An Architecture for Highly Concurrent, Well-Conditioned Internet Services

by

Matthew David Welsh

B.S. (Cornell University) 1996
M.S. (University of California, Berkeley) 1999

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

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Fall 2002

The dissertation of Matthew David Welsh is approved:

Chair	Date
	Date
	Date

University of California at Berkeley

Fall 2002

An Architecture for Highly Concurrent, Well-Conditioned Internet Services

Abstract

An Architecture for Highly Concurrent, Well-Conditioned Internet Services

by

Matthew David Welsh
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor David Culler, Chair

This dissertation presents an architecture for handling the massive concurrency and load conditioning demands of busy Internet services. Our thesis is that existing programming models and operating system structures do not adequately meet the needs of complex, dynamic Internet servers, which must support extreme concurrency (on the order of tens of thousands of client connections) and experience load spikes that are orders of magnitude greater than the average. We propose a new software framework, called the *staged event-driven architecture* (or SEDA), in which applications are constructed as a network of event-driven stages connected with explicit queues. In this model, each stage embodies a robust, reusable software component that performs a subset of request processing. By performing admission control on each event queue, the service can be well-conditioned to load, preventing resources from being overcommitted when demand exceeds service capacity. SEDA employs dynamic control to tune runtime parameters (such as the scheduling parameters of each stage) automatically, as well as to manage load, for example, by performing adaptive load shedding.

In this dissertation, we show that the SEDA design yields higher performance than traditional service designs, while exhibiting robustness to huge variations in load. We begin by evaluating existing approaches to service design, including thread-based and event-driven concurrency mechanisms, and demonstrate that these approaches fail to meet the concurrency and load conditioning demands of busy network services. Next, we present the SEDA design, and motivate its use through a set of design patterns that describe how to map an Internet service onto the SEDA structure. We also derive a performance model for SEDA-based Internet services based on queueing networks; this model is used to highlight the performance and load aspects of the architecture, as well as to drive design decisions affecting the decomposition of services into stages.

We present an implementation of an Internet services platform, called *Sandstorm*, based on the SEDA architecture. Sandstorm is constructed entirely in Java and makes use of nonblocking I/O primitives for sustaining high concurrency. We evaluate the use of SEDA through several applications, including a high-performance HTTP server, a packet router for the Gnutella peer-to-peer file sharing network, and a Web-based e-mail service using dynamic scripting and database access. Finally, we describe several control-based mechanisms for automatic tuning and load conditioning of busy Internet services in the SEDA design, including thread pool sizing, event batching, and queue-based admission control for overload management. Our results show that SEDA is an effective design approach for complex Internet services, yielding high performance and robust behavior under heavy load.

Professor David Culler
Dissertation Committee Chair

If you want to be free,
Get to know your real self.
It has no form, no appearance,
No root, no basis, no abode,
But is lively and buoyant.
It responds with versatile facility,
But its function cannot be located.
Therefore when you look for it,
You become further from it;
When you seek it,
You turn away from it all the more.

Rinzai Gigen Zenji (d. 886)

Contents

List of Figures

Acknowledgments

First and foremost, I would like to thank David Culler, my advisor and mentor, for lending his extensive experience and incredibly broad vision to this thesis work, as well as all of my research at Berkeley. David has an amazing way of cutting to the core of complex subjects and focusing on the important details. David is also unafraid to work on very hard problems, as well as to drastically change research directions—traits that I can only hope have rubbed off on me.

I owe a great deal of thanks to Eric Brewer, one of the principal investigators on the Ninja project, under which much of my research at Berkeley was conducted. Eric always seems to have something insightful and interesting to say about any given research problem, and I am grateful to have his critique on this thesis. I am also indebted to Marti Hearst for her valuable feedback on this dissertation, as well as for early comments on the SEDA project.

I have had the pleasure of working with a number of talented undergraduate students over the years, and several of them have contributed substantial code to the SEDA project. Eric Wagner implemented the PyTeC service construction language and ported Sandstorm's sockets interface to JDK 1.4. Dennis Chi did a heroic job implementing the asynchronous TLS/SSL library described in Chapter 6. Jerrold Smith ported the nonblocking I/O layer to Windows 2000.

Other thanks are due to Steve Czerwinski for providing the IMAP traces used to develop the client load generator in Chapter 6; Stan Schwarz at the USGS Pasadena Field office for providing the Web server logs in Chapter 2; and Mor Harchol-Balter at CMU for her course notes and thorough review of Chapter 4. Many of the experiments in this dissertation were carried out on the UC Berkeley Millennium cluster, and would not have been possible without the support of Eric Fraser, Matt Massie, and Albert Goto.

Many others at Berkeley and elsewhere provided much advice and feedback on my work over the years. In particular, Joe Hellerstein offered a good deal of critical analysis of my research, and the occasional (much-needed) grilling. Steve Gribble was deeply involved in many aspects of this research, providing countless hours of discussion, debate, and counsel during my first few years at Berkeley.

I owe a great deal of inspiration to Thorsten von Eicken and Dan Huttenlocher, who found numerous ways to keep me busy while I was an undergrad at Cornell. I will never forget being boxed in by six workstations, running my first ATM network benchmarks, and wiring an oscilloscope to the PCI bus to get measurements for a paper. My fellow grad students, including Jason Hill, Phil Buonadonna, Fredrick Wong, Rich Martin, Brent Chun, Kamin Whitehouse, and Phil Levis, provided a constant source of entertainment and encouragement that kept me going through the many ups and downs of the graduate student experience.

Amy Bauer, my best friend, partner in crime, and fiancée, helped me in more ways than I can possibly recount here, and I am eternally grateful for her love and support. Last but not least, my parents are the ones to thank for getting me here in the first place—it all started with that VIC-20 we bought when I was nine.

This research was supported by several grants, fellowships, and equipment donations from the Defense Advanced Research Projects Agency, the National Science Foundation, Intel Corporation, Nortel Networks, and Royal Philips Electronics.

Chapter 1

Introduction and Motivation

This dissertation presents an architecture for handling the massive concurrency and load conditioning demands of busy Internet services. Our thesis is that existing programming models and operating system structures do not adequately meet the needs of complex, dynamic Internet servers, which must support extreme concurrency (on the order of tens of thousands of client connections) and experience load spikes that are orders of magnitude greater than the average. We propose a new software framework, called the *staged event-driven architecture* (or SEDA), in which applications are constructed as a network of event-driven stages connected with explicit queues [147]. In this model, each stage embodies a robust, reusable software component that performs a subset of request processing. By performing admission control on each event queue, the service can be well-conditioned to load, preventing resources from being overcommitted when demand exceeds service capacity. SEDA employs dynamic control to tune runtime parameters (such as the scheduling parameters of each stage) automatically, as well as to manage load, for example, by performing adaptive load shedding. In this dissertation, we show that the SEDA design yields higher performance than traditional service designs, while exhibiting robustness to huge variations in load.

1 Introduction: The rise of Internet services

The explosive growth of the Internet in the last few years has given rise to a vast range of new services being deployed on a global scale. No longer dominated by Web sites with static content, Internet services span a diverse range of categories including stock trading, live media broadcasts, online shopping, instant messaging, peer-to-peer file sharing, and application hosting. In contrast to static content sites, this new class of dynamic services requires significant computational and I/O resources to process each request. An increasingly complex array of systems are involved in delivering these services, including Web servers, caches, middle-tier application servers, databases, and legacy systems. At the same time, exponential growth of the Internet population is placing unprecedented demands upon the scalability and robustness of these services. Yahoo! receives over 1.2 billion page views daily [150], and AOL's Web caches service over 10 billion hits a day [6].

Internet services have become critical both for driving large businesses as well as for personal productivity. Global enterprises are increasingly dependent upon Internet-based applications for e-commerce, supply chain management, human resources, and financial accounting. Many individuals consider e-mail and Web access to be indispensable lifelines. This growing dependence upon Internet services underscores the importance of their availability, scalability, and ability to handle large loads. Such popular sites as EBay [94], Excite@Home [55], and E*Trade [18] have had embarrassing outages during periods of high load. An extensive outage at E*Trade resulted in a class-action lawsuit against the online stock brokerage by angry customers [130]. Likewise, during a week-long outage of MSN Messenger [148], many users expressed a great amount of desperation and hostility from being disconnected from the instant-messaging service. As more people begin to rely upon the Internet for managing financial accounts, paying bills, and potentially even voting in elections, it is increasingly important that these services perform well and are robust to changes in load.

This challenge is exacerbated by the burstiness of load experienced by Internet services. Popular services are subjected to huge variations in demand, with bursts coinciding with the times that the service has the most value. The well-documented "Slashdot Effect"[1] shows that it is not uncommon to experience more than 100-fold increases in demand when a site becomes popular [142]. The events of September 11, 2001 provided a poignant reminder of the inability of Internet services to scale: virtually every Internet news site was completely unavailable for several hours due to unprecedented demand following the terrorist attacks on New York and Washington. CNN.com experienced a two-and-a-half hour outage with load exceeding 20 times the expected *peak* [83]. Although the site team managed to grow the server farm by a factor of 5 by borrowing machines from other sites, even this was not sufficient to deliver adequate service during the load spike. CNN.com came back online only after replacing the front page with a text-only summary in order to reduce load [22].

Apart from these so-called flash crowds, sites are also subject to denial-of-service attacks that can knock a service out of commission. Such attacks are increasingly sophisticated, often being launched simultaneously from thousands of sites across the Internet. Denial-of-service attacks have had a major impact on the performance of sites such as Buy.com, Yahoo!, and whitehouse.gov [86], and several companies have been formed to help combat this problem [85].

The number of concurrent sessions and hits per day to Internet sites translates into a large number of I/O and network requests, placing enormous demands on underlying resources. Unfortunately, traditional operating system designs and common models of concurrency do not provide graceful management of load. Commodity operating systems focus on providing maximal transparency by giving each process the abstraction of a virtual machine with its own CPU, memory, disk, and network. Processes and threads are traditionally used for concurrency, but these mechanisms entail high overheads in terms of memory footprint and context switch time. Although simplifying the programming model, transparent resource virtualization prevents applications from making informed decisions, which are vital to managing excessive load.

[1]This term is often used to describe what happens when a site is hit by sudden, heavy load. This term refers to the technology news site `slashdot.org`, which is itself hugely popular and often brings down other less-resourceful sites when linking to them from its main page.

2 Thesis summary

This dissertation proposes a new design framework for highly concurrent server applications, called the *staged event-driven architecture* (SEDA).[2] SEDA combines the use of threads and event-based programming models to manage the concurrency, I/O, scheduling, and resource management needs of Internet services. In SEDA, applications are constructed as a network of *stages*, each with an associated *incoming event queue*. Each stage represents a robust building block that may be individually conditioned to load by thresholding or filtering its event queue. In addition, making event queues explicit allows applications to make informed scheduling and resource-management decisions, such as reordering, filtering, or aggregation of requests.

An important aspect of SEDA is that it relies on *feedback-driven control* for managing resources and adapting to overload conditions. This approach avoids the use of static resource limits and “knobs”, subject to error-prone manual configuration, that can have a serious impact on performance. Rather, SEDA-based services are instrumented to observe their own performance, using feedback to adjust resource allocations and perform admission control when the service is overloaded. In this way, dynamic control keeps the service within its ideal operating regime despite fluctuations in load.

This dissertation describes the design, architecture, and implementation of a SEDA-based Internet services platform. This platform provides efficient, scalable I/O interfaces as well as several resource-control mechanisms, including thread pool sizing and adaptive load shedding. We present a queue-theoretic performance model for SEDA-based systems that formally describes the effects of application structure, queue management, and thread allocations. This model serves as a guideline for SEDA service developers to understand the performance of a given application design.

Our prototype of SEDA, called Sandstorm, is implemented in Java and exhibits good performance and scalability, even surpassing two popular Web servers implemented in C. We also develop a family of overload control mechanisms based on adaptive admission control and service

[2] *Seda* is also the Spanish word for *silk*.

degradation. These mechanisms allow the service to meet administrator-specified performance targets despite enormous load spikes.

We evaluate the SEDA framework through several applications, including a high-performance HTTP server, a packet router for the Gnutella peer-to-peer file sharing network, and a Web-based e-mail service making use of dynamic scripting and database access. We present performance and scalability results demonstrating that SEDA achieves robustness over huge variations in load and outperforms other service designs.

We argue that using SEDA, highly concurrent applications are easier to build, more efficient, and more robust to load. With the right set of interfaces, application designers can focus on application-specific logic, rather than the details of concurrency and resource management. The SEDA design is based on the concept of exposing overload to an Internet service, allowing it to adapt and participate in load-management decisions, rather than taking the more common view that overload protection is only the responsibility of the underlying operating system. At the same time, the SEDA design attempts to shield application designers from many complex aspects of system architecture, such as thread management and scheduling. Our goal is to strike the right balance between exposing resource control and burdening the application designer with unneeded complexity.

3 Background: Internet service properties

The Internet presents a systems problem of unprecedented scale: that of supporting millions of users demanding access to services that must be responsive, robust, and always available. This work is motivated by three fundamental properties of Internet services: massive concurrency demands, an increasing trend towards complex, dynamic content, and a need to be extremely robust to load. In this section we detail each of these properties in turn.

3.1 High concurrency

The growth in popularity and functionality of Internet services has been astounding. While the Web itself is growing in size, with recent estimates anywhere between 1 billion [64] and 2.5 billion [114] unique documents, the number of users on the Web is also growing at a staggering rate. In April 2002, Nielsen//NetRatings estimates that there are over 422 million Internet users worldwide [104], and a study in October 2000 [48] found that there are over 127 million adult Internet users in the United States alone.

As a result, Internet applications must support unprecedented concurrency demands, and these demands will only increase over time. As of December 2001, Yahoo! serves 1.32 billion pages a day, and delivers over 19.1 billion messages through its e-mail and instant messenger services monthly [150]. Internet traffic during the 2000 U.S. presidential election was at an all-time high, with ABC News reporting over 27.1 million pageviews in one day, almost 3 times the peak load that the site had ever received. Many news and information sites were reporting a load increase anywhere from 130% to 500% over their average [92].

3.2 Dynamic content

The early days of the Web were dominated by the delivery of static content, mainly in the form of HTML pages and images. More recently, dynamic, on-the-fly content generation, which requires significant amounts of computation and I/O to generate, has become more widespread. A typical dynamic Internet service involves a range of systems, including Web servers, middle-tier application servers, and back-end databases, to process each request. The processing for each request might include encryption and decryption (e.g., if the SSL protocol is being used); server-side scripting using frameworks such as Java 2 Enterprise Edition (J2EE) [123] or PHP [134]; database access; or access to legacy systems such as mainframes (still commonly used for airline ticketing). In contrast to static Web pages, dynamic services require greater resources, and the resource demands for a given service are difficult to predict. Moreover, the content generated by a dynamic Internet service is often not amenable to caching, so each request demands a large amount of server resources.

The canonical example of a highly dynamic Internet service is a large “mega-site” such as Yahoo! [149], which provides a wide range of services, including a search engine, real-time chat, stock quotes, driving directions, and access to online retailers. In addition to consumer-oriented sites, specialized business-to-business applications, ranging from payroll and accounting to site hosting, are becoming prevalent. The trend towards dynamic content is also reflected in industry standard benchmarks for measuring Web server performance, such as SPECweb99 [119], which includes a significant dynamic content-generation component as part of the standard workload.

Another aspect of the increased demand on Internet services is that the size of Web content is growing, with streaming media, MP3 audio files, videos, and large software downloads becoming increasingly common. In October 2000, not long before the Napster file-trading service was shut down, CNET [78] reported 1.3 billion MP3 files were exchanged over the service in one month, with over 640,000 users downloading songs at any given time. Apart from placing new demands on networks, this growth in content requires that Internet services be responsible for dedicating large numbers of resources for storing and serving vast amounts of data.

3.3 Robustness to load

Demand for Internet services can be extremely bursty, with the peak load being orders of magnitude greater than the average load. Given the vast user population on the Internet, virtually any site can be suddenly inundated with a surge of requests that far exceed its ability to deliver service. The media has reported on numerous examples of popular Internet sites being brought down by excessive load. We have already discussed the effects of the September 11 attacks on news sites. In September 1998, the U.S. Government released Ken Starr’s report on President Clinton on the Internet, causing seven government sites to be swamped with requests from over 20 million users [34]. During the February 1999 Internet broadcast of a Victoria’s Secret Fashion show, over

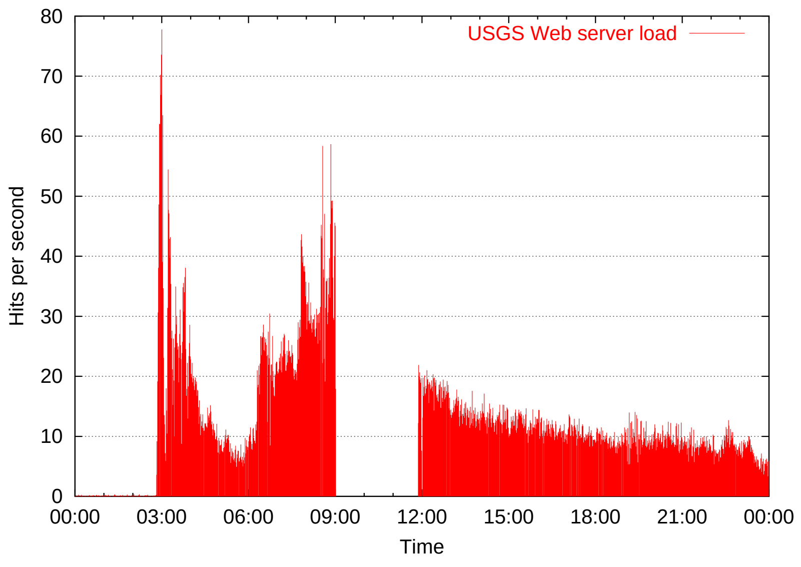


Figure 1: **The effect of sudden load on a Web server:** *This is a graph of the Web server logs from the USGS Pasadena Field Office Web site after an earthquake registering 7.1 on the Richter scale hit Southern California on October 16, 1999. The load on the site increased almost 3 orders of magnitude over a period of just 10 minutes. Before the earthquake, the site was receiving about 5 hits per minute on average. The gap between 9am and 12pm is a result of the server's log disk filling up. The initial burst at 3am occurred just after the earthquake; the second burst at 9am when people in the area began to wake up the next morning. (Web server log data courtesy of Stan Schwarz, USGS.)*

1.5 million users logged in simultaneously to view the broadcast, causing the site to melt down under the overload. According to Keynote systems, a Web site monitoring firm, only 2 percent of requests to the site succeeded during the live show [19].

