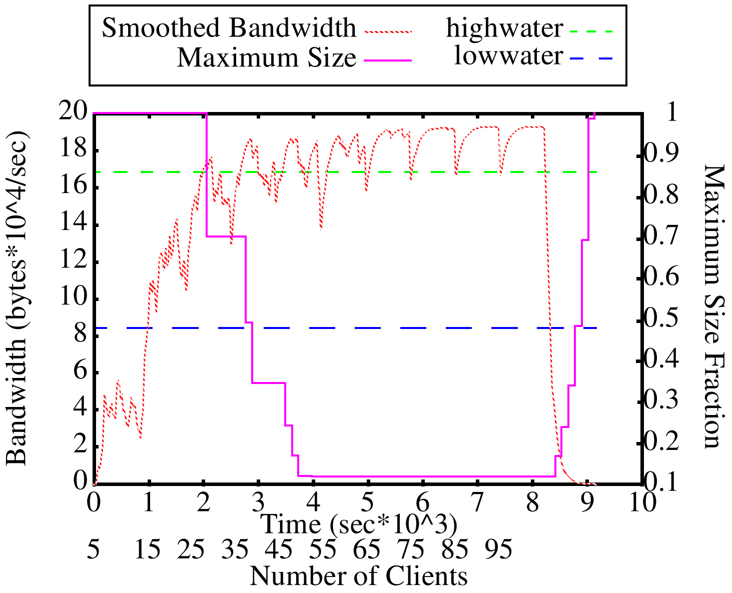

Figure 5.12: Bandwidth leaving Apache as a function of number of clients with content negotiation

negotiable objects such that 90% of the bytes served by Apache were negotiable. To further stress our implementation, we added twice as many clients every 800 seconds as in the 60% case.

The results of this experiment are shown in Figure 5.14. The x axis shows elapsed time and the number of clients accessing the web server, and the y axis shows the web server's throughput in connections per second. The lower x axis shows the number of clients for the Non-negotiable and 60% negotiable configurations, and the upper x axis shows the number of clients for the 90% negotiable case. We see that the bandwidth stays below the peak bandwidth for a larger number of clients, and this leads to a greater throughput for the web server. It takes a client population almost four times as large to saturate the link as in the case without content negotiation.

Figure 5.15 shows the throughput for the version of Apache without content negotiation, the version where 60% of the bytes are negotiable, and the version where 90% of the bytes are negotiable. The 90% version of Apache has a peak throughput four times the peak throughput of Apache without content negotiation.

From these results, we can conclude that there is potential to dramatically increase the throughput of a web server and the number of clients that the server can using support by using content negotiation. The actual benefits are very dependent on the fraction of bytes that are negotiable. Servers that have a high degree of negotiable content such as images will see the most benefit, while servers that have a smaller degree of negotiable content will see more limited benefits.

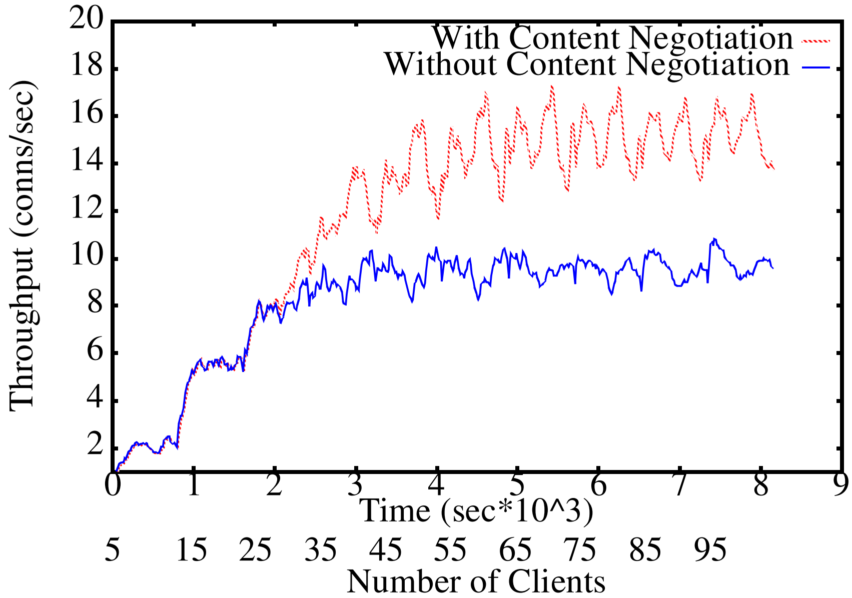

Figure 5.13: Apache throughput with and without content negotiation

5.7 Conclusion

In this chapter, we have examined the effectiveness of using HTTP Content Negotiation to improve the response time observed by web users. We presented an implementation of an adaptive web client that reduces web page download times at the expense of lower content fidelity, using SPAND's network measurements to drive adaptation decisions. In addition, web servers can use content negotiation to reduce bandwidth requirements under periods of heavy request load. We presented a simple mechanism and policy that reacts to increases in client population by decreasing the quality of served web objects. This allows servers to handle greater number of clients and have increased throughputs than servers that do not perform negotiation.

We can now answer the questions posed in the beginning of this chapter:

- *Is the first bottleneck of a web server complex likely to be its computing resources or its network connection to the Internet?*

By examining published benchmark results for the SpecWeb96 benchmark, we saw that a well designed web server can easily saturate up to 20 45 MBit/sec T3 lines. This implies that the first bottleneck of most web server complexes will be their network connection to the Internet.

- *How often do mismatches between expected and actual available bandwidth lead to excessively long (i.e. more than 30 seconds) transfer times for web clients today?*

By examining the actual behavior of web clients, we found that some clients often have to wait excessively long for web pages to download. Even for a well-connected group of clients like those at IBM, 5% of clients had to wait more than 30 seconds for web pages.

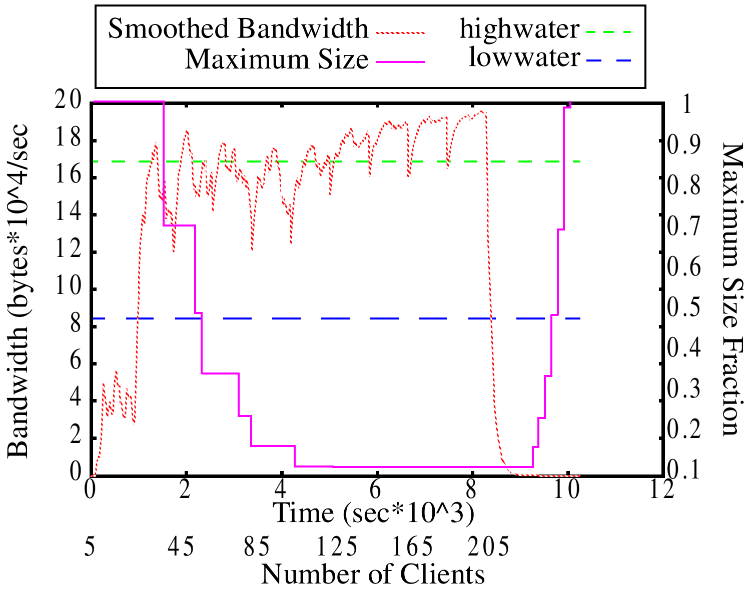

Figure 5.14: Bandwidth leaving Apache when 90% of the bytes come from negotiable documents.

- *Client-side initiated negotiation requires some estimate of available bandwidth and other network characteristics to work. How often is this estimate accurate, and do clients obtain acceptable response times using these estimates?*

Web clients that use content negotiation can achieve a lower likelihood of excessive download times as compared to clients that do not use content negotiation. 35% of the time, clients that do not negotiate when they should must wait for 30 seconds or more to download pages. Clients that do use content negotiation reduce the chances of this to 10%. Clients also observe better average-case behavior. The median transfer time of a web page drops from 16 to 6 seconds using content negotiation. Our system only does an somewhat acceptable job of keeping download time constant at the expense of content fidelity, however, due in most part to the amount of network noise in our network measurements. Using SpandConneg, approximately 60% of downloads met the user-specified goal of a download time less than ten seconds. Applications that wish to obtain hard guarantees for application level performance must be very conservative about network statistic measurements.

- *What is the overhead of using content negotiation in a real web server implementation?*

We implemented and measured Content Negotiation in Apache, a commonly used web server. We found that in most cases, a web server serving negotiated content has equivalent throughput to a web server serving non-negotiated content. Care must be taken, however, to carefully manage operating system resources such as the number of open file descriptors, as limits on these resources can adversely affect performance.

