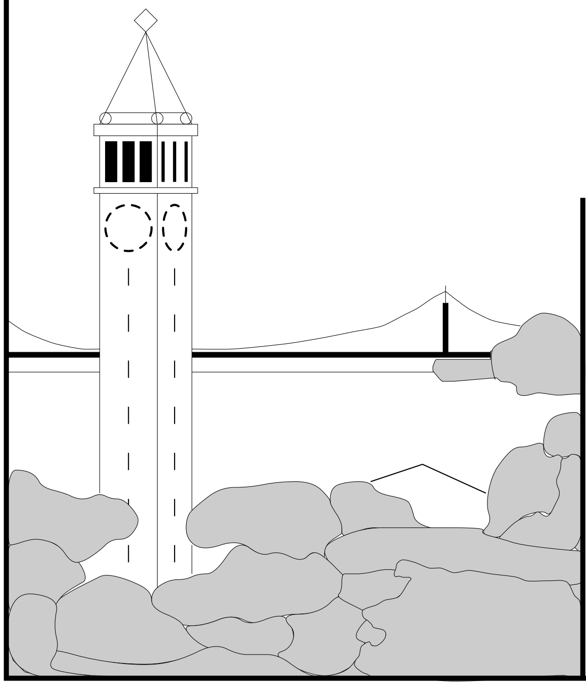
Response Time as a Performability Metric for Online Services

Peter M. Broadwell
Computer Science Division
University of California at Berkeley



Report No. UCB//CSD-04-1324

May 2004

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Response Time as a Performability Metric for Online Services

Abstract

Response Time as a Performability Metric for Online Services

by

Peter M. Broadwell
Master of Science in Computer Science

University of California at Berkeley

Professor David Patterson, Research Advisor

Evaluations of the behavior of Internet services during partial outages tend to rely upon system throughput as their primary unit of measurement. Other metrics such as response time and data quality may be equally as important, however, because of their ability to describe the performance of the system from the user's perspective. Because the success of most online services depends upon user satisfaction, such user-oriented metrics may be a more appropriate way to rate a system's effectiveness.

This report investigates how system evaluators can use the performance metric of response time when conducting reliability benchmarks and long-term performability modeling of online systems. Specifically, this report describes the results of fault-injection tests run on a cluster-based web server under an emulated client workload and explores various ways of presenting and summarizing data about the observed system behavior. The report concludes with a discussion of how online services can improve their effectiveness by adopting and employing more useful and descriptive performability measurements.

Contents

Chapter 1

Introduction

The user experience that an online service offers is an integral component of that service's overall reliability. If customers of an interactive, Internet-based business experience slowness or other difficulties with the business's web site, they will be less likely to frequent that site in the future, regardless of the amount of total processing and storage resources that the service possesses. It is thus in the best interest of the service to provide a consistent user experience at all times, even under adverse conditions such as component failures or high demand.

To build services that provide consistent, positive user experiences, designers need metrics that let them reason about what does and does not constitute "good" service under varying conditions. Currently, the reliability metrics that designers commonly use to evalute online services do not satisfy this requirement. This study works to address this deficiency by seeking more advanced methods for quantifying the failure-related behavior of online services as perceived by end users.

1.1 Motivation

As giant-scale Internet services like Google, Yahoo! and MSN grow in size and importance, so does the demand for continuous availability and full functionality from these services. But recent high-profile outages of some prominent online services have highlighted the variable quality and occasional unreliability of these services [29]. Recognition of the need to improve computer system reliability has led to much recent academic and industrial interest in this area. Research projects from the computing industry include Microsoft's "Trustworthy Computing," which focuses on security and source code quality [24], and IBM's "Autonomic Computing," which aims to create self-configuring, self-healing, self-optimizing and self-protecting systems to ease the burdens of managing large, complex systems [32]. In the academic realm, the Berkeley/Stanford Recovery-Oriented Computing (ROC) project [14] seeks to improve online service reliability by increasing the speed and effectiveness with which a service can recover from a failure.

Numerous techniques for prolonging a service's time *between* failures already exist, thanks to researchers in the venerable "fault-tolerant computing" field [35], so why does the ROC project choose to focus on recovery time? For one, the fast rate of development that is necessary to keep an online service up to date with new markets and technologies makes the application of most traditional fault-tolerant computing techniques impractical. Also, measuring and reasoning about the recovery time of systems is easier than monitoring the time between failures [5], meaning that recovery-oriented improvements can be implemented more quickly and effectively. Finally, real-world experiences suggest that rare but large-scale service outages may have a greater negative psychological effect on user perception than numerous small-scale outages that are quickly repaired [15].

To determine the success of the projects mentioned above, it is necessary to rate the degree to which their novel design concepts actually increase the reliability of a service. Again, the fault-tolerant computing community provides a wealth of ideas in this area, only some of which have been applied to online services [21]. This report argues that the reliability metrics currently used by most online services are not detailed enough to be effective aids to developers. Because of the unsatisfactory nature of current reliability metrics, other concepts must be examined, especially those that deal with the behavior of systems during partial outages.

Consider *availability*, the most widely-used online service reliability metric. Here, we define the term *reliability* to be identical to the technical meaning of *dependability* – the ability of a system to deliver a specified service [21]. A popular method method of representing online service availability is to calculate the fraction of the service's operational lifetime during which it has been accessible, generally yielding some number of "nines" in the resulting decimal representation.

$$service\ availability\ (nines) = \frac{uptime}{(downtime + uptime)}$$

Representing a service's availability with a simple count of "nines" gives a rather coarse-grained indication of its overall quality of service. In particular, this version of availability fails to capture the *duration* and *frequency* of specific outages. As mentioned above, these figures may have a large impact on the user perception of a service.

Also, availability metrics may not capture such elements of the user experience as the *severity* of a given outage. Outage severity is an important metric because many services have the ability to provide partial functionality during outages. For example, a service could cope with temporarily reduced network, computation or storage resources by reducing the number of probable matches served in response to a search query or by providing lower streaming audio or video quality. Because the service is still technically "available" in these situations, this behavior would not directly affect its availability score but would nonetheless have a considerable effect on the experience of anyone using the service at that time. Finally, time-based availability may misrepresent the actual service quality experienced by users: an outage of short duration might have a negligible effect on the service's "nines," but the actual damage to the *user-perceived availability* of the service could be much greater if the outage occurs at a time when the service is experiencing high customer demand.

Compared to basic availability, alternative metrics that are able to quantify outage duration, frequency and severity will prove much more useful to developers, who use such evaluations to determine the best ways to improve their systems. As discussed above, a primary benefit of detailed failure-related metrics is that they are capable of describing the experience offered to the end user, both during normal behavior and in the presence of faults. Several studies of e-commerce sites have shown that customer perceptions of the "usefulness" of a service ultimately determine its income [39, 40]. In addition, reliability measurements that operate on a smaller scale than availability are necessary for the establishment of useful service-level objectives for online services [9].

Previous studies that have attempted to rate reliability with metrics other than availability [38, 7] have primarily measured the average throughput provided by a service in the presence of faults.

Throughput metrics such as *requests served per second* are certainly useful, but they do not tell the whole story. In particular, throughput fails to give much information about how an outage affects the end-to-end qualities of the service, which are best measured at the client. This report addresses the shortcomings of throughput as a means to describe the severity of service outages.

1.2 Overview of Research

Any metric that considers how failures may influence the performance of a system is, by definition, a *performability* metric. Throughput, the most commonly-used performability metric, is only applicable to the server-side processing components of an online system. Because online services conduct interactions with their clients through large-scale networks such as the Internet, they can benefit greatly from the use of alternative performability metrics such as *response time (latency)* and *data quality* to give an accurate description of the end user's experience during an outage.

The primary contributions of this study are

- An exploration of how to quantify the severity of online service outages, using concepts from the field of performability.
- Arguments and experimental data to describe the efficacy of user-observed response time as an indicator of end-to-end service quality during an outage.
- A discussion of how best to present response time measurements and how they can be combined with other metrics to describe a service's ability to support degraded service.

This report continues in Chapter 2 with a discussion of reliability metrics, a brief overview of the field of performability and a review of the previous research that has been conducted on the subject of user-oriented performability metrics. Chapters 3 and 4 describe the setup and results, respectively, of the experiments conducted for this study. Chapter 5 discusses the implications of the experimental results, and Chapter 6 concludes the report and poses questions for future research.

Chapter 2

Background and Related Work

Before beginning a discussion of failure-related reliability metrics and the previous research that has occurred in this field, it will be useful to define some terms that pertain to service outages:

- *online service* - for the purposes of this study, an online service or *Internet service* is any organization that provides a service to customers through the primary medium of the Internet. The hardware and software necessary to support the service may range from a single workstation running a commodity operating system to a complex clustered or federated environment. This definition does not extend to systems with no central authority, such as true peer-to-peer networks. Figure 1 shows a hypothetical online service of the type that this research is meant to address.
- *failure* - the inability of a service to provide some expected level of functionality.
- *fault* - an event (e.g., a natural disaster, hardware breakdown, human error) that leads to a failure.
- *outage* - the period of time during which a system is experiencing a failure. Note that the level of functionality provided during an outage may not always be zero; in fact, it may fluctuate wildly. An outage may consist of several stages, separated by events such as the initial fault, detection of the failure, diagnosis of the failure root cause, fault repair, and recovery.

2.1 Failure-Related Reliability Metrics

The following sections describe classes of reliability metrics that can be applied to online services.

2.1.1 Outage frequency and duration

Capturing outage frequency and duration involves recording the initial failure time and the final recovery time for each outage. These data can then be used to calculate the *maximum, minimum and mean outage duration* for the service, as well as information about the *outage duration variance.*

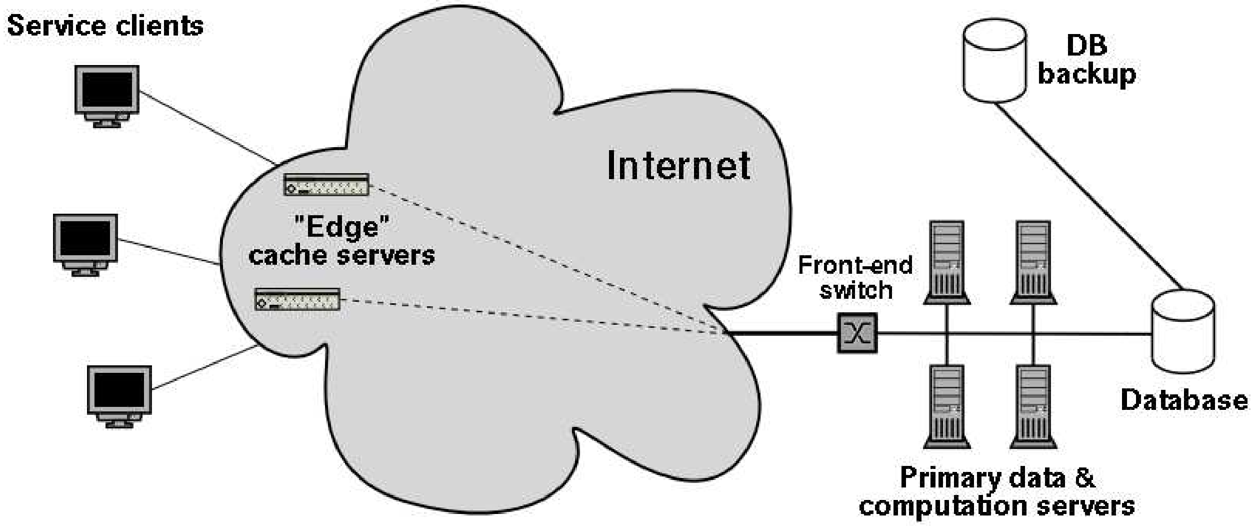


Figure 1: **The components and configuration of a hypothetical online service.** A cluster of computation and data nodes interacts with a database to provide authoritative responses to client requests, which are sent and received via the Internet. The clients may be home users or businesses with broadband or dial-up connections. The service may cache static content on "edge" servers distributed throughout the Internet. Clients consult these caches instead of the central servers to reduce load on the primary servers and to cut down on client-perceived response time. The central compute notes may be concentrated at a single data center, or the service may consist of a number of cooperating data centers and backup sites.

Another important metric might be the *outage frequency* for different types of outages – it can be useful to know how often the service experiences "large" or "small" outages the over time.

As stated in Chapter 1, the frequency and duration of outages may influence the user perception of a service and also figure prominently in the service-level objectives set by an online service. Outage frequency and duration, although important components of any real-world failure analysis, are not the central focus of this research.

2.1.2 Fraction of affected users

As Chapter 1 mentions, availability calculations based upon the raw uptime and downtime of a system may ignore or misrepresent the actual experiences of the users of the system. For example, an eight-hour outage of a service would have a markedly negative effect on a service's "nines" of total availability. If, however, the outage occurs at night and the customers of the service primarily use it during the day, the actual impact of the failure on the system's perceived reliability would be negligible.

An alternate representation of long-term reliability is a comparison of the number of users affected by an outage to the total number of customers who visit the site over a period of time. Computing the fraction of affected users, however, requires that the service be able to record the number of clients who attempted to access it during an outage. Since the monitoring mechanisms of the service may themselves be unavailable during an outage, such data can be difficult to record directly and may require estimation.

2.1.3 Service quality during partial outages

All metrics that describe the quality of service that a system provides in the presence of failures are *performability metrics*. These metrics are the result of work in the fault-tolerant computing field on how to quantify a system's "ability to perform" under both normal and exceptional conditions. As such, peformability metrics present both performance and reliability-related data.

Performability metrics provide several advantages that are not offered by coarse-grained reliability metrics like availability or raw performance metrics such as best-case throughput. Because performability metrics capture the behavior of a system during outages, they allow direct comparisons of the reliability benefits offered by different system designs and recovery mechanisms. In addition, performability measurements may provide enough detailed information about failure-related behavior to facilitate the identification and analysis of recovery phases, so that designers can isolate the weak points of their recovery systems.

These are the classes of performability metrics that apply to online services:

- **Throughput-based metrics**. These metrics describe the raw server-side performance of the entire service. Given data about component failure rates and the throughput levels offered under both failures and normal conditions, it is possible to compute the *average number of responses served per unit time* for the service, a commonly-used performability metric [26].

- **Data quality metrics**. These metrics apply if the business model of a service allows it to vary the amount of work completed per query in response to environmental conditions. For example, a search engine may return fewer than the optimum number of matches to a query if part of its data set is unavailable. Two data quality metrics that may directly be affected by outages are *yield* – the percentage of successfully-answered queries, and *harvest (query completeness)* – the percentage of all applicable data that is returned [5]. Changes in system behavior due to failures can also indirectly influence the system's *recall* – the number of "hits" returned in response to a multi-answer query, and *precision* – the odds that a response will contain the desired information.

- **Latency metrics**. Latency (*response time*) metrics quantify how long the user must wait for a response to a query, regardless of the quality of the response. Together with data quality metrics, latency metrics provide the best indication of the end-user experience under normal conditions and during outages. For web-based requests, query response time often includes at least two elements: *fetch latency* – the time to load the page HTML into the browser, and *render time* – the time to receive all elements referenced by the page HTML, such as images, and display the page in the browser [10].

This study uses the terms *latency* and *response time* interchangeably when discussing the behavior of Internet services. In this context, latency is defined as the wait time experienced by the user after requesting data, rather than the low-level definition of latency as the time necessary to send data from one end of a physical network link to the other.

2.1.4 Metrics used in this study

Because the aim of this project is to explore metrics that describe the end-user experience, latency-based metrics will receive the most attention, although data quality metrics will be considered as well. Throughput-related data will also be recorded to allow direct comparisons with the results of previous studies.