As a more concrete example, Figure 1 shows the load on the U.S. Geological Survey Pasadena Field Office Web site after a large earthquake hit Southern California in October 1999. The load on the site increased almost 3 orders of magnitude over a period of just 10 minutes, causing the Web server's network link to saturate and its disk log to fill up [142]. Note that this figure shows only the number of requests that were successfully logged by the server; due to the overload, it is likely that an even larger number of requests were present but not recorded. During the load spike, the system administrator who was responsible for the Web site was unable to login to the system remotely to clear up the disk log, and had to physically reboot the machine in order to regain control.

The most common approach to dealing with heavy load is to overprovision resources. In the case of a Web site, the administrators simply buy enough Web server machines to handle the peak load that the site could experience, and load balance across them. However, overprovisioning is infeasible when the ratio of peak to average load is very high; it is not practical to purchase 100 or 1000 times the number of machines needed to support the average load case. This approach also neglects the cost issues which arise when scaling a site to a large "farm" of machines; the cost of managing a large cluster of machines is no doubt much higher than the multiplicative cost of the machines themselves.

Given that we cannot expect most Internet services to scale to peak demand, it is critical that services are designed to be *well-conditioned to load*. That is, when the demand on a service exceeds its capacity, a service should not overcommit its resources and degrade in a way that all clients suffer. Rather, the service should be aware of overload conditions and attempt to adapt to them, either by degrading the quality of service delivered to clients, or by (predictably) shedding load, such as by giving users some indication that the service is saturated. It is far better for an overloaded service to inform users of the overload than to silently drop requests.

Nevertheless, replication is a key aspect of service scalability, and is commonly employed using both scalable clusters to obtain replication within a service site [45], as well as in the wide area, as with content-distribution networks [5, 39, 49]. Despite such scaling, we expect the individual nodes within a scalable system to experience large load spikes. Our goal in this thesis is to complement the use of replication by addressing the problem of load management within an individual node. Therefore we focus on developing a general framework for authoring highly concurrent and well-conditioned service instances that can potentially be deployed in a replicated system.

4 Trends in the Internet service design space

The systems challenge associated with robust Internet service design is magnified by two trends that increase the generality of services: rapid evolution of service logic and the drive for general-purpose platforms for hosting services.

4.1 Rapid evolution of service logic

Internet services experience a high rate of change and innovation, driven by the constantly evolving Internet marketplace. Popular sites are constantly adding new functionality to stay competitive, a trend which is supported by the increasing ease of Internet service development using simplified Web scripting languages such as PHP [134] and Java servlets [127]. It is excessively time-consuming to carefully engineer and tune each new service component to deal with real-world loads, and as a result, deployment of new service functionality often takes precedence over robustness.

4.2 General-purpose hosting platforms

Internet services are increasingly hosted on general-purpose facilities, rather than on platforms that are specifically engineered for a particular service. A number of companies, such as Exodus, EDS, and ProTier, are now providing managed Internet hosting services, which are based in replicated Internet data centers with carefully managed hardware and software platforms. Accordingly, the Internet services market is dominated by generic frameworks, often based upon scalable workstation clusters running commodity operating systems, using a standard software platform such as Java 2 Enterprise Edition (J2EE) [123]. In this environment, service authors have little control over the underlying medium upon which the service will be deployed and run.

As these trends continue, we envision that a rich array of novel services will be authored and pushed into the infrastructure where they may become successful enough to scale to millions of users. Several investigations are addressing the high-level aspects of service authorship, including naming, lookup, composition, and versioning [36, 53, 59, 122, 128]. Our focus is on the performance and robustness aspect of the problem: achieving well-conditioned performance on a wide range of services subject to huge variations in load, while preserving ease of authorship. We argue that the right way to approach this problem is through a generic service platform that manages load in a manner that is cleanly separated from the service-specific logic, while giving services an indication of load and the ability to participate in load-management decisions.

5 Thesis summary and contributions

Much work has focused on performance and robustness for specific services [4, 61, 106, 151]. However, with services becoming increasingly dynamic and flexible, the engineering burden required to make services robust to heavy demands becomes excessive. Few tools exist that aid the development of highly concurrent, well-conditioned services; our goal is to reduce this complexity by providing general-purpose mechanisms that aid software developers in obtaining these properties.

An additional hurdle to the construction of Internet services is that there is little in the way of a systematic approach to building these applications, and reasoning about their performance or behavior under load. Designing Internet services generally involves a great deal of trial-and-error on top of imperfect OS and language interfaces. As a result, applications can be highly fragile—any change to the application code or the underlying system can result in performance problems, or worse, total meltdown.

The goal of this thesis is to design a generic software architecture that can:

- Handle the massive concurrency demands of Internet services;
- Deal gracefully with large variations in load;
- Generalize to a broad class of Internet services; and,
- Simplify the construction of services by decoupling load management from service logic.

In this dissertation, we present the design an implementation of the *staged event-driven architecture* (or SEDA), a software platform designed to meet these goals. We present a detailed application and performance study, demonstrating that SEDA yields higher performance than traditional service designs, allows services to be well-conditioned to overload, and simplifies service design.

This dissertation makes the following contributions:

The Staged Event-Driven Architecture: The focus of this work is on the SEDA architecture, which decomposes a complex Internet service into a network of event-driven *stages* connected with explicit *queues*. We present a detailed look at related approaches to concurrency and load management, proposing the SEDA approach as a general-purpose model for Internet service design. Also, we present a set of *design patterns* that describe how to map a given Internet service onto the SEDA design to achieve high performance, robustness to load, and code modularity. To round out our discussion of SEDA, we present a queue-theoretic performance model that describes the behavior of SEDA-based services in terms of request load, service time distribution, stage graph structure, and thread allocations. This model can be used by a system designer to understand the various factors that affect SEDA performance.

The *Sandstorm* Internet service platform: We describe a Java-based implementation of SEDA, called *Sandstorm*, that provides a rich set of programming interfaces for developing SEDA-based services. Sandstorm is intended to reduce the complexity of service development by hiding the details of thread allocation, scheduling, and resource control from application code. Sandstorm provides built-in interfaces for stage graph construction, asynchronous network and file I/O, and per-stage resource control.

Feedback-driven overload control: We investigate a family of overload prevention techniques based on per-stage admission control. These mechanisms automatically shed load from an overloaded service by monitoring stage performance and adapting admission control parameters accordingly. We present techniques for meeting a 90th-percentile response time target, class-based service differentiation, and application-specific service degradation.

Detailed application evaluation: Finally, we present a detailed evaluation of several significant applications built using the Sandstorm runtime. These include *Haboob*, a high-performance Web server; a packet router for the Gnutella peer-to-peer file sharing network; and *Arashi*, a Web-based e-mail service. Our results show that SEDA-based services exhibit high performance and are well-behaved under extreme variations in load. The Haboob Web server outperforms the popular Apache and Flash Web servers, which are implemented in C, and exhibits a great deal of fairness to clients. The Gnutella packet router is capable of driving a load of over 20,000 packets a second, and is used to demonstrate the use of automatic resource tuning to overcome an artificial bottleneck in the service. The Arashi e-mail service is an example of a complex service making use of dynamic page generation and database access. Arashi is used to evaluate the feedback-driven overload control mechanisms in a number of scenarios, including graceful degradation under a massive load spike.

6 Dissertation roadmap

The rest of this dissertation is organized as follows.

In Chapter 2, we present the motivation and background for this work, discuss previous work in Internet service construction, and describe the shortcomings of existing approaches to concurrency and load management for Internet services. Chapter 3 describes the SEDA architecture in detail, and presents a set of design patterns that one can use to map a service from the traditional single-task model into the SEDA design. Chapter 4 presents a performance model for SEDA based on results from queueing theory, which is useful for motivating an intuition for the performance of SEDA-based applications.

In Chapter 5, we present the design and implementation of Sandstorm, our SEDA-based Internet services platform. Sandstorm has been used to build a number of complex Internet services, which are described in Chapter 6. This chapter also presents a detailed performance study of several SEDA applications, demonstrating scalability and robustness to load. In Chapter 7 we present a feedback-driven approach to overload management through the use of adaptive admission control and service degradation. This chapter explores a range of load conditioning mechanisms that can be employed in SEDA, and discusses the tradeoff between load shedding and application-specific adaptivity to overload.

Finally, in Chapters 8 and 9, we present reflections on the SEDA design, outline several areas for future work, and conclude.

Chapter 2

Background and Related Work

This chapter develops the lineage of the SEDA approach by outlining the previous steps towards solving the performance, scalability, and robustness problems faced by Internet services. The key requirements and challenges for Internet service design have been discussed in the previous chapter. In summary, the primary goals of an Internet service are to support massive concurrency (in the form of many simultaneous requests or user connections); an extremely dynamic environment in terms of the request distribution as well as the resources required to process each request; and predictable, well-conditioned performance despite large variations in offered load.

Related work towards supporting these goals can be broadly categorized as follows. We begin by discussing the two most common programming models for expressing concurrency: thread-based (or process-based) concurrency, which is generally used for ease of programming, and event-driven concurrency, which is used for scalability and performance. SEDA draws together these two approaches and proposes a hybrid model that exhibits features of both. Next, we discuss resource virtualization and identify several specific problems raised by existing operating system designs with respect to resource management. Finally, we describe existing approaches to load management in busy Internet services and argue that dynamic control is the right way to manage resources in this regime.

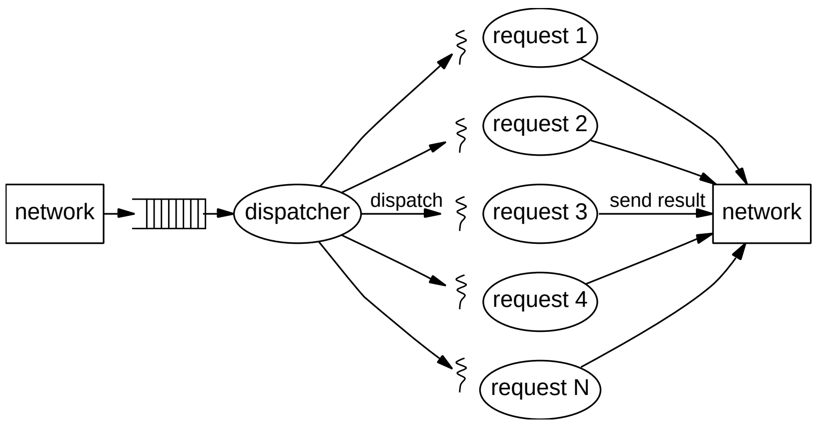


Figure 2: **Threaded server design:** *Each incoming request is dispatched to a separate thread, which performs the entire processing for the request and returns a result to the client. Edges represent control flow between components. Note that other I/O operations, such as disk access, are not shown here, but are incorporated within each threads' request processing.*

1 Concurrency programming models

A critical aspect of service design is the means by which concurrency is represented, as services must support many simultaneous requests from a large and ever-changing population of users. Internet services push the envelope in terms of the degree of concurrency required; a typical site may experience tens of thousands (or more) of simultaneous user sessions. The highly dynamic nature of incoming requests also raises challenges for resource provisioning, as the resource needs of individual requests are difficult to predict. In this section we discuss the two most commonly employed concurrent programming models: thread-based concurrency and event-driven concurrency.

1.1 Thread-based concurrency

The most commonly used design for server applications is the thread- or process-per-request model, as embodied in systems such as RPC packages [129], Java Remote Method Invocation [125], and DCOM [95]. This model is well-supported by modern languages and programming environments, and threading primitives are provided by virtually every commodity operating system.

A *thread* is a context for computation, typically consisting of a program counter, set of registers, private stack, address space, and operating system or language state associated with the thread, such as scheduling and accounting data structures. Multiple threads can share a single address space, thereby communicating directly through shared data structures. A *process* is simply a thread with its own private address space.

Throughout this thesis, we make no distinction between a thread and a process; we consider the two to be identical for the purposes of concurrency and load management, although threads are more flexible in that they can share an address space. It is often the case that threads are more scalable than processes, though this performance gap seems to be narrowing. Whereas in earlier operating systems, there was a strong distinction made between (relatively) light-weight threads and heavy-weight processes, in modern operating systems this distinction is evaporating due to the widespread incorporation of threading mechanisms into the kernel. For example, in the Linux operating system a thread and a process are virtually identical save for the address space ownership.

In the thread-per-request model, shown in Figure 2, each accepted request is dispatched to a thread which processes it. Synchronization operations, such as locks and semaphores, are used to protect shared resources and data structures that may be accessed during request processing. The operating system allows system resources to be time-shared across multiple requests, and overlaps computation and I/O, by transparently context switching between threads.

The thread-per-task model is relatively easy to program, allowing the application developer to implement request processing as a “straight-line” code path, which executes (virtually) in isolation from all other requests in the system. The existence of other requests is only evident when dealing with shared resources or state.

Threads raise a number of challenges for Internet service design in terms of resource management and scalability. The most serious issue with threads is that they do not provide adequate control over resource usage; we will discuss this problem in Section 1.4. First, we focus on the problem that many thread implementations exhibit serious scaling issues, making them impractical

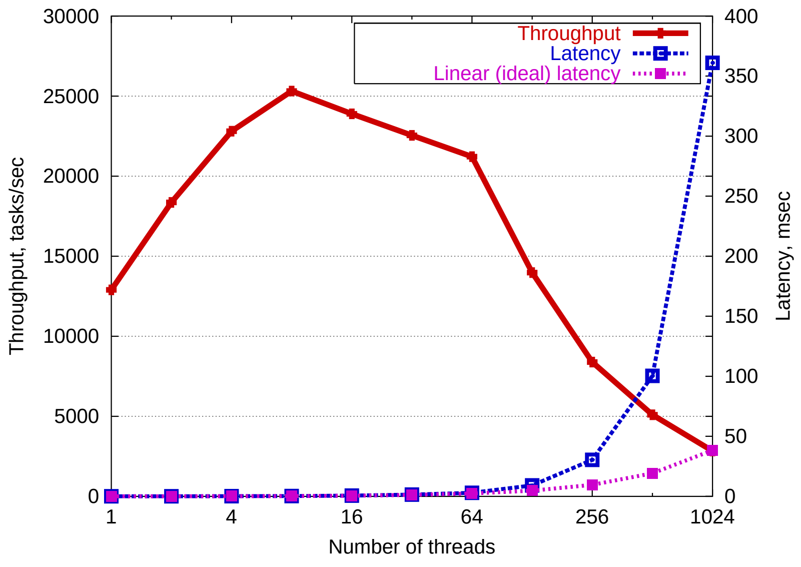


Figure 3: **Threaded server throughput degradation:** *This benchmark measures a simple threaded server that dispatches a separate thread for each concurrent request in the system. After receiving a request, each thread performs an 8 KB read from a disk file; all threads read from the same file, so the data is always in the buffer cache. Threads are pre-allocated in the server to eliminate thread startup overhead from the measurements, and requests are generated internally to negate network effects. The server is implemented in C and is running on a 4-way 500 MHz Pentium III with 2 GB of memory under Linux 2.2.14. As the number of concurrent requests increases, throughput initially increases until about 8 threads are in use. Adding additional threads causes throughput to degrade substantially. Response time becomes unbounded as request queue lengths increase; for comparison, we have shown the ideal linear response time curve (note the log scale on the horizontal axis).*

for large-scale Internet services.

It is important to stress that threads are primarily designed to support timesharing. Thread-based concurrency entails high overheads as the number of threads grows to be large, which can lead to disastrous results for performance. As the number of threads grows, the system experiences increases in scheduling overheads, memory pressure due to thread footprints, cache and TLB misses, and contention for shared resources, such as locks. As a concrete example, Figure 3 shows the performance of a simple threaded server as the number of threads increases. Although the number of threads that the server can support before experiencing performance meltdown would be large for general-purpose timesharing, it is not adequate for the tremendous concurrency requirements of an Internet service.

A considerable amount of prior work has attempted to reduce the overheads associated with thread-based concurrency. The majority of this work approaches the problem from the point of view of lightweight threading for shared-memory multiprocessors, in which a separate thread is spawned for any operation (such as a function call) that can be logically performed in parallel. Although this approach can reduce the overhead of a thread system designed specifically for compute-bound applications, it typically does not benefit threads that perform arbitrary system calls or I/O, or that hold locks during blocking operations. For example, Lazy Task Creation [99], Lazy Threads [50], and Stack Threads [131] optimistically perform thread creation as a sequential function call within the context of the current thread, lazily allocating a new thread context only when the spawned operation blocks. In an Internet service, however, threads are often long-running and may perform numerous blocking operations during their lifetime. Lightweight threading techniques designed for parallel computation do not address the more general class of multithreading used by Internet services.

1.2 Bounded thread pools

To avoid performance degradation due to the overuse of threads, a number of systems adopt a coarse form of resource management that serves to bound the number of threads associated with a service, commonly referred to as *thread pooling*. In this approach, requests are dispatched to a bounded pool of threads. When all threads in the pool are busy processing requests, new requests are queued for service. Request queueing can be accomplished in several ways, and the form of queueing can have a serious impact on client-perceived performance.

A number of Web servers use a thread-per-connection model in which each thread processes a single client TCP connection; this is the model used by Apache [8], Microsoft Internet Information Server [96], and Netscape Enterprise Server [103]. Web application servers such as BEA Weblogic [15] and IBM WebSphere [62] also use the thread-per-connection approach. In this model, each thread processes a single client TCP connection to the service. When all threads are busy, additional connections are not accepted by the server. Under heavy load, this can cause the operating system queues for pending connections to fill, in turn causing the client's connection request (SYN packet) to be dropped by the server. Because TCP uses an exponentially increasing timeout for retransmitting connection attempts, this can lead to very long delays on the client side: several minutes may pass before a retried connection is established or aborted. This effect also leads to great deal of *unfairness* to clients, as clients that happen to have their connections accepted rapidly will continue to receive good service, while others will wait for long periods of time to establish a connection. Given that many modern Web browsers open multiple simultaneous connections to a service, it is often the case that every connection must receive good service for the user to experience acceptable response times. We study this effect in detail in Chapter 6.

A better model for thread pooling is to queue new requests within the service, where they can be served according to a policy dictated by the service itself, rather than by the underlying network. Queueing requests within the service also facilitates the early classification or prioritization of requests for overload management, as discussed in Section 3. This approach has been adopted by several Web servers, such as Web2k [17].

If used correctly, bounded thread pools are an effective means to represent concurrency in an Internet service. However, the thread pool approach as described here presents several significant challenges. The first problem is choosing the right thread pool size. Most systems use a static (administrator-specified) limit on the maximum number of threads, though this depends greatly on the resource usage of each request and the workload experienced by the service. Setting the thread limit too low underutilizes resources, while setting the limit too high can cause severe degradation under heavy load.