- *For server-side negotiation, can outgoing bandwidth stay relatively constant as client*

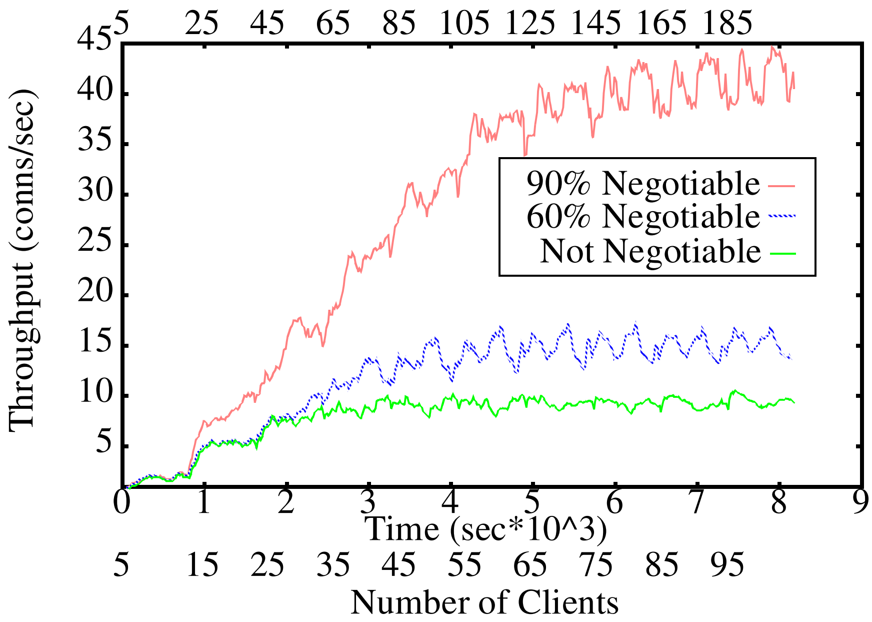

Figure 5.15: Apache throughput for varying fractions of negotiable bytes

load is added? What is the increase in web server throughput by using content negotiation?

The degree to which throughputs can be increased depends significantly on the fraction of served bytes that come from negotiable documents. For servers with a relatively low fraction of negotiable documents, this increase is rather modest. For example, if 60% of bytes come from negotiable documents, a 50% increase in throughput is obtained by using Content Negotiation. However, servers with a relatively high fraction of negotiable documents may see much more significant increases in throughput. Servers with 90% of bytes coming from negotiable documents can see increases of 400% or more by using content negotiation.

In the next chapter, we describe *LookingGlass*, the second adaptive application that we developed to take advantage of SPAND's network performance measurements.

Chapter 6

SPAND and LookingGlass: A Mirror Selection Tool

In this chapter, we present a solution to the problem of selecting between multiple mirror servers that replicate the same content. *LookingGlass* is a Mirror Selection Tool that uses SPAND to drive its choices for Mirror Selection. We begin in Section 6.1 by presenting background and motivating why a Mirror Selection tool is necessary. In Section 6.2, we describe existing solutions to the problem of Mirror Selection and their limitations. In Section 6.3, we describe LookingGlass and its advantages over existing solutions. In Section 6.4, we describe how we evaluate the performance of LookingGlass, and in Section 6.5, we present results showing how well our system performs at choosing mirrors to contact when a client is presented with content available at multiple locations.

6.1 Background and Motivation

The number of Internet users has grown at a phenomenal rate over the past few years. As the number of online users increases, the load placed on the Web's busiest web servers has also increased. One of the common methods to improve web performance has been to replicate popular data items on multiple mirror sites. For example, the most popular download from `www.download.com` on Feb 1, 1999 (a chat program called ICQ) is available from 17 distinct sites in 13 countries. Similarly, the mirror page for www.redhat.com (`http://www.redhat.com/mirrors.html`) lists 82 distinct sites in 28 countries. The complete problem of mirroring web content at multiple locations can be broken down into three sub-problems:

- *Mechanisms for Mirror Advertisement:* There must be a way for web servers to advertise to web clients where the mirrored content is located.
- *Metrics for Mirror Ranking:* A web client must decide how to rank the mirror locations in terms of their quality of connectivity to local clients.
- *Mirror Selection Algorithm:* Given a ranking, there must be an algorithm for web clients to select the most appropriate mirror from which to download the object.

We describe existing solutions to these subproblems below.

6.2 Existing Solutions to Mirror Selection

Currently, the above three subproblems are usually solved in manual and ad-hoc ways. In this section, we describe existing solutions to these subproblems and their limitations:

6.2.1 Existing Mechanisms for Mirror Advertisement

In one existing mechanism for mirror advertisement, the user is presented with an HTML page that contains information about the mirror servers as well as links to the mirrored data. The disadvantage of this solution is that the administrator of the primary site must be notified every time a mirror is added or deleted to update the HTML page. If a mirror site is added without the knowledge of the primary site administrator, there is usually no way for web clients to find out about its existence. Another disadvantage is that there is no way for the primary mirror site to disseminate the mirror information to secondary sites. If a secondary mirror administrator wants to display the full list of mirror locations on its own site, it must manually copy the contents of the HTML page at the primary site to its own location.

6.2.2 Existing Metrics for Mirror Ranking

The mirror ranking process is currently manually performed by the end user, using imperfect heuristics such as geographic location or advice from the web site administrator listed on the HTML page. For example, the web site `www.download.com` ranks mirror locations on a one-star to four-star ranking based on its own evaluation of the reliability and quality of connectivity to a particular site. Although these metrics can often be a hint for good performance, the information provided usually does not enable the user to select the server from which the data can be retrieved most quickly, because it does not take current web server load, network conditions, or other actual performance along the network path from the client to that particular mirror site into account. The burden is on the primary site administrator to keep these hints up-to-date with actual performance observed by clients.

6.2.3 Existing Algorithms for Mirror Selection

The mirror selection algorithm is also handled manually today. A user uses the links on the HTML page to manually select one of the mirror sites. The disadvantage of this approach is that it does not involve any amount of load balancing or randomization to evenly spread request load across the collection of mirror locations. For example, if each user manually selects the first location on a list of mirror sites, this would lead to a "hotspot" where all users attempt to download the object from the same mirror location. This would leave one mirror location overloaded and other mirror locations underutilized.

6.3 Our Solution: LookingGlass

LookingGlass is a HTTP server selection tool that addresses the above problems in the mirroring of web content. It addresses the problem of mirror advertisements by automatically collecting information about the location of mirrored objects and distributing this information to other mirror locations as well as web clients. It addresses the problem of mirror ranking by using SPAND's application-level metrics such as response time as the metric for mirror ranking. It addresses the problem of mirror selection by using randomization with weightings in proportion to the rankings returned by SPAND in its mirror selection. LookingGlass uses SPAND as the repository for both mirror location and mirror location performance information. In the following sections, we describe these mechanisms and algorithms in more detail.

The web client component of LookingGlass is implemented as a Muffin filter [52] that intercepts HTTP requests, identifies mirrored objects, selects the most appropriate mirror location for those objects and downloads objects from that mirror location. The web server component of LookingGlass is a server side daemon that periodically communicates with other mirrors to exchange mirror location information.

6.3.1 Mechanisms for Mirror Advertisement

There are two major flaws with the current method for advertising mirrored objects. First, the mirrors are manually advertised by creating a HTML page that lists the mirror locations, which burdens the user with the responsibility of keeping the list up-to-date. Second, the mirrors are advertised from a single location, usually from the primary site. Our solution addresses both of these problems by providing a transparent way to advertise mirrored objects and a distributed algorithm for disseminating this information.

We can make the process of advertising mirrors transparent by using an existing mechanism in HTTP for variant selection. As described in the last chapter, HTTP's Transparent Content Negotiation mechanism [37] is used to advertise alternate versions of web objects by adding an additional response type (a "Multiple Choices" response) to HTTP requests. This response lists the possible choices for the requested object and their characteristics. An example of this is shown below, where the web page `http://www.imdb.org` is mirrored at three locations in the United States, England, and Japan:

```
HTTP 300 Multiple Choices:
Date: Tue, 11 Jan 1996 20:02:21 GMT
TCN: List
Alternates:
 {``http://us.imdb.org/index.html''
          {type text/html} {length 2176}},
 {``http://uk.imdb.org/index.html''
          {type text/html} {length 2176}},
 {``http://jp.imdb.org/index.html''
          {type text/html} {length 2176}}
```

When a web client receives a multiple choices response, it chooses one of the alternates and retrieves it from the specified location. This mechanism allows for transparent advertisements of mirrored web objects in a way that is completely transparent to the user.

One important advantage of this technique is that it enables mirroring on a very fine-grained (i.e. per web object) basis. This has significant advantages over host-based solutions such as mapping a single Domain Name System (DNS) name to multiple IP addresses. The disadvantage of a DNS-based solution is that the entire web or FTP site must be replicated at each mirror location using exactly the same pathnames. If someone wants to mirror a single object using the host-based solution, they have no choice but to mirror the entire web or FTP site.

Using HTTP Transparent Content Negotiation provides a transparent way for mirror locations to notify web clients about the locations of mirrored objects. In this approach, however, web clients must still contact the primary site to receive the list of alternates, and the administrator of the primary site must still manually create the list of alternate objects. To address this problem, we use a distributed algorithm similar to USENET to disseminate mirror information, as shown in Figure 6.1.