Chapter 5 applies a large set of performability metrics to the data gathered during the experiments described in Chapter 3. Some of these metrics were chosen post hoc for their ability to highlight specific aspects of the test systems' behavior; others are standard performance metrics. They are all provided here for reference:

- *total (raw) throughput*: The sum of all user requests that the service satisfies successfully during a test period.
- *total (raw) latency*: The sum of the response times (in seconds) observed by all users during a test.
- *average (mean) throughput*: The total throughput of the system divided by the duration of the test, yielding the average number of requests served successfully per time unit.
- *average (mean) latency*: The total latency for the test divided by the number of requests (successful or not) submitted by the users, producing the average response time per request.
- *trimmed-mean latency*: The average per-request latency of the middle 95% of all observed response times. Excluding 5% of the highest and lowest response times reduces the odds that a small number of extreme outliers will distort the mean.
- *goodput*: The observed average throughput divided by the ideal throughput of the system, producing some fraction of 1. This metric occasionally appears in computer and network performance evaluations as a way to describe the actual progress made by a system rather than the total processing done by the system [3].
- *punctuality*: A proposed latency-oriented analogue to goodput, defined as the observed average per-request response time divided by the system's best-case response time.
- *variance of response times ("jitter")*: Quantifies the degree to which the response time for a particular action, such as a page request, varies during a test period. It is possible to describe this variance via metrics such as the standard deviation of the response time from the mean, or by providing graphical representations such as box-and-whiskers plots of the observed response time distributions.

- *probabilities of exceeding latency thresholds*: Since users may react differently to different response times, it can be useful to count how many individual requests exceeded certain "watershed" wait time limits and use this data to compute the probability that a single request will exceed a given time limit.

- *demerits*: The collection of service events that result in user dissatisfaction, such as dropped connections, long wait times and incomplete or incorrect data, can be combined into a single metric by arbitrarily assigning a certain number of *demerits* to each event. The sum of all demerits that a system accrues during operation thus estimates the total amount of user dissatisfaction it has generated.

2.2 Performability Background

The concept of performability first emerged in the mid-1970s, when system designers began to deal routinely with systems that were large and complex enough to maintain some degree of functionality after a fault [23]. Designers realized that their understanding of a system would be more extensive if they could compare its *degraded* behavior during outages to its performance during normal operation.

Simply calling performability a measure of performance after faults – the "ability to perform" – does not adequately represent this class of metrics, however. To understand their full set of applications, it is necessary to examine how performability metrics have come to be used in the field of fault-tolerant computing. This field uses performability metrics extensively to provide a composite measure of both performance and dependability over the *entire lifetime of a system* [17].

The need to depict a system's life cycle is a result of the lengthy design verification process that fault-tolerant system designers must conduct prior to building and deploying a system. The fault-tolerant computing field deals with mission-critical systems for demanding environments such as aerospace and high-performance transaction processing. Due to the immense cost of building and servicing such systems, designers strive to model the behavior of a system in great detail before they actually build it. Performability metrics provide a way to quantify this behavior.

2.2.1 Performability examples

Consider a weather satellite that is to be launched into orbit. The satellite consists of a set of redundant components such as cameras, antennas and solar cells. Each component has an associated failure rate over time, as well as a performance contribution, which defines the degree to which the overall performance of the satellite will be degraded upon the failure of that component. Given these statistics, service designers can build a model to determine the average level of performance that the satellite can be expected to provide for part or all of its operational lifetime. This metric, which might be a throughput measurement such as *images transmitted per day*, is a performability metric because it captures the system's behavior given faults.

In practice, performability metrics can be used to represent system behavior on either large or small time scales. Figures 2 and 3 illustrate this concept through the more down-to-earth example of a RAID-5 disk array. Figure 2 shows a Markov reward model that estimates the long-term performability of a RAID-5 array. Markov models, as well as more detailed models like stochastic Petri nets, are

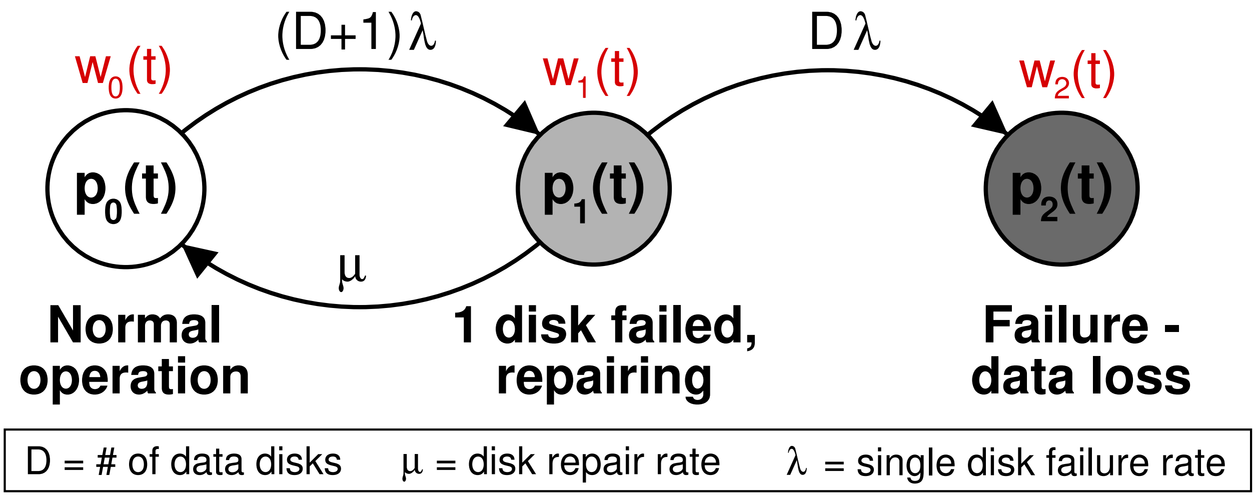


Figure 2: **A discrete-time Markov chain model of a RAID-5 disk array**, from Kari [18]. The nodes represent the state of the array at different points in its operational lifetime. The expected disk failure and repair rates, shown as arcs, determine each $p_i(t)$, the probability that the system is in state i at time t. Each state i has an associated "reward" value $w_i(t)$, which represents performance in I/O operations per second. The array's expected performability at any time t is described by *performability(t)* $= \sum_i p_i(t)w_i$. This model also describes other reliability-oriented features of the array, such as its *mean time to data loss*.

commonly used for performability modeling in the fault-tolerant computing field [17]. Figure 3 depicts a short-term performability evaluation: empirical measurements of the throughput levels that an array provides during the failure and eventual reconstruction of a volume.

The two relative time scales that can be covered in performability evaluations, here termed "macro-" and "micro-" scale, highlight the primary purposes of these metrics. Macro-scale analyses determine how well a complete system design meets some established criterion, such as the number of weather readings that our hypothetical satellite takes over a four-year period, and micro-scale analyses are conducted in order to gather data about some specific failure-related behavior of a system.

2.2.2 New applications of performability metrics

Commercial Internet service developers have recently begun to apply performability metrics to their services. Most commonly, these metrics are used to evaluate the large-scale storage systems that form the crucial database backbone of an online service. An example of a storage-related project that employs performability metrics is the self-managing storage effort at HP Laboratories, which uses performability metrics to describe aspects of storage systems such as *mean time to data access* during outages [19]. Performability metrics are also appearing in evaluations of the front-end machines that directly serve web pages and other content to the user. One such effort, a study from Rutgers University that used fault injection to evaluate the reliability of the PRESS web server [26], was the direct precursor to this report.

A significant difference between how performability metrics are employed in the online systems

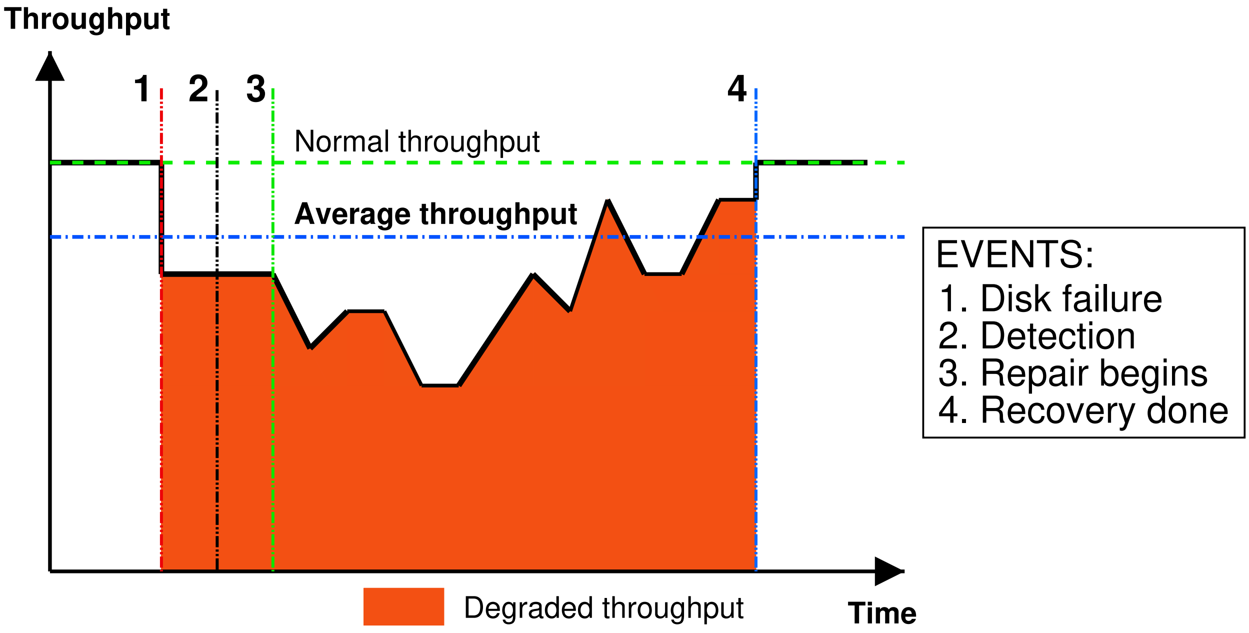


Figure 3: **Hypothetical throughput of a RAID-5 array during a disk failure and subsequent volume reconstruction.** After the failure, the array's throughput drops because one of the disks of the array is no longer available to serve read requests, placing a higher load on the rest of the array. Repair of the array requires that a replacement disk be added to the array and populated with the data previously stored on the failed disk *while the array continues to service user requests*. The missing information is reconstructed based on data stored in the other disks of the array, resulting in higher aggregate load levels on the array during the reconstruction. As a result, the throughput of the array may fluctuate considerably during reconstruction, before returning to its ideal level once recovery has completed. See Brown and Patterson [6] for real-world throughput measurements of software RAID systems.

field as opposed to the fault-tolerant systems community is that Internet systems often employ a "build-and-see" mentality, but fault-tolerant systems designers must perform extensive modeling before actually constructing a system. Online service designers, stimulated by the rapid development cycle of Internet applications and the availability of cheap hardware components, are likely to build a prototype of a system and then run performability tests on it, rather than modeling the system first and deriving performability numbers from the model. This report explores both approaches to obtaining performability measurements.

An important but often neglected role of performability metrics is to quantify the end-user experience that a service offers in the presence of faults [23]. As is true of the examples presented in this chapter, most practical performability tests measure system throughput, a metric that is not intrinsically user-oriented. Designers focus on throughput for the simple reason that there is little end-user interaction with systems like satellites and storage arrays, so the metric of greatest importance is the throughput that the service can provide during its operational lifetime.

The recent application of performability metrics to Internet systems offers new opportunities for the application of user-oriented performability metrics. Because providing a consistent user experience equates to higher income for online services, it is important that designers of these services consider how failures influence the end-user experience. The primary purpose of this report is to explore how one user-oriented metric, response time, can be used to describe online service outages.

2.3 Related Work

Numerous studies of both large- and small-scale systems have identified latency, rather than throughput, as the most important performance metric for interactive services. One of the earlier studies on this topic was Endo et al. [13], which explored latency-based metrics for evaluating desktop operating systems. Neilsen [27] also investigated the degree to which response times of different durations could influence the usability of a computer system. In addition, Douceur and Bolosky [12] proposed the use of response time measurements to evaluate the effectiveness of scheduling policies for streaming media servers.

More recently, latency metrics have seen increased use by web performance monitoring companies like Keynote Systems [1], which employ far-flung networks of client computers to perform real-time latency tests on their customers' sites. Keynote has published several reports [39, 40] that document the deleterious effect of long page-load times on the income of a web service. In these reports, Keynote establishes several rules of thumb regarding user behavior and response time. The most significant of these principles is the so-called "eight-second rule", which states that most Internet users – especially those with broadband connections – are likely to become annoyed and eventually abandon a site if the response time for a particular page exceeds eight seconds. Subsequent user studies, including Bhatti et al. [4], further refined these rules.

In addition to the Keynote Systems project, Rajamony and Elnozahy at IBM [31] and Chen and Perkowitz at Appliant, Inc. [10] have outlined latency monitoring mechanisms that can provide information about service failures. The latter studied how to use deviations in client page-load times to pinpoint infrastructure-related misconfigurations and failures in large-scale Internet services. Although these studies do not mention the term, they are using latency as a performability metric, since they employ response time to describe the abnormal behavior of their systems.

Papers about online services that explicitly use latency as a performability metric tend to originate from storage-related research projects. As discussed previously, designers of online storage systems were among the first to adopt performability metrics, due to their need for detailed failure-related measurements that would help them avoid losing data. Several projects from the HP Labs Storage Systems program have used both latency and throughput as performability metrics. These projects include work on automating data dependability [19] and providing performance guarantees for storage arrays [20].

Projects that attempt to do any sort of performability analysis, latency-based or otherwise, on web service "front-ends" such as load spreaders and web servers are rather rare. One such study by Candea et al. [7] used throughput to evaluate the reliability improvements offered by JAGR, a self-recovering application server. Similarly, researchers at Rutgers University used the Mendosus fault-injection platform to evaluate the throughput of the PRESS web server during failures [26]. This project aims to reproduce the experiments of the PRESS/Mendosus research, focusing on latency rather than throughput as the primary performability metric.

This report also will consider the practice of "load shedding" – immediately terminating new connections, often with a "Server too busy" message, during times of high load. Many services adopt this policy during load spikes to maintain a reasonable level of service for the clients who are already connected, rather than subjecting all customers to the same unacceptably long response times. The experience of being denied access to a service, although inconvenient for the user, may in some cases be preferable to waiting an excessive amount of time for a response. Even though load shedding may result in 0% complete query responses to a large subset of prospective customers, it is possible that these users will be more likely to try back later in this scenario than if they are subjected to inordinately slow service. Papers that consider the tradeoffs inherent in reducing data quality include Brewer [5] and Abdelzaher and Bhatti [2].

Finally, any discussion of new metrics to evaluate computer systems should consider the examples of the current industry standard benchmarks and the ongoing efforts to develop new benchmarks. Most of the standard performance benchmarks that are applicable to Internet services, such as SPECweb [34], are primarily server- and throughput-oriented. Others, like TPC-C [37] and TPC-W [36], do include some concepts of user wait time and reliability in their specifications. For example, TPC-C does not measure the reliability of the online transaction processing systems it evaluates, but does require that the systems meet some basic reliability standards such as the ability to do log-based recovery. In addition, TPC-C and TPC-W both place some constraints upon the user-oriented response times that the test system is allowed to produce. The international DBench project [11] and the SIGDeB [16] group have made significant progress toward the definition of dependability benchmarks that incorporate fault injection and the monitoring of throughput, latency and data quality. Both projects, however, have tended to focus upon upon transaction-processing systems and generic desktop operating systems rather than online services.

Chapter 3

Experimental Setup and Methods

The investigative goals of this project are to reproduce the experimental results from an earlier performability study conducted at Rutgers University [26] and to supplement the throughput metrics gathered during the experiments with additional response time and data quality measurements. This second objective requires a few minor alterations to the experimental setup from the Rutgers study, but the basic configuration remains the same. The experimental infrastructure consists of the PRESS cluster-based web server, the Mendosus fault injection platform and a set of load generation and performance monitoring tools.

3.1 The PRESS Web Server

The PRESS web server, as described by Carrera and Bianchini [8], is a cluster-based, locality-aware HTTP server. The server can be deployed on an arbitrarily large number of commodity PCs linked together by a high-speed network. Once the PRESS software has been configured and activated, the member nodes of the cluster act as a single web server, communicating with each other to facilitate cooperative caching of page content and load balancing of incoming requests.