Another issue with thread pooling is that it is possible for all threads in the pool to become consumed with long-running requests, thereby starving other requests that may have higher priority or fewer resource requirements. Depending on the characteristics of the workload and the implementation of the service, the actual number of threads required may therefore be very large. Consider a case where a fixed-size thread pool is used to process requests, and additional requests are processed in FIFO order from a queue. If all threads are busy processing long-running requests (say, blocked on I/O or access to a bottleneck resource such as a database), then no additional requests—regardless of priority or resource requirements—can be processed. The issue here is that the concurrency model provides no mechanism to allow a busy thread to be preempted for a shorter or higher-priority request.

1.3 Resource management challenges of threading

Even if the performance overhead of threading could be eliminated, we argue that threads are fundamentally an inappropriate mechanism for concurrency and resource management in Internet services. In the thread-per-request model, important information about the application's resource usage and request stream are hidden within the thread scheduler; it is generally not possible for a service to inspect or control the operation of the thread system to perform load conditioning. Likewise, the thread scheduler has very limited information about the application's behavior and resource needs.

For example, consider a service that uses a thread-per-task concurrency model with a (hypothetical) scalable threading system. If a large number of requests block on a shared resource, such as a disk, the corresponding threads queue up within the thread scheduler; there is no way for the application to be aware of or effect the bottleneck. In this case, it might be desirable to shed load by performing admission control on the requests that require access to the bottleneck resource. However, the server is unable to inspect the internal request stream to implement such a policy. Moreover, once a request has been dispatched to a thread, the only load-management operations that can be performed are to kill the thread (aborting its processing regardless of progress) or to change its scheduling priority (in the extreme case, putting it to sleep for some time). This interface makes it difficult to implement effective load-management policies, such as offloading a resource bottleneck or rejecting requests that fail to meet a performance target.

One way to effect greater control over the operation of threads would be to instrument the application code with explicit control points. For example, before accessing a potential resource bottleneck, each thread would invoke a function that guards access to the resource, allowing an alternate action to be performed if the resource is saturated. Another approach is to give applications greater control over the internals of the threading system by allowing applications to specify their own scheduling algorithm, either by implementing thread dispatch at user level (in the case of scheduler activations [7]) or by pushing application-specific thread management code into the kernel (in the case of Strands [118]). However, these mechanisms are intended to allow applications to specify scheduling algorithms that have infrequently changing policies, not for implementing complex load conditioning policies based on detailed runtime information.

In many ways, these various techniques represent a departure from the simplicity of the thread-per-request model. We claim that it is better to consider an approach to concurrency that is more suited to Internet service design than to lash complex policies onto existing approaches.

1.4 Event-driven concurrency

The scalability limits of threads have led many developers to eschew them almost entirely and employ an event-driven approach to managing concurrency. In this approach, shown in Figure 4, a server consists of a small number of threads (typically one per CPU) that loop continuously, processing events of different types from a queue. Events may be generated by the operating system or internally by the application, and represent triggers that some form of processing needs to occur: network and disk I/O readiness and completion notifications, timers, or other application-specific events.

In the event-driven approach, each request in the system is represented as a *finite state machine* (FSM) or *continuation*. Each state of the FSM represents a set of processing steps to be performed on the request. For example, the FSM for a static Web page request might contain the

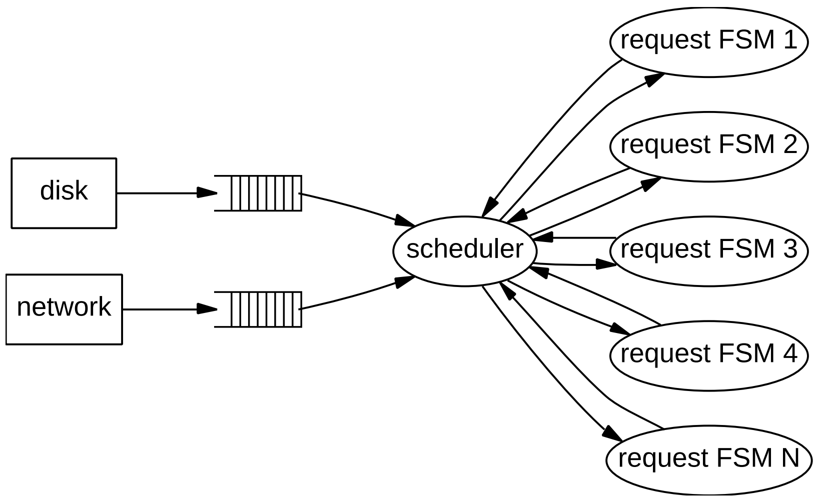


Figure 4: **Event-driven server design:** *This figure shows the flow of events through an event-driven server. The main thread processes incoming events from the network, disk, and other sources, and uses these to drive the execution of many finite state machines. Each FSM represents a single request or flow of execution through the system. The key source of complexity in this design is the event scheduler, which must control the execution of each FSM.*

following states, as shown in Figure 5:

1. Read request from network;
2. Parse request (e.g., process HTTP headers);
3. Look up disk file corresponding to request;
4. Read file data;
5. Format reply packet;
6. Write reply to network.

When an event arrives corresponding to some request, it is dispatched to one of the (small number) of threads, which processes the request according to its current state. The processing for each state is generally very short and runs to completion; when processing is complete, the FSM may transition to a new state, and the thread is dispatched to the next request FSM. A request may remain in the

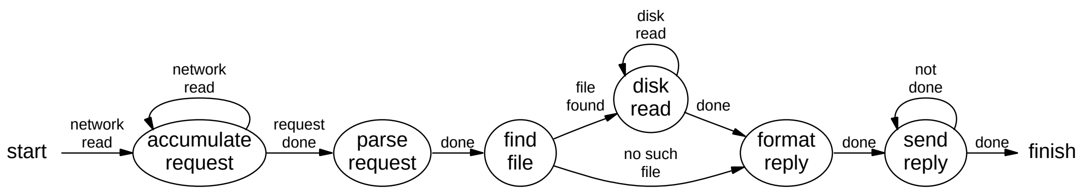


Figure 5: **Finite state machine for a simple HTTP server request:** *This figure depicts a static HTTP server request as a finite state machine (FSM) as used in an event-driven system. Each state represents some aspect of request processing, and edges represent transitions between states, triggered by incoming events or the completion of the processing for a given state.*

same state across multiple event arrivals. For example, an HTTP request described by the FSM given above may remain in the "read request from network" state until all network packets corresponding to the request have been received.

An event-driven system multiplexes the operation of many request FSMs over a small number of threads by "context switching" across requests after each iteration of the request-processing loop. This requires that the processing required for each state be short and not block, for example, when performing I/O. For this reason, event-driven systems make use of *nonblocking I/O interfaces* that operate in a split-phase fashion: the application performs a I/O operation (for example, reading data from a socket) that returns immediately whether the I/O request has completed or not. If the operation completes at a later time, the operating system generates an I/O completion event that the FSM scheduler can use to continue request processing. The dispatching of events to FSMs is therefore very similar to thread scheduling, and an event-driven system effectively implements its own application-level scheduling policy. Event-driven concurrency is typically more scalable than thread-driven concurrency, as the FSM state representing each request is much smaller than a thread context, and many of the overheads involving large numbers of threads are avoided.

If designed correctly, event-driven systems can support very high levels of concurrency, and experience little performance degradation as the number of simultaneous requests increases. Figure 6 shows the throughput achieved with an event-driven implementation of the service from Figure 3. The efficiency gain of the event-driven approach is mainly due to the reduced overhead

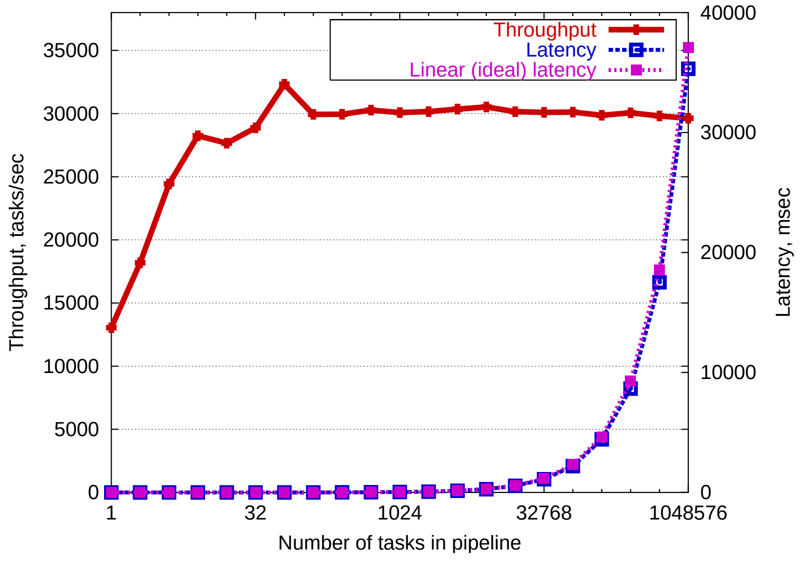


Figure 6: **Event-driven server throughput:** *This benchmark measures an event-driven version of the server from Figure 3. In this case, the server uses a single thread to process tasks, where each task reads 8 KB from a single disk file. Although the filesystem interface provided by the operating system used here (Linux 2.2.14) is blocking, because the disk data is always in the cache, this benchmark estimates the best possible performance from a nonblocking disk I/O layer. As the figure shows, throughput remains constant as the load is increased to a very large number of tasks (note the change in the horizontal axis scale from Figure 3), and response time is linear (note the log scale on the horizontal axis).*

of representing request state as a lightweight continuation, rather than as a heavyweight thread context. As the number of requests increases, the server throughput increases until the pipeline fills and the bottleneck (the CPU in this case) becomes saturated. If the number of requests is increased further, excess work is absorbed as a continuation in the server's event queue; this is in contrast to the thread-per-request model in which a new thread is forked for each incoming request. The throughput remains constant across a huge range in load, with the latency of each task increasing linearly.

It is important to note that while event-driven systems are highly efficient in terms of managing concurrency, it is still possible for such a system to exhibit performance degradation due to other resource bottlenecks in the system. For example, if each request consumes a great deal of memory, then admitting too many simultaneous requests can lead to VM thrashing. Therefore the event-driven design only addresses the concurrency aspect of scaling service capacity.

The event-driven design is used by a number of systems, including the Flash [106], thttpd [4], Zeus [151], and JAWS [61] Web servers, and the Harvest [25] Web cache. Each of these systems uses event-driven concurrency in a slightly different way; here we focus on the Flash Web server as the canonical example that closely represents the implementation of the other systems.

Flash consists of a set of event handlers that respond to particular types of events, for example, read-readiness or write-readiness on a particular socket. The main server process runs in a loop, collecting events (using the UNIX *select()* system call) and invoking the appropriate event handler for each event using a function call. Flash can be configured to use multiple server processes, for example, to take advantage of multiprocessor systems. Because most UNIX systems do not provide nonblocking file I/O interfaces, Flash makes use of a pool of of *helper processes*, which issue (blocking) I/O operations and signal an event to the main process upon completion. Helper processes are also used for pathname conversions (mapping URLs and symbolic links onto absolute pathnames for file access), listing the contents of directories, and dynamic page generation. The rationale is that these operations are filesystem-intensive and may block the main server processes, so it is desirable to perform them asynchronously. Flash uses UNIX domain sockets to communicate between the processes, rather than shared memory, which requires relatively high overhead as the operating system must be involved in interprocess communication. This design also allows completion events from a helper process to be picked up in the main process's *select()* loop.

The Harvest Web cache is very similar in design: it is single-threaded and event-driven, with the exception of the FTP protocol, which is implemented by a separate process. In this case, a helper process is used not because FTP is inherently blocking, but rather due to the difficulty of writing an asynchronous FTP implementation.

The use of helper processes in Flash and Harvest underscores the occasional need for an event-driven system to resort to blocking operations, either to reduce complexity or to make use of legacy interfaces and libraries. Most event-driven systems develop *ad hoc* mechanisms for integrating blocking code into the service, requiring careful management of thread/process resources to perform the dispatch.

The tradeoffs between threaded and event-driven concurrency models have also been studied in the JAWS Web server [60, 61]. The focus in JAWS is on the software engineering and reusability aspects of Web server design, and makes heavy use of *design frameworks* [41]. JAWS consists of a framework for Web server construction allowing the concurrency model, protocol processing code, cached filesystem, and other components to be customized. JAWS uses the *Proactor pattern* to represent event dispatching and completion notification; the resulting design is very similar to Flash. To our knowledge, JAWS has only been evaluated under light loads (less than 50 concurrent clients) and has not addressed the use of adaptivity for conditioning under heavy load.

This body of work has realized the importance of using scalable primitives to support large degrees of concurrency. However, these systems have been constructed to support a specific application, such as as serving static Web pages, and have not approached the problem of concurrency and load management for a broad class of Internet services. The focus has been primarily on achieving the highest levels of raw performance and scalability, and less on managing variations in load or handling extreme overload conditions.

Event-driven design raises a number of challenges for the application developer. An important limitation of this model is that it requires event-handling code to be short and run to completion, to avoid stalling the event-processing threads, and to ensure fairness across a large number of requests. This typically requires that nonblocking, asynchronous I/O interfaces be used. While much prior work has explored scalable I/O interfaces [12, 13, 84, 111, 113], and nonblocking socket interfaces are common, many operating systems do not provide true nonblocking I/O for disk access. Flash, Harvest, and other systems address this challenge using separate threads or processes to perform long-running operations asynchronously. However, it is also possible for event-handling threads to block due to other events, such as interrupts, page faults, or garbage collection, so much care must be taken in the event-driven model to avoid stalling the system due to one of these unavoidable events.

A related issue is that of scheduling and ordering of event processing within the application. An event-driven system effectively implements its own scheduler that carefully multiplexes many request FSMs over limited thread resources, and must consider request prioritization, fairness, and resource consumption. The design of the event scheduler is one of the most subtle aspects of an event-driven system and is often tailored for a specific application with known resource demands for each request. However, this can lead to very brittle system design, as any changes to the application code may necessitate a redesign of the event scheduler. We refer to this design as a *monolithic* event-driven system, since service logic, scheduling, and resource management are bundled together in an application-specific way.

For these reasons, event-driven systems are often considered to be difficult to engineer, tune, and reason about. One of our goals in SEDA is to obtain the performance and scalability benefits of event-driven concurrency, but to develop a structured framework that eases the development of such systems. Also, event-driven concurrency by itself does not address the issue of overload management and resource control; this is the other major goal of the SEDA framework.

The classic work on the threads-versus-events dichotomy is Lauer and Needham's paper from 1978 [81]. This work carefully discusses the differences between "processes" and "procedures"—here, "processes" corresponds to event-driven concurrency while "procedures" corresponds to threading. The main claim is that the process-based and procedure-based models are duals of each other, in that any program implemented in one model can *just as efficiently* be implemented in the other. The paper presents a simple transformation from one concurrency model to the other and argues that the only reasons to choose processes over procedures (or vice versa) are based on external factors, such as which is more straightforward to program in a given environment.

While we agree with the basic argument, and it is clearly possible to transform a system between thread-based and event-driven concurrency models, we dispute the claim that performance can be preserved across the mapping. The main issue that this work overlooks is the complexity of building scalable, general-purpose multithreading; while in a simplistic sense, threads and events are identical, in practice threads embody a great deal of additional complexity that limit their scalability. Also, Lauer and Needham were not concerned with the difficulty of managing extreme loads in an environment such as an Internet service, which weighs the argument in favor of the event-driven approach.

1.5 Structured event queues

To counter the complexity and brittleness of event-driven design, a number of systems have proposed variants of the model. A common aspect of these designs is to structure an event-driven application using a set of event queues to improve code modularity and simplify application design.

One of the starting points of the SEDA design was the I/O core framework [54] used by the Ninja system at UC Berkeley. This framework was used in Gribble's Distributed Data Structures (DDS) [52] layer and initial versions of the vSpace [139] cluster-based service platform. In the Ninja I/O core, a limited number of threads is used to process requests flowing through the system, and blocking socket and disk I/O operations are converted into asynchronous, split-phase operations using a dedicated pool of threads, similar to the Flash use of helper processes. One of the limitations of this design was that true nonblocking I/O was not used for socket connections, so the I/O core makes use of one thread per socket for reading data from the network, and a (large) pool of threads for writing data to the network—thereby limiting the number of simultaneous socket connections that can be handled. This is not a serious issue for DDS, since it requires only a small number of socket connections within a cluster. However, for general-purpose Internet services, such as those hosted by vSpace, this is an important restriction.

In the I/O core, each software component exports an event-handling interface with a single method that accepts an event to process. Applications are constructed by composing these components together using either an upcall interface (in which one component directly invokes the event-handling method of another component) [31] or using a queue (in which one component enqueues an event to be processed asynchronously by another component). The basic programming model makes no distinction between upcalls and queues, requiring application designers to be very careful when composing two event-processing components. If an upcall is used, then a component that performs a long, resource-intensive operation will stall the caller until its return. Likewise, enqueueing an event for asynchronous processing entails additional context switch and synchronization overhead. The SEDA model was developed with the goal of generalizing and improving upon this basic framework, in addition to applying it to a wider class of services. In Chapter 8 we elaborate on the differences between SEDA and the Ninja I/O core.

The Click modular packet router [77] is another example of a system making use of structured event queues. In Click, packet processing components are implemented as separate code modules each with private state. Like the Ninja I/O core, Click components can be composed using direct function calls or queues. One of the most critical aspects of application design is determining how components should be composed. Click is targeted at a specific application (packet routing) and uses a single thread to process all queues; a multiprocessor extension to Click [26] uses a thread for each processor and performs load-balancing across threads.

A primary goal in Click is to optimize per-packet routing latency, so it is often desirable to have a single thread call directly through multiple packet-processing components to avoid queueing and scheduling delays. For this design to be feasible, Click makes the assumption that modules have bounded processing times, leading to a relatively static determination of resource-management policies. In Click, there are effectively no long blocking operations, so asynchrony is used only for decoupling the execution of multiple packet-processing components; for example, a queue is often used for temporary packet storage before an outgoing network interface.

StagedServer [80] also makes use of modules communicating using explicit event queues. Like SEDA, this work introduces the notion of a stage as the unit of request processing within a server application. StagedServer's goal is to maximize processor cache locality by carefully scheduling the order in which stages are executed, as well as the order in which requests within a stage are processed. This is accomplished using *cohort scheduling*, which batches the execution of related aspects of request processing. For example, requests within a stage are processed in last-in-first-out order, increasing the likelihood that recently processed request data is in the cache.