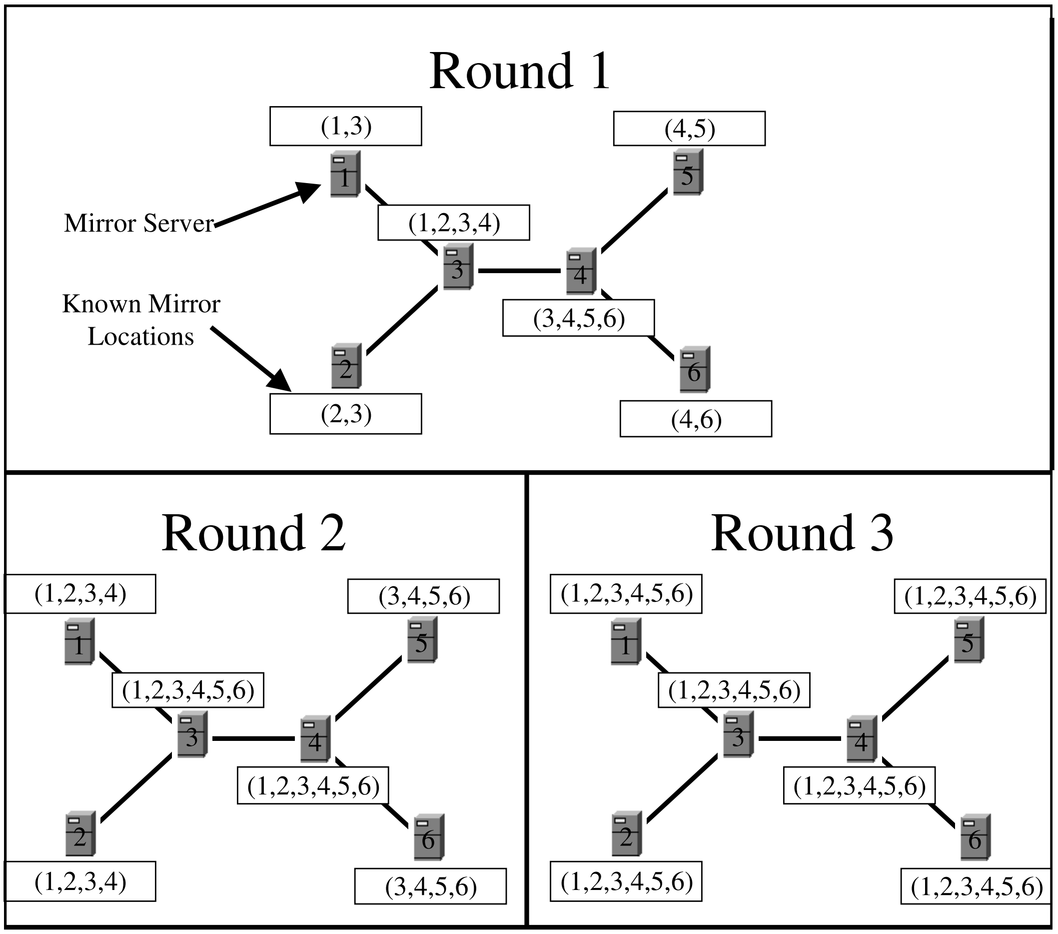

Figure 6.1: Distributed algorithm for disseminating mirror information

The set of mirror sites cooperate in a distributed way to exchange mirror information. For each locally maintained web object, the server keeps a permanent list of neighbor mirror locations for the locally maintained object. The server also keeps a soft state list of all mirrored locations for the object. This list is initially the same as the permanent list. Periodically, the server contacts the neighbor locations and they exchange and merge soft state lists of mirror locations. With each round of exchanges, each object's mirror information spreads throughout the network of mirror servers. A client can then contact any mirror to obtain the full list of mirror locations for a particular object.

Using this technique, a web client can attempt to fetch any mirrored web object and receive the full list of mirror locations for that object. However, this involves an unnecessary extra client-to-mirror interaction that we would like to avoid if possible. To solve this problem, we cache the list of mirror locations for each object in the SPAND repository, and the client-side component of LookingGlass consults this cache before contacting a distant mirror to determine if alternate locations exist for a particular web object. This avoids unnecessary communication with distant mirrors that may add to the total response time.

6.3.2 Metrics for Mirror Ranking

The primary metric we use for the ranking of mirrors is the typical document download time as described in Section 4.3.2. In cases where a client experiences a failure in retrieving a document from a mirror, we report a large constant value which is greater than the worst possible download time.

To describe "typical" download time, we can either use the median or mean download time. By using the median, our metric is not influenced by outlier values that result from very slow download times or failures. By using the mean, our metric will be influenced by these outlier values. In Section 6.5, we show how the choice of median versus mean in determining typical behavior affects client performance.

6.3.3 Algorithm for Mirror Selection

Once a client has the list of mirror locations, it must select one of the locations from which to download the web object. Regardless of the metric used to choose the best mirror, our algorithm must have the following features:

- The algorithm should recommend well-connected mirror locations over poorly-connected mirror locations.
- The algorithm should avoid recommending a single best mirror location to clients. This could create a "hotspot" where all clients attempt to use a single mirror and overload it.
- Because our system relies on passive measurements to make decisions about the choice of mirror location, we must make sure that the network performance information for lowly ranked mirror locations does not become out-of-date.

These goals can sometimes conflict. For example, to keep timely performance information for lowly ranked mirror locations, it will be necessary to occasionally direct clients to these mirrors to refresh their performance information. This may degrade client performance.

To achieve these goals, we use a combination of randomization and the offering of multiple choices in our algorithm. For each mirror location, we use the average response time x_i to construct a weight for that mirror w_i. The weights of all locations sum to one (i.e., $\Sigma_j w_j = 1$), and the weight is a proportional to the inverse of the response time, i.e. a mirror with a lower median response time should receive more weight than a mirror with a higher median response time. We experimented with five different weighting functions, ordered below by the degree to which they weight lower response times:

- Uniform: $w_i = 1$
- Inverse: $w_i = 1/x$
- Inverse Squared: $w_i = 1/x^2$
- Exponential: $w_i = e^{x_i}$
- Exponential-Uniform Hybrid: $w_i = e^{x_i}$ or $w_i = 1$ with probability z.
- Hyperexponential: $w_i = e^{e^{x_i}}$

In Section 6.5, we show how the choice of weighting function affects client performance.

Once we calculate the normalized weights, we randomly select a mirror location according to the weights and recommend that mirror to the client. This distributes load across multiple mirrors and assures that lowly ranked mirrors are still visited occasionally.

However, sending clients to lowly-ranked mirrors may force them to take much longer to retrieve objects than if they did not use our system at all and simply chose a mirror at random. Because we rely on a large client population to maintain our repository of performance measurements, we must provide an incentive for clients to use our system even in cases where they visit lowly-ranked mirror locations. As a result, we provide clients with a backup mirror location to use if the primary mirror's performance is much worse than the best possible performance.

More specifically, our algorithm returns the following information:

- A primary location, chosen using the weighting function described above.
- A backup location, which is always the best known mirror location.
- An experiment time, which indicates the minimum time that the client must attempt to retrieve content from the primary location.
- A target number of bytes, which indicates a performance level that should be considered acceptable for the primary location.

The target number of bytes is initially fixed at a fraction of the total document size. The results in Section 6.5 use a fraction of 10%. To calculate the experiment time, we calculate how long it would take for the best known mirror to transfer that fraction of the document. When the client-side LookingGlass component receives this information, it starts downloading the content from the primary location. If LookingGlass has not received the target number of bytes from the primary location at the end of the experiment time, it switches to the backup location to complete the transfer. The proxy always generates a performance report for the primary mirror. In addition, the proxy also generates a performance report for the backup mirror if it was forced to use it.

This algorithm has the attractive property that it bounds the amount of time that a client will attempt to use a possibly lowly-ranked mirror before giving up on it and switching to the backup mirror. Regardless of how poor the performance is for a particular lowly-ranked mirror, a client will not wait for more than the experiment time before switching to another mirror. This meets both goals of maintaining reasonable client performance and keeping up-to-date network performance statistics.

To account for variations in network performance, our algorithm also divides the target number of bytes by a constant called the *aggressiveness factor* before returning it to the LookingGlass. This constant can be any value greater than 1. By changing the value of the aggressiveness factor, we make clients less or more aggressive in giving up on a lowly ranked mirror and switching to the backup location. Greater values of the aggressiveness factor make clients less aggressive, because this decreases the target number of bytes. In turn, this makes it more likely that the client will receive the target number of bytes during the experiment time and will continue to use the primary mirror. In Section 6.5, we show how the choice of this constant affects client performance.