Any member of the PRESS cluster may receive an HTTP request from a client. When this node (hereafter referred to as the "primary" node) first receives a request, it consults an in-memory table to determine if it has the requested file cached in memory. If so, it immediately sends the file to the client. If the primary node does not have the file cached, it checks the table to determine if another node in the cluster has the requested file in its cache. If such a secondary node exists, the primary node requests the file from the secondary node and then caches the file in memory before sending a reply to the client. If the requested file is not cached anywhere on the cluster, the primary node retrieves it from disk, caches the file, and sends it to the client. Whenever a node's cache contents change, it immediately notifies all other nodes in the cluster of the change.

Figure 4 shows the basic hardware components of a PRESS cluster. For storage of web page data, cluster nodes may share a single network-attached storage array or possess individual copies of a read-only page set. To ensure an equal distribution of client requests across the nodes of the cluster, the system administrator may choose to install load-balancing hardware switches between the PRESS cluster machines and external network. Given such an equal distribution of requests, the PRESS

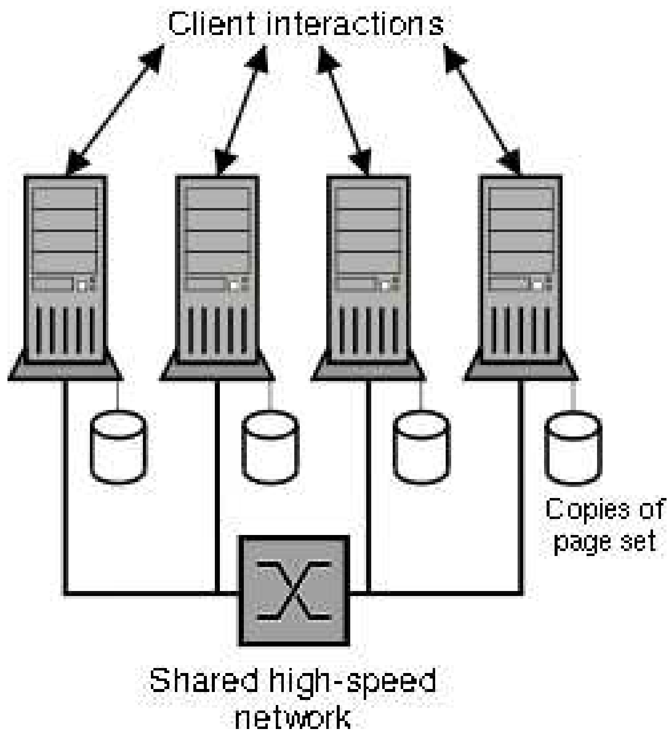


Figure 4: **The hardware components of the PRESS cluster-based web server.** The PRESS software allows a potentially unlimited number of physical server nodes to act together as a single web server via cooperative in-memory page caching over a shared, high-bandwidth network. This network may carry client requests as well, or the server nodes may receive requests on a different network interface. Although the behaviors of the two PRESS versions used in this study, Perf-PRESS and HA-PRESS, differ considerably, they both run upon the hardware setup shown above.

software provides automatic load balancing via its in-memory cooperative caching scheme, resulting in high CPU utilization and high overall performance for the cluster.

PRESS offers further customization choices via software configuration options. This study uses two versions of the PRESS software, both of which run on the same physical hardware. These two versions, called Perf-PRESS and HA-PRESS, were chosen for their differing design goals: the Perf-PRESS software configuration is set up to deliver the best performance in an error-free setting, but the HA-PRESS configuration seeks to provide dependability in the presence of faults.

In the tightly-coupled Perf-PRESS configuration, the entire cluster acts as a single entity. A failure of any component of the cluster will seriously degrade the availability and performance of the server as a whole. HA-PRESS, however, incorporates several design concepts for high availability, providing service robustness in the presence of faults at the cost of lower peak performance due to monitoring overhead. Specifically, the nodes in HA-PRESS monitor each other via cyclic heartbeat checks and periodic broadcasts that are sent in response to queries from the current master node. When a component node is detected to have failed, the other nodes cooperate to reconfigure the cluster, choosing a replacement master node if necessary. Once the failed node has recovered, the HA-PRESS software can automatically re-incorporate it into the cluster. As a result of these capabilities, the HA-PRESS server recovers quickly from some types of partial failures.

Perf-PRESS incorporates no such self-monitoring and self-healing mechanisms, so a failure of a single node is likely to disrupt the operation of the entire cluster until the failure is repaired and the entire cluster is restarted. Problem detection and troubleshooting on the part of a human operator are usually necessary to resolve a Perf-PRESS failure, but HA-PRESS is able to detect and recover from many classes of failures automatically. The use of two versions of PRESS in this study, one designed for high performance and the other for availability, makes possible a statistical comparison of their performability ratings. These ratings provide a way to determine which server configuration provides a higher overall quality of service in a particular environment.

The ability of the PRESS web server to be configured in these two different modes – tightly coupled versus high availability – was a primary reason why the original Rutgers fault-injection study used PRESS as its evaluation target. By virtue of its design, the HA-PRESS version can be expected to perform significantly better than Perf-PRESS in most fault-injection tests, thus providing a convenient way to gauge how well such performability evaluations can highlight the differences between the versions. Also, the cluster-based nature of PRESS made it ideal for use with the fault injection system developed at Rutgers, called Mendosus.

3.2 The Mendosus Fault Injection System

Zhang, et al. describe Mendosus as a "SAN-based fault injection testbed for the construction of highly-available network services" [38]. Its purpose is to coordinate the injection of a variety of faults into the component nodes of small to medium-sized cluster-based online services in order to evaluate their behavior in the presence of faults. Because Mendosus usually runs on physical, rather than simulated nodes, it can generate near real-world performance and dependability measurements, albeit at the cost being somewhat hardware specific. Mendosus is limited in its scalability primarily by the number of physical nodes available to the service, although it is possible to escape this restriction by running the nodes inside virtual machines. Introducing this level of virtualization, however, reduces the ability of Mendosus to mimic strictly true-to-life system behavior.

A Mendosus testbed consists of several components: a user-level Java *daemon* on each node, which can start and stop applications as well as coordinate fault injection, individual *fault injectors* on each node – modified kernel modules and drivers that inject specific faults when invoked by the daemon – and finally the global *controller*, which is another Java program that typically runs on a separate machine and communicates with the per-node daemons to coordinate fault injection across the testbed. See Figure 5 for a diagram of a standard Mendosus setup. The most important fault injector on each node is the *network emulation* module, a modified version of the node's NIC driver that emulates a specified network topology and applies network-level faults to traffic entering or leaving the node.

Currently, Mendosus is able to inject a wide variety of faults, including cluster-wide network switch or link failures, node-level crashes, reboots and hangs, application failures and lock-ups, OS-level memory exhaustion, and both SCSI and IDE disk subsystem failures. For the sake of brevity, this study focuses on node, application and network-level faults – see Table 1 for a complete list of the faults used in this study.

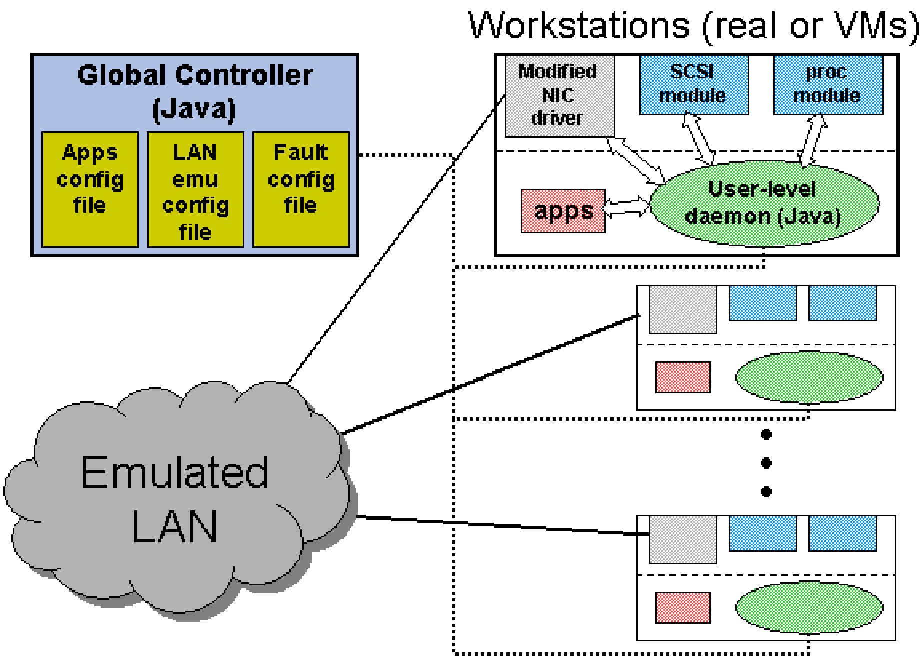


Figure 5: **The primary components of the Mendosus fault injection platform.** Each node in the cluster runs a user-level Mendosus daemon, which communicates with the instrumented kernel modules on that node to perform network emulation and fault injection. The actions of the per-node daemons are coordinated by a global controller daemon, which runs on a separate node. For this study, it is important to note that the applications running on the instrumented nodes are not restricted to communicating only with hosts on the emulated LAN; they can access external networks as well through the modified NIC driver.

One justification for a simplified fault scheme is that many of the lower-level faults that Mendosus emulates, such as SCSI timeouts and OS memory exhaustion, eventually manifest themselves externally as application and network-level faults of the types included in this study: application slowdowns, node failures and network problems. This observation relates to the concept of "fault model enforcement" as described by Nagaraja et al. [25], which explains that system reliability can be improved by amplifying unusual fault behavior to conform to more common, large-scale faults that the system already is programmed to handle gracefully.

Mendosus allows faults to be either "transient," existing for a limited period of time, or "sticky," remaining in effect for the duration of the simulation. Additionally, each fault can be assigned a specific time at which to occur, or else be made to recur at pseudo-random intervals according to a

Category	Fault	Possible Root Cause
Node	Node crash	Operator error, OS bug, hardware component failure or power outage
	Node freeze	OS or kernel module bug
Application	Application crash	Application bug or resource unavailability
	Application hang	Application bug or resource contention with other processes
Network	Link down or flaky	Broken, damaged or misattached cable
	Switch down or flaky	Damaged or misconfigured switch, power outage

Table 1: **The types of faults injected into a running PRESS cluster during the performability experiments**, along with their classifications and possible root causes. This list is a subset of the fault classes provided by Mendosus [26]. Specifically, Mendosus also provides the ability to inject faults that mimic disk subsystem failures and operating system bugs.

given occurrence distribution, such as Poisson or Gaussian.

The faults used in this study are injected via several different mechanisms. Application-level faults, such as crashes or hangs, are enacted by sending signals from the per-node daemon to a running process. The full-node ”freeze” effect is accomplished by loading a module into the kernel that goes into a continuous loop for a specified period of time, effectively halting all other processing on the machine. Node crashes are achieved by invoking the system `reboot` command. Network-level faults such as failed links or dropped packets are enabled through communication between the node daemon and the the instrumented NIC driver. This communication takes place via ioctls sent through a character device that the NIC driver has exported to the kernel's `/proc` file system.

3.3 The Test Environment

3.3.1 Servers and clients

Figure 6 shows the testbed setup for the experiments in this study. Four cluster nodes are instrumented with Mendosus fault injection and network emulation modules and run both the PRESS software and a Mendosus user-level daemon. A separate machine on the same network runs the Mendosus global controller to coordinate the tests.

Four other computers, which are on the same physical network as the PRESS nodes but do not run under the Mendosus emulated network, act as client emulators, generating page requests to be fulfilled by the PRESS cluster. A fifth node acts as the client coordinator, collecting and combining the quality-of-service metrics observed by the client machines.

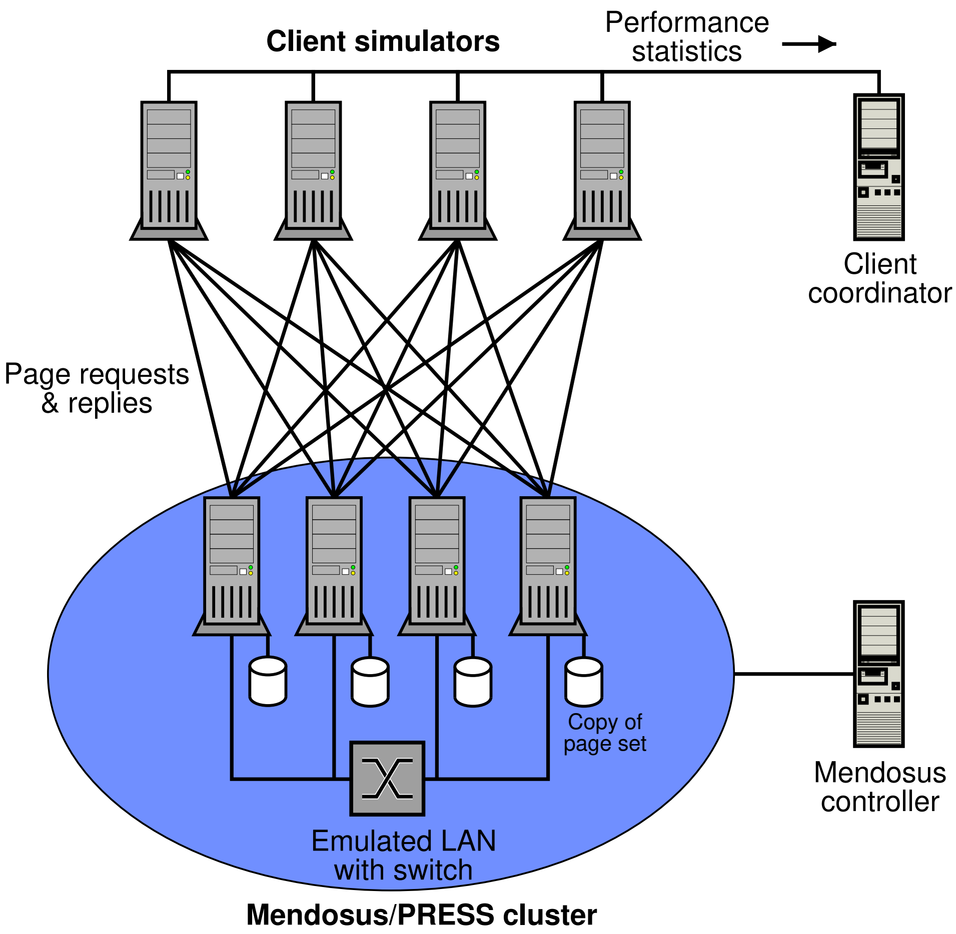


Figure 6: **A logical view of the PRESS cluster, Mendosus infrastructure and client load generation and monitoring systems** as deployed for the fault injection experiments. The client machines emulate the behavior of a large pool of users and send their observations about the performance of the PRESS cluster to the client coordinator machine. The Mendosus controller coordinates the execution of fault injection tests on the four PRESS server nodes, which each are instrumented with the Mendosus fault injection modules and daemons. Note that only the processes and components within the Mendosus sphere of influence are eligible for fault injection – neither the client machines nor their connections to the PRESS servers can be directly affected by simulated faults.

3.3.2 Page set and workload

Each PRESS server node has its own local copy of a read-only set of web pages, consisting of approximately 18,000 HTML files of 26 KB each. This set represents the page set size and average HTML file size of the Rutgers Computer Science Department web site. Representing the pages by an average size, as opposed to the actual distribution of page sizes, runs counter to real-world observations that web page sizes often follow heavy-tailed distributions, wherein a few exceptionally large pages make up the bulk of all available data [30]. The simplification of the pages to an average size was necessary, however, to reduce the variability of performance measurements conducted over short time scales.

Each client emulation node makes requests for pages from the test set based upon traces of actual page requests made on the Rutgers CS web site. Thus, this portion of the simulation accurately reflects the access patterns of a set of real-life users.