Work Crews [137] is another system that makes use of structured event queues and limited numbers of threads to manage concurrency. Work Crews deals with the issue of exploiting potential (compute-bound) parallelism within a multiprocessor system while avoiding the creation of large numbers of threads. A Work Crew consists of a bounded number of worker threads (typically one per CPU) that dispatch work requests from a single queue. Rather than forking a new thread to subdivide a parallel operation, a help request is placed onto the queue for servicing by a worker thread. Work Crews supports various synchronization primitives allowing task processing to be subdivided in various ways and to collect the results from dispatched help requests. This work identified the importance of limiting thread contexts due to resource contention, and the value of exposing queues for performance gains: in this case, allowing workers to process tasks serially when additional processors are unavailable for parallel execution. The ALARM concurrency mechanism proposed by Vajracharva and Chavarría-Miranda [135] is very similar to Work Crews.

Each of these systems employs structured event queues to solve a specific problem, such as packet processing, cache locality, or multiprocessor scheduling. Our goal in SEDA is to extend this approach to provide a general framework that addresses the concerns of large-scale Internet services: massive concurrency, load management, request-processing efficiency, and code modularity.

2 Challenges posed by OS virtualization

An additional hurdle to the design of well-conditioned services is the lack of resource control exposed by commodity operating system designs. The design of most current operating systems is primarily derived from a heritage of multiprogramming: allowing multiple applications, each with distinct resource demands, to share a single set of resources in a safe and efficient manner. As such, existing OSs strive to virtualize hardware resources, and do so in a way that is transparent to applications. Applications are rarely, if ever, given the opportunity to participate in system-wide resource management decisions, or given indication of resource availability in order to adapt their behavior to changing conditions. Virtualization fundamentally hides the fact that resources are limited and shared [145].

The focus on resource virtualization in existing operating systems presents a number of challenges for scalability and load conditioning:

Concurrency limitations: Internet services must efficiently multiplex many computational and I/O flows over a limited set of resources. Given the extreme degree of concurrency required, services are often willing to sacrifice transparent virtualization in order to obtain higher performance. However, contemporary operating systems typically support concurrency using the process or thread model, and as discussed earlier, these abstractions entail high overheads.

I/O Scalability limitations: The I/O interfaces exported by existing OSs are generally designed to provide maximum transparency to applications, often at the cost of scalability and predictability. Most I/O interfaces employ blocking semantics, in which the calling thread is suspended during a pending I/O operation. Obtaining high concurrency requires a large number of threads, resulting in high overhead.

In order to sidestep the scalability limits of threads, nonblocking I/O interfaces are often used. Nonblocking socket interfaces are common, though most commodity operating systems still do not provide nonblocking I/O interfaces for disk access. Many nonblocking I/O interfaces tend to degrade in performance as the number of I/O flows grows very large [13, 111, 74, 97]. Likewise, data copies on the I/O path (themselves an artifact of virtualization) have long been known to be a performance limitation in network stacks [112, 140, 141].

This problem has led to the development of a range of scalable I/O primitives [13, 40,

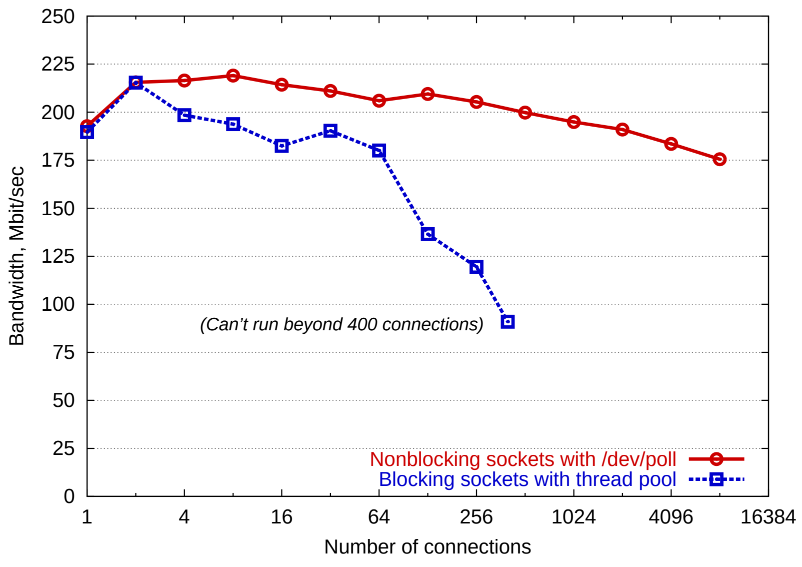


Figure 7: **Performance degradation of nonblocking sockets:** *This graph shows the aggregate bandwidth through a server making use of either nonblocking or blocking socket interfaces. Each client opens a connection to the server and issues bursts of 1000 8 KB packets; the server responds with a single 32-byte ACK for each burst. All machines are 4-way Pentium III systems running Linux 2.2.14 connected using Gigabit Ethernet. Two implementations of the server are shown: one makes use of nonblocking sockets with the* `/dev/poll` *mechanism for event delivery, and the other makes use of blocking sockets and a bounded thread pool to emulate asynchrony. The latter implementation allocates one thread per socket for reading packets, and uses a fixed-size thread pool of 120 threads for writing packets. The threaded implementation could not support more than 400 simultaneous connections due to thread limitations under Linux, while the nonblocking implementation degrades somewhat due to lack of scalability in the network stack.*

71, 84, 107, 111, 113], although these mechanisms can also face scalability challenges if the underlying operating system is not designed to support massive scalability. To demonstrate this fact, we have measured the performance of the nonblocking socket interface in Linux using the `/dev/poll` [111] event-delivery mechanism, which is intended to provide a greater degree of scalability than the standard UNIX *select()* and *poll()* interfaces [74]. As Figure 7 shows, the performance of the nonblocking socket layer still degrades somewhat when a large number of connections are established; we believe this is due to inherent scalability limitations in the Linux TCP/IP stack. Also shown is the same benchmark using blocking I/O and a bounded thread pool to perform I/O, which exhibits very severe performance degradation.

Transparent resource management: Internet services must be in control of resource usage in order to make informed decisions affecting performance. Virtualization implies that the OS will attempt to satisfy any application request regardless of cost, and the performance aspects of the operating system's interfaces are often hidden from applications.

For example, a request to allocate a page of virtual memory may require other pages to be swapped out to disk; this can cause a very serious performance problem if the number of VM pages requested exceeds the size of physical memory, a problem known as *thrashing*. Likewise, the existence of (or control over) the filesystem buffer cache is typically not exposed to applications. This can have a serious impact on application performance if the resource usage and access patterns used by the application do not map well onto the operating systems' policies. This is one of the major issues with operating system design often cited by database designers [121].

Internet services do not have the luxury of paying an arbitrary penalty for processing such requests under heavy resource contention. In many cases it is necessary for a service to prioritize request processing based on resource availability. For instance, a service may be able to process "cheap" requests during a period of high load while "expensive" requests (those requiring many resources) are delayed or rejected. Unfortunately, without knowledge of or control over the resource management policies of the OS, it is difficult for an application to implement such a policy.

An interesting way around this problem is to adopt a "gray box" approach, as proposed by Arpaci-Dusseau [10], in which the application infers the underlying OS policies and adapts its behavior to optimize for them. Burnett *et al.* use this idea to optimize the order in which Web page requests are serviced based on knowledge of the OS buffer cache policy [23].

Coarse-grained scheduling: The thread-based concurrency model yields a coarse degree of control over resource management and scheduling decisions. Although it is possible to control the prioritization or runnable status of an individual thread, this is often too blunt of a tool to implement effective load conditioning policies. Instead, it is desirable to control the flow of requests through a particular resource.

As an example, consider the page cache for a Web server. To maximize throughput and minimize latency, the server might prioritize requests for cache hits over cache misses; this is a decision that is being made at the level of the cache by inspecting the stream of pending requests. Such a policy would be difficult, although not impossible, to implement by changing the scheduling parameters for a set of threads, each representing a different request in the server pipeline. The problem is that this model only provides control over scheduling of individual threads, rather than over the ordering of requests for a particular resource.

2.1 Approaches to customized resource management

A number of systems have attempted to remedy this problem by exposing greater resource control to applications. Scheduler activations [7], application-specific handlers [143], user-level virtual memory managers [58, 93], and operating systems such as SPIN [16], Exokernel [70], and Nemesis [87] are all attempts to augment limited operating system interfaces by giving applications the ability to specialize the policy decisions made by the kernel.

Application-specific handlers allow applications to push fast message handling code into the kernel, allowing data-copying overheads to be avoided, as well as customized protocols to be implemented. User-level VM systems allow the application to tailor the virtual memory paging policy, for example, by managing control over the set of physical pages allocated to a process. SPIN and Exokernel take these ideas much further, allowing the resource-management policies of the OS, including scheduling, memory management, and I/O, to be tailored for a specific application. In SPIN, applications can push safe code (in the form of Modula-3 components) into the kernel, while Exokernel takes the approach of using a minimal kernel to safely export resources to applications to control. Nemesis is an operating system designed for delivering quality-of-service guarantees to applications, and uses ideas similar to Exokernel in terms of fine-grained multiplexing of system resources, and allowing applications to participate in resource-management decisions.

These systems have realized the need to give applications greater control over resource usage, though they do not explicitly focus on providing customized resource management for the purposes of overload control. Much of the design of these systems is still based on the multiprogramming mindset, in that the focus continues to be on safe and efficient (though customized) resource virtualization, rather than on high concurrency and graceful management of load. In addition, we think it is a worthwhile question to consider whether it is possible to build effective resource control on top of a commodity operating system, without resorting to a major redesign of the OS. The approach taken in SEDA can be thought of as a middleware layer between the application and the operating system, monitoring application behavior and effecting control over resource consumption. However, giving the application greater control over resource management is an interesting area for future work, and we expect that SEDA would benefit from the techniques proposed by systems such as Exokernel and Nemesis. We return to this issue in Chapter 8.

3 Approaches to overload management

Overload is an inescapable reality for popular Internet services, and several approaches to overload management in Web services have been proposed. In this section we survey the field of Internet service overload control, discussing prior work as it relates to four broad categories: resource containment, admission control, control-theoretic approaches, and service degradation.

3.1 Resource containment

The classic approach to resource management in Internet services is static resource containment, in which *a priori* resource limits are imposed on an application or service to avoid overcommitment. We categorize all of these approaches as *static* in the sense that some external entity (say, the system administrator) imposes a limit on the resource usage of a process, application, or other resource-owning entity. Although resource limits may change over time, they are typically not driven by monitoring and feedback of system performance; rather, the limits are arbitrary and rigid.

In a traditional thread-per-connection Web server design, the only overload mechanism generally used is to bound the number of processes (and hence the number of simultaneous connections) that the server will allocate. When all server threads are busy, the server stops accepting new connections; this is the type of overload protection used by Apache [8]. There are three serious problems with this approach. First, it is based on a static thread or connection limit, which has a complex relationship with the performance of the service based on user load, the length of time a given connection is active, and request type (e.g., static versus dynamic pages). Secondly, not accepting new TCP connections gives the user no indication that the site is overloaded: the Web browser simply reports that it is still waiting for a connection to the site. This wait time can grow to be very long, as TCP uses an exponentially increasing timeout for establishing new connections, meaning that several minutes may pass before the connection attempt is aborted. Finally, this approach is extremely unfair to clients, as some clients will be able to establish connections quickly, while others will experience large backoff delays.

Zeus [151] and thttpd [4] provide mechanisms to throttle the bandwidth consumption for certain Web pages to prevent overload, based on a static bandwidth limit imposed by the system administrator for certain classes of requests. A very similar mechanism has been described by Li and Jamin [88]. In this model, the server intentionally delays outgoing replies to maintain a bandwidth limit, which has the side-effect of tying up server resources for greater periods of time to deliver throttled replies.

Somewhat related to the issue of bandwidth throttling is the use of network scheduling techniques to give priority to some responses over others. One technique that has been explored is the use of shortest-remaining-processing-time (SRPT), alternately called shortest-connection-first (SCF) scheduling [35, 57, 116]. In this technique, packets for connections with less remaining outgoing data are scheduled first for network transmission. In [116], Schroeder and Harchol-Balter investigate use of SRPT network scheduling for managing overload; they show that under a heavy-tailed request size distribution, SRPT greatly reduces response times and does not penalize long responses. Their study is limited to static responses and does not look at extreme overload cases where some other form of overload control (such as admission control) would be needed in addition to network scheduling.

Another style of resource containment is that typified by a variety of real-time and multimedia systems. In this approach, resource limits are typically expressed as reservations or shares, as in "process P gets X percent of the CPU." In this model, the operating system must be careful to account for and control the resource usage of each process. Applications are given a set of resource guarantees, and the system prevents guarantees from being exceeded through scheduling or forced termination. Examples of systems that have explored reservation- and shared-based resource limits include Scout [100], Nemesis [87], Resource Containers [11], and Cluster Reserves [9].

This kind of resource containment works well for real-time and multimedia applications, which have relatively static resource demands that can be expressed as straightforward, fixed limits. For example, a multimedia streaming application has a periodic time deadline that it must meet to (say) continue displaying video at a given framerate. For this class of applications, guaranteeing resource availability is more important than ensuring high concurrency for a large number of varied requests in the system. Moreover, these systems are focused on resource allocation to processes or sessions, which are fairly coarse-grained entities. In an Internet service, the focus is on individual requests, for which it is permissible (and often desirable) to meet statistical performance targets over a large number of requests, rather than to enforce guarantees for particular requests.

In general, resource containment has the fundamental problem that it is generally infeasible to determine ideal *a priori* resource limitations in a highly dynamic environment. Setting limits too low underutilizes resources, while setting them too high can lead to oversaturation and serious performance degradation under overload. Returning to the example of bounding the number of simultaneous connections to a Web server, the right limit depends greatly on load: for static Web page accesses, a limit of several thousand or more connections may be acceptable; for resource-intensive dynamic content, the feasible limit may be on the order of tens of connections. More critically, given the rapidly evolving nature of application code (and hence resource usage) in a typical Internet service, it seems likely that any fixed “solution” for the ideal resource limits will change whenever the service logic changes.

3.2 Admission control

Another way to approach overload management in Internet services is to perform admission control, in which the system restricts the set of requests entering the system according to some policy. Admission control has been effectively employed in networks, for example, to provide quality-of-service guarantees to applications by restricting the set of flows that are admitted to the network [21, 20]. Several approaches to per-request and per-session admission control for Internet services have been proposed.

Many of these systems are *parameter based*, in that they operate based on fixed parameters that define the admission control policy. Like resource containment, admission control can lead to inflexible overload management if it is driven by static policies that are not ultimately tied to the performance of the service and the resource usage of admitted requests. Also, many of these systems consider only static Web page loads, and others are studied only in simulation.

Another common aspect of these approaches is that they often reject incoming work to a service by refusing to accept new client TCP connections. As we have discussed previously, we argue that this is an inappropriate way to perform load shedding, as it gives users no indication of overload, and leads to even longer client-perceived response times. Such an approach pushes the burden of dealing with excess demand for a service into the network.

Iyer *et al.* [65] describe a simple admission control mechanism based on bounding the length of the request queue within a Web server. Load shedding is accomplished by dropping incoming connections when the queue length exceeds an administrator-specified discard threshold, and new connections are allowed when the queue length falls below an abatement threshold. Alternatively, the system can inform a Web proxy to delay new requests, though this only pushes the problem of connection management to the proxy. This work analyzes various settings for the discard and abatement thresholds, though does not specify how these thresholds should be set to meet any given performance target.

In [30], Cherkasova and Phaal present *session-based* admission control, which performs an admission decision when a new session arrives from a user, rather than for individual requests or connections. In many Internet services, it is undesirable to reject requests from users that have already established an active session with the service; this work realizes the importance of sessions as the unit of load management. The proposed technique is based on CPU utilization and causes new sessions to be rejected (by sending an explicit error message to the client) when CPU utilization exceeds a given threshold. The paper presents simulation studies that explore the tradeoffs of several parameter settings.

Many approaches to overload management are performed entirely in the application or in a user-level middleware component that does not require specific operating system functionality to accomplish its goals. In contrast, Voigt *et al.*[138] present several kernel-level mechanisms for overload management: restricting incoming connections based on dropping SYN packets; parsing and classification of HTTP requests in the kernel; and ordering the socket listen queue by request URL and client IP address. SYN rate policing is accomplished using static token-bucket parameters assigned to IP address ranges. Ordering the socket listen queue according to request type or IP address causes the Web server to give priority to certain requests over others. However, this technique would not appear to be effective for a Web server that rapidly accepts all pending connections (as is the case in SEDA). Here, the admission control rules are statically determined, and the question arises as to how well these different techniques work together in a realistic setting. Another traffic-shaping approach is described in [69], which drives the selection of incoming packet rates based on an observation of system load, such as CPU utilization and memory usage.

Web2K [17] brings several of these ideas together in a Web server “front-end” that performs admission control based on request type or user class (derived from IP address, cookie information, and so forth). Incoming connections are rapidly accepted, which avoids dropped SYN packets, and requests are read and placed onto an internal queue based on priority. An unmodified Web server (in this case, Apache) pulls requests from these queues through a provided library. Admission control is performed that attempts to bound the length of each priority queue; as in [65], the issue of determining appropriate queue thresholds is not addressed. The admission control scheme makes use of a simple controller that predicts how many requests of each priority class can be accepted based on past observations, though we believe this mechanism could be greatly simplified (for example, using the technique from [65]). A form of session-based admission control is used in that requests from old sessions are prioritized before new ones.

In each of these systems, the problem arises that it is difficult to map from low-level observations of system resource availability (such as queue length or CPU utilization) to performance targets that are meaningful to users or system administrators. Relationships between an Internet service provider and a customer (e.g., a client or hosted service) are often expressed in terms of a *service level agreement* (SLA) that is based on client-perceived performance, not the internal activity of the service. A simple example of an SLA might be "for each request that meets a response-time bound of T we will pay D dollars." To effectively meet these kinds of SLAs, admission control techniques need to take client-perceived performance into account.

Several systems have taken such an approach. In some cases, control theory is used to design and tune an admission control technique; we describe a number of control-theoretic approaches together in Section 3.3. PACERS [28] is a capacity-driven admission control technique that determines the number of requests to admit based on expected server capacity in each time period. By admitting a limited number of requests during some time period T, the response time of each request is bounded by T. Requests are prioritized and higher-priority requests are admitted before lower-priority requests. This paper only deals with simulation results using a simple service model in which static page requests require processing time that is linear in the page size, and dynamic page requests consume constant time. The mechanism described here seems overly complex, and details on request classification and the actual rejection mechanism are not discussed.