6.3.4 Putting it All Together: Example Object Download

Figure 6.2 shows an example web object download using LookingGlass. At the server side, the collection of mirrors exchange lists of mirror locations for each web object. At the client side, the process begins when the web client starts to fetch a web object. This HTTP request is intercepted at the web client by the LookingGlass client-side component. Before completing the request, the LookingGlass first contacts the SPAND performance server to determine if the desired object has any alternate locations. If no list exists at the performance server, the client-side component contacts the original mirror location to see if alternates exist for the desired web object. Using the list of alternate locations, the client-side LookingGlass component sends performance queries to the performance server to obtain estimates of the download time for each of the mirror locations. Using these estimates, the client-side component uses the weighted randomized selection algorithm described above to choose the primary location, backup location, experiment time, and target number of bytes. The client-side component then contacts the primary location and downloads the object. If after the experiment time has passed, the number of bytes received from the primary location is lower than the target number of bytes, the LookingGlass client switches to the backup location. Otherwise, it continues to download the object from the primary location. After the transfer has completed, the client-side component sends performance reports to the performance server that indicate the response time for the primary location

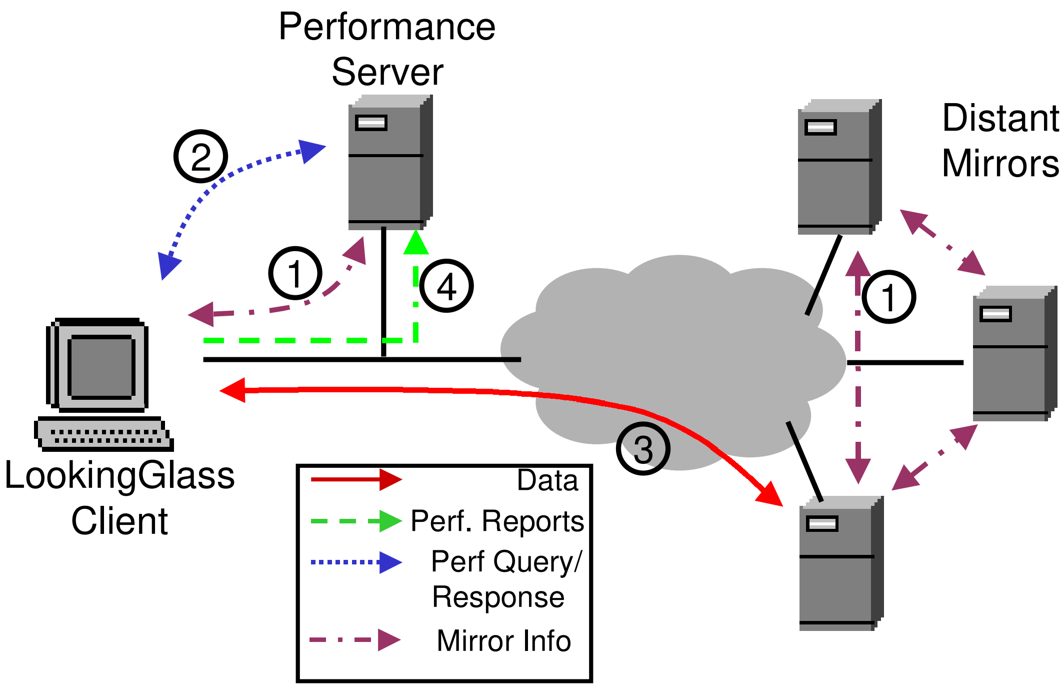

Figure 6.2: Example download using LookingGlass.

and (if used) the backup location.

6.4 Experimental Methodology

To test the effectiveness of LookingGlass, we used a combination of trace analysis and implementation for our experiments. To simulate application-level behavior, we used application-level traces collected at Carnegie-Mellon University [54]. The traces were generated by using `lynx`, a text-based web client application, to download web objects from a collection of web servers that mirrored the same content. The clients downloaded objects in a series of rounds. In each round, a client retrieved the same object from each mirror location and recorded the document download time for each mirror. After a round was complete, the client slept for a random amount of time taken from an exponential distribution with a mean of 30 minutes added to a constant 30 minutes. In total, the data trace spanned 3 weeks.

The results we present here are from a UC Berkeley client that downloaded a 400 KByte image from the Mars Pathfinder mission from a collection of 20 mirror locations. The primary mirror location was at the Jet Propulsion Laboratory, and the alternate mirror locations were as close as the Bay Area (e.g., Sun, HP, SGI) and as far away as Hawaii (The Hawaii Institute of Geophysics and Planetology). Table 6.1 shows the complete list of mirror sites.

To conduct our experiments, we use the traces to indicate what the client performance would be if it used LookingGlass to select the most appropriate mirror. At the

mars.sgi.com	www.sun.com/mars
entertainment.digital.com/mars/JPL	mars.novell.com
mars.primehost.com	mars.hp.com
mars.excite.com/mars	mars1.demonet.com
mars.wisewire.com	mars.ihighway.net
pathfinder.keyway.net/pathfinder	mpfwww.arc.nasa.gov
mars.jpl.nasa.gov	www.ncsa.uiuc.edu/mars
mars.sdsc.edu	laguerre.psc.edu/Mars
www.ksc.nasa.gov/mars	mars.nlanr.net
mars.catlin.edu	mars.pgd.hawaii.edu

Table 6.1: Mirror Locations used for Experiments

beginning of each round, the client uses LookingGlass to determine the primary and backup mirror locations, the experiment time, and the target number of bytes. We use the information from the trace to determine how long the client would actually have taken to complete the transfer, including downloading the document from the primary mirror and, if necessary, switching to the backup mirror if the client has not received the target number of bytes by the end of the experiment time. The download times for the primary (and if the client switched, the backup) mirror locations *only* are added to the collection of performance information for the next round. To make sure that every mirror has some performance information for each mirror, we initially visit each mirror location exactly once before using consulting the SPAND performance server and using the randomized selection algorithm.

For each round, we report the ratio of the actual time taken by the client and the minimum observed download time for all mirrors in that round. Ideally, our system will direct clients to well-connected mirror locations, resulting in download times that are close to the minimum time for any mirror and as a result, ratios that are close to one. We then examine the distribution of these ratio statistics to determine how well our algorithm does at selecting well-connected mirrors. A distribution with a majority of its ratio measurements near one implies a configuration that chooses near-optimal mirror locations.

6.4.1 Operating in the Presence of Isolated Infrequent Measurements

There are two limitations with the trace we used that limit the accuracy of our results. First of all, the trace we used had relatively infrequent data samples. A particular mirror in the trace was only visited on average once per hour. In addition, our trace only includes measurements made by a single host. There is no actual sharing of performance information between a population hosts.

To overcome these two limitations, we modified the SPAND performance server to use all previous performance reports for a given web object when calculating a performance response. This means that the results below are very conservative, because they incorporate a significant amount of temporal noise in the form of daily and weekly variations in performance and do not benefit from the collective knowledge gained from a large client population. In a real implementation, we would have more up-to-date information about these mirror sites because we could benefit from the shared access patterns of a larger client population.

Despite this amount of temporal noise and lack of shared measurements, however, we see in the next section that our system still does an excellent job at identifying near-optimal mirrors from which to download web objects.

6.5 Results

In this section, we evaluate the performance of LookingGlass in returning good mirror locations to clients. We begin by examining the sensitivity of LookingGlass's mirror selection algorithm to several control parameters:

- The use of the mean vs. median to represent "typical" behavior for a particular mirror location.
- The choice of weighting function (i.e., uniform, exponential, etc.) in ranking mirror locations to determine the "best" mirror to contact.
- The scaling constant for the experiment time that controls the aggressiveness of clients in giving up on mirror locations.

For each control parameter, we hold two parameters fixed and vary the third to determine the best choice for that parameter. This allows us to determine the best possible configuration of LookingGlass. Then, using the best choices for these control parameters, we compare the performance of LookingGlass against alternate algorithms for mirror ranking and selection. In particular, we consider the following alternate metrics:

- Using geographic location to approximate good performance. In this approach, the client chooses a mirror that is closest geographically.
- Choosing the mirror that is the least number of network hops away from the client.
- Randomly selecting a different mirror for each round.
- Always choosing the primary mirror site for the content.

This allows us to compare the improvement that a client would observe as a result of using LookingGlass instead of an alternate criteria for mirror selection.

For each experiment, we repeated the trace analysis described in Section 6.4 and calculated the Cumulative Distribution Function (CDF) of ratios between optimal and actual download time for a given round. By comparing the CDF curves for alternate configurations or policies, we can determine which configuration or policy performs better.