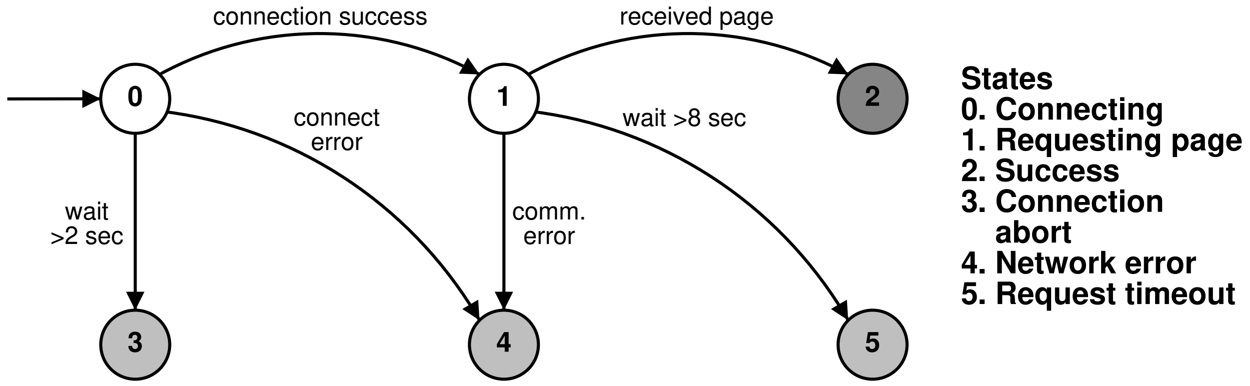


Figure 7: **A state diagram of the emulated behavior of a single PRESS user.** In the PRESS/ Mendosus experiments, each of the four client emulation machines may produce several dozen overlapping page requests per second, each of which follows the state transition rules given in this figure. Note that the client will only wait six more seconds to receive the actual page content after connecting, resulting in a cumulative timeout limit of eight seconds.

The client processes are directed to make requests from individual server nodes in a round-robin fashion, emulating the actions of a "load spreading" network switch or round-robin DNS scheme. Because the page set in these experiments is static, each emulated request occurs independently of any other request, and thus each request could represent the actions of a single real-world user. To mimic the behavior of a large number of multiple remote users, each client emulator program requests a given number of web pages from the PRESS server machines in each logging period. For example, to achieve a target load of 800 page requests per second, each of the four client machines would generate 200 concurrent requests per second. The actual number of requests made in each logging period is not constant, however, because each client emulator is programmed to wait a variable amount of time before initiating each new request in order to generate load levels that fluctuate realistically, but without being so erratic that they complicate data analysis. The wait time between each request is determined by a Poisson process, with an average wait interval equal to the amount of time necessary to approximate the requested load level. Again, this model may not accurately reflect the "burstiness" of traffic on the Internet, where request arrival distributions may also be heavy-tailed [30], but the simplification is necessary to produce useful experimental data on small time scales.

Performance data are monitored and recorded at the clients, rather than at the servers, in the interests of obtaining performability statistics that reflect the end-user experience. The clients keep track of the average throughput and latency of requests over a given logging period (here set to be five seconds), and also record how many requests completed successfully, failed because of a connection error, timed out in the process of connecting to the server, or timed out after connecting to the server.

Figure 7 shows the states that each emulated client can inhabit while making a page request. The initial connection abort threshold, which would presumably be enforced at the user's web browser, is set at two seconds. After connecting, the client will wait six more seconds before failing the request with a timeout. This limit is a slightly more stringent requirement than the well-known "eight second" Internet user impatience threshold [40], but the dual two-second/six-second timeout may more accurately capture the patience levels of modern Internet users, given the growing numbers of broadband customers who are accustomed to high-speed data connections. Such users may terminate a page request quickly if a connection is not made within a few seconds, and after a successful connection may only be willing to wait a few more seconds for the page content to appear before giving up on the request.

The client machines record the latency of every attempted connection from the instant a request is submitted, regardless of whether or not the requested page is returned successfully. Under this scheme, transactions that fail immediately due to a connection fault will be recorded as lasting only a fraction of a second and providing 0% data quality. Requests that time out during the connection attempt will have a recorded duration of two seconds and be labeled as the result of an aborted transaction.

Once in each logging period (set to five seconds in the experiments), each client emulator sends its quality-of-service observations for that time slice to the client coordinator, which aggregates the client statistics across that period and logs them to disk.

3.3.3 Hardware and software

Each of the machines used in this study is a dual 1 GHz Pentium III workstation with 1.5 GB of RAM, connected via Gigabit Ethernet and utilizing a SCSI disk subsystem. The cluster nodes run Debian GNU/Linux, kernel version 2.4.20 SMP, as well as the custom PRESS, Mendosus and load generation software. In addition to the Gigabit Ethernet links, the cluster nodes are connected to each other by a separate Myricom network fabric, which is not utilized in these experiments.

3.4 Experiments

Each experimental test is designed to observe and record in real time the behavior that a version of PRESS (Perf-PRESS or HA-PRESS) would exhibit under a single type of component failure at a given workload level. Mendosus is capable of injecting multiple faults of different types at once, but before testing such non-independent fault models, it is necessary first to ascertain how each single fault type affects the behavior of the server. This study considers only single-fault results.

A test run consists of the following primary stages:

1. *Warm-up* (ca. 60 seconds) - Client programs make requests to the PRESS servers at a fixed average rate, populating the caches of the servers and establishing baseline throughput and latency levels for the cluster.

2. *Fault injection* (30-90 seconds) - A singe fault is enabled for an arbitrary length of time (between 30 and 90 seconds in the experiments), appearing constantly or at intermittent intervals

determined by a Poisson process. To compare the effects of brief faults to those of longer durations, the same fault may be injected for only 30 seconds in one test but persist for the entire 90-second injection phase in another test. Depending upon its configuration, the PRESS server may recover fully or partially from a failure, or else stop serving pages altogether. The emulated clients continue to request pages at a predetermined rate.

3. *Recovery* - If the effects of a fault continue for longer than 90 seconds, regardless of how long the fault actually was in effect, Mendosus mimics the actions that a human operator would take to return the system to a fully operational state. Such corrective actions may include rebooting a node or restarting some or all of the server applications on the cluster. All repair options that do not require a node reboot take 10-20 seconds. After a reboot, it may be several minutes before the cluster returns to a steady state.

4. *Post-recovery* (60 seconds) - Clients continue to submit requests to the server and any deviations from the baseline quality of service levels are noted.

Figure 8, adapted from a figure in the initial PRESS/Mendosus study [26], outlines a possible profile that the performance measurements might exhibit during the stages of a fault injection test run. Note the similarity to Figure 3.

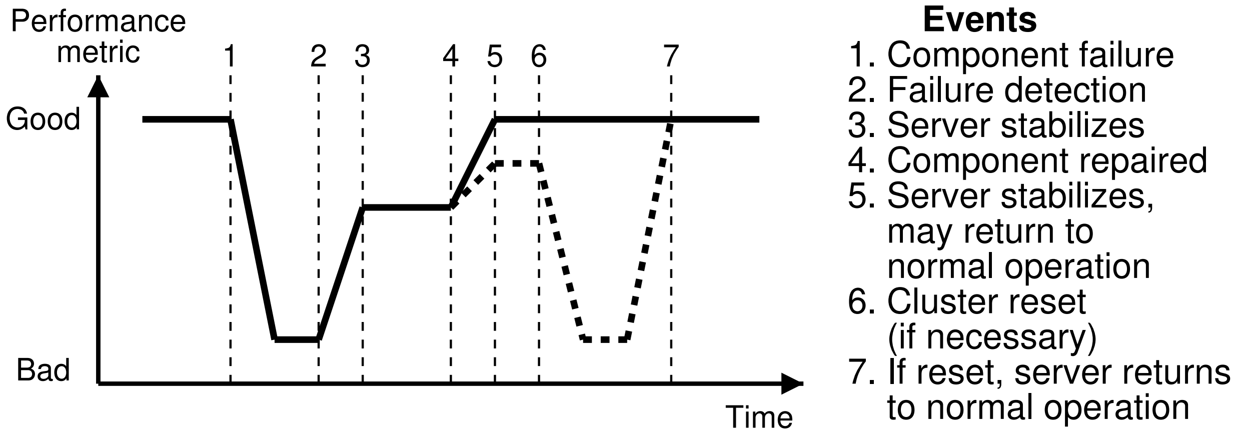


Figure 8: **The stages of recovery and possible behavior of a performance metric after a fault is injected into a running PRESS server.** This behavior will vary considerably based upon the metric chosen, the fault injected and the degree of recoverability offered by the version of PRESS being used in the experiment. For example, if the PRESS version supports automatic fault recovery without a full reset of the server processes, the behavior of the performance metric is likely to follow the solid line, but if the version requires a complete cluster reset to recover, the results will resemble the shape of the dashed line. Note that the latter version exhibits a period of complete cluster unavailability while the servers are restarting. This figure is a modification of the original, which appeared in the first PRESS/Mendosus report from Rutgers University [26].

The "signatures" of the faults injected in these experiments – that is, their duration and manner of appearance (transient or sticky) – are very similar to those used in the initial PRESS/Mendosus fault-injection study. These fault behaviors are themselves based somewhat upon speculation and hearsay, due to the scarcity of detailed system logs that record specific details about what occurs within an Internet service during an actual hardware or software failure. For this reason, the test stage durations listed above represent only a single point in the relatively unexplored space of real-world fault scenarios.

3.5 Differences from the Original PRESS/Mendosus Study

Because one of the primary goals of this project is to replicate the experiments of the initial Rutgers PRESS/Mendosus study, it is important to enumerate the ways in which the experimental setup for this study differs from that of the original:

- **Test hardware –** This study and the initial Rutgers study both feature four cluster nodes and four client emulation nodes. The workstations used in both studies have similar CPU speeds and memory capacities. The network setup is also quite similar; both studies feature 1 Gb/s connections between all nodes, although this study does not make use of a second networking protocol in some configurations, as the Rutgers study does with the VIA protocol.

- **Test software –** This study uses the same client emulation software as the original, as well as the same page set and request traces. The original study featured four different software configurations of the PRESS cluster, but this study only uses two of them: Perf-PRESS (called TCP-PRESS in the original) and HA-PRESS (originally called ReTCP-PRESS). One of the unused configurations has each of the four server nodes act independently of the rest, and the other relies upon the VIA protocol for communication between cluster nodes.

- **Fault types and behavior –** This study uses all of the Mendosus-emulated failure types from the original except for the SCSI disk faults. Fault durations and recovery behavior are very similar between the studies.

- **Data collection –** This study extends the data reported by the emulated clients to include response time measurements. In addition, this study examines the number of failed, timed out and aborted requests during the tests, where the previous study considered only the system's successful throughput.

- **Data analysis –** A central focus of the Rutgers study was the use of data from the fault-injection experiments, combined with the expected occurrence rates of the faults tested, to build a model of the long-term performability of each of the different configurations of PRESS. This study avoids making such long-term predictions and instead concentrates on how best to present and compare the latency-oriented performability results of each individual test.

3.6 Data Analysis

One of the advantages that fault injection systems such as Mendosus offer is the ability to automate the running of a large number of test cases. This feature allows researchers to collect data from multiple tests runs of a single fault scenario and then average the results from these test runs to develop a single representative measurement of the system's behavior under a given fault. This report, however, makes use of data collected during single test runs that nevertheless are considered to be representative of the system's average behavior unless otherwise noted.

During an experiment, the client emulators record the quality-of-service metrics described above and transmit them to the coordinator, which subsequently logs them to a file. After the experiment, these observations are compared with the fault injection time line to determine how the behavior of the cluster changed in response to the events of the test run. Further analyses compare the levels of service that the two different versions of PRESS were able to provide under identical fault scenarios. Having established a baseline normal performance level for each PRESS version, the analysis considers only the data collected during the actual fault injection phase of each experiment. Assuming that the service levels observed during the fault were relatively stable, the actual duration of the fault in the test case is less important to the analysis, since it becomes possible to estimate the performability of a PRESS version under longer and shorter fault durations given its average performance during the fault phases of the initial experiments.

In the following chapter, the data gathered during the experiments will be examined to determine how injected faults and recovery behavior affected the user-perceived performance of the system.

Chapter 4

Experimental Results

This chapter presents specific results from the PRESS/Mendosus fault injection experiments. This description will form the basis for the subsequent discussions in Chapter 5 of how to present latency data in an end-to-end performability evaluation.

4.1 Behavior of Response Times During Injected and Spontaneous Failures

Figure 9 compares the performance of an HA-PRESS system during a spontaneous deviation from normal processing to the behavior of the system during an actual injected fault. The first scenario, the server "hiccup," is a result of a temporary exhaustion of processing resources on the server machines, and the fault is a temporary "hang" of one of the four PRESS server processes. In the second scenario, the fault is considered transient and disappears automatically after 30 seconds. The "fault phase" of the experiment is outlined by the two dashed vertical lines passing through the graphs.

The injected fault results in an immediate spike in the number of connection timeouts as the server becomes unable to satisfy requests from already-established connections. Once this initial set of requests has timed out, any subsequent connection requests abort at the client's two-second threshold. The non fault-related "hiccup," however, causes a more gradual increase in the response time of the server until resources become available and operation returns to normal. This sequence of events usually does not result in full eight-second timeouts.

The most important feature to note in the figure is that the overall throughput of the servers, including all satisfied requests, connection timeouts and aborts, remains nearly constant, but the quality-of-service metrics such as dropped connections and response time vary considerably.

A note on the latency measurements: PRESS is designed to deny incoming requests when the number of open connections on the server reaches a certain pre-established threshold. The way that PRESS performs this version of "load shedding" is rather unusual, however. Rather than returning a "Server too busy" message or even a socket error, the server simply ignores the client's connection request. This behavior requires a minimum of effort from the server, but relies upon the client to decide when to terminate the connection attempt. In the case of the automated experiments described here, the connection timeout threshold is set at two seconds. This aspect of the client/server interaction explains

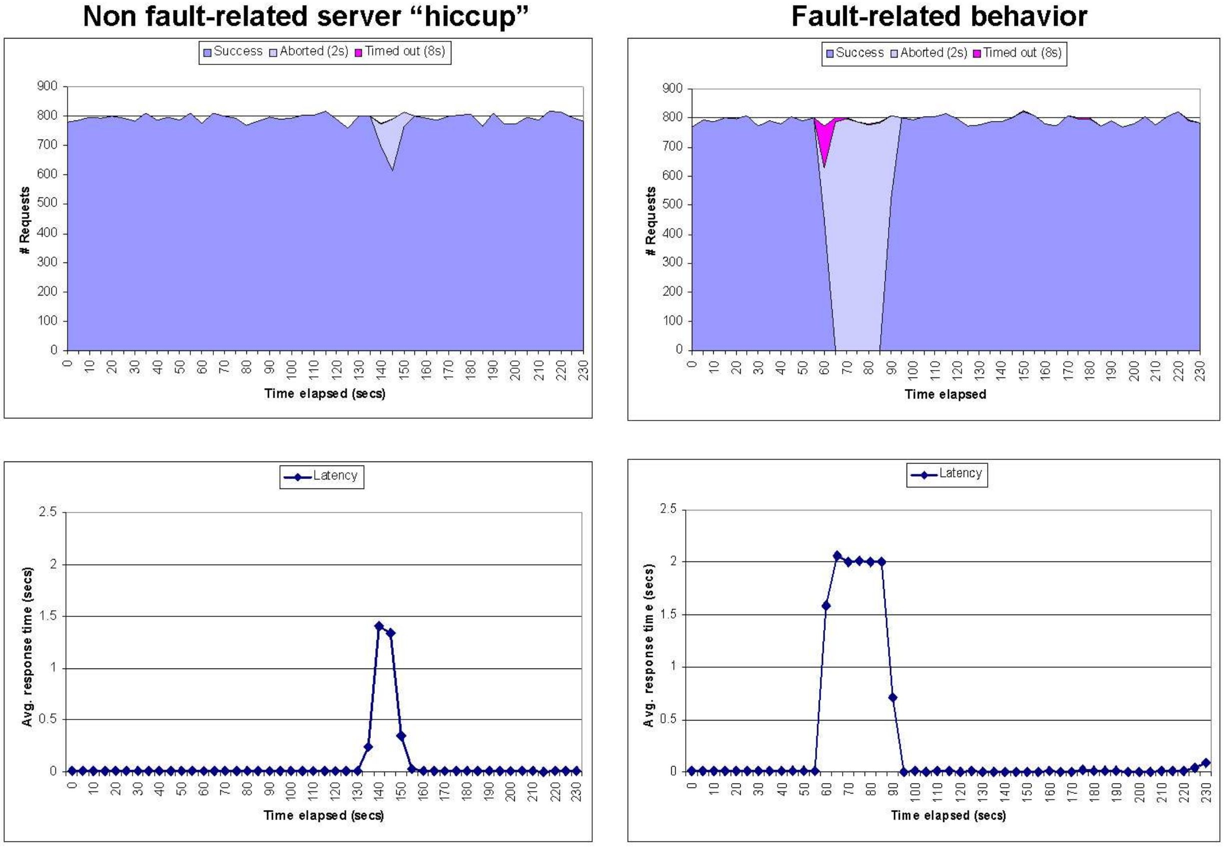


Figure 9: **A comparison of the throughput components and latencies of an HA-PRESS system during a spontaneous “hiccup” caused by resource exhaustion and an actual fault.** Note that the response times reach the connection abort time limit of two seconds during the actual failure, resulting in a 30-second interval during which all client requests either abort or time out. The service disruption that the spontaneous “hiccup” causes is comparatively minor.

why the average response time during failures appears clamped at two seconds; the appearance of this phenomenon indicates that most of the requests to the server are being ignored (presumably because it is experiencing fault-related difficulties) and the client connection attempts are simply timing out.