A related technique, proposed by the same group, allocates requests to Apache server processes to minimize per-class response time bounds [27]. A weight is assigned to each class of requests to maximize a "server productivity" function, defined in terms of the number of pending requests in each class, a (fixed) per-class processing-time requirement, and a (fixed) delay bound. The paper is very unclear on implementation details and how the assigned weights are actually employed. This paper considers only static Web pages and silently drops requests if delay bounds are exceeded, rather than explicitly notifying clients of overload. This technique appears to be similar to [89], discussed in Section 3.3 below.

Finally, Kanodia and Knightly [72] develop an approach to admission control based on the use of *service envelopes*, a technique used in networking to characterize the traffic of multiple flows over a shared link. Service envelopes capture the server's request rate and service capacity without requiring detailed modeling techniques. The admission control technique attempts to meet response-time bounds for multiple classes of service requests, so is more closely tied to the kind of SLAs that real systems may employ. However, the technique is only studied under a simple simulation of Web server behavior.

3.3 Control-theoretic approaches to resource management

Control theory [105] provides a formal framework for reasoning about the behavior of dynamic systems and feedback-based control. A number of control-theoretic approaches to performance management of real systems have been described [108, 90], and in this section we survey several control-theoretic techniques specifically focused on overload control for Internet services.

Abdelzaher and Lu [3] describe a control-based admission control scheme that attempts to maintain a CPU utilization target of $\ln 2$. This particular value is derived from results in real-time scheduling theory, which do not seem to apply in a standard operating system (which is the evaluation environment used in the paper). A proportional-integral (PI) controller is designed using a very simplistic linear model of server performance, based on request rate and network bandwidth. Apart from ignoring caching, resource contention, and a host of other effects, this model is limited to static Web page accesses. The paper does not elaborate on exactly how requests are rejected.

An alternative approach to overload management is proposed by the same group in [89]. Here, the overload control technique is based on allocating server processes (assuming a thread-per-connection server design, e.g., Apache) to each class of pending connections, where classification is performed by client IP address. The controller attempts to maintain a given *relative delay* between classes: for example, a low-priority class experiences twice the delay of a high-priority class. This is an interesting choice of performance metric, and it is unclear whether such an SLA would ever be used in practice. A PI controller is designed and white-noise system identification (a technique commonly employed in controller design for physical systems) is used to determine a second-order model of system performance.

Finally, Diao *et al.* describe a control-based mechanism for tuning Apache server parameters to meet resource utilization targets [37]. A multi-input-multi-output (MIMO) PI controller design is presented that adjusts both the number of server processes and the per-connection idle timeout in the Apache server to obtain given levels of CPU and memory utilization. Recall that in Apache, reducing the number of server processes leads to increased likelihood of dropped incoming connections; while this technique effectively protects server resources from oversaturation, it results in poor client-perceived performance. This paper considers a static Web page workload and uses system identification techniques from control theory to derive a linear model of server performance.

Although control theory provides a useful set of tools for designing and reasoning about systems subject to feedback, there are many challenges that must be addressed in order for these techniques to be applicable to real-world systems. One of the biggest difficulties in applying classic control theory to complex systems is that good models of system behavior are often difficult to derive. Unlike physical systems, which can often be described by linear models or approximations, Internet services are subject to poorly understood traffic and internal resource demands. As a result, Internet services tend to be highly nonlinear, which limits the applicability of standard modeling and control techniques. The systems described here all make use of linear models, which may not be accurate in describing systems with widely varying loads and resource demands. Moreover, when a system is subject to extreme overload, we expect that a system model based on low-load conditions may break down.

Many system designers resort to *ad hoc* controller designs in the face of increasing system complexity. Although such an approach does not lend itself to formal analysis, careful design and tuning may yield a robust system regardless. Indeed, the congestion-control mechanisms used in TCP were empirically determined, though some recent work has attempted to apply control-theoretic concepts to this problem as well [73, 75]. A benefit to *ad hoc* controller design is that it does not rely on complex models and parameters that a system designer may be unable to understand or to tune. A common complaint of classic PID controller design is that it is often difficult to understand the effect of gain settings.

3.4 Service degradation

Rather than rejecting requests, an overloaded Internet service could take an alternate action that requires fewer resources, but delivers a lower fidelity of service to the client. We use the term *service degradation* to describe a class of overload management policies that take this approach.

The most straightforward form of service degradation is to reduce the quality of static Web content, such as by reducing the resolution or compression quality of images delivered to clients. This approach has been considered in a number of projects [45, 2, 24], and has been shown to be an effective form of overload control. In many cases the goal of image quality degradation is to reduce network bandwidth consumption on the server, though this may have other effects as well, such as memory savings.

A more sophisticated example of service degradation involves replacing entire Web pages (with many inlined images and links to other expensive objects) with stripped-down Web pages that entail fewer individual HTTP requests to deliver. This is exactly the approach taken by CNN.com on September 11, 2001, when server load spiked to unprecedented levels following terrorist attacks on the World Trade Center and the Pentagon. CNN replaced its front page with simple HTML page that that could be contained in a single Ethernet packet [83]. However, this was an extreme measure that was put in place manually by the system administrators. A better approach would be to degrade service gracefully and automatically in response to load. In [24], for example, the Web server degrades the compression quality of JPEG images when bandwidth utilization exceeds a target.

In some cases it is possible for a service to make performance tradeoffs in terms of the freshness, consistency, or completeness of data delivered to clients. Brewer and Fox [44] describe this tradeoff in terms of the *harvest* and *yield* of a data operation; harvest refers to the amount of data represented in a response, while yield (closely related to availability) refers to the probability of completing a request. It is often possible to achieve better performance from an Internet service by reducing the harvest or yield requirements of a request. For example, a Web search engine could reduce the amount of the Web database searched when overloaded, and still produce results that are good enough such that a user may not notice any difference.

One disadvantage to service degradation is that many services lack a “fidelity knob” by design. For example, an e-mail or chat service cannot practically degrade service in response to overload: “lower-quality” e-mail and chat have no meaning. In these cases, a service must resort to admission control, delaying responses, or one of the other mechanisms described earlier. Indeed, rejecting a request through admission control is the lowest quality setting for a degraded service.

4 Summary

In this chapter we have surveyed the two main challenges for this dissertation: scaling to extreme levels of concurrency and maintaining robust performance under heavy load. The traditional thread-based programming model exhibits problems along both axes, in that such systems typically do not scale well and yield little control over resource usage, as required for load conditioning. Although explicit event-driven concurrency mechanisms mitigate many aspects of the scalability issue, they entail a great deal of complexity in terms of application design and scheduling. Also, event-driven concurrency alone does not address the more general issue of load management; it simply alleviates the scalability limitations of the threaded design.

Graceful management of load is hindered by resource virtualization, which makes it difficult for a service to observe or react to overload conditions, as physical resource limits are hidden. A number of systems have explored ways to expose greater information and control over resource usage to applications, though few of these techniques have specifically looked at the problem of massive scale and dynamicism in an Internet service environment.

A critical aspect of service design is overload prevention, which comes in many forms. The most widely used approaches are based on static resource limits (e.g., bounding the number of connections to a service) or admission control (e.g., bounding the rate at which new requests enter the service). Static resource limitations are inflexible, and it is difficult to determine how such limits should be set. Many overload control schemes are based on metrics such as CPU utilization or queue length, which are not tied to client-perceived performance.

A better way to approach overload is based on feedback and control, in which the system observes its own behavior and adjusts resource usage or admission control parameters to meet some performance target. A number of systems have taken such an approach, using either *ad hoc* controller designs, or more formal techniques based on control theory. Control theory raises a number of challenges when applied to complex software systems, though the tools it provides may be useful even if complete system models cannot be derived.

Finally, an alternative to rejection of work in an overloaded service is to degrade the quality of service delivered to clients. This can be performed in a number of ways, such as by reducing the quality of images or the freshness of data. However, not all services can be effectively shielded from overload using degradation, so other techniques must be employed as well.

Bringing all of these ideas together, the goal of this thesis is to design a *general-purpose* framework for handling massive concurrency and load in complex Internet services. Much of the prior work described here as addressed aspects of this problem, often in the context of specific applications under limited conditions (e.g., a Web server for static pages). Our focus is on providing a framework that can host a wide range of dynamic Internet services and manage much of the complexity of service construction, tuning, and conditioning to heavy load conditions. The SEDA approach, described in the following chapters, incorporate aspects of many of the techniques described here, including efficient event-driven concurrency; structured queues for modularity; self-monitoring for detecting overload conditions; as well as feedback, admission control, and service degradation for prevention of overload.

Chapter 3

The Staged Event-Driven Architecture

In this chapter we propose a new software architecture, the *staged event-driven architecture* (SEDA), which is designed to enable high concurrency, load conditioning, and ease of engineering for Internet services. SEDA decomposes an application into a network of *stages* separated by *event queues*. SEDA makes use of *dynamic resource control* to allow applications to adapt to changing load and resource conditions. An overview of the SEDA approach to service design is shown in Figure 8.

1 Design overview

Before presenting the SEDA architecture in detail, here we present a high-level overview of the main aspects of the design:

Efficient, event-driven concurrency: To support massive degrees of concurrency, SEDA relies on event-driven techniques to represent multiple flows through the system. This design makes use of a small number of threads, rather than one thread per request. Nonblocking I/O primitives are used to eliminate common sources of blocking.

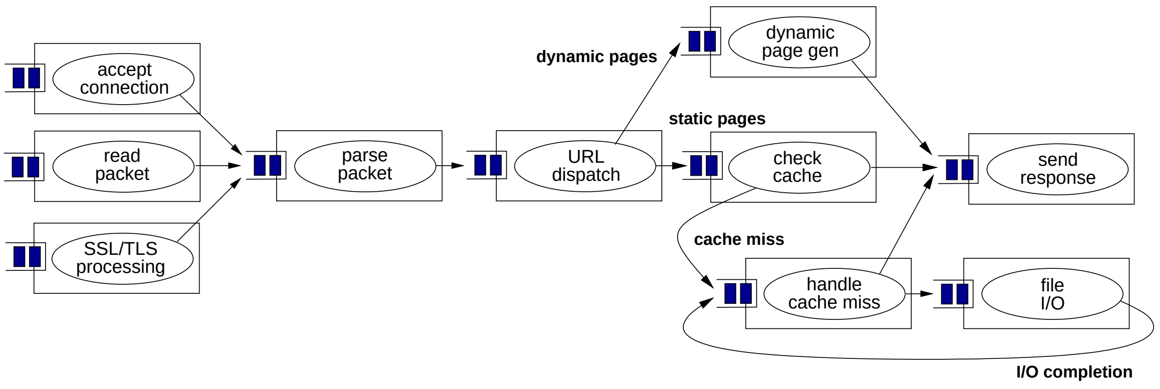


Figure 8: **Staged event-driven (SEDA) HTTP server:** *This is a structural representation of Haboob, the SEDA-based Web server, described in detail in Chapter 6. The application is composed as a set of* stages *separated by* queues. *Edges represent the flow of events between stages. Each stage can be independently managed, and stages can be run in sequence or in parallel, or a combination of the two. The use of event queues allows each stage to be individually load-conditioned, for example, by performing admission control on its event queue. For simplicity, some event paths and stages have been elided from this figure.*

Dynamic thread pooling: To relax the rigid scheduling and nonblocking requirements for event-driven concurrency, SEDA uses a set of *thread pools*, one per stage, to drive execution. This not only frees the application programmer from having to implement event scheduling (as the operating system handles scheduling threads), but also allows event-handling code to block for brief periods of time, as additional threads can be allocated to a stage.

Structured queues for code modularity and load management: By partitioning an application into a set of stages with explicit queues between them, application designers can focus on the service logic and concurrency management for individual stages, “plugging” them together into a complete service later. Queues decouple the execution of each stage, allowing stages to be developed independently. Queues provide a point of control over the request stream in a service, as requests flowing across stages can be inspected and managed by the application. Likewise, admission control can be performed on a per-stage basis.

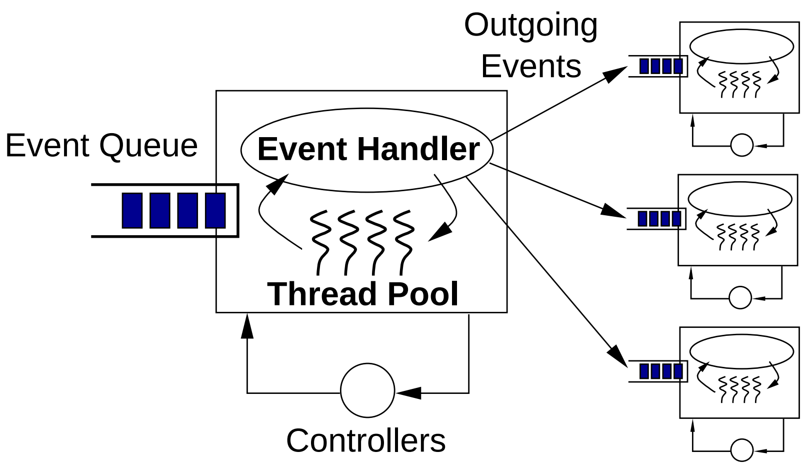


Figure 9: **A SEDA Stage:** *A stage consists of an* incoming event queue*, a* thread pool*, and an application-supplied* event handler*. The stage's operation is managed by a set of* controllers*, which dynamically adjust resource allocations and scheduling.*

Self-tuning resource management: Rather than mandate *a priori* knowledge of application resource requirements and client load characteristics, SEDA makes use of feedback and control to automatically tune various resource usage parameters in the system. For example, the system determines the number of threads allocated to each stage based on perceived concurrency demands, rather than relying on a hard-coded value set by the programmer or administrator.

2 Stages as robust building blocks

The fundamental unit of processing within SEDA is the *stage*. A stage is a self-contained application component consisting of an *event handler*, an *incoming event queue*, and a *thread pool*, as depicted in Figure 9. Each stage is managed by one or more *controllers* that affect resource consumption, scheduling, thread allocation, and admission control, as described below. Threads within a stage operate by pulling a *batch* of events off of the incoming event queue and invoking the application-supplied event handler. The event handler processes each batch of events, and dispatches zero or more events by enqueueing them on the event queues of other stages.

2.1 Events and batching

Each *event* processed by a stage is typically a data structure representing a single client request to the Internet service, for example, an HTTP request for a Web page. However, events may represent other information relevant to the operation of the service, such as a network connection being established, the firing of a timer, or an error condition. Throughout this thesis we often use the terms *event* and *request* interchangeably, though technically a *request* is generated by a client, while other types of events may be generated by the service internally.

The SEDA programming model exposes batches of events, rather than individual events, to the application. This allows the application to exploit opportunities for processing multiple events together, which can lead to higher performance. For example, if the event handler is able to amortize an expensive operation across multiple events, throughput can be increased. Likewise, data and cache locality may benefit if the processing for multiple events shares common code and data structures [80].

2.2 Event handlers

In SEDA, a service designer implements an *event handler* for each stage, which represents the core service-processing logic for that stage. An event handler is simply a function that accepts a batch of events as input, processes those events, and (optionally) enqueues outgoing events onto other stages. The event handler itself does not have direct control over threads within a stage, the input queue, and other aspects of resource management and scheduling. In some sense, event handlers are passive components that are invoked by the runtime system in response to event arrivals, and the runtime makes the determination of when and how to invoke the event handler.

The motivation for this design is separation of concerns: by separating core application logic from thread management and scheduling, the runtime system is able to control the execution of the event handler to implement various external resource-management policies. For example, the number and ordering of events passed to the event handler is controlled by the runtime, as is the allocation and scheduling priority of threads operating within the stage. Our goal is to simplify service design by providing resource management within a generic framework that frees service developers from much of the complexity of load conditioning. However, this approach does not strip applications of all control over the operation of the service: within a stage, applications may make decisions as to request prioritization, ordering, or service quality.

2.3 Threading model

Threads are the basic concurrency mechanism within SEDA, yet their use is limited to a small number of threads per stage, rather than allocating a separate thread per request in the system. This approach has a number of advantages. First, the use of threads relaxes the constraint that all event-processing code be non-blocking, allowing an event handler to block or be preempted if necessary. In a traditional event-driven system, all blocking operations must explicitly capture the state of the computation and restore it when the operation completes. In SEDA, threads act as *implicit continuations*, automatically capturing the execution state across blocking operations.

SEDA applications create *explicit continuations* when they dispatch an event to another stage; the benefit of this approach is that continuation management can be performed by the application when it is most convenient (i.e., at stage-boundary crossings) rather than at any time during the event handler's operation. Creating such a continuation typically consists of allocating a (small) event data structure to represent the operation to be performed by the next stage, and filling it in with whatever state the stage will need to process it. In many cases the same event structure can be passed along from stage to stage, as stages perform transformations on its state.

Blocking effectively reduces the number of active threads within a stage, thereby degrading the stage's request-processing rate. To maintain high concurrency, additional threads may be allocated to a stage that exhibits blocking. However, to avoid overuse of threads (and attendant performance degradation), it is necessary that the number of threads needed to overlap blocking operations not be too large. In general this means that most blocking operations should be short, or that long blocking operations should be infrequent. Page faults, garbage collection, and occasional blocking calls to the operating system typically do not require a large number of threads to maintain a high degree of concurrency. As discussed below, SEDA relies on nonblocking I/O primitives to eliminate the most common sources of blocking. In addition, our implementation of SEDA automatically reduces the number of threads allocated to a stage when throughput degradation is detected.

The second benefit of this thread model is that SEDA can rely upon the operating system to schedule stages transparently on the underlying hardware. This is in contrast to the monolithic event-driven design, in which the service designer must determine the order and policy for dispatching events. In SEDA, the use of threads allows stages to run in sequence or in parallel, depending on the operating system, thread system, and scheduler. Given the small number of threads used in a typical SEDA application, it is acceptable to use kernel-level threads, which (unlike many user-level threads packages) can exploit multiple CPUs in a multiprocessor system.

The alternative to operating-system-managed scheduling would be for the SEDA runtime to schedule event handlers directly on the physical CPUs in the system. This difficulty with this approach is that it causes SEDA to duplicate most of the functionality of the OS scheduler, and to handle all aspects of blocking, preemption, prioritization, and so forth. However, this approach would yield a great degree of control over the ordering and prioritization of individual requests or stages in the system. Although there may be some benefit in exploring alternative scheduler designs for SEDA, our experiences with OS threads have been good and not necessitated application-level scheduling. In this dissertation we focus on the use of preemptive, OS-supported threads in an SMP environment, although this choice is not fundamental to the SEDA design.

It is not strictly necessary for each stage to have its own private thread pool; an alternate policy would be to have a single global thread pool and allow threads to execute within different stages based on demand. To simplify the discussion, we focus on the case where each stage contains a private thread pool, noting that other policies can be approximated by this model. Other thread allocation and scheduling policies are interesting areas for future work, and we discuss them in more detail in Chapter 8.