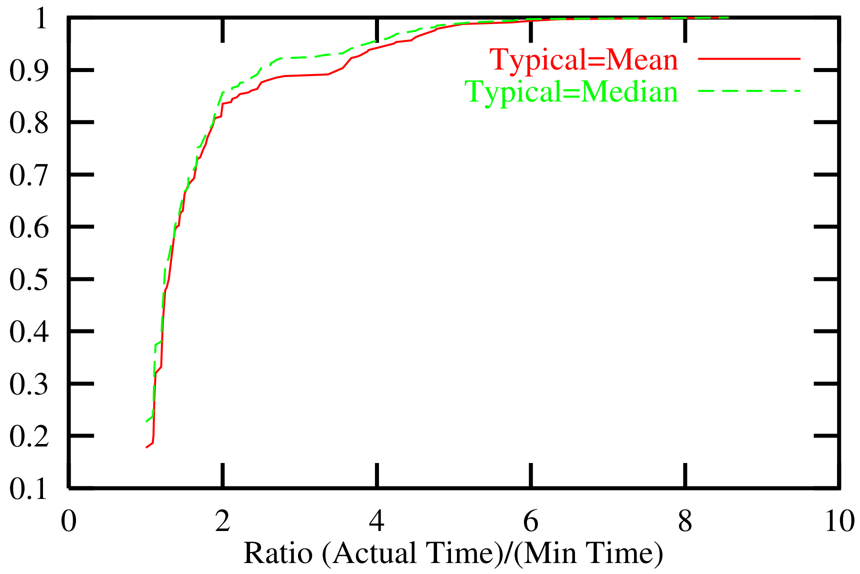

Figure 6.3: Effect of using the mean vs. median to report typical client performance.

6.5.1 Using Median vs. Mean for Ranking Metric

Figure 6.3 shows the results of using the median vs. the mean in reporting typical client download times. We assume that the weighting function is an exponential-uniform hybrid with a 5% probability of choosing a uniform distribution. We assume that the aggressiveness constant is 5–if a client receives less than 20% of the target number of bytes in the experiment time, it gives up and switches to the backup mirror.

The graph consists of two CDF curves, one for the system that uses the mean, and one for the system that uses the median. The X axis shows the ratio between the actual and minimum possible download time for a given round, and the Y axis represents cumulative probability. We see that there is little difference between using the mean or median in reporting typical performance. The median works slightly better because it it is not influenced by outlier values. Based on this information, we chose to use the median for subsequent experiments.

6.5.2 Choice of Weighting Function

Figure 6.4 shows the results of using different weighting functions in ranking mirror locations. We use the median to report typical performance, and we assume that the aggressiveness factor is 5.

We see that the hyperexponential and uniform weighting functions lead to the worst performance, as they place too much and too little weight, respectively, on the tails of the distribution of hosts. The uniform weighting function does not take any performance information into account, and clients often visit mirror locations that have bad performance. The hyperexponential weighting function assigns too much weight to hosts that are initially

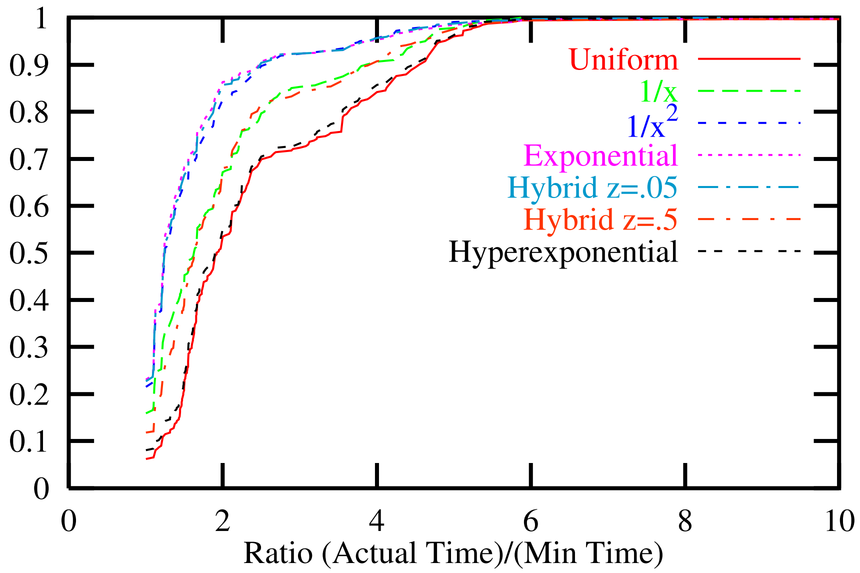

Figure 6.4: Effect of choice of weighting function in ranking mirror locations.

highly ranked. As a result, clients continually visit the same small set of mirrors and do not discover other mirrors that may have better performance.

The $1/x$ weighting function performs better than the uniform and hyperexponential functions but worse than the exponential and $1/x^2$ functions. As expected, the hybrid function ranges between the exponential and uniform functions depending on the value of the mixing constant z.

Of all these functions, we see that the $1/x^2$, exponential, and exponential-uniform hybrid weighting functions perform equally well at maximizing client performance. We chose to use an exponential-uniform hybrid weighting function with $z = .05$ for the rest of our experiments.

6.5.3 Choice of Aggressiveness Factor

Figure 6.5 shows the sensitivity of the aggressiveness factor on client performance. As the factor decreases, clients become more aggressive, switching to the backup host even for small differences in performance. As the factor increases, clients become more tolerant of significant differences between the current and backup mirror's performance. We use the median to report typical performance, and we assume that the weighting function is a exponential-uniform hybrid with a 5% probability of choosing a uniform distribution.

As expected, as clients become less aggressive, they continue to download content from poorly-performing mirrors and their performance degrades. Based on this information, we chose to use an aggressiveness factor of 1 for subsequent experiments

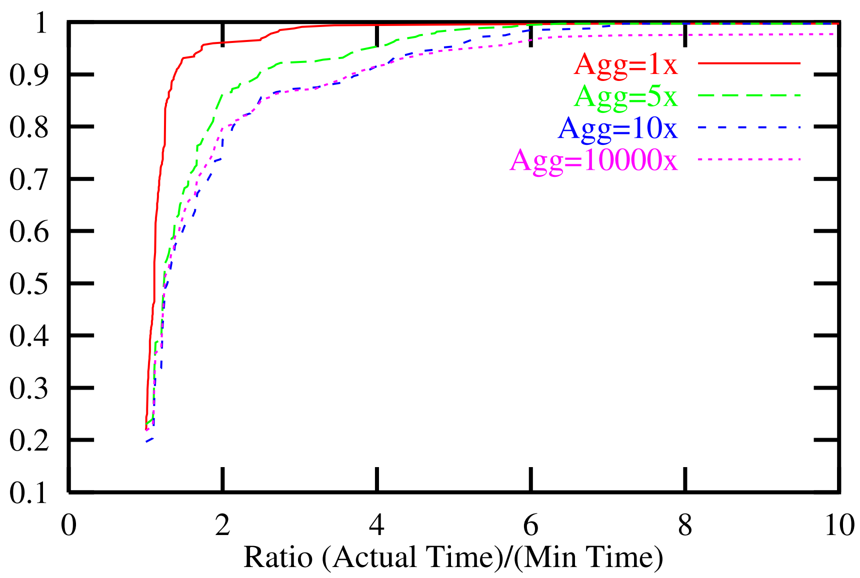

Figure 6.5: Effect of aggressiveness factor on client performance.

6.5.4 Choice of Mirror Selection Policy

Figure 6.6 summarizes the maximum benefits of using LookingGlass over alternate ranking metrics that do not take network performance information into account. There are several CDF curves, each representing a different policy in choosing a mirror to contact. When using LookingGlass, we use the median to report typical performance, assume that the weighting function is a exponential-uniform hybrid with a 5% probability of choosing a uniform distribution, and use an aggressiveness factor of 1. As described previously, we compared LookingGlass to several alternate policies:

- Using geographic location to approximate good performance. In this approach, the client chooses a mirror that is closest geographically.
- Choosing the mirror that is the least number of network hops away from the client.
- Randomly selecting a different mirror for each round.
- Always choosing the primary mirror (The Jet Propulsion Lab) of the content.

Because there were several mirror locations in the same geographic area as the client, we show the performance for the best performing and worst performing mirror locations that could easily be identified (by domain name) as located in the Bay Area.

We see that the policy that uses hop count to select a mirror location performs the worst of all policies considered. The median download time using this policy is a factor of four away from the best possible download time. This is especially interesting because several commercial wide-area server selection systems such as Cisco's DistributedDirector product use hop count and other routing metrics as the selection policy.