4.2 Advantages of the HA-PRESS Server Over Perf-PRESS

Figures 10 and 11 illustrate the performability advantages that the heartbeat monitoring and liveness checks of the HA-PRESS configuration provide over the less advanced Perf-PRESS.

Figure 10 displays the throughput- and latency-oriented behavior of both systems when one of the four PRESS server processes crashes unexpectedly. In both cases, the remaining three PRESS servers detect the failure of the fourth and remove it from the cooperative caching and load balancing pool. Since IP takeover of the crashed node is not enabled in either version of PRESS, however, all client

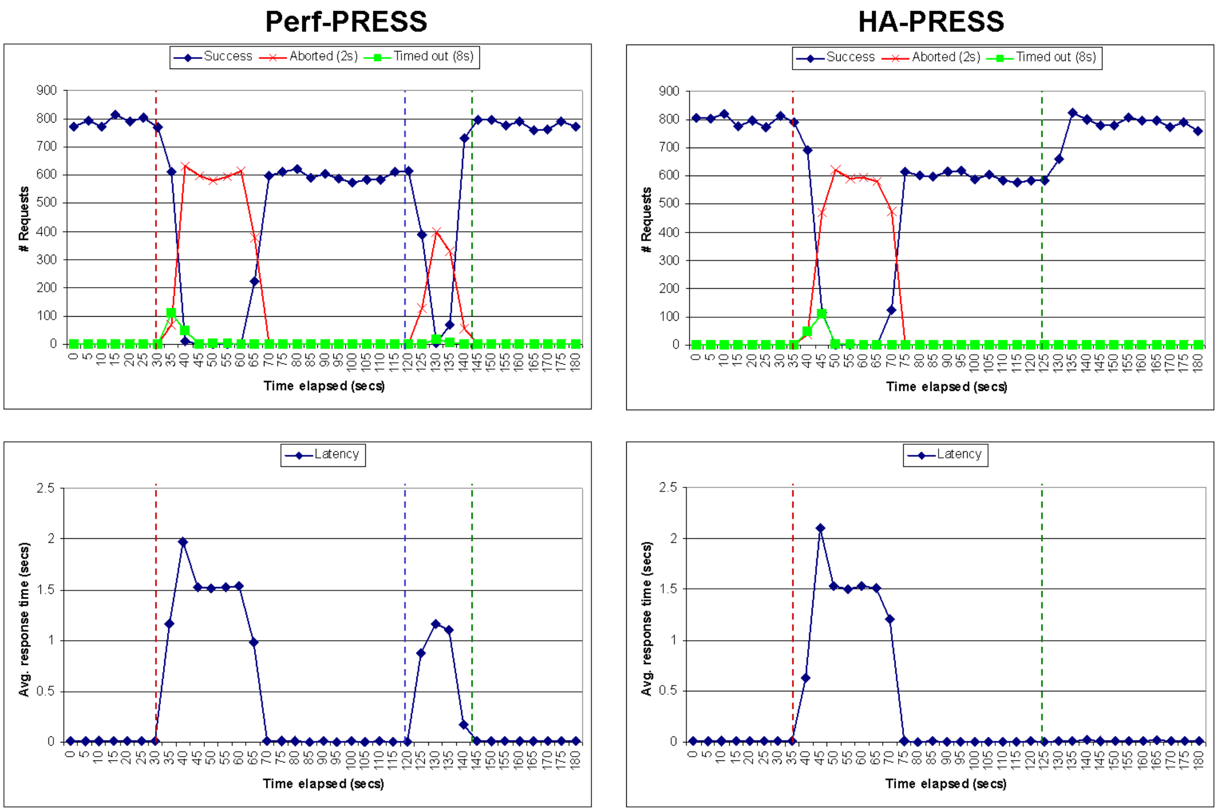


Figure 10: **Behavior of Perf-PRESS and HA-Press following the crash of one of the cluster server processes** (first dashed line). 90 seconds after the crash, the administrator takes action to repair the system. In the case of Perf-PRESS, this action involves restarting all of the servers (second dashed line), while HA-PRESS requires only that the failed server be restarted before the cluster returns to normal (final dashed line). Note the extra, severe disruption of service that Perf-PRESS incurs during the full-server reset, which HA-PRESS avoids entirely.

requests to the failed node return immediate socket errors, which explains why the overall throughput of the cluster drops to 75%. Since these connection attempts fail immediately, the average response time of the cluster returns to its usual low level after an initial reconfiguration period of around 40 seconds, during which a small number of connections time out or are aborted.

The server is restarted automatically when the simulated administrator "detects" the failure at the arbitrary 90-second detection threshold. Here the difference in server designs becomes evident. HA-PRESS was designed explicitly to allow nodes to leave and rejoin the cluster on the fly. Perf-PRESS has no such capability, so the administrator must restart all of the server processes at the same time in order to return a node to the cluster and thus allow the cluster to regain its optimal service levels requires. This action results in a short period of complete system unavailability, causing more failed connections and an increase in overall response time compared to HA-PRESS, which requires only that the failed process be restarted.

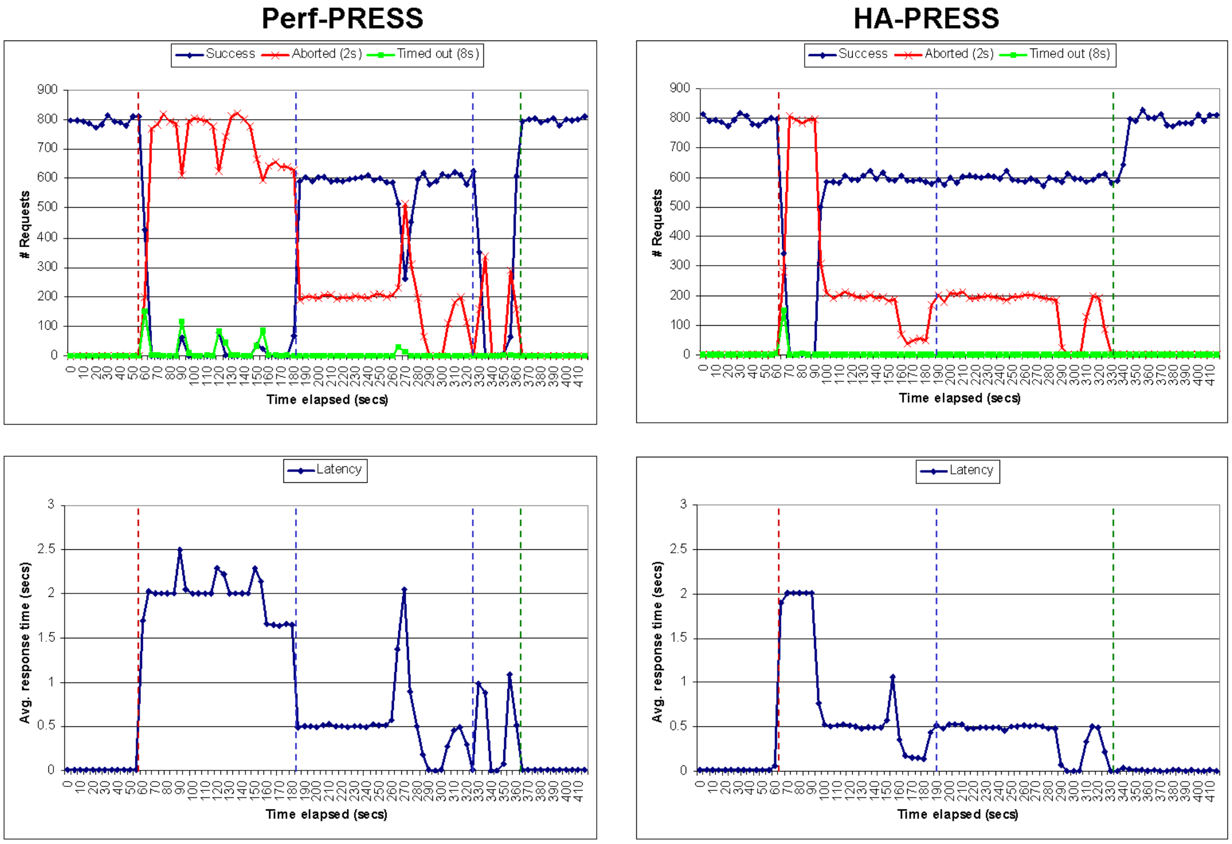


Figure 11: **Behavior of Perf-PRESS and HA-Press following a lock-up of one of the server nodes**, which is remedied by rebooting the afflicted node after 90 seconds (second dashed line). The fluctuations in the quality-of-service metrics during the reboot are the result of the node regaining partial levels of functionality at various points during its reboot. Once the node has returned to operation, the administrator must restart the entire PRESS cluster if the Perf-PRESS server version is being used (third dashed line). HA-PRESS automatically returns the rebooted node to its cluster topology once its server daemon restarts (final dashed line), once again avoiding the disruption of service that Perf-PRESS incurs during the cluster-wide reset.

Figure 11 displays the behavior of Perf-PRESS and HA-PRESS when one node of the cluster has become completely “frozen” due to resource exhaustion or an operating system-related bug. The frozen node is rebooted after the 90-second detection limit has passed. Once it has come back up, its PRESS process starts automatically. Once again, HA-PRESS is able to re-integrate the server into the pool seamlessly, while Perf-PRESS requires that all of the server processes be restarted.

The graph also indicates that HA-PRESS is better able to determine that the frozen node is unavailable, due to its heartbeat monitoring mechanisms. Perf-PRESS does not notice this fault until the node has been rebooted and temporarily removed from the network topology. Note also that connection attempts made on the rebooting node do not return errors because the operating system is not available to send a reply – instead, the connections simply time out at the pre-established two-second threshold.

4.3 Shortcomings of HA-PRESS

Under most reasonable workloads, HA-PRESS does not display significantly lower performance levels than Perf-PRESS. There are a few types of fault conditions, however, that cause more problems for HA-PRESS than Perf-PRESS. Figure 12 shows such a fault appears. In this situation, one of the PRESS server processes locks up for a short amount of time (here, 30 seconds) before continuing its processing. This scenario causes no particular problems for Perf-PRESS: the other nodes fail to notice the error and so they do not remove the stricken node from the server pool. Once the process has returned to normal operation, the cluster continues processing as usual.

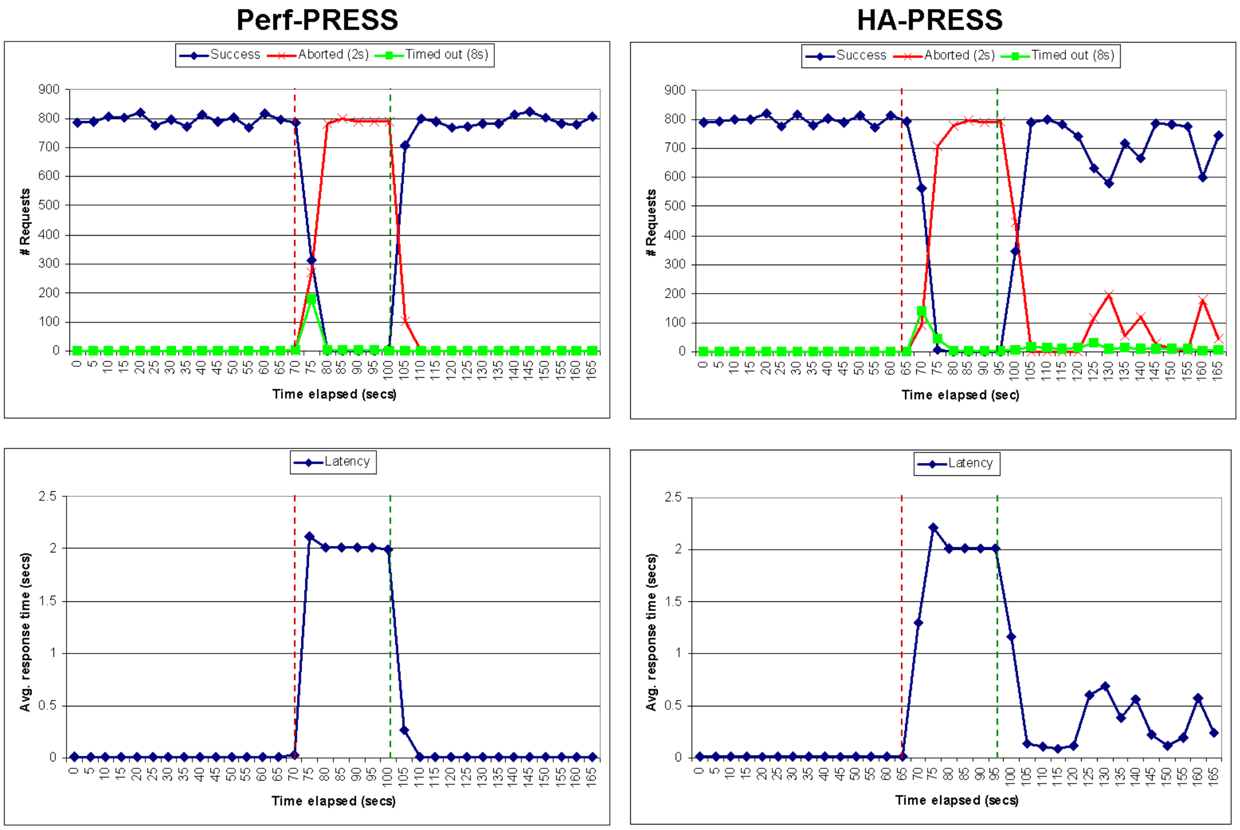


Figure 12: **Behavior of Perf-PRESS and HA-Press following a 30-second hang of one of the server processes.** Both server versions recover automatically, but HA-PRESS subsequently experiences difficulties because the messages sent by the affected node are out of sequence with the rest of the cluster. Although the graph seems to suggest that the HA-PRESS system will return to a steady state with time, the situation will not be completely remedied unless an administrator restarts the problematic server process.

This situation causes difficulties for HA-PRESS, however. HA-PRESS also does not remove the node from its server pool because the 30-second failure window is shorter than the time limit within which the node must reply to an active heartbeat check. And since the underlying operating system and hardware continue running during the application hang, there are no signals to alert the passive monitoring systems on other nodes that the process is frozen. Once the stopped process resumes, the cluster enters an impaired mode because the heartbeat, caching and load broadcasts from the affected node are all out of sequence with the rest of the cluster, causing messages to be dropped and cache entries to be erroneously invalidated. This behavior manifests as higher computational delays, resulting in connection timeouts and increased average response times.

HA-PRESS exhibits similar behavior when the network link of one of the server processes goes down for a short period of time. The resulting erratic behavior of the server can always be remedied by restarting the afflicted process, thus prompting it to resynchronize with the other servers. But since the design of HA-PRESS is supposed to minimize administrator involvement, this particular failure-related behavior of the server may be considered a design defect.

4.4 Sensitivity of Results to Changes in Experimental Variables

The experimental results described thus far are the products of tests in which the environmental variables that could affect the behavior of the test systems, such as the hardware components, software configuration, client request workload, page set and fault behavior, were carefully controlled. Between individual tests, the only variables that changed were the fault type and the version of PRESS under test, allowing direct comparisons of the behaviors of the two PRESS versions given various fault scenarios.