3 Applications as a network of stages

A SEDA application is constructed as a network of stages, connected by event queues. Event handlers may enqueue events onto another stage by first obtaining a handle to that stage's incoming event queue (through a system-provided lookup routine), and then invoking an *enqueue* operation on that queue.

An important aspect of queues in SEDA is that they are subject to *admission control*. That is, a queue may reject an enqueued event in order to implement some resource-management policy, such as preventing response times from growing above a threshold. A rejected enqueue operation is signaled immediately to the originating stage, for example, by returning an error code. Enqueue rejection acts as an explicit overload signal to applications and should be used by the service to adapt behavior in some way. The particular admission control mechanism used, and the response to a queue rejection, depends greatly on the overload management policy and the application itself. This is discussed in more detail in Section 5 below.

The network of stages may be constructed either statically (where all stages and connections between them are known at compile or load time) or dynamically (allowing stages to be added and removed at runtime). There are advantages and disadvantages to each approach. Static network construction allows the service designer (or an automated tool) to reason about the correctness of the graph structure; for example, whether the types of events generated by one stage are actually handled by stages downstream from it. Static construction may also permit compile-time optimizations, such as short-circuiting an event path between two stages, effectively combining two stages into one and allowing code from one stage to be inlined into another. The *Combine* design pattern, described in Section 6.4 below, formalizes the effect of this transformation.

Dynamic network construction affords much more flexibility in application design, permitting new stages to be added to the system as needed. For example, if some feature of the service is rarely invoked, the corresponding stages may only be instantiated on demand. Our implementation of SEDA, described in Chapter 5, makes use of dynamic stage graph construction and imposes few restrictions on how stages are composed.

3.1 Haboob: An example SEDA application

Figure 8 illustrates the structure of an example SEDA-based service, the Haboob Web server described in detail in Chapter 6. The service consists of a number of stages for handling network I/O, parsing HTTP requests, managing a cache of recently accessed pages, and so forth. Several stages, such as the network and file I/O interfaces, provide generic functionality that could be used by a range of services. These generic interfaces are described further in Chapter 5. Other stages, such as the static Web page cache, are specific to the Haboob service.

3.2 Service structuring issues

The particular structure of the Haboob server, and indeed any SEDA-based service, is based on a number of factors. Important issues to consider include performance, load conditioning, and code modularity. In Section 6 we present a set of design patterns that captures these tradeoffs in service design.

A fundamental tradeoff to consider is whether two request-processing code modules should communicate through a queue (i.e., implemented as separate stages), or directly through a function call (implemented as one stage). For raw performance reasons, it may be desirable to use direct invocation to avoid the synchronization and scheduling overhead of dispatching an event to a queue. This is the approach taken in Scout [100] and Click [77], which focused on reducing the latency of requests flowing through the system.

However, the use of queues has a number of benefits, including isolation, independent load management, and code modularity. Introducing a queue between two code modules decouples their execution, providing an explicit control boundary. The execution of a thread is constrained to a given stage, bounding its execution time and resource usage to that consumed within its own stage. A thread may only pass data across the boundary by enqueueing an event for another stage. As a result, the resource consumption of each stage can be controlled independently, for example, by performing admission control on a stage's event queue. An untrusted, third-party code module can be isolated within its own stage, limiting adverse effects of interference with other stages in the system.

Queues are also an excellent mechanism for structuring complex applications. Rather than exposing a typed function-call API, stages accept events of certain types and emit events of certain types; there need not be a one-to-one matching of event reception and emission. Stages are therefore composed using a *protocol*, rather than merely type-matching of function arguments and return values, admitting a flexible range of composition policies. For example, a stage might aggregate data across multiple events over time, only occasionally emitting a "summary event" of recent activity. Matching of event types across a queue interface need not be strict; for instance, a stage could transparently pass along event types that it does not understand.

Queues also facilitate debugging and performance analysis of services, which have traditionally been challenges in complex multi-threaded servers. Monitoring code can be attached to the entry and exit points of each stage, allowing the system designer to profile the flow of events through the system and the performance of each stage. It is also straightforward to interpose proxy stages between components for tracing and debugging purposes. Our implementation of SEDA is capable of automatically generating a graph depicting the connectivity of stages and the number of events passing through each queue. The prototype can also generate visualizations of event queue lengths, memory usage, and other system properties over time.

4 Dynamic resource control

A key goal of enabling ease of service engineering is to shield programmers from the complexity of performance tuning. In order to keep each stage within its ideal operating regime, SEDA makes use of *dynamic resource control*, automatically adapting the behavior of each stage based on observed performance and demand. Abstractly, a controller observes runtime characteristics of the stage and adjusts allocation and scheduling parameters to meet performance targets. Controllers can operate either with entirely local knowledge about a particular stage, or work in concert based on global state.

A wide range of resource control mechanisms are possible in the SEDA framework. One example is tuning the number of threads executing within each stage. If all operations within a stage are nonblocking, then a stage would need no more than one thread per CPU to handle load. However, given the possibility of short blocking operations, additional threads may be needed to maintain concurrency. Likewise, allocating additional threads to a stage has the effect of giving that stage higher priority than other stages, in the sense that it has more opportunities to execute. At the same time, it is important to avoid overallocating threads, which leads to performance degradation. Therefore the ideal number of threads per stage is based on the stage's concurrency demands a well as overall system behavior. Rather than allocate a static number of threads to each stage, our implementation of SEDA, described in Chapter 5, uses a *dynamic thread pool controller* to automatically tune thread allocations.

Another example is adjusting the number of events contained within each batch passed to a stage's event handler. A large batch size allows for increased locality and greater opportunity to amortize operations across multiple events, while a small batch size more evenly distributes work across multiple threads in a stage. In the prototype, a controller is used to tune the batch size based on the measured performance of each stage. Both of these mechanisms are described in more detail in Chapter 5.

Dynamic control in SEDA allows the application to adapt to changing conditions despite the particular algorithms used by the underlying operating system. In some sense, SEDA's controllers are naive about the resource management policies of the OS. For example, the SEDA thread pool sizing controller is not aware of the OS thread scheduling policy; rather, it influences thread allocation based on external observations of application performance. Although in some cases it may be desirable to exert more control over the underlying OS—for example, to provide quality of service guarantees to particular stages or threads—we believe that the basic resource management mechanisms provided by commodity operating systems, subject to application-level control, are adequate for the needs of Internet services. We return to this issue in Chapter 8.

5 Overload protection

Apart from tuning runtime parameters, another form of resource management in SEDA is *overload control*. Here, the goal is to prevent the service from exhibiting significantly degraded performance under heavy load due to overcommitting resources. As a service approaches saturation, the response times exhibited by requests can grow exponentially. To address this problem it is often desirable to shed load, for example, by sending explicit rejection messages to users, rather than cause all users to experience unacceptable response times.

Overload protection in SEDA is accomplished through the use of fine-grained admission control at each stage, which can be used to implement a wide range of policies. Generally, by applying admission control, the system can limit the rate at which that stage accepts new requests, allowing performance bottlenecks to be isolated. A simple admission control policy might be to apply a fixed threshold to each stage's event queue; unfortunately, it is very difficult to determine what the ideal thresholds should be to meet some performance target.

A better approach is for stages to monitor their performance (for example, response-time distributions) and trigger rejection of incoming events when some performance threshold has been exceeded. Alternately, an admission controller could assign a cost to each event in the system, and prioritize low-cost events (e.g., inexpensive static Web page requests) over high-cost events (e.g., expensive dynamic pages). SEDA allows the admission control policy to be tailored for each individual stage, and admission control can be disabled for any stage.

A fundamental property of SEDA service design is that stages must be prepared to deal with enqueue rejection. Rejection of events from a queue indicates that the corresponding stage is overloaded, and the service should use this information to adapt. This explicit indication of overload differs from traditional service designs that treat overload as an exceptional case for which applications are given little indication or control. In SEDA, overload management is a first-class primitive in the programming model.

Rejection of an event from a queue does not imply that the user's request is rejected from the system. Rather, it is the responsibility of the stage receiving a queue rejection to perform some alternate action. This action depends greatly on the service logic. For example, if a static Web page request is rejected, it is usually sufficient to send an error message to the client indicating that the service is overloaded. However, if the request is for a complex operation such as executing a stock trade, it may be necessary to respond in other ways, such as by transparently re-trying the request at a later time. More generally, queue rejection can be used as a signal to *degrade service*, by performing variants of a service that require fewer resources.

We defer a detailed discussion of overload control in SEDA to Chapter 7. There, we present mechanisms that control the 90th-percentile response time exhibited by requests through the system. We also present a technique for class-based service differentiation that drops lower-priority requests over higher-priority requests, and demonstrate the use of service degradation.

6 Design patterns for structured service design

Our initial investigation into the spectrum of concurrency models from Chapter 2 indicates the need for systematic techniques for mapping an application design onto an implementation that provides efficiency and robustness despite the inherent limits of the underlying operating system. Here, we propose a set of *design patterns* [46] that may be applied when constructing SEDA-based services. A design pattern is a specification for a recurring solution to a standard problem [115]. The process of mapping complex Internet services onto the SEDA model can be represented as a set of design patterns to achieve certain performance and robustness goals.

6.1 Definitions

We define a *task* as the fundamental unit of work to be performed by a service in response to a given client request. Abstractly, a task consists of a set of processing *steps* that may involve computation, I/O, network communication, and so forth. For example, a Web server responding to a request to retrieve a static HTML page must first parse the request URL, look up the page in the local cache, read the page from disk (if necessary), and send a formatted response to the client. The steps of a task can either be executed in sequence or in parallel, or a combination of the two. By decomposing a task into a series of steps, it is possible to distribute those steps over multiple physical resources, and reason about the control flow of each task for load-balancing and fault-isolation purposes.

We begin with a simple formulation of a service in which each request is processed by a separate thread. This is the standard approach to service design, as discussed in Chapter 2. Our goal in this section is to show how to map the *thread-per-task* processing model onto a scalable, robust service using the SEDA framework.

6.2 The *Wrap* pattern

The *Wrap* pattern places a queue in front of a set of threads performing task processing, thereby "wrapping" the task processing logic into a SEDA stage. Each thread processes a single task through some number of steps, and may block. The *Wrap* operation

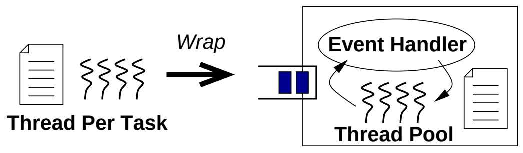


Figure 10: **The *Wrap* design pattern.**

makes the resulting stage robust to load, as the number of threads inside of the stage can now be fixed at a value that prevents thread overhead from degrading performance, and additional tasks that cannot be serviced by these threads will accumulate in the queue.

Wrap is the basic stage construction primitive in our design framework. The simplest SEDA-based server is one that consists of a single stage with a bounded thread pool and request queue. *Wrap* is identical to thread pooling as described in Chapter 2; a typical example is the Apache [8] Web server, which uses a static pool of UNIX processes to perform request processing. In Apache, the request queue is implemented using the *accept()* system call; as discussed previously, this can lead to dropped TCP connections if the listen queue fills up.

Wrapping the entire service logic in a single stage is typically too coarse-grained of an approach to effect adequate performance and load conditioning properties. This is because such an approach only exposes control to two variables: the number of threads in the stage, and the admission control policy over the single request queue. To carefully manage resource bottlenecks within the service, it is often desirable to control the flow of requests *within* a service, for example, by identifying those requests which cause performance degradation and perform selective load shedding upon them. As discussed in Chapter 2, a single thread pool and request queue does not yield enough "external" control over the behavior of the service logic. Decomposing a service into multiple stages permits independent resource provisioning and load conditioning for each stage. This is the purpose of the remaining design patterns.

6.3 The *Pipeline* and *Partition* patterns

The *Pipeline* pattern takes a single stage and converts it into a chain of multiple stages in series, introducing event queues between them. Abstractly, if the task processing within the original stage consists of processing steps S_1, S_2, and S_3, *Pipeline* can be used to convert this into three stages each performing a single step.

The *Partition* pattern takes a single stage and converts it into a tree of multiple stages in parallel. This transformation is generally applied at a branching point in the stage's task processing; if a stage performs processing steps S_1 followed by a branch to either S_2 or S_3, then *Partition* converts this into a tree of three stages, where S_1 feeds events to stages S_2 and S_3.

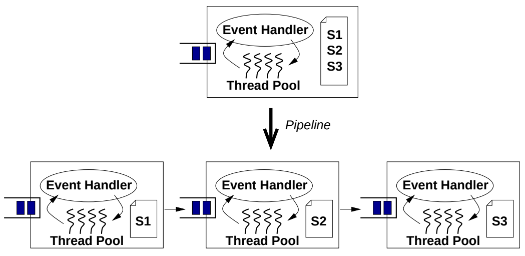


Figure 11: **The *Pipeline* design pattern.**

The *Pipeline* and *Partition* patterns have a number of uses. First, they can be used to isolate blocking or resource-intensive code into its own stage, allowing it to be independently provisioned and conditioned to load. Breaking the processing for a task into separate stages allows the size of the thread pool to be tailored for that stage, and allows that stage to be replicated across separate physical resources (as we will see below) to achieve greater parallelism. This approach also allows admission control decisions to be made based on the resource needs of each particular stage.

Consider a use of *Partition* that converts a stage performing the logical operations

1. Do task S_1;
2. If some condition holds, do task S_2;
3. Otherwise, do task S_3.

into three stages, as shown in Figure 12. Say that S_2 performs a blocking operation that takes a long time to complete, and that for some request load becomes a bottleneck requiring the use of admission control. Without applying *Partition*, it would be necessary to apply admission control on the single stage performing steps S_1, S_2, S_3, without any information on which requests actually performed step S_2. However, by placing this code in its own stage, requests that flow only through stages S_1 and S_3 are unaffected; admission control can be performed on S_2 separately from the rest of the service.

Pipeline and *Partition* can increase performance by causing the execution of service-processing code to be "batched" for greater cache locality. As the performance gap between cache and main memory increases, improving cache locality may yield important performance gains. The impact of cache locality on server performance was the chief motivation for the StagedServer [80] design, which resembles SEDA in some respects. In a thread-per-task system, the instruction cache tends to take many misses as the thread's control passes through many unrelated code modules to process the task. When a context switch occurs (e.g., due to thread preemption or a blocking I/O call), other threads will invariably flush the waiting thread's state out of the cache. When the original thread resumes execution, it will need to take many cache misses in order to bring its code and state back into the cache. In this situation, all of the threads in the system are competing for limited cache space.

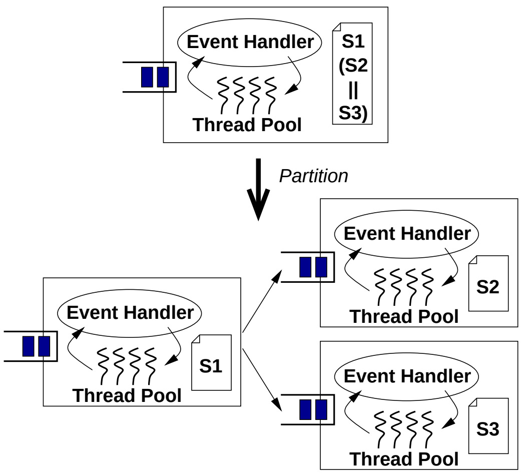


Figure 12: **The *Partition* design pattern.**

Applying the *Pipeline* and *Partition* patterns can increase data and instruction cache locality to avoid this performance hit. Each stage can process a convoy of tasks at once, keeping the instruction cache warm with its own code, and the data cache warm with any shared data used to process the convoy. In addition, each stage has the opportunity to service incoming tasks in an order that optimizes for cache locality. For example, if queues are serviced in last-in-first-out order, as proposed by [80], then the tasks that arrived most recently may still be in the data cache.

Finally, isolating code into its own stage facilitates modularity and independent code development. Because stages communicate through an explicit task-passing interface, it is straightforward to experiment with new implementations of a stage, or to introduce a proxy stage between two other stages. In contrast to the thread-per-task model, in which a single thread may invoke many code modules to process a task, isolating service components within separate stages isolates performance bottlenecks or failures in a given stage. Decomposition also has the benefit of greatly reducing code complexity, which is often an issue for monolithic event-driven systems. The fact that each stage uses its own thread pool to drive execution allows the underlying thread system to make scheduling decisions across a set of stages, rather than having to implement an application-specific scheduler within a monolithic event loop.

6.4 The *Combine* pattern

The *Combine* pattern combines two stages into a single stage. It is the inverse of the *Pipeline* and *Partition* stages, and is used to aggregate the processing of separate stages. One benefit of this pattern is to reduce code complexity. Consider a set of three sequential stages (S_1, S_2, and S_3), resulting from the use of *Pipeline*, as shown in Figure 13. If S_1 and S_3 perform closely related actions, such as the dispatch and completion of an asynchronous operation to S_2, then it makes sense to combine the logic for these stages into a single stage. Also, combining stages conserves threads, as multiple components share a single thread pool. For example, if S_1 and S_3 are CPU bound, both stages need no more than one thread per CPU; there is no value in allocating a separate pool to each stage. *Combine* has benefits for locality as well. If S_1 and S_3 share common data structures or code, then combining their processing into a single stage

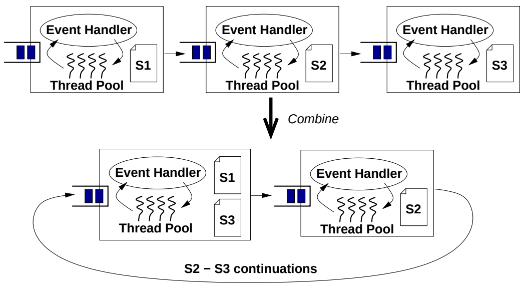


Figure 13: **The *Combine* design pattern.**

may improve cache locality, leading to higher performance.

6.5 The *Replicate* pattern

The *Replicate* pattern replicates a given stage, introducing a failure boundary between the two copies, possibly by instantiating the new stage on a separate set of physical resources. Whereas *Pipeline* and *Partition* are used to achieve functional decomposition of a stage, *Replicate* is used to achieve parallelism and fault isolation. The canonical use of *Replicate* is to distribute stages in a SEDA-based application across a set of physical nodes, for example, in a workstation cluster. This allows the processing capacity of the service to scale with the number of nodes in the cluster, and increases fault tolerance by allowing stage replicas to fail independently on separate nodes.