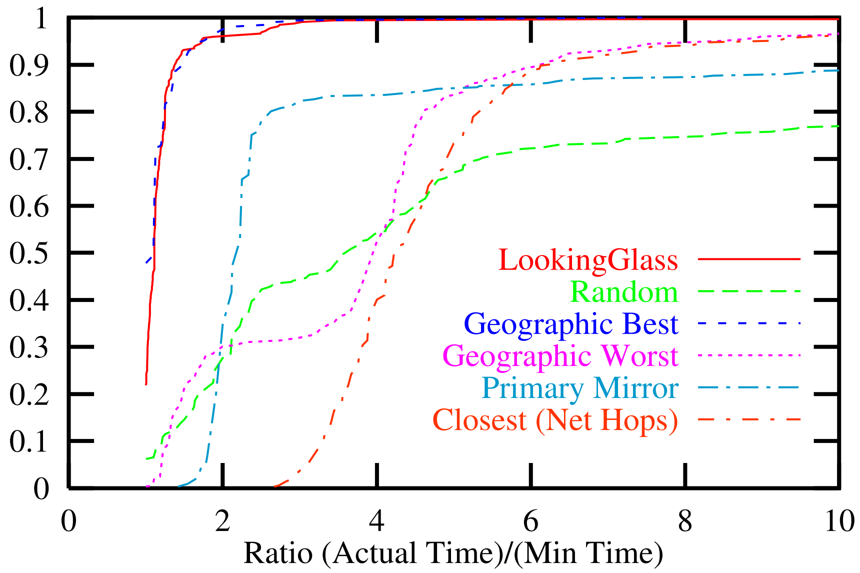

Figure 6.6: Effect of choice of ranking metric on performance

A random policy actually results in a median download time that is slightly better than the hop count policy (3.5 as compared to 4.0). However, using a random policy results in much worse worst-case behavior, which can be seen by examining the tail of the distribution. Approximately 20% of the time, using a random policy leads to more than a factor of 10 degradation in download as compared to the best possible download time.

A policy that always chooses the primary location results in relatively consistent but suboptimal performance. 80% of the time, the download time is approximately a factor of 2 away from the best possible download time. However, this policy also exhibits bad worst-case behavior. 15% of the time, choosing the primary mirror leads to performance that is a factor of 6 or more away from the best possible performance.

A policy that depends on geographic hints may or may not lead to good performance, depending on the "nearby" mirror that was chosen. We see that the best-performing geographically close host, the mirror located at Hewlett Packard, has consistently superior performance and a very low likelihood of poor performance. If a client chose to use this mirror, they would see very good performance. However, if a host chose a different "near-by" mirror, they would observe see such good performance. The worst-performing nearby mirror, the mirror located at Sun Microsystems, has two distinct modes, one at a factor of 1.5, and one at a factor of 4. The median download is a factor of four away from the best possible download time. From this, we see that geographic hints are sometimes indicative of good performance, but sometimes lead to suboptimal results.

We see that LookingGlass performs equal to or better than every other policy we considered. In addition, LookingGlass performs very well in an absolute sense, as a large fraction of the ratios are close to one. For 90% of the rounds, the download time using LookingGlass is within 37.5% of the minimum possible download time for that round (i.e.

the ratio is 1.375 or less).

The main result is that by using network performance information, LookingGlass allows clients to download mirrored web objects faster than using other policies for mirror selection, and in near-optimal time.

6.6 Conclusion

In this Chapter, we have described LookingGlass, a Mirror Selection Tool that uses SPAND to drive its policies for Mirror Selection. We divided the generic problem into three sub-problems: *Mechanisms for Mirror Advertisement*, *Metrics for Mirror Ranking*, and *Mirror Selection Algorithms.* We described the limitations of current techniques for wide-area mirror selection, showing that in most cases, current systems solve these sub-problems via manual techniques for mirror advertisement and relatively static policies for mirror ranking and selection.

We then presented how LookingGlass solves the problems of mirror advertisement, ranking, and selection. To solve the problem of mirror advertisement, we re-use the Transparent Content Negotiation framework described in Chapter 5 to inform web clients of mirror locations. To solve the problem of mirror ranking, we use SPAND's application-level performance metrics. To solve the problem of mirror selection, we use a randomized selection scheme that meets the dual goals of selecting well-connected mirrors, balancing client load across mirrors to avoid hotspots, and keeping SPAND's repository of network performance information up-to-date.

We then presented trace analysis experiments that show the performance of LookingGlass compared to alternate policies for mirror selection. We found that LookingGlass performed better than every other policy we considered, and that 90% of the time, LookingGlass chose a mirror location whose performance was within 37% of the optimal mirror location.

In the next chapter, we summarize the thesis and present ideas for future work.

Chapter 7

Conclusions and Directions for Future Work

In this chapter, we summarize the thesis, as well as general principles learned while developing the thesis, and present ideas for future work.

7.1 Conclusion

In this thesis, we have presented a solution to the problem of enabling and developing adaptive networked applications. The Internet's decentralized nature and emphasis on simplicity and efficiency over features and complexity leads to inherent network heterogeneity, both in the access technologies used to connect hosts to the Internet and the capabilities of Internet Service Providers that connect hosts to each other. For applications to effectively communicate in this heterogeneous environment, they need to be *adaptive*, measuring the state of the network and making application-level decisions based on those changes. Examples are:

- *Mirror Selection* to choose the best of a number of servers that replicate the same content.
- *Content Negotiation* to match object fidelity to current network characteristics.
- *Feedback* to the user indicating the expected impact of user-level choices, allowing the user to make informed choices in the absence of automatic adaptation.

However, these adaptation mechanisms are not a complete solution for building adaptive applications. Applications also need a way to measure the characteristics of the network path between hosts and to use this information to drive their adaptation mechanisms. Providing mechanisms and policies to solve this problem has been the goal of our thesis.

This thesis presents three main contributions toward solving this problem. The first contribution is a Network Measurement Service called SPAND (**S**hared, **PA**ssive, **N**etwork performance **D**iscovery). Using SPAND, applications can determine the network performance to distant hosts and use these measurements to drive their adaptation decisions. In SPAND, applications make passive measurements of network performance by observing their own application-to-application communications. They create *performance reports* that summarize this performance and send these reports to a per-domain centralized repository of performance information called a *Performance Server*. The performance server is responsible for maintaining the collection of reports. In addition, the performance server responds to requests for network performance information called *Performance Queries*. The performance server determines the relevant performance reports to use in answering the query and returns a *Performance Response* that indicates the expected performance to a distant host.

SPAND incorporates three key design decisions that contrast with the design choices made by previous network measurement efforts. First, SPAND relies on *Shared* measurements. Applications explicitly cooperate together to increase the accuracy and availability of network performance information. Second, SPAND relies on *Passive* measurements. Our system does not introduce unnecessary probe traffic into the network that could slow down the useful work of routers or servers. Instead, SPAND only uses application-to-application traffic to make measurements of network performance. Third, SPAND relies on *Application Specific* measurements. Instead of using network-level measurements such as latency, hop count, or network available or peak bandwidth as approximations to actual application level performance, we measure exactly what applications are most interested in–application-level performance.

In this thesis, we also introduced the concept of *Measurement Noise*, the difference between predicted and actual performance, and showed how measurement noise affects the granularity of application level decisions. We categorized measurement noise into *network noise*, the variation that is inherent in the network, *sharing noise*, the variation that results from sharing performance information between hosts, and *temporal noise*, the variation that results from using past information to predict current performance. We presented measurements of network performance for actual network clients and categorized measurement noise into network, sharing, and temporal sources, showing that network noise is usually the largest contributor to variation in performance.

We also presented application-independent results of SPAND that show that our system works well at providing clients with relevant, accurate network performance information. Our system can quickly provide meaningful performance responses to over 95% of performance requests, and these performance responses are usually within a factor of 2 of actual observed performance. This difference can be attributed to the *network noise*, or inherent variation, in the state of the network.

The second contribution of our thesis is *SpandConneg*, a HTTP Content Negotiation application that uses SPAND to drive its choice of data representation. SpandConneg utilizes content negotiation at the both at the client side of the network and at the server side of the network. At the client side of the network, SpandConneg allows clients to trade off web object quality for a consistent response time. Using SpandConneg, web clients specify the maximum time they are willing to wait to download a web page. Using SPAND's network performance measurements, SpandConneg determines the highest quality representation of the web object that can be downloaded from the web server and still meet the user's time constraint. SpandConneg then downloads that variant of the web object from the web server.

Measurements of the client-side implementation of SpandConneg show that it reduces the likelihood of excessive download times as compared to not performing content negotiation at all. 35% of the time, clients that do not negotiate must wait for 30 seconds or more to download pages. Clients that use content negotiation, on the other hand, reduce the chances of this to 10%. In addition, SpandConneg improves the typical page download time observed by clients. The median transfer time of a web page drops from 16 to 6 seconds using content negotiation. The presence of network noise in our performance measurements limits the effectiveness of completely meeting the user-specified goal of a constant download time, however. Using SpandConneg, approximately 60% of downloads met the user-specified goal of a download time less than ten seconds. Users must be conservative about network performance if they wish to obtain hard guarantees for application level performance.

At the server side of the network, SpandConneg is used to allow web servers to handle an unexpected burst of web clients by reducing the fidelity (and as a result, the size) of documents sent to individual clients. This reduces the outgoing traffic requirements of the web server and allows it to handle a greater number of clients. Our implementation includes simple mechanisms for generating multiple representations of web objects as well as limiting the size of negotiated web objects. It also includes a policy program that observes the outgoing traffic from the web server and adjusts the quality of web objects as necessary.