An important question to ask is how the system might react to changes in the other environmental variables. If the behavioral differences between Perf-PRESS and HA-PRESS remain fairly constant across a range of load levels, data sets and hardware versions, system designers (whose operating environments will invariably differ from those of these tests) can feel more confident about applying the design modifications and concepts that these tests find to be beneficial.

It is possible to use observations made in the course of this study to make several statements about the sensitivity of the PRESS server to changes in the operating environment, although these observations do not constitute a full sensitivity analysis of the test system. The first such observation involves the effect that changes in the client workload levels have on the behavior of the system. As discussed in Chapter 3, the nature of the user workload being applied to a system is of primary importance to any discussion of the overall user experience that the service offers. Even when conducting comparative experiments for which the user workload remains constant, as in the case of the PRESS/Mendosus experiments, it is important to consider if a change in the workload could significantly affect the evaluation of a service's performability.

Figure 13 displays the results of the same fault injection test applied to identical versions of the PRESS web server (Perf-PRESS in this case). The only difference between the tests is the workload applied to the systems. The workload in the first experiment averages 400 requests/second, while the second is 800 requests/second. The higher level is the default workload level for all of the other performability experiments presented here. The fault being injected in this case is a temporary "hang"

Figure 13: **A comparison of Perf-PRESS systems under two different workload levels, during and after an intermittent 30-second "hang" of a server process.** The first dotted vertical bar passing through the graphs indicates the time at which the fault was injected, while the second shows the time at which the fault ceased to be in effect. In this situation, the server recovered automatically from the failure, so no administrator intervention was required. Although only one out of four PRESS server processes was directly affected by the fault, the throughput of the entire system dropped to zero for the duration of the fault because such a hang on the part of one server disrupts intracluster communications to the point that no server is able to make progress. Specifically, the nodes of the cluster use TCP to send caching and load updates as well as page contents to each other. When one node in the system unexpectedly ceases transmitting such a message, the other nodes quickly freeze as well, waiting for the rest of the message from the afflicted node. The TCP-based version of PRESS used in these experiments does not implement any sort of timeout mechanism other than the standard TCP connection timeouts, which take several minutes to be triggered, so the system does not begin recovery until the fault itself has been remedied.

of one of the four PRESS server processes.

As Figure 13 shows, the fault-related behavior of the system does not change significantly when the workload is doubled, other than that the throughput success and failure rates both roughly double, as would be expected. One possibly significant deviation, however, is that the response time appears to reach its peak of two seconds more quickly under heavier load.

From this test and from other test results not shown, it appears that the overall fault-oriented behavior of the two versions of the PRESS server used in this study does not change significantly unless an abnormally high workload level is applied. Since an abnormally large workload would preclude the normal functioning of the server, such a “load spike” could itself be considered a type of fault condition. Because the present study requires near-constant workload levels to isolate behavioral differences between the two versions of PRESS, a discussion of the effects of abnormally high workloads is beyond the scope of this paper.

Other, more informal tests have indicated that both versions of PRESS may be highly sensitive to changes in the amount of processing overhead required to service a request. The effect of adding even an addition hundredth of a second of processing overhead to each request is immediately noticeable when PRESS is running at workload levels of hundreds or thousands of requests per node per second. Because each cluster node has only two CPUs, the extra processing overhead may, in the worst case, result in an inability of the cluster to satisfy all incoming requests in a timely manner, causing several aborted client connection attempts in each time period.

Because the behavior of the PRESS server seems quite sensitive to increases in per-request processing, it is important to view the results obtained under the static page set used in this paper as not representative of the abilities of the server to support high volumes of CGI-based queries or other types of interactive workloads. Under such workloads, it is quite possible that the maximum throughput of the server is a mere fraction of its capacity under a static workload, with per-request latencies also increasing significantly.

Changes that result in increased utilization of the network (for example, increases in the average file size or decreases in network bandwidth) would have a similarly deleterious effect on the user experience, resulting in more connection aborts and longer wait times for pages. In this case, though, the network, rather than the cluster itself, becomes the bottleneck. As with increases in the client workload, PRESS appears to exhibit few behavioral changes from the client’s perspective as the file size increases or the network capacity decreases. Once the network saturates, however, the overall quality of service that PRESS can provide to all of its clients drops dramatically.

4.5 Summary of Experimental Behavior

From the experimental results described above and the outcomes of other tests, it is possible to make several statements about the latency- and throughput-oriented effects of fault injection on the two versions of the PRESS web server that were tested.

First is the observation that varying the workload has no unexpected effect on the behavior of the metrics during failures unless the workload level is severe enough to cause impaired behavior on its own. Doubling the workload to a higher but sustainable level has the effect of doubling the successful throughput and increasing the average response time by a negligible amount. Ultimately, this trend continues up to a point at which a further increase in workload results in a catastrophic reduction in successful throughput and a corresponding spike in response time to its maximum level, both of which constitute a complete failure of the server due to overload.

Second, the automatic failure detection and self-recovering features of the HA-PRESS sever version allow it to provide considerably higher levels of service than Perf-PRESS during fault tests. For a few classes of injected faults, however, HA-PRESS's recovery mechanisms prove to be more of a hindrance than a help.

Finally, a comparison of the fault injection graphs reveals that the successful throughput of the server at a given point in the simulation tends to mirror its response time. Due to the nature of the test system, an event that causes a reduction in successful throughput usually causes a simultaneous increase in response time. Because the server in these simulations performs no client-specific processing and, moreover, is explicitly stateless with respect to client connections, it is not surprising that we do not see variations in response times that are independent of reductions in successful throughput. Another consequence of the test setup is that response time during normal operation remains consistently near optimal, but quickly degrades to the worst possible average value (2 seconds, in this case) once an error occurs. Test scenarios that may exhibit less of a correlation between response time and successful throughput are described in the Further Work section of Chapter 6.

Chapter 5

Discussion

Although the latency-based performability measurements described in the previous chapter were not as interesting or widely applicable as might be hoped, further analysis of the data can still aid the development of better performability metrics for Internet services. An important step toward achieving this goal is to consider the various ways in which latency-related performability measurements may be presented, both on their own and combined with throughput and data quality measurements.

This chapter will explore the advantages and disadvantages of different methods of presenting performability results, taking cues from metrics already employed in industrial benchmarks. The results of a handful of the fault-injection tests described in Chapter 3 will be used as data points to facilitate this comparison. After outlining the basic methods used to compare the output of complementary test runs, this discussion will begin its consideration of different performability metrics by applying the simplest, such as total throughput and response time, to results from the fault injection tests. Subsequently, this chapter will examine conceptually more complicated evaluation methods, commenting upon their descriptiveness, ease or difficulty of application, and their suitability for different types of performability summaries.

5.1 Comparing Tests of Different PRESS Versions

Each simulated component fault lasts for either 30 or 90 seconds in the PRESS/Mendosus tests. Faults that persist for only 30 seconds are assumed to be transient, and those lasting up to 90 seconds are presumed to be repaired at the end of the time period, usually via a reset of the afflicted node or process. The *fault duration* for each test – the amount of time during which part of the system is not operating properly – does not change between the Perf-PRESS and HA-PRESS tests. The varying behaviors of the PRESS versions, however, produce differences in the *failure duration* metric – the amount of time for which the service is not operating correctly. In fact, if repairing the fault requires that all or part of the system be rebooted, the failure may last much longer than the initial fault. Because Perf-PRESS and HA-PRESS handle faults differently, the failure durations observed while testing a single fault type vary considerably between the two versions of PRESS. Table 2 shows these differences.

These differing failure durations can pose problems for evaluators who wish to compare the performability measurements of different PRESS versions under the same fault. Artificially elongating

Fault Type	Version	Fault Duration	Failure Duration	Reason for Difference
Application hang	Perf-PRESS	30 sec	35 sec	N/A
	HA-PRESS	30 sec	35 sec	
Application crash	Perf-PRESS	90 sec	105 sec	Must restart Perf servers but not HA
	HA-PRESS	90 sec	90 sec	
Node crash	Perf-PRESS	90 sec	195 sec	Must restart all Perf servers after reboot
	HA-PRESS	90 sec	170 sec	
Node freeze	Perf-PRESS	90 sec	300 sec	Must restart all Perf servers after reboot
	HA-PRESS	90 sec	280 sec	
Link down	Perf-PRESS	90 sec	105 sec	Must restart Perf servers but not HA
	HA-PRESS	90 sec	95 sec	

Table 2: **The differences in fault duration (length of component failure) and failure duration (length of service outage or degradation) for different versions of PRESS under the faults included in this analysis.** Note that Perf-PRESS exhibits noticeably longer failure durations because it does not automatically re-configure the cluster when a repaired node rejoins the pool. Instead, all of the PRESS processes must be restarted simultaneously. Note also that the long failure durations after node lock-ups are due to the time needed to detect the failure (90 seconds) and the time required for the node reboot.

one system's failure duration to match the other permits direct comparisons between the two, but this modification sacrifices the descriptiveness of the measurements by mixing failure- and non-failure oriented measurements for the system with the shorter failure duration. The approach taken in this report is to present only those statistics gathered during the service failure, which requires that the system evaluator be aware of the differences in failure durations between versions of PRESS.

Analysis of only the behavior recorded during failures facilitates summarization and comparison of fault-related behavior but does not take into account the behavior of the system after the fault is repaired. As noted previously, the HA-PRESS server continues to exhibit occasionally unstable behavior even after it "recovers" from a short application hang or link failure. This behavior is not detrimental enough to qualify as a failure, though, so it is disregarded in our performability analysis. A more thorough presentation of the system's performability over time may, however, choose to incorporate such post-fault behavior.

5.2 Raw vs. Average Throughput and Latency

An obvious way to present both throughput- and latency-oriented test results is simply to compare the successful throughput, defined as the number of requests successfully or partially satisfied, and total response time (in seconds) of each test system during the failure period of the fault-injection tests. Figure 14 displays these values as recorded in ten representative test scenarios, showing the behavior of both versions of PRESS under five different fault conditions.

One drawback to raw metrics is that although they may be useful for illustrating the differences in

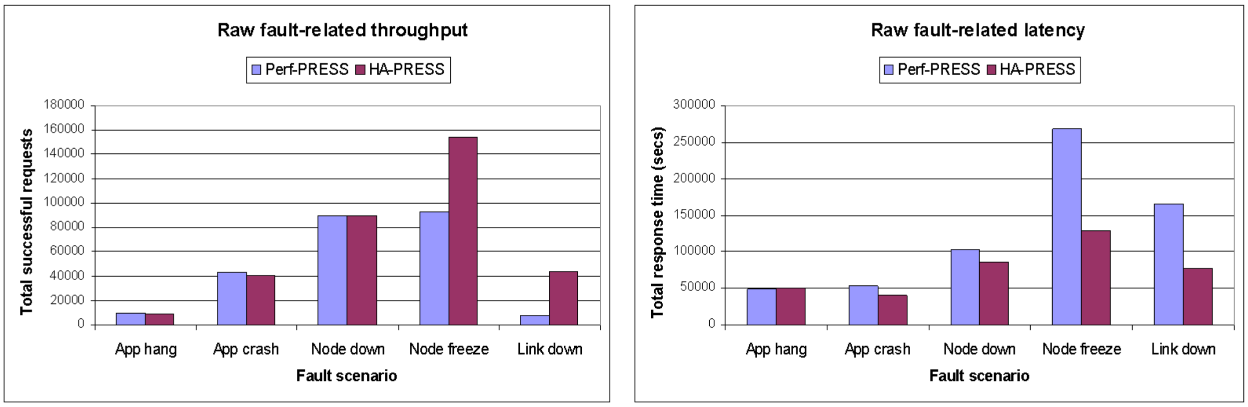


Figure 14: **The total throughput and client-perceived latency measurements for each version of PRESS in five fault-injection experiments.** Each bar of throughput represents the total requests handled successfully during a single fault injection test. Each latency bar represents the total wait time experienced by all clients during a single test. Because the different fault injection scenarios did not exhibit identical running times, this type of graph only permits direct performability comparisons between different PRESS versions under the same fault type. Chapter 3 describes the experimental methods used to collect these results.

performance between two configurations under the same fault scenario, they are not as successful at facilitating comparisons across different fault scenarios. The origin of this shortcoming is that raw metrics do not take into account the fact that some fault tests run for different lengths of time. In Figure 14, for example, it may not be immediately apparent that the first test scenario was of a much shorter duration than the last, even though this is indeed the case. If observers erroneously assume that all test scenarios run for the same length of time, they may draw the wrong conclusions about the comparative performabilities of the systems being tested.

An obvious way to remove test duration as a variable from the presentation of results is to compute the average throughput and response time for each test. This conversion enables direct comparisons and combinations of results from different test cases. Figure 15 gives these average values for the test cases, calculated as the mean number of requests served per unit time and the 95% trimmed-mean wait time for all requests. The use of a trimmed mean for latency, rather than a simple mean, may help prevent a few outlying data points from having an inordinately large effect on the average.

Although averages efficiently distill multiple performability measurements down to a single, universal metric, they also disregard nuances in the system behavior that may also affect the user experience significantly. Nevertheless, summary metrics are necessary to facilitate large-scale modeling of the system being tested. For example, the original Rutgers PRESS/Mendosus study modeled the long-term performability of four different PRESS versions by combining average throughput values recorded in fault injection tests with the estimated failure rates of each component in the system. Although such models are limited in their ability to predict the actual long-term fortunes of the system in question, they can be used to justify the deployment of one version of PRESS over another, or to do pre-deployment modeling of the expected effect of a new design upon the service's expected long-

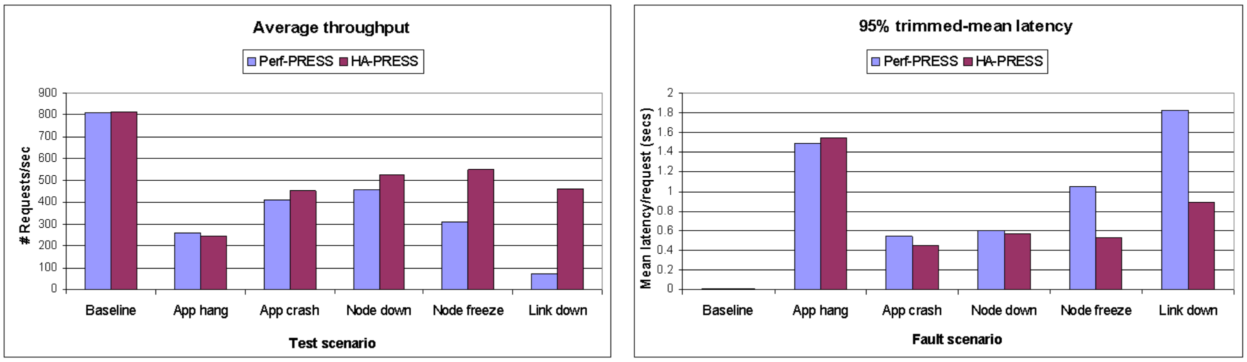


Figure 15: **The average throughput and 95% trimmed-mean response time measurements for both versions of PRESS in five fault injection experiments.** The baseline (ideal) average throughput and latencies from non-fault control tests are given for reference. Because this type of graph represents average, rather than absolute results from each test, it allows comparisons across fault types. For example, it is possible to conclude from the graphs that a "link down" fault has a much more severe impact on the performability of Perf-PRESS than an application crash.

term reliability [26]. Researchers at Rutgers University have also proposed the inverse of the average latency as an alternate summary metric that can be used in place of or in addition to mean throughput in such long-term models [22].

5.3 Goodput and Punctuality

An issue with mean response time as a latency-based performability metric is that the mean of the per-request latency does not supply any information about how close that average lies to the response time that the system can provide under ideal (fault-free) conditions. If the ideal, or "baseline," performance of the system during normal operation is also presented, as in Figure 15, the viewer may be able to infer this relation. In some cases, though, it may be helpful to quantify this difference as a single metric. A proposed metric to provide this information is *punctuality*, defined as

$$punctuality = \frac{fault\!-\!related\ mean\ latency}{ideal\ latency}$$

where the *ideal latency* is not zero (since that would be impossible and also would result in divide-by-zero errors in the punctuality equation) but rather the average response time of the system during a fault-free run of the test setup. The higher a system's punctuality, the more successful it was at preventing the effects of the failure from manifesting themselves as longer wait times.