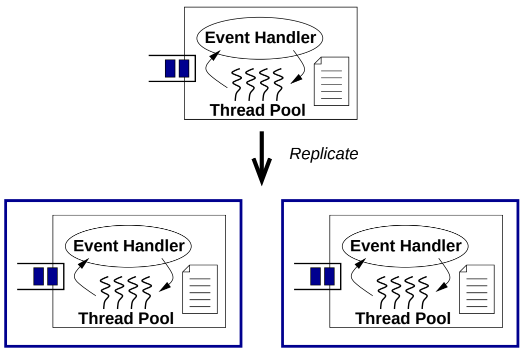


Figure 14: **The *Replicate* design pattern.**

By replicating a stage across physical resources, the combined processing capability of the replicas is increased; this can be used to eliminate a bottleneck in the system by devoting more resources to the replicated stage. For example, if a stage is CPU bound and is unable to deliver expected performance with available resources, *Replicate* can be used to harness additional CPUs across multiple nodes of a cluster. Replication can also be used to introduce a failure boundary between copies of a stage, either by running them on separate machines, or even within different address spaces on the same machine.

Stage replication raises concerns about distributed state management. The failure of a network link within a cluster can lead to partitioning, which is troublesome if stages residing on different cluster nodes need to maintain consistent state. There are several ways to avoid this problem. One is to employ one of various distributed consistency or group membership protocols [101, 136].

Another is to engineer the cluster interconnect to eliminate partitioning. This is the approach taken by DDS [52] and the Inktomi search engine [63].

Although we include replication of stages in our design framework, this dissertation focuses on the properties of SEDA as they pertain to a single-node Internet service. However, it is relatively straightforward to replicate a SEDA-based service, for example, by implementing event queues using network connections between cluster nodes. This is the approach taken by the Ninja *vSpace* system [139], which makes use of the SEDA design in a cluster environment.

7 Additional design principles

Apart from the design patterns presented above, there are several other principles that should be considered when engineering an Internet service under the SEDA design framework.

Avoiding data sharing: The data associated with tasks within the service should be passed by value, rather than by reference, whenever possible. Data sharing between two stages raises a number of concerns. Consistency of shared data must be maintained using locks or a similar mechanism; locks can lead to race conditions and long acquisition wait-times when contended for, which in turn limits concurrency. Also, passing data by reference is problematic when two stages are located in different addresses spaces or machines. Although Distributed Shared Memory (DSM) [1] can be used to make cross-address-space sharing transparent, DSM mechanisms are complex and raise concurrency concerns of their own. Data sharing also requires stages to agree upon who is responsible for deallocating data once it is no longer used. In a garbage-collected environment (within a single address space) this is straightforward; without garbage collection, more explicit coordination is required. Perhaps most importantly, data sharing reduces fault isolation. If a stage fails and leaves shared data in an inconsistent state, any other stages sharing that data must be able to recover from this situation or risk failure themselves.

An alternative to passing by value is to pass by reference with the originator relinquishing access. Another means of reducing data sharing is to space-partition application state, in which multiple stages process their own private partition of the application state, rather than sharing state and using locks to maintain consistency.

Stateless task processing: A *stateless* stage is one that maintains no internal state that needs to be maintained across the processing of multiple tasks. Note that a stateless stage may still make use of *per-task state*, for example, state carried in the data structure representing a request. Although stateless processing may be difficult to achieve in practice, it has several benefits. A stateless stage can be lock-free, as no state is shared between threads within the stage or other stages. In addition, a stateless stage can be easily created or restarted on demand, for example, in response to a load spike or failure.

Avoiding fate sharing: In a clustered environment where disjoint physical resources are available, it is important to avoid *fate sharing* of different stages. If two stages share physical resources, they also share their fate: that is, if the physical resources fail (e.g., the cluster node crashes), both stages will also fail. To avoid fate sharing, replicas of the same stage should be kept on separate physical nodes, or at least in separate address spaces on the same node.

Another form of fate sharing is *load sharing*, in which the performance of two stages are subject to dependent load conditions. For example, if one replica of a stage fails, other replicas will be require to take on its load in order to maintain throughput. Likewise, the resource consumption of one stage on a node can affect the resources available to other stages on that node. SEDA attempts to mitigate this issue by applying load conditioning and admission control to each stage independently, with the goal of avoiding resource starvation.

Well-formed structure: One should consider whether a service constructed using the SEDA framework is *well-formed*, that is, that it meets certain criteria that enable the system to function properly. Many such constraints are easy to state: for example, a stage with an empty thread pool will clearly not make progress. Two stages that are composed through a queue need to agree on the types of events that are produced and consumed by the participants.

In general, the structure of a SEDA application should not allow deadlock or starvation.

Deadlock can occur for various reasons. For example, if two stages share common resources or state, then the use of locks can lead to deadlock, unless the stages agree on a well-defined order for acquiring and releasing exclusive access. Cycles in a SEDA stage graph are generally unproblematic as long as certain properties are satisfied. For example, consider a stage cycle such as that shown in Figure 13, arising from the use of *Combine*. If the rightmost stage (S_2) becomes a bottleneck requiring admission control, and the leftmost stage (S_1/S_3) reacts to a queue rejection by blocking its threads, then requests flowing from S_2 to S_1/S_3 will back up in the stage graph.

Chapter 4

A Performance Model for SEDA-based Services

One of the benefits of SEDA is that it provides a structured framework for reasoning about the performance of complex Internet services. The core structure proposed by SEDA, a network of service components with associated request queues, has been studied extensively in the context of queue-theoretic performance models [14, 66, 76]. In many cases, these performance models rely on simplifying assumptions in order to yield a closed-form solution; a common assumption is that interarrival and service times are exponentially distributed. However, many of the "interesting" cases that arise in real systems, such as the effects of scheduling, resource contention, complex service time distributions, and so forth, are not amenable to direct formal analysis. Despite these simplifications, however, queueing networks provide a useful formal structure in which to analyze the behavior of a SEDA application.

In this chapter, we derive a simple performance model for SEDA based on a queueing network of load-dependent service centers. This model describes the performance aspects of SEDA in terms of external load, stage performance, thread behavior, and the structure of the stage network. Using this model we motivate the use of the SEDA design patterns described in the previous chapter, demonstrating the performance effects of each transformation on the structure of a simple service.

The performance model given here makes several simplifying assumptions, and real services no doubt exhibit dynamics that are not captured by it. Regardless, we expect that the performance model can provide a valuable approximation and intuition for the expected performance of a SEDA-based service. At the end of the chapter we discuss the limitations of the model and several potential extensions.

1 Introduction: Basic queueing models

In the terminology of queueing theory, a SEDA application can be modeled as a network of *service centers*, with each service center exhibiting some service time distribution S,

$$S(x) = \Pr[\text{service time} \leq x]$$

Requests (also called *customers* or *jobs*) arrive at each service center according to an arrival process characterized by an interarrival time distribution A,

$$A(t) = \Pr[\text{time between arrivals} \leq t]$$

The bulk of queueing theory related to networks of queues deals with service centers exhibiting exponential service and interarrival time distributions. These two assumptions permit a straightforward analysis of various aspects of the queueing system, such as the mean number of requests in the system, the waiting time and response time distributions, and so forth.

The exponential distribution is characterized by the probability density function

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

where λ is the *rate* of the associated distribution. The cumulative distribution function is given by

$$F(x) = \Pr[X \leq x] = \int_{-\infty}^{x} f(y)dy = \begin{cases} 1 - e^{-\lambda x} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

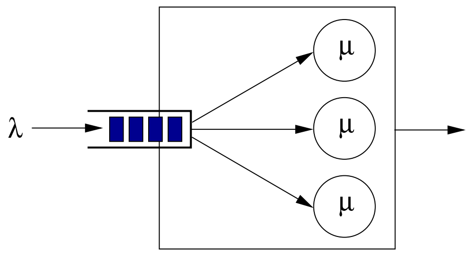


Figure 15: **M/M/m queueing system:** *Requests arrive at the system according to a Poisson arrival process with average rate λ, and m servers process each request with an exponentially distributed service time with mean $1/\mu$.*

The exponential distribution exhibits the *memoryless property* in that

$$\Pr[X > s + t \,|X > t] = Pr[X > s] \quad \forall s, t \geq 0$$

This property allows the sequence of arrivals to and departures from a service center to be modeled as a continuous-time Markov chain, which is the basis for many of the results in basic queueing theory.

The simplest formulation for a single SEDA stage is an M/M/m queueing system as shown in Figure 15. In this system, request interarrival times are exponentially distributed with mean $1/\lambda$, service times are exponentially distributed with mean $1/\mu$, and there are m *servers* within the service center available to process requests concurrently. In the SEDA model, m represents the number of threads within the stage. For the moment, let us assume that the processing capacity of a stage is linear in the number of threads; in Section 2.1 we discuss the load-dependent case in which stage performance depends on other factors.

The ratio $\rho = \lambda/(m\mu)$ is referred to as the *traffic intensity* of the system; when $\rho < 1$, the system is *stable* in that the steady-state waiting time of requests entering the system is finite. When $\rho \geq 1$, the system is *unstable* (or overloaded), and waiting times are unbounded. We define the *maximum stable arrival rate* as the largest value of λ for a given system such that $\rho < 1$.

Rather than reiterate the derivations for these fundamental results, we refer the reader to any text on basic queueing theory, such as [76]. The derivations of the theorems below can be found in [67], pages 519–534.

It can be shown that the probability of n requests in an M/M/m queueing system is

$$p_n = \begin{cases} p_0 \frac{(m\rho)^n}{n!} & n < m \\ p_0 \frac{\rho^n m^m}{m!} & n \geq m \end{cases} \tag{4.3}$$

where p_0 is the probability of the system being empty, given as

$$p_0 = \left[1 + \frac{(m\rho)^m}{m!(1-\rho)} + \sum_{n=1}^{m-1} \frac{(m\rho)^n}{n!} \right]^{-1} \tag{4.4}$$

Another important quantity is the probability that an incoming request must queue (that is, that all m servers are busy); this is given by Erlang's C formula,

$$\varrho = \Pr[\geq m \text{ jobs in system}] = p_0 \frac{(m\rho)^m}{m!(1-\rho)} \tag{4.5}$$

The mean number of requests in the system is denoted by the random variable N, where

$$E[N] = m\rho + \frac{\rho\varrho}{1-\rho} \tag{4.6}$$

with variance

$$Var[N] = m\rho + \rho\varrho \left[\frac{1+\rho-\rho\varrho}{(1-\rho)^2} + m \right] \tag{4.7}$$

The *response time* (time waiting in the queue plus time to complete service) exhibited by a request is denoted by the random variable R, with mean

$$E[R] = \frac{1}{\mu} \left(1 + \frac{\varrho}{m(1-\rho)} \right) \tag{4.8}$$

and cumulative distribution function

$$F(r) = \Pr[R \leq r] = \begin{cases} 1 - e^{-\mu r} - \frac{\varrho}{1-m+m\rho} e^{-m\mu(1-\rho)r} - e^{-\mu r} & \rho \neq (m-1)/m \\ 1 - e^{-\mu r} - \varrho\mu r e^{-\mu r} & \rho = (m-1)/m \end{cases} \tag{4.9}$$

Figure 16 shows the mean response time for three M/M/m queueing systems; as the arrival rate λ approaches the service rate μ, response times grow to infinity.

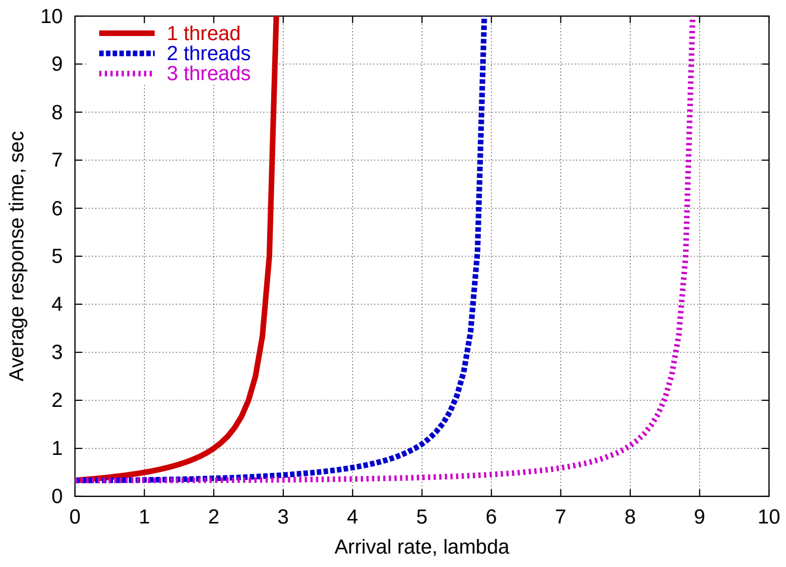


Figure 16: **Mean response time for an M/M/m queueing system:** *This graph shows the mean response time (time in queue plus service time) for requests entering an M/M/m queueing system with an average service rate $\mu = 3.0$, under a varying average arrival rate λ and 1, 2, or 3 servers. As the graph shows, the system becomes unstable as $\lambda \to m\mu$.*

It is important to note that these results hold in steady state; that is, when λ and μ are constant as $t \to \infty$. In a real system, both the arrival rate and service rate will vary considerably over time. Unfortunately, these basic results do not address the transient behavior of the system under heavy load.

2 Modeling resource contention and thread limitations

While the M/M/m queueing system is simple to analyze, it has several drawbacks. Modeling each thread within a stage as a separate server fails to capture resource contention that may occur between threads, either within the same stage or in different stages. As discussed in Chapter 2, most threading systems exhibit performance degradation as the number of threads in the system grows; under the simplistic model here, the best way to meet performance targets for SEDA is to simply allocate more threads! In this section we extend the basic model to address resource contention and the inherent performance overheads of threads.

2.1 Load-dependent service centers

.

A *load-dependent service center* is a queueing system in which the average service rate μ depends upon the number of requests in the system. In general μ can be any function of the number of customers n; for M/M/m, service times are exponentially distributed with mean

$$\mu(n) = \begin{cases} n\mu & \text{if } n < m \\ m\mu & \text{otherwise} \end{cases}$$

Where μ is the base service rate of the service center.

Rather than attempt to express all of the possible resource contention scenarios that may affect a stage, we choose to limit the our model to capturing performance degradation as the total number of threads in the system increases. First we consider a single stage in isolation; in Section 3 we extend the model to a network of stages.

We expect that for a given stage, there is some *inherent parallelism* that can be exploited by allocating some number of threads to the stage. Beyond a certain limit, adding additional threads does not yield higher performance, and in fact may incur a performance penalty due to thread overhead. Leaving aside the number of requests in the system for the moment, we define μ_m as the base service rate for a stage with m threads.

$$\mu_m = \begin{cases} \max(0, m\mu - \phi(m_t)) & \text{if } m < \alpha \\ \max(0, \alpha\mu - \phi(m_t)) & \text{if } m \geq \alpha \end{cases} \tag{4.10}$$

α represents the “saturation point” of the stage, beyond which all of the inherent parallelism has been exploited. For example, if a stage is strictly CPU-bound, then α is equal to the number of CPUs in the system. $\phi(m_t)$ is the performance penalty incurred for m_t threads, where m_t is the total number of threads in the system. Given N stages where the ith stage has m_i threads allocated

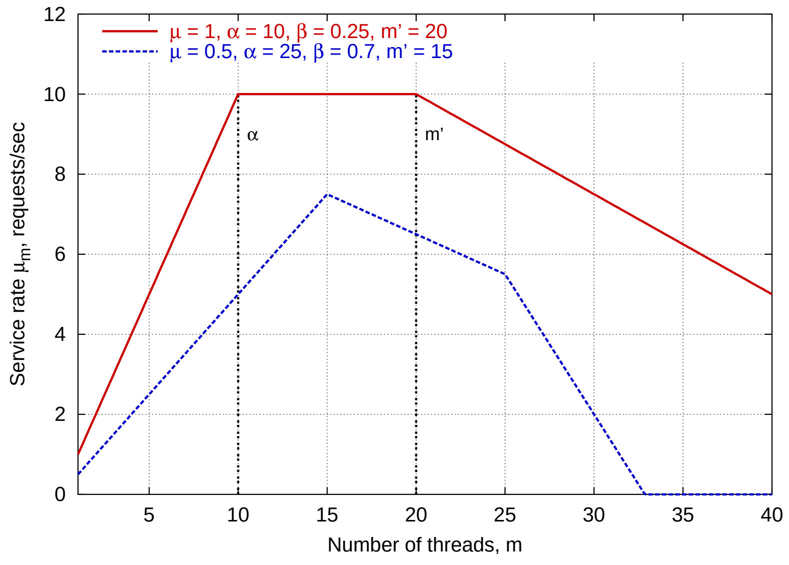


Figure 17: **Load-dependent servicing rate as a function of the number of threads:** *This graph shows sample plots for the base service rate μ_m for two different settings of the parameters α, β, and m'. In the upper curve, as m increases, performance increases until the natural parallelism limit α is reached, and degrades once m' has been exceeded. In the lower curve, $m' < \alpha$, so performance degradation begins earlier, although additional threads still benefit performance until α has been reached.*

to it,

$$m_t = \sum_{i=0}^{N} m_i$$

The effective processing rate is bounded to prevent the performance penalty from driving the effective processing rate below 0.

A range of penalty functions ϕ could be considered. Suppose that beyond some number of threads m', there is a linear performance penalty β incurred for each thread:

$$\phi(m) = \begin{cases} 0 & \text{if } m < m' \\ \beta(m - m') & \text{if } m \geq m' \end{cases} \tag{4.11}$$

A plot of μ_m for representative values of α, β, and m' is shown in Figure 17.

The actual service rate exhibited by a stage is dependent both upon n, the number of pending requests, and m, the number of threads in the stage. Let us define $\mu_m(n)$ as the service rate for a stage with m threads and n pending requests. For given values of m and n, then, there are four cases to consider when request enters the system:

$$\mu_m(n) = \begin{cases} \max(0, n\mu - \phi(m_t)) & \text{if } m < \alpha \text{ and } n < m \\ \max(0, m\mu - \phi(m_t)) & \text{if } m < \alpha \text{ and } n \geq m \\ \max(0, n\mu - \phi(m_t)) & \text{if } m \geq \alpha \text{ and } n < \alpha \\ \max(0, \alpha\mu - \phi(m_t)) & \text{if } m \geq \alpha \text{ and } n \geq \alpha \end{cases} \tag{4.12}$$

The application of $\phi(m_t)$ represents the fact that performance degradation occurs regardless of whether all threads are in use.

2.2 Solving the load-dependent service center

We wish to find the limiting probability $p_n = \lim_{t\to\infty} P_n(t)$ of there being n requests within an M/M/m load-dependent service center with average service rate $\mu(n)$ as given by Eq. (4.12). There are various ways to solve for the limiting probabilities p_n; we adopt Harchol-Balter's method of solving for the limiting probabilities in a continuous-time Markov chain (CTMC), each state of which represents the number of requests in the system [56]. In this technique, the CTMC is modeled as a discrete-time Markov chain, where we allow the time quantum δ to approach 0. It can be shown that the resulting CTMC is ergodic for $\rho < 1$, which establishes the existence of the limiting probabilities.