Measurements of the server-side component of SpandConneg show that it effectively allows web servers to increase the number of clients they can support. If 60% of outgoing traffic comes from negotiable documents, the web server throughput and maximum client population increases by 50%. If 90% of outgoing traffic comes from negotiable documents, the increases are even more dramatic. The throughput and client population increases by 450% or more when using content negotiation.

The third contribution of our thesis is LookingGlass, a web mirror selection tool that uses SPAND's network performance measurements to drive its choice of web server to contact. LookingGlass solves the problem of dissemination of web objects from multiple locations. LookingGlass incorporates a transparent mechanism to inform web clients about the location of mirrored web objects as well as an automated process to disseminate locations of mirrored objects between mirror locations. To rank mirror locations, LookingGlass uses SPAND's network performance measurements of actual observed network performance. LookingGlass also uses a randomized mirror selection algorithm that meets the conflicting goals of evenly spreading client requests across the mirror locations, keeping passively collected measurements of performance up to date, and bounding worse-case client performance.

Measurements of LookingGlass show that it does a near-optimal job of selecting fast mirrors to contact. When presented with a choice of mirror locations that replicate the same object, 90% of the time, clients that use LookingGlass can download objects within 37.5% of the best possible download time. In addition, LookingGlass's use of actual network performance measurements allows it to do much better than other metrics such as hop count, geographic location, or using the primary mirror site. Without LookingGlass, 90% of transfers are at a minimum 600% and at a maximum infinite (failure) away from the optimal download time, an improvement of ¿15-fold. In addition, the median LookingGlass download time is within 10% of optimal, compared to 200%-400% for other approaches, an improvement of 20-40 fold. This shows that despite the presence of temporal and network noise, SPAND works well at selecting ideal mirrors.

7.2 General Principles

While developing SPAND and its applications, we synthesized several general principles and lessons that can be used by other developers of network measurement systems and adaptive applications:

- **Passive measurements are are effective in measuring performance:** One important lesson we learned was the value of making passive measurements of network performance. In addition to placing less load on the network than active measurements, passive measurements automatically measure the "right" network performance statistic, because actual application-to-application traffic is used to measure performance. In addition, the use of passive measurements made the experimental evaluation process easier because we could collect offline packet traces and use them as a repeatable workload for our evaluations.

- **Trace analysis is better than simulation for evaluating systems:** This principle is closely related to the use of passive measurements. While measuring the performance of SpandConneg and LookingGlass, we deliberately chose to rely on a combination of trace analysis and implementation whenever possible instead of using network simulations. By using trace analysis, we did not have to generate artificial workloads that accurately reflected actual performance. In addition, by using trace analysis, we could more realistically measure the implications of application level adaptation. For example, in SpandConneg, we measured the download time for negotiated web objects by examining the packet trace to determine how long it took for the network to complete a fractional portion of a web object transfer. Not all systems can be evaluated in this way, but for those that can, the use of traces leads to a more realistic evaluation process.

- **You have to implement the system more than once to get it right:** The design and implementation of the SPAND architecture went through several revisions. The performance server was implemented twice in C++ and finally re-implemented in Java. The packet capture host was constantly reworked as we added support for new application classes. Each redesign of the architecture benefited from the lessons learned in the previous implementation. For example, the performance report format was initially inflexible and not application specific. We constantly found ourselves adding new fields to the report format as we added new types of network measurements. We finally decided to throw out the initial implementation and replace it with the more generic flexible messaging interface described in Section 4.2. By starting over from scratch, but keeping in mind the lessons learned from previous implementations, we resulted in a developing a more robust and better designed architecture.

7.3 Directions for Future Work

There are several possible directions for future work in improving the performance of the SPAND network measurement service:

- SPAND's current performance server relies on relatively static policies when determining the relevant set of performance reports to use when calculating a performance response. It assumes that the H most recent performance reports are the relevant set of reports. Instead of relying on static policies, the performance server could incorporate feedback by comparing performance responses with the actual performance experienced by applications. For example, it could change the cut-off time for garbage collecting old performance reports based on feedback from clients. This potentially improves the accuracy of SPAND's responses by reducing the amount of temporal noise in performance measurements.

- If SPAND's performance server does not have any performance reports for a given host, it returns that no information is available. As an alternative, it could perform a lightweight active probe such as Packet Pair [44] or Cprobe [19] to construct an approximation for application-level performance. This significantly increases the availability of information for infrequently visited distant hosts. The challenge is performing these active probes in a way that does not significantly disturb application-level traffic on the path to the distant host.

In addition, there are examples of new adaptive applications that can utilize SPAND's network measurement service:

- Many search engines currently return documents ordered by their relevance to the search query, with no consideration of the time it would take for clients to retrieve them. For example, a highly relevant document that takes a long time to download may actually be less useful than a less relevant document that could be retrieved more quickly. As an application of SPAND, a web client could receive a list of documents from a search engine and re-score the documents based on the expected time to retrieve the documents before presenting them to the user.

- The current classes of performance reports and applications of SPAND have focused on measuring and improving the performance of web object retrievals. Another datatype that is increasingly used is streaming media clips such as RealMedia or Microsoft NetShow streams. SPAND could also measure application-level performance for these. Many of the same types of adaptation decisions, such as content negotiation and server selection, could be performed. The challenge is in devising an application-level performance metric that is useful for these data types.

7.4 Software availability

Parts of the software used in this dissertation is available in source-code form in several locations. Implementations of the SPAND architecture and the SpandConneg application are available as a part of the SPAND 2.0b2 toolkit distribution. This is available from the SPAND web site at:

`http://spand.cs.berkeley.edu/software/spand-2.0b2.tar.gz.`

An additional copy of the SPAND distribution is also at a mirror of the SPAND web site:

`http://www.cs.berkeley.edu/ stemm/spand/software/spand-2.0b2.tar.gz.`

Documentation for the software is automatically created as a part of building the distribution. In addition, on-line documentation for the software toolkit is available at the SPAND site and its mirror:

`http://spand.cs.berkeley.edu/doc`

`http://www.cs.berkeley.edu/ stemm/spand/doc`

Bibliography

[1] P. Albitz and C. Ciu. *DNS and BIND in a Nutshell.* O'Reilly & Associates, 1992.

[2] E. Amir, S. McCanne, and R. H. Katz. An Active Service Framework and its Application to Real-time Multimedia Transcoding. In *Proc. ACM Sigcomm*, September 1998.

[3] E. Amir, S. McCanne, and H. Zhang. An Application Level Video Gateway. In *Proc. ACM Multimedia*, November 1995.

[4] M Arlitt and C. L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proc. ACM SIGMETRICS '96*, May 1996.

[5] H. Balakrishnan, S. Seshan, M. Stemm, and R.H. Katz. Analyzing Stability in Wide-Area Network Performance. In *Proc. ACM SIGMETRICS '97*, June 1997.

[6] P. Barford. surge – Scalable URL Reference Generator. http://www.cs.bu.edu/students/grads/barford/Home.html, 1998.

[7] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proc. ACM Sigmetrics '98*, 1998.

[8] S. Bhattacharjee, M. H. Ammar, E. W. Zegura, V. Shah, and Z. Fei. Application layer anycasting. In *Proc Infocom '97*, April 1997.

[9] Bing Home Page. http://spengler.econ.duke.edu/ ferizs/bing.html, 1996.

[10] J.C Bolot. Characterizing End-to-End Packet Delay and Loss in the Internet. *Journal of High Speed Networks*, 2(3):305–323, 1993.

[11] J.C Bolot. End-to-End Packet Delay and Loss Behavior in the Internet. In *Proc. ACM SIGCOMM '93*, San Francisco, CA, Sept 1993.

[12] N. Borenstein and N. Freed. *MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies*, Sep 1993. RFC 1521.

[13] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F Schwartz. The Harvest Information Discovery and Access System. *Computer Networks and ISDN Systems*, 1995(28):119 – 125, 1995.

[14] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, M. F Schwartz, and D. Wessels. Harvest: A scalable, customizable discovery and access system. Technical Report CU-CS-732-94, Computer Science Department, University of Colorado, March 1995.

[15] R. T. Braden. *Requirements for Internet Hosts – Communication Layers*. Information Sciences Institute, Marina del Rey, CA, October 1989. RFC-1122.

[16] S. R. Caceres, N. Duffield, J. Horowitz, D. Towsley, and T. Bu. Multicast-Based Inference of Network-Internal Characteristics: Accuracy of Packet Loss Estimation. In *Proc. IEEE Infocom*, March 1999.

[17] Caida Taxonomy of Network Probing Tools. http://www.caida.org/Tools/taxonomy.html, 1999.

[18] R. L. Carter and M. E. Crovella. Dynamic Server Selection using Bandwidth Probing in Wide-Area Networks. Technical Report BU-CS-96-007, Computer Science Department, Boston University, March 1996.

[19] R. L. Carter and M. E. Crovella. Measuring bottleneck-link speed in packet switched networks. Technical Report BU-CS-96-006, Computer Science Department, Boston University, March 1996.