Figure 16 shows the punctuality of the test systems as computed according to the equation given above, as well as the *goodput* of the system. In these experiments, goodput is defined by the following ratio, which produces results quite similar to the mean successful throughput values in Figure 15:

$$goodput = \frac{fault\!-\!related\ average\ throughput}{ideal\ average\ throughput}$$

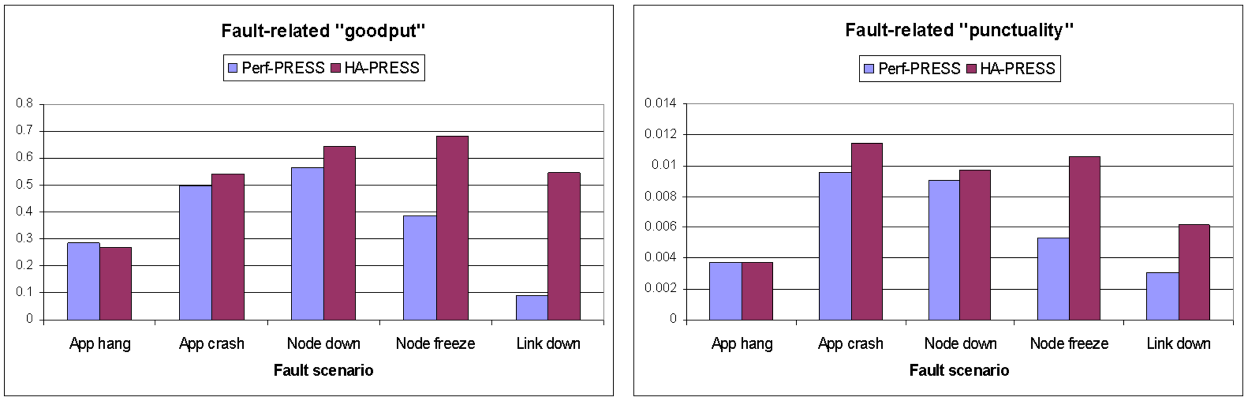


Figure 16: **The *goodput* and *punctuality* measurements of the test systems**, where goodput is the fraction of the ideal throughput the system achieved during the test and punctuality is the fraction of the ideal response time the system provided in the presence of faults. Graphs of these metrics allow a system evaluator to see how well each system version was able to approximate its best-case performability (which would be 1 on each graph) under each fault scenario, without requiring the presence of an "ideal" (baseline) entry on each graph. The absolute values of the punctuality scores are much lower than the goodput scores because the average time necessary to serve page requests under ideal conditions is quite small compared to the failure-related response time. The ratios between ideal and failure-oriented throughput levels, however, are not as great.

The ideal average throughput of PRESS would represent the successful completion of all requests submitted in a time period. Much like raw availability (the vaunted "nines"), both goodput and punctuality can be expressed as fractions, where 1 is the ideal value.

5.4 Response Time Distributions and Standard Deviations

The use of punctuality as a performability metric raises another issue: whether latency-oriented performability metrics should attempt to capture the variance in response times that end users experience during a failure – often referred to as the system's response time "jitter." A general consensus exists among operating system designers that minimizing the variability of an interactive system's response time may be more important than minimizing the average response time of that system [33]. In other words, it may be more important that the system's behavior be consistent, rather than faster overall but occasionally inconsistent.

The justification for this viewpoint arises from experiments with human subjects, which have shown that computer users have certain thresholds of patience beyond which their behavior toward the system will change. Nielsen [27] explains that the time limit after which a user will no longer feel that the system is reacting instantly to input is around .1 seconds. In addition, the wait time limit beyond which the user's "flow of thought" is interrupted is approximately 1 second. Finally, the maximum time after which a waiting user's attention will no longer remain focused on the current task is about 10 seconds.

These observations, combined with speculation about the possible negative effects on a commercial web site's business when a user loses interest in a purchase, have led companies such as Keynote Systems to assign monetary penalties to unacceptably long wait times [40]. In addition, certain benchmarks, such as the TPC-C online transaction processing benchmark [37], stipulate limits on the variability of the user-oriented wait times associated with certain actions. The TPC-C benchmark will summarily disqualify any candidate system that does not keep the 90th percentile of its response times below a given threshold. The goal of this requirement is to ensure that the system's response time curve is not so heavy-tailed that a handful of very long-running transactions make up most of the total wait time – in short, the response time of the system must be consistent.

Given that users may lose interest in a site if faced with inconsistent behavior, it is important to consider how to represent the variability in response time that a service exhibits during failures. One way to represent this value would be through a calculation of the sample standard deviation of the latencies observed during each test. Such a comparison would show how consistently one system configuration behaves compared to another, but the resulting graphical representation would provide no information about the actual response times offered by those systems. An alternative representation that provides much more information about the response time distribution in each test is a box-and-whiskers plot of these distributions, as shown in Figure 17. This graph, plotted here on a logarithmic scale due to the low median response times in the tests, provides information about the average response times, variances, outliers and interquartile distances of the underlying response time distributions.

5.5 Probabilities of Exceeding Thresholds

The box-and-whiskers graph in Figure 17 provides a great deal of information about the underlying response time distributions in each test, but it is not particularly helpful at reducing the latency-related behavior of the system down to a single statistic. Such a reduction would be necessary for talking-point benchmark results and long-term performability modeling. A candidate method for describing the response time distribution in a small number of figures was proposed by Douceur and Bolosky in their evaluation of a streaming media server [12]. Citing the user-related wait time thresholds established by Nielsen, they suggest that a service's latency-oriented performability can be represented by the probability that the service will exceed one of the three major wait time thresholds given above. Figure 18 gives these values for each of the test scenarios examined in this study. Since 10-second wait times were not possible in the PRESS/Mendosus tests due to the simulated clients' 8-second timeout threshold, the final latency threshold was set to > 2 seconds. Any request that reached this threshold probably was an aborted connection request or an eventual 8-second timeout.

Of course, there are situations in which system evaluators may want all of the available response time data from each test. In this case, a histogram of response times or a continuous graph of the number of requests that exceeded each possible response time would be the most appropriate representation. The viewer would then be free to examine the distribution in detail, possibly identifying response time thresholds at which the system's performance becomes markedly better or worse. Unfortunately, the data reporting method used in the PRESS/Mendosus experiments precludes such a representation: the test clients reported only the mean response time and total number of successful, aborted and timed out requests for each 5-second time slice, rather than providing the wait time for every individual request. It was possible to "reconstruct" the distributions from these data to produce a

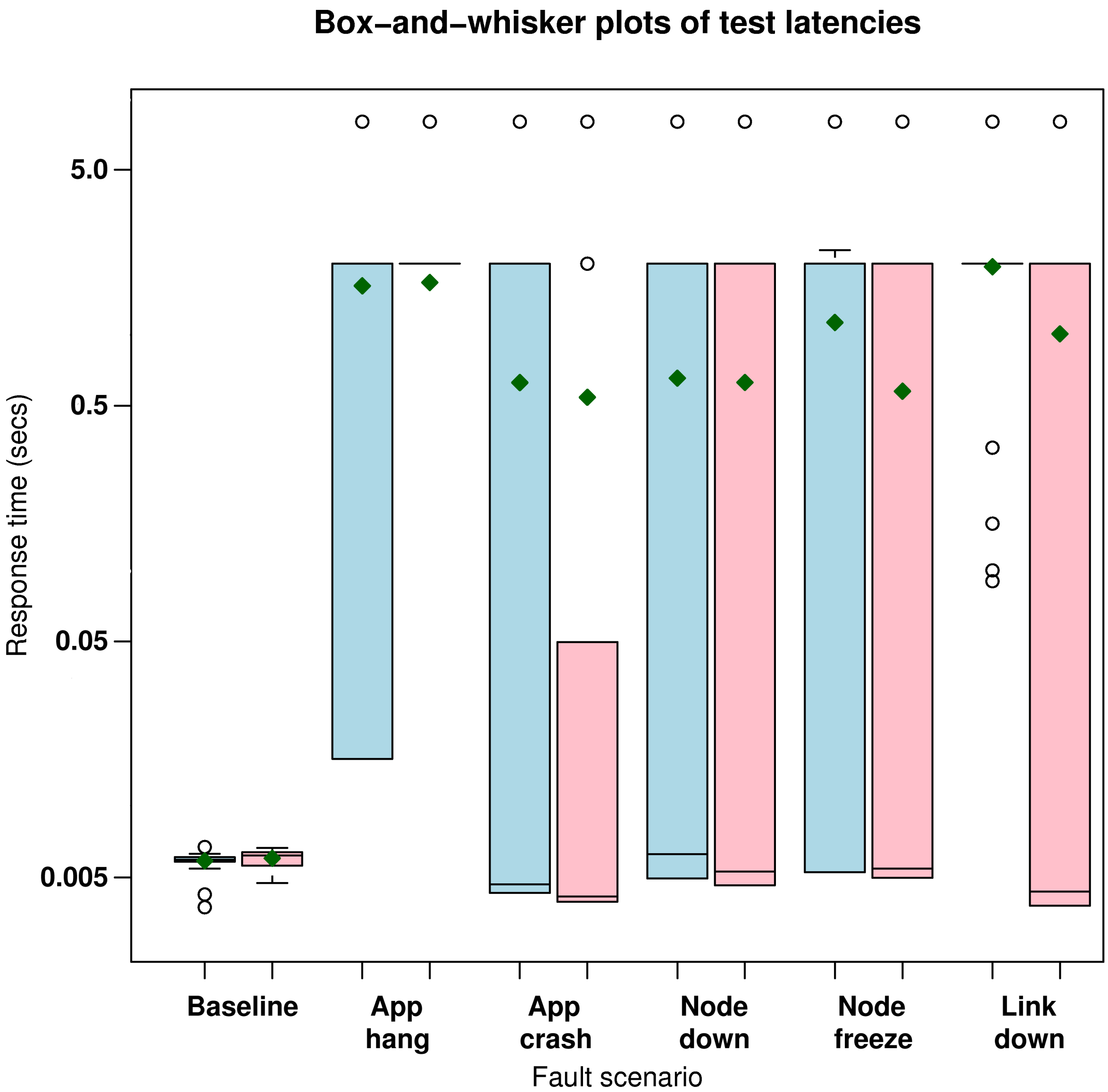


Figure 17: **A box-and-whiskers plot of the response time distributions observed during the PRESS/Mendosus fault injection tests.** The left box of each pair describes the latency-oriented behavior of Perf-PRESS during a fault injection test. The right box represents the resulting response time distribution under HA-PRESS. The dark-colored diamond in each column represents the simple mean of the response time for that fault scenario, and the line through each box corresponds to the median of that distribution. Note that the response time axis is on a logarithmic scale to display more clearly the relatively small interquartile ranges of the response time distributions. The outliers of these distributions (empty circles) appear around the intervals of 2 seconds and 8 seconds as a consequence of the programmed behavior of the clients. Note also that the median response times of most distributions are considerably shorter than their mean response times, indicating that the simple means are rather skewed toward longer wait times. This graph also provides visual evidence that the differences between the response time distributions provided by Perf-PRESS and HA-PRESS tend to be significant.

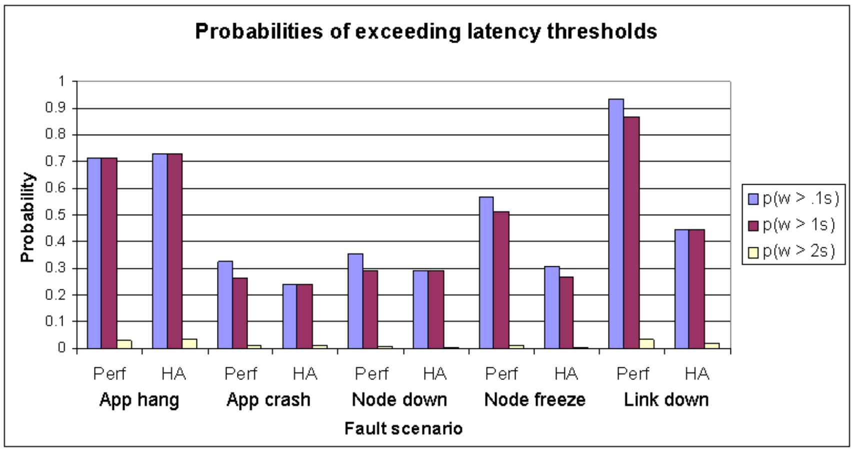


Figure 18: **The probabilities, based on the PRESS/Mendosus experiments, that a given request during a fault injection test scenario will exceed any of three wait time thresholds**: .1 second, 1 second and 2 seconds. Note that this representation implies double counting by definition: a response time that exceeds 1 second also must contribute to the count of wait times that exceed .1 seconds. This type of graph provides a less complicated representation of the response time distribution observed in each tests than a box-and-whiskers plot or a response time histogram, but still provides vital information about the ability of the test system to satisfy a user's response time expectations.

reasonably accurate box-and-whiskers graph for Figure 17, but these reconstructed distributions were not accurate enough to produce useful histograms.

5.6 Combined Demerits

The methods described above, especially the punctuality, boxplot and threshold probability graphs, provide useful ways to analyze and present the user-oriented response times offered by services during failures. But what if a system evaluator wants to combine throughput, response time and possibly data quality-related metrics into a single numeric descriptor of a system's performability? In this case, some kind of arithmetic method for weighting and combining these disparate results into a single figure is necessary. The basic unit of such a summary is often called a "demerit," meaning an abstract quantification of something that degrades the user experience [40]. According to the whim of the system evaluator, deficiencies in latency, throughput and data quality may be assigned varying numbers of demerits, which are then combined in arbitrarily complex ways.

Here is a sample demerit system that combines all of the possibly detrimental events recorded by the client simulators in the PRESS/Mendosus tests, including data quality-related connection aborts and errors as well as response times:

- Each aborted connection: 2 demerits
- Each connection error: 1 demerit

- Each user timeout: 8 demerits
- Each second of average request latency above the ideal level: 1 demerit $*$ a scaling factor to equalize the contributions of data quality and latency to the overall demerit value.

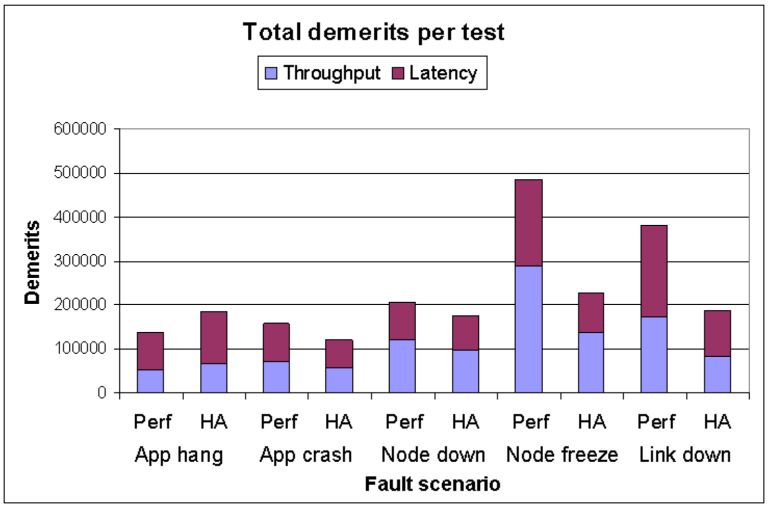


Figure 19: **The total user-experience demerits accrued by Perf-PRESS or HA-PRESS during each test scenario**, according to the demerit equation given above.

Figure 19 shows the results of computing these demerits for each test scenario.

Given the necessarily arbitrary and application-specific nature of demerit-based metrics, however, it is quite possible that system designers may derive more accurate impressions from an evaluation that presents throughput, latency and data quality via separate figures and summary statistics.

Chapter 6

Conclusions and Future Work

The primary purpose of this work is to contribute to the discussion of how computer scientists should measure, analyze and present performability data for online services. In particular, this study explored how to gather and present experimental data about response time in order to describe the end-user experience an online service offers in the presence of faults.

The method for facilitating this discussion was to observe the outcome of fault injection experiments performed on a prototype web server. The study then presented several methods for summarizing the failure-related throughput and latency measurements collected during the experiments, in the hopes of producing ideas and metrics that may be applicable to subsequent performability evaluations of other systems.

6.1 Conclusions

In general, the experiments conducted for this study did not reveal any particular descriptive advantage that latency as a performability metric offers over throughput. The ultimate usefulness of response time measurements, however, remains an open question, because the rather simplistic and artificial nature of the experimental setup was not representative of the operating conditions of most real-world online services.

6.1.1 Test system behavior

PRESS, the prototype cluster-based web server examined in this study, tended to maintain a steady state of very low latencies and consistent throughput levels until a failure occurred. During a failure, the server often experienced a period of near or total unavailability, characterized by very low throughput levels and mostly timed out or unsuccessful connection attempts. Depending on the version of the server being tested, PRESS could recover partially to some degraded level of service prior to being repaired. During the partial recovery phase, the server would either provide a constant level of degraded service, such as satisfying 75% of all incoming requests, or else oscillate between periods of partial and complete unavailability.

The system behavior observed under the described request and fault loads was very similar to the behavior noted in the previous Rutgers study. This study was able to replicate all of the fault-related changes in system throughput described in the original PRESS/Mendosus paper /citemendosus-press, with the exception of the SCSI fault effects, since this investigation did not include disk faults. The original Rutgers study did not describe the response times provided by the cluster under faults, but it may be assumed that this behavior, too, was not significantly different from the response times observed in this study.

During failures, the response times provided by the PRESS cluster tended to mirror its throughput, in that the server would exhibit low latencies when it was able to maintain a high volume of throughput and vice versa. This inverse relationship between latency and throughput suggests that one metric may be derived from the other and raises questions about the necessity of collecting both response time and throughput metrics during failures. The server and the client models used in the tests were very simple compared to those present in most Internet services, however, so it seems unwise to draw conclusions about the relative merits of throughput and latency metrics based upon these results alone.

The experimental results themselves emphasize the importance of minimizing the time to fault detection and reducing the impact of recovery operations (such as reboots) on the provided quality of service. As would be expected, the PRESS version that provided faster and more complete recovery with a minimum of downtime scored more highly than the other in nearly every performability comparison.

6.1.2 Applicability of test results

The particular abilities of the PRESS web server – high-bandwidth, low-latency transmission of a mostly static page set – occupy a rather small niche in the larger world of online services. Although the ability to handle successfully a large number of static content requests is a necessary capability of many web sites, this feature alone is not sufficient to enable most online businesses to be successful. More important to the viability of a complete online service design is the ability to provide specialized computation or data manipulation services to individual clients. These requirements call for a greater number of system components than PRESS offers, including a database backplane, dedicated computation and session-state storage servers, and designated management and monitoring computers to assist human operators in managing such systems.

Certainly, PRESS could be modified to provide some of these services – for example, a service could implement server-side processing on the PRESS cluster machines, as depicted in Figure 1 in Chapter 2, or else simply use a PRESS cluster as the service's "front-end" machines, which serve static data to clients and forward more complicated requests to dedicated computation and database interface servers. A service might even deploy a PRESS cluster as a heavy-duty, reliable "edge" cache server to serve recent copies of the service's data to the set of clients that is closest to the cluster on the Internet.

In general, the limited capabilities of PRESS as a web server raise questions as to whether the design strategies that result in better performability on the part of PRESS are more widely applicable to the design of other online services. At the least, however, the improvements suggested by a performability analysis of PRESS can be tested on different types of services as part of future research. This avenue has already been explored, with considerable success, in the efforts of the Rutgers researchers to apply the idea of fault model enforcement to other types of online services [25]. And, indeed, the other design issues highlighted by this research – the importance of redundancy, the need for fast error detection and diagnosis, the great reliability gains that fast, accurate recovery mechanisms can provide – echo ideas that are currently topics of much research in the field of computer system reliability.

Finally, it is entirely possible that the most widely-applicable contribution of this report may not be its performability analysis of the PRESS web server and subsequent design suggestions, but rather the performability metrics and benchmark presentation techniques described in Chapter 5.

6.1.3 Presenting performability results

Of the methods described for representing latency, some were better suited to facilitating comparisons between different versions of the server under the same fault, and others proved more useful for comparing the behavior of PRESS versions across several different faults. The latter type of metric is more suitable for use in models of long-term system performability because such metrics allow for the synthesis of failure data from multiple tests. Raw throughput and response time measurements, for example, do not provide information about the duration of the outage over which they were gathered and do not indicate what the ideal behavior of the system should be. More sophisticated measurements like goodput and punctuality satisfy these two requirements and still are able to describe the behavior of the system via a single figure.

Some methods of representing response time (and thus describing the end-user experience offered by the service) were best employed to describe very specific aspects of the overall behavior of the system. For example, a box-and-whiskers plot of the response time distributions in each test is a visually appealing and fairly intuitive way to describe the variations and ranges of response times observed during tests. And although a histogram or cumulative graph of the response times recorded during individual tests would be the best way to present a wealth of data about the precise frequencies of the observed latencies, a graph showing the probability that a request will exceed certain significant response time thresholds is a good way to summarize this information.

Although the experiments and subsequent data analyses described in this study do not point to the supremacy of response time as the preferred metric for evaluating an online service's performability, they do demonstrate how latency-based measurements can be included with other quality of service metrics to produce a comprehensive representation of a service's end-to-end performability.

Ultimately, advances in the descriptiveness and usefulness of performability metrics will increase their visibility in the realm of online business. The adoption of these metrics by large numbers of online services will help raise awareness of the reliability-oriented challenges facing large-scale services and hopefully encourage the development of new techniques for meeting these challenges. In addition, online businesses may find that performability metrics, combined with knowledge of the operational costs of a service, can be useful tools for modeling both the income and expenses of an Internet-based company.

6.2 Future Work

The issue of dependability benchmarking is of paramount importance to the design of highly reliable online services, and it is questionable whether the field can make significant progress until there are accepted, straightforward ways to compute, analyze and compare the reliability "scores" of online services. Previous research on this topic has suggested that designers of such metrics should set the development of large-scale, standard dependability benchmarks, similar to the TPC-C and TPC-W benchmarks for online transaction processing [28], as their ultimate goal. In the process of working toward such benchmarks, however, designers should first experiment with new techniques for evaluating small portions of the overall reliability of systems. Such "micro-benchmarks" might include fault injection-based performability tests or usability evaluations of system administrator interfaces.

By exploring response time, a previously neglected dependability metric, this report represents a small step toward the development of more comprehensive dependability benchmarks. Significant advances in the fields of realistic fault injection and system modeling, however, are still necessary to describe the complex interactions of user workloads, system configurations and unexpected occurrences that conspire to threaten system dependability in the real world. As discussed in Chapter 2, the DBench [11] and SIGDeB [16] projects constitute the most organized and forward-thinking efforts to develop a common framework and language for describing dependability benchmarks. Although these projects tend to focus upon transaction-processing systems and general-purpose operating systems, it is important to consider their conclusions and experiences when discussing how to implement realistic and useful fault injection tests for online systems.

Improvements to the PRESS/Mendosus infrastructure would result in more realistic and informative results from future experiments that use the test systems described in this study. Specifically, the use of a programmable switch or IP takeover scheme to divert requests from failed servers to viable ones would produce better failure-oriented measurements. In addition, a modification of the PRESS load-shedding policy to make it explicitly deny client requests or return some kind of "Server too busy" message when the server is approaching overload would more accurately reflect realistic operating conditions and could potentially result in more informative response time measurements.

Another issue left unexplored by this study is the effect that support for dynamic content creation and stateful user sessions by online services would have on elements of the end-user experience, particularly response time. Although the PRESS server contains provisions for CGI scripting, dynamic load balancing and the caching of dynamic content, these abilities were not utilized for this study. Many more observations about the behavior of latency metrics during Internet service failures could be gathered if such capabilities in PRESS or some other, more interactive Internet service platform were tested as part of a comprehensive fault injection evaluation. In particular, it is quite possible that the inverse relation between successful throughput and latency observed during failures of PRESS would become much less evident on a test system in which the response time also changes in response to high processing load on the server side.

Further performability evaluations should also recognize the benefits offered by comprehensive data collection. Due in part to the large volume of requests the test clients generated, the PRESS/Mendosus study described in this paper collected only basic summary statistics (average response time, total throughput, number of failed and timed out connections) during each five-second time slice. Considerably more sophisticated analyses and representations of the data could have been provided if the test setup had recorded more per-request data, especially regarding response time and data quality. Such a policy may require compression and logging of results during the tests, but the potential value of the data collected far outweighs the programming challenges posed by these requirements.

Acknowledgements

First of all, I wish to thank my advisor, David Patterson, for his guidance and support throughout this project, as well as all of the other ROC project members at Berkeley and Stanford for their dedication, enthusiasm and willingness to provide feedback. Many thanks especially to Mike Howard for helping me deploy the test system and for putting up with the numerous unexpected node reboots that followed. Thanks also to the industrial visitors who have attended the semi-annual ROC research retreats. Finally, thanks to Professor Anthony Joseph, my second reader for providing many helpful comments and suggestions.

I owe a great debt of gratitude to the students and faculty of the Rutgers University Computer Science Department, particularly Rich Martin, Thu Nguyen, Kiran Nagaraja, Konstantinos Kleisouris, Bin Zhang and Xiaoyan Li. This study would not have been possible without their prior work on the Mendosus fault injection platform and their willingness to help others use their research software. I thank also professors Ricardo Bianchini and Enrique V. Carrera and the other members of the Rutgers Network Servers Project for making available the source code of PRESS.

Finally, I would like to thank my family and friends for all their help and support during my time at Berkeley.

This work was supported in part by NSF grant no. CCR-0085899, the NASA CICT (Computing, Information & Communication Technologies) Program, an NSF CAREER award, Allocity, Hewlett Packard, IBM, Microsoft, NEC, and Sun Microsystems

Bibliography

[1] Keynote Systems. `http://www.keynote.com`, 2003.

[2] T. F. Abdelzaher and N. Bhatti. Adaptive Content Delivery for Web Server QoS. In *International Workshop on Quality of Service*, June 1999.

[3] J. Basney and M. Livny. Improving Goodput by Coscheduling CPU and Network Capacity. 13(3):220–230, Fall 1999.

[4] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating User-Perceived Quality into Web Server Design. In *9th International World Wide Web Conference*, Amsterdam, Netherlands, May 2000.

[5] E. Brewer. Lessons from Giant-Scale Services. *IEEE Internet Computing*, July/August 2001.

[6] A. Brown and D. Patterson. Towards Availability Benchmarks: A Case Study of Software RAID Systems. In *USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[7] G. Candea, E. Kiciman, S. Zhang, P. Keyani, and A. Fox. JAGR: An Autonomous Self-Recovering Application Server. In *5th International Workshop on Active Middleware Services*, Seattle, WA, June 2003.

[8] E.V. Carrera and R. Bianchini. Efficiency vs. Portability in Cluster-Based Network Servers. In *8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, June 2001.

[9] J.B. Chen. SLA Promises. `http://www.appliant.com/docs/pdf/slapromises.pdf`, June 2002.

[10] J.B. Chen and M. Perkowitz. Using End-User Latency to Manage Internet Infrastructure. In *2nd Workshop on Industrial Experiences with Systems Software (WIESS)*, Boston, MA, December 2002.

[11] DBench Project Group. DBench. `http://www.dbench.org`.

[12] J. R. Douceur and W. J. Bolosky. Improving the Responsiveness of a Stripe-Scheduled Media Server. In D. Kandlur, K. Jeffay, and T. Roscoe, editors, *Multimedia Computing and Networking*, volume 3654, pages 192–203, 1999.

[13] Y. Endo, Z. Wang, J. B. Chen, and M. Seltzer. Using Latency to Evaluate Interactive System Performance. In *2nd Symposium on Operating Systems Design and Implementation (OSDI*, Seattle, WA, October 1996.

[14] D. Patterson et al. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques and Case Studies. Technical Report UCB||CSD-02-1175, UC Berkeley Computer Science, March 2002.

[15] A. Fox and D. Patterson. When does fast recovery trump high reliability? In *2nd Workshop on Evaluating and Architecting System Dependability (EASY)*, San Jose, CA, October 2002.

[16] IFIP WG 10.4 Dependability Benchmarking SIG. SIGDeB. `http://www.ece.cmu.edu/~koopman/ifip_wg_10_4_sigdeb/`.

[17] F. Abou Jawad and E. Johnsen. Performability Evaluation. `http://www.doc.ic.ac.uk/~nd/surprise_95/journal/vol4/eaj2/report.html`, June 1995.

[18] H. Kari. *Latent Sector Faults and Reliability of Disk Arrays*. PhD thesis, Helsinki University of Technology, Espoo, Finland, April 1997.

[19] K. Keeton and J. Wilkes. Automating Data Dependability. In *10th ACM SIGOPS European Workshop*, September 2002.

[20] C. Lamb, A. Merchant, and G. A. Alvarez. Facade: Virtual Storage Devices with Performance Guarantees. In *2nd USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, April 2003.

[21] J.C. Laprie, editor. *Dependability: Basic Concepts and Terminology*. Springer-Verlag, 1992.

[22] Richard Martin. Personal correspondence, February 2004.

[23] J.F. Meyer. Performability Evaluation: Where It Is and What Lies Ahead. In *International Computer Performance and Dependability Symposium (IPDS)*, Erlangen, Germany, April 1995.

[24] C. Mundie, P. de Vries, P. Haynes, and M. Corwine. Trustworthy Computing. *Microsoft White Paper*, October 2002.

[25] K. Nagaraja, R. Bianchini, R. Martin, and T. Nguyen. Using Fault Model Enforcement to Improve Availability. In *2nd Workshop on Evaluating and Architecting System Dependability (EASY)*, San Jose, CA, October 2002.

[26] K. Nagaraja, X. Li, B. Zhang, R. Bianchini, R. Martin, and T. Nguyen. Using Fault Injection to Evaluate the Performability of Cluster-Based Services. In *4th USENIX Symposium on Internet Technologies and Systems (USITS)*, Seattle, WA, March 2003.

[27] Nakob Nielsen, editor. *Usability Engineering*. Academic Press, 1993.

[28] D. Oppenheimer, A. Brown, J. Traupman, P. Broadwell, and D. Patterson. Practical Issues in Dependability Benchmarking. In *Second Workshop on Evaluating and Architecting System Dependability (EASY)*, October 2002.

[29] D. Oppenheimer and D. Patterson. Studying and Using Failure Data from Large-Scale Internet Services. In *ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[30] V. Paxson and S. Floyd. Wide-Area Traffic: The Failure of Poisson Modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1994.

[31] R. Rajamony and M. Elnozahy. Measuring Client-Perceived Response Times on the WWW. In *3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, San Francisco, CA, March 2001.

[32] IBM Research. Autonomic Computing. `http://research.ibm.com/autonomic`, 2003.

[33] A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley & Sons, 6th edition, 2004.

[34] Standard Performance Evaluation Corporation. SPECweb99 Benchmark. `http://www.specbench.org/osg/web99/`.

[35] W. Torres-Pomales. Software Fault Tolerance: A Tutorial. Technical Report TM-2000-210616, NASA, October 2000.

[36] Transaction Processing Performance Council. TPC Benchmark W (Web Commerce) Specification, Version 1.8. `http://www.tpc.org/tpcw/spec/tpcw_V1.8.pdf`, February 2002.

[37] Transaction Processing Performance Council. TPC Benchmark C Standard Specification, Revision 5.2. `http://www.tpc.org/tpcc/spec/tpcc_current.pdf`, December 2003.

[38] B. Zhang, K. Nagaraja, X. Li, R. Martin, and T. Nguyen. Mendosus: A SAN-Based Fault-Injection Test-Bed for the Construction of Highly Available Network Services. In *Novel Uses of System Area Networks (SAN-1)*, Cambridge, MA, February 2002.

[39] Zona Research, Inc. The Economic Impacts of Unacceptable Web-Site Download Speeds. *Zona White Paper*, April 1999.

[40] Zona Research, Inc. The Need for Speed II. *Zona Market Bulletin*, (5), April 2001.