[20] CERT. CERT Advisory CA-98.01, smurf IP Denial-of-Service Attacks. http://www.cert.org/advisories/CA-98.01.smurf.html, 1998.

[21] J. Challenger, P. Dantzig, and A. Iyengar. A Scalable and Highly Available System for Serving Dynamic Data at Frequently Accessed Web Sites. In *Proc. SC98*, 1998.

[22] A. Chankhunthod, P. Danzig, C. Neerdaels, M.F. Schwartz, and K.J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings 1996 USENIX Symposium*, San Diego, CA, Jan 1996.

[23] Cisco Distributed Director Web Page. http://www.cisco.com/warp/public/751/distdir/, 1997.

[24] Z. Fei, S. Bhattacharjee, E. W. Zegura, and M. H. Ammar. A novel server selection technique for improving the response time of a replicated service. In *Proc Infocom '98*, March 1998.

[25] Project Felix Home Page. ftp://ftp.bellcore.com/pub/mwg/felix/index.html, 1999.

[26] A. Fox. *A Framework for Separating Server Scalability and Availability From Internet Application Functionality.* PhD thesis, University of California at Berkeley, 1998.

[27] A. Fox, S. Gribble, E. Brewer, , and E. Amir. Adapting to Network and Client Variability via On-Demand Dynamic Transcoding. In *Proc. ASPLOS*, 1996.

[28] A. Fox, S. Gribble, Y. Chawathe, and E. Brewer. Cluster-based Scalable Network Services. In *Proc. 16th ACM Symposium on Operating Systems Principles*, October 1997.

[29] Fping Program. ftp://networking.stanford.edu/pub/fping/, 1997.

[30] P. Francis. *http://www.ingrid.org/hops/wp.html*, 1997.

[31] P. Francis, S. Jamin, V. Paxson, L. Zhang, D. F. Gryniewicz, and Y. Jin. An Architecture for a GLobal Internet Host Distance Estimation Service. In *Proc Infocom '99*, 1999.

[32] S. Gribble and E. Brewer. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *Proc. 1997 Usenix Symposium on Internet Technologies and Systems*, December 1997.

[33] J. Guyton and M. Schwartz. Locating Nearby Copies of Replicated Internet Servers. In *Proc. SIGCOMM '95*, September 1995.

[34] J. Gwertzman and M. Seltzer. The Case for Geographical Push-Caching. In *Proc. Fifth IEEE Workshop on Hot Topics in Operating Systems*, May 1995.

[35] R. Hinden and S. Deering. *IP Version 6 Addressing Architecture.* RFC, Dec 1995. RFC-1884.

[36] K. Holtman and T. Hardie. Content Feature Tag Registration Procedure. http://genis.win.tue.nl/ koen/conneg/draft-ietf-http-feature-reg-03.txt, November 1997.

[37] K. Holtman and A. Mutz. Tranparent Content Negotiation in HTTP. http://genis.win.tue.nl/ koen/conneg/draft-ietf-http-negotiation-06.html, January 1998.

[38] K. Holtman and A. Mutz. *Transparent Content Negotiation in HTTP*. RFC, Mar 1998. RFC-2295.

[39] Internet Performance Measurement and Analysis Project Home Page. http://www.merit.edu/ipma, 1999.

[40] V. Jacobson. Congestion Avoidance and Control. In *Proc. ACM SIGCOMM 88*, August 1988.

[41] V Jacobson. pathchar – A Tool to Infer Characteristics of Internet Paths. ftp://ee.lbl.gov/pathchar, 1997.

[42] E. Katz, M. Butler, and R. McGrath. A Scalable HTTP server: The NCSA prototype. *Computer Networks and ISDN systems*, pages 240–249, November 1994.

[43] S. Keshav. Packet-Pair Flow Control. *IEEE/ACM Transactions on Networking*, February 1995.

[44] S. Keshav, A. Agrawala, and S. Singh. Design and Analysis of a Flow Control Algorithm for a Network of Rate Allocating Servers. In *Proc. Second International Workshop on Protocols for High Speed Networks*, 1990.

[45] Keynote Home Page. http://www.keynote.com, 1999.

[46] M. Mathis and M. Allman. Empirical Bulk Transfer Capacity. http://www.ietf.org/internet-drafts/draft-ietf-ippm-btc-framework-00.txt, January 1999.

[47] M. Mathis and J. Mahdavi. Diagnosing Internet Congestion with a Transport Layer Performance Tool . In *Proc. INET '96*, Montreal, Canada, June 1996.

[48] M. Mathis, J. Semke, J. Mahdavi, and Ott. T. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *Computer Communications Review*, 27(3), July 1997.

[49] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *Proc. Winter '93 USENIX Conference*, San Diego, CA, January 1993.

[50] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven Layered Multicast. In *Proc ACM SIGCOMM*, August 1996.

[51] J. C. Mogul. Network Behavior of a Busy Web Server and its Clients. Technical Report 95/5, Digital Western Research Lab, October 1995.

[52] Muffin Home Page. http://muffin.doit.org, 1999. Muffin Home Page.

[53] MultiNet Home Page. http://www.process.com/multinet, 1997.

[54] A. Myers, P. Dinda, and H. Zhang. Characteristics of Mirror Servers on the Internet. In *Proc. Infocom '99*, March 1999.

[55] NetNow Probing Program. http://www.merit.edu/ipma/netnow/daemon/, 1999.

[56] Network Wizards Homepage. http://www.nw.com/, 1999. Network Wizard's WWW page.

[57] B. Noble. *Mobile Data Access.* PhD thesis, Carnegie Mellon University, 1998.

[58] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile Application-Aware Adaptation for Mobility. In *Proc. 16th ACM Symposium on Operating Systems Principles*, Oct 1997.

[59] NUA Survey of Worldwide Internet Users. http://www.nua.ie/surveys/how_many_online/index.html, 1999. NUA Internet Survey.

[60] Project Octopus Home Page. http://www.cs.cornell.edu/ cn-rg/topology_aware/topology/Default.html, 1999.

[61] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. TCP Throughput: A Simple Model and its Emperical Validation. In *Proc. Sigcomm '98*, August 1998.

[62] C. Partridge, T. Mendez, and W. Milliken. *Host Anycasting Service*. RFC, Nov 1993. RFC-1546.

[63] V. Paxson. End-to-End Routing Behavior in the Internet. In *Proc. ACM SIGCOMM '96*, August 1996.

[64] V. Paxson. End-to-End Internet Packet Dynamics. In *Proc. ACM SIGCOMM '97*, September 1997.

[65] V. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, U. C. Berkeley, May 1997.

[66] V. Paxson, J. Mahdavi, A. Adams, and M. Mathis. An Architecture for Large-Scale Internet Measurement. *IEEE Personal Communications*, 36(8):48–54, August 1998.

[67] QuickWeb DNS Pro 2.0Home Page. http://www.menandmice.com/products/quickdnspro, 1997.

[68] Netscape Corp. http://www.realnetworks.com, 1999.

[69] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. *RTP: A Transport Protocol for Real-Time Applications*. RFC, Jan 1996. RFC-1889.

[70] J. Sedayao and K. Akita. LACHESIS: A Tool for Benchmarking Internet Service Providers. In *Proc. 9th System Administration Conference (LISA 95)*, September 1995.

[71] Servicemetrics Home Page. http://www.servicemetrics.com, 1999.

[72] Socks Home Page. http://www.socks.nec.com, 1997.

[73] Sonar home page. http://www.netlib.org/utk/projects/sonar, 1999.

[74] W. R. Stevens. *TCP/IP Illustrated, Volume 1*. Addison-Wesley, Reading, MA, Nov 1994.

[75] D. L. Tennenhouse and D. J. Wetherall. Towards an Active Network Architecture. *Computer Communication Review*, 26(2):5–18, April 1996.

[76] Timeit 2.1 Software Distribution. ftp://ftp.va.pubnix.com/pub/uunet/timeit-2.1.tar.gz, 1999.

[77] UC Berkeley Annex WWW Traces. http://www.cs.berkeley.edu/ gribble/traces/index.html, 1997.

[78] VitalSign's NetMedic Product. http://www.vitalsigns.com/products/nm/index.html, 1997.

[79] T. Von Eicken, D. Culler, S. C. Goldstein, and K. E. Schauer. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. International Symposium on Computer Architecture*, 1992.

[80] R. Wolski. Dynamically forecasting network performance to support dynamic scheduling using the network weather service. In *Proc. 6th High-Performance Distributed Computing Conference*, August 1997.

[81] G.R. Wright and W. R. Stevens. *TCP/IP Illustrated, Volume 2.* Addison-Wesley, Reading, MA, Jan 1995.

[82] The Emerging Digital Economy. http://www.ecommerce.gov/emerging.htm, April 1998. Commerce Department Report on Future of Electronic Commerce.

[83] B. Zenel and D. Duchamp. A General Purpose Proxy Filter Mechanism Applied to the Mobile Environment. In *Proc. ACM Mobicom*, October 1997.