To solve for p_n, we begin with the ergodicity condition that the rate at which the system enters state j is equal to the rate at which the system leaves state j. Thus:

$$\begin{aligned} \lambda p_0 &= \mu(1)p_1 \\ \lambda p_1 &= \mu(2)p_2 \end{aligned}$$

and so forth. Rewriting in terms of the p_i's,

$$\begin{aligned} p_1 &= \frac{\lambda}{\mu(1)} p_0 \\ p_2 &= \frac{\lambda}{\mu(2)} p_1 = \frac{\lambda^2}{\mu(1)\mu(2)} p_0 \end{aligned}$$

In general,

$$p_n = \frac{\lambda^n}{\prod_{j=1}^{n} \mu(j)} p_0$$

We solve for p_0 using the condition that the probabilities must all sum to 1:

$$p_0 + \sum_{i \neq 0} p_i = 1$$

to obtain

$$p_0 = \frac{1}{1 + \sum_{i=1}^{\infty} \frac{\lambda^i}{\prod_{j=1}^{i} \mu(j)}} \tag{4.13}$$

From these probabilities the mean number of jobs in the system can be determined using

$$E[N] = \sum_{n=0}^{\infty} n p_n \tag{4.14}$$

and the average response time from Little's Law,

$$E[R] = \frac{E[N]}{\lambda} \tag{4.15}$$

It is difficult to obtain a closed-form solution for these expressions when $\mu(n)$ is complicated, as is the case here. Although these expressions can be solved numerically, we chose to rely on simulation results, which are more readily obtained. The simulation models a load-dependent M/M/m system with $\mu(n)$ as given in Eq. (4.12).

Figure 18 shows the mean response time of a single load-dependent service center as the arrival rate λ and number of threads varies. As the figure shows, with a small number of threads the system is underprovisioned, and with a large number of threads the thread system overhead ϕ dominates.

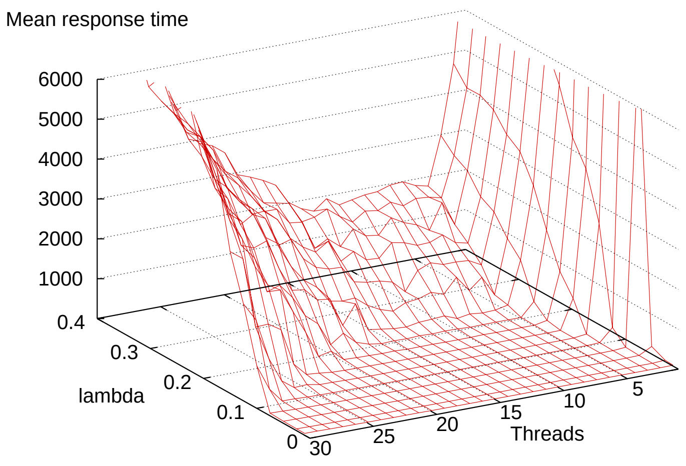


Figure 18: **Mean response time of a load-dependent M/M/m queueing system:** *This graph shows the mean response time as a function of the mean arrival rate λ and number of threads for a simulated load-dependent M/M/m queueing system with $\mu = 0.05$, $\alpha = 4$, $m' = 15$, and $\beta = 0.01$. With a small number of threads, the system is underprovisioned so response time is large. With a large number of threads, the overhead of the threading system ϕ dominates, driving response time up.*

3 Open Jackson networks of load-dependent service centers

Having established results for a single stage in isolation, we are now interested in the behavior of a SEDA application consisting of multiple stages. Such a system can be modeled as a queueing network of load-dependent service centers, essential results for which have been established by Jackson [66] and Baskett *et al.* [14]. In this model, shown in Figure 19, the system consists of N service centers. Each service center has a load-dependent, exponentially-distributed service time distribution as given by Eq. (4.12). Stage i has m_i threads. For convenience, we denote the average service rate of the ith stage simply as μ_i, dropping the m_i subscript; it is implied that the average service rate of the stage depends upon m_i according to Eq. (4.12).

Arrivals are Poisson, and requests may arrive into the system at any of the N service centers; r_i is the average rate at which requests arrive at server i. Requests are routed through

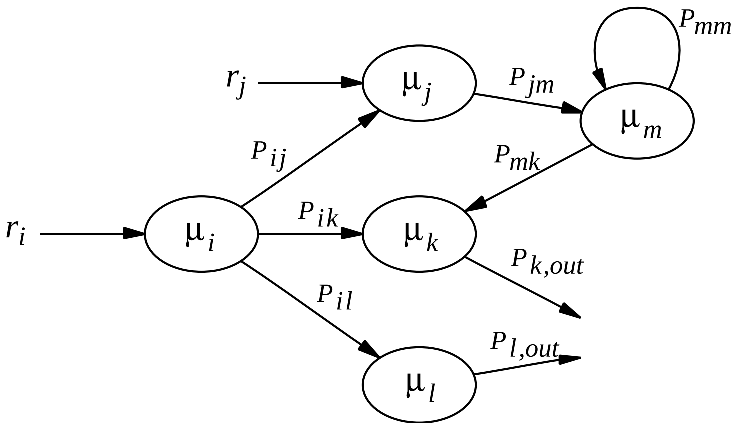


Figure 19: **Open Jackson queueing network:** *An example Jackson queueing network consisting of five service centers with respective exponential service time distributions μ_i, μ_j, μ_k, μ_l, and μ_m. Poisson job arrivals enter the system at service centers i and j with respective average rates r_i and r_j. P_{ij} represents the probability of a request being routed between service centers i and j; $P_{i,out}$ is the probability of a job leaving the system after being processed at service center i.*

the network in a probabilistic fashion; for each job that completes service at server i, it is routed to server j with probability P_{ij}, or exits the system with probability $P_{i,out}$. The connectivity in a typical queueing network is sparse, so many of the P_{ij} values are 0. Here we study *open* networks in which external arrivals and departures are allowed; this is in contrast to *closed* networks in which customers circulate entirely within the network.

First we consider a Jackson network of M/M/m service centers, where m_i represents the number of servers at service center i. The total arrival rate into each server is the sum of the external arrival rate plus the internal arrival rate:

$$\lambda_i = r_i + \sum_{j=1}^{k} \lambda_j P_{ji} \tag{4.16}$$

As with the M/M/m case, for the corresponding Markov chain to be ergodic, we require that $\lambda_i < m_i\mu_i$ for all i. We denote the state of the queueing network as $(n_1, n_2, \ldots, n_N)$ where n_i represents the number of requests (including those in service) at the ith service center. $p(n_1, n_2, \ldots, n_N)$ is the limiting probability of the system being in state $(n_1, n_2, \ldots, n_N)$. Jackson [66] shows that

$$p(n_1, n_2, \ldots, n_N) = p_1(n_1)p_2(n_2)\cdots p_N(n_N) \tag{4.17}$$

where each $p_i(n_i)$ is the equilibrium probability of finding n_i requests at service center i, the solution to which is identical to that of the M/M/m system. In other words, the Jackson network acts as a set of independent M/M/m queueing systems with Poisson arrival rates λ_i as given by Eq. (4.16). Note that the actual arrival process to each service center is not Poisson (due to request routing between service centers), however, the queueing network behaves as though they are. We say that Jackson networks exhibit a *product-form solution*: the limiting probability of the system being in state $(n_1, n_2, \ldots, n_N)$ equals the product of the individual limiting probabilities of there being n_1 requests at server 1, n_2 requests at server 2, and so forth.

A Jackson network of load-dependent service centers also exhibits a product-form solution. This can be shown by exploiting *local balance* within the structure of the queueing network; the results in this section come from Harchol-Balter's course notes (Solutions to Homework 5) [56].

To demonstrate local balance, we solve for $p(n_1, n_2, \ldots, n_N)$ using two conditions:

(1) The rate at which the system leaves state z due to a departure from server i is equal to the rate at which the system enters state z due to an arrival at server i:

$$\begin{aligned}
&p(n_1, n_2, \ldots, n_N)\mu_i(n_i) = \\
&\quad \sum_{j=1}^{k} p(n_1, \ldots n_i - 1, \ldots n_j + 1, \ldots n_N)\mu_j(n_j + 1)P_{ji} + \\
&\quad p(n_1, \ldots n_i - 1, \ldots n_N)
\end{aligned}$$

(2) The rate at which the system leaves state z due to an arrival from the outside is equal to the rate at which the system enters state z due to a departure to the outside:

$$\begin{aligned}
&p(n_1, n_2, \ldots, n_N) \sum_{i=1}^{k} r_i = \\
&\quad \sum_{i=1}^{k} p(n_1, \ldots n_i + 1, \ldots n_N)\mu_i(n_i + 1)P_{i,out}
\end{aligned}$$

The solution comes from [56] and is given as

$$p(n_1, n_2, \ldots, n_N) = C \cdot \frac{\lambda_1^{n_1}}{\prod_{j=1}^{n_1} \mu_1(j)} \cdot \frac{\lambda_2^{n_2}}{\prod_{j=1}^{n_2} \mu_2(j)} \cdots \frac{\lambda_N^{n_N}}{\prod_{j=1}^{n_N} \mu_N(j)} \tag{4.18}$$

where C is a normalizing constant used to cause

$$\sum p(n_1, n_2, \ldots, n_N) = 1$$

given as

$$C = \frac{1}{\sum_{n_1 \ldots n_N} \frac{\lambda_1^{n_1}}{\prod_{j=1}^{n_1} \mu_1(j)} \cdots \frac{\lambda_N^{n_N}}{\prod_{j=1}^{n_N} \mu_N(j)}} \tag{4.19}$$

The probability of n_i jobs at server i is

$$p_i(n_i) = \sum_{n_1=0}^{\infty} \cdots \sum_{n_{i-1}=0}^{\infty} \sum_{n_{i+1}=0}^{\infty} \cdots \sum_{n_N=0}^{\infty} p(n_1, n_2, \ldots, n_N) \tag{4.20}$$

that is, the equilibrium probabilities where all n_j ($j \neq i$) are summed from 0 to ∞. Algebraic manipulation yields

$$p_i(n_i) = \frac{\frac{\lambda_i^{n_i}}{\prod_{j=1}^{n_i} \mu_i(j)}}{\sum_{n_i=0}^{\infty} \frac{\lambda_i^{n_i}}{\prod_{j=1}^{n_i} \mu_i(j)}} \tag{4.21}$$

4 Examples

In this section we make use of the SEDA performance model and simulation results to demonstrate the performance impact of the SEDA design patterns described in Chapter 3. These examples are necessarily simple and serve to illustrate the essential features of the SEDA performance model, rather than attempt to capture the full complexity of a real system.

As with Eq. (4.12), there is no closed-form solution to Eq. (4.21) when $\mu(n)$ is sufficiently complicated. Our initial results were initially obtained using numerical analysis, though later we relied on simulation, which is more flexible and produces results more quickly. Our simulator models a Jackson network of stages with load-dependent behavior as described by Eq. (4.12). We have verified that the simulation and numerical analysis results agree.

4.1 The *Combine* and *Pipeline* patterns

Consider a simple SEDA application where each request must have two operations, A and B, performed on it. Let A and B each be characterized by an exponential service time distribution

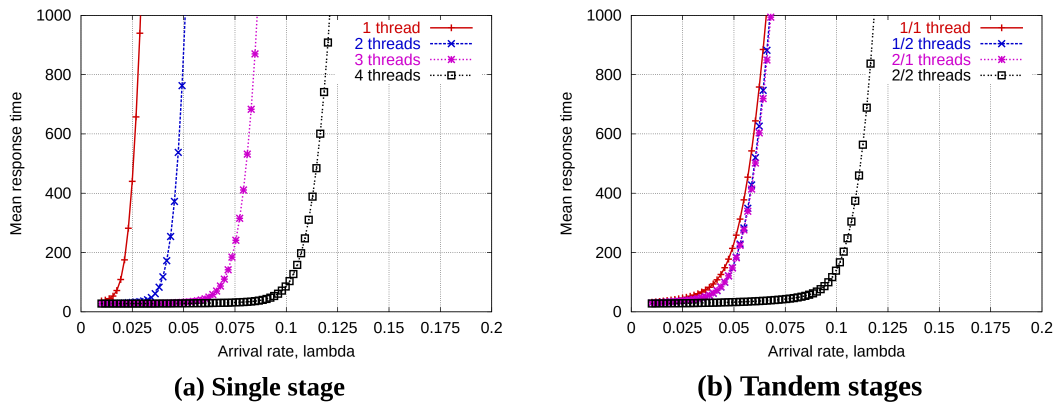


Figure 20: **Effect of stage pipelining:** *This figure shows the response time as a function of the arrival rate λ for two servers: (a) a single M/E$_r$/m server where the service time is the sum of two exponentially distributed steps with $\mu = 0.07$; and (b) two M/M/m stages in tandem where each stage has an exponentially distributed service time with $\mu = 0.07$. In (a), the number of threads in the stage ranges from 1 to 4; in (b), the number of threads in each stage ranges from 1 to 2. As the figure shows, adding threads to the M/E$_r$/m server scales the peak rate λ, while in the M/M/m case, only when both stages are well-provisioned does the pipeline avoid overload.*

with mean μ_A and μ_B, respectively. First, consider a single stage performing both operations A and B for each request. If $\mu_A = \mu_B = \mu$, the resulting service time distribution is an r-stage Erlangian distribution with $r = 2$ and rate μ,

$$f(x) = \frac{r\mu(r\mu x)^{r-1}e^{-r\mu x}}{(r-1)!} = 4\mu^2 x e^{-2\mu x} \tag{4.22}$$

Taking into account the SEDA performance model, this application can be described as a load-dependent M/E_2/m queueing system. A single stage processing both steps is representative of the *Combine* design pattern from the previous chapter.

Second, consider a different implementation of the service consisting of two stages in tandem, where the first stage performs step A and the second performs B. In this case we have a simple Jackson network of two load-dependent M/M/m servers. This is an example of the *Pipeline* design pattern at work.

We are interested in studying the behavior of these two systems as a function of the (Poisson) request arrival rate λ as well as the number of threads in each stage. Here, let us assume that

α and m' are sufficiently large such that thread overhead can be neglected. Figure 20 shows the mean response times for each server with $\mu_A = \mu_B = 0.07$, and an increasing number of threads per stage. In the case of *Combine*, adding threads to the stage scales the maximum stable arrival rate that the system can handle. In the case of *Pipeline*, the peak stable arrival rate is bounded by the minimum processing capacity of both stages, as we would expect. With only a single thread in either of the two stages, the maximum stable arrival rate is $\lambda = \mu$; with two threads in each stage it is 2μ.

This result might indicate that there is no benefit to pipelining stages, or that all service logic should be contained in a single stage. However, in a realistic configuration, the number of threads required to provision both pipelined stages for high performance is small; as the figure shows, 4 threads split between two stages yields identical performance to 4 threads in a single stage. Also, there are other benefits to using *Pipeline* that are not captured by this model, such as increasing cache locality, increased code modularity, and better isolation of resource bottlenecks. These results do point out that whenever *Pipeline* is used, it is important to adequately provision the resulting stages.

4.2 The *Partition* pattern

Next we consider the use of the *Partition* pattern, which converts a branch within a stage's processing into a tree of stages. In this example, each request performs step A ($\mu_A = 0.9$) followed by either a "slow" processing step B ($\mu_B = 0.02$) or a "fast" step C ($\mu_C = 0.07$). 50% of the requests require processing by the bottleneck task B. Figure 21 shows the mean response time for two versions of this server: one in which all tasks are performed by a single stage, and another in which requests are processed by a tree of three stages, with A at the root and B and C at the leaves.

As the figure shows, adding threads to the single-stage server increases the maximum stable arrival rate λ. In the case of the partitioned server, it suffices to add threads only to the "slow" stage B, up to a certain point when "fast" stage becomes the bottleneck. With 4 threads allocated to

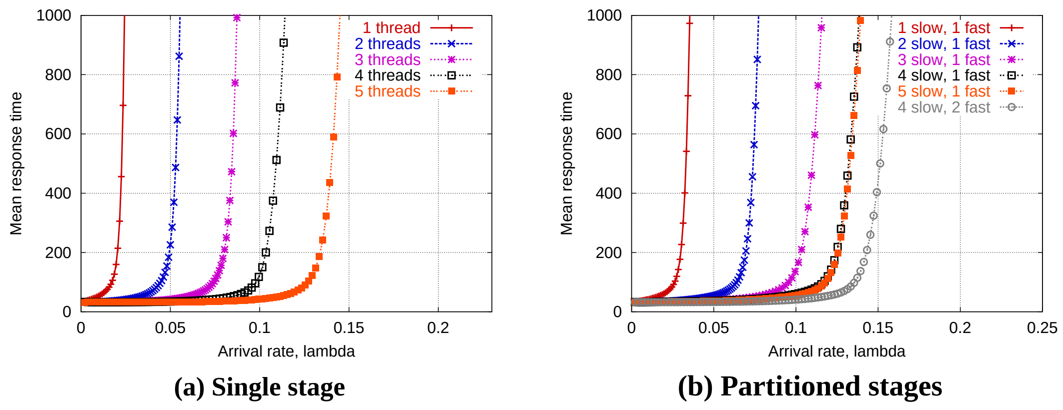


Figure 21: **Effect of stage partitioning:** *This figure shows the response time as a function of the arrival rate λ for two servers. (a) shows a single server where each request requires initial processing ($\mu_A = 0.9$), followed by either a "slow" (bottleneck) step ($\mu_B = 0.02$) or a "fast" step ($\mu_C = 0.07$). 50% of the requests require processing at the bottleneck. (b) shows the effect of the* Partition *pattern, in which request processing is performed by a tree of three stages, with initial processing at the root; the "slow" and "fast" steps are performed in separate stages at the leaves. In (a), adding threads to the stage increases the maximum stable arrival rate λ, while in (b), it suffices to add threads only to "slow" stage, up to a certain point when the "fast" stage becomes the bottleneck.*

B and 1 thread to C, the maximum stable arrival rate is

$$\lambda_{\max} = \min(2(4\mu_B), 2\mu_C) = \min(0.16, 0.14) = 0.14$$

In other words, stage C is the bottleneck in this case ($2\mu_C = 0.14$). Adding an additional thread to C yields

$$\lambda_{\max} = \min(2(4\mu_B), 2(2\mu_C)) = \min(0.16, 0.28) = 0.16$$

making B the bottleneck once again ($8\mu_B = 0.16$). This analysis shows that *Partition* allows stages to be independently provisioned for a given load, by allocating threads only to the stages where they are needed.

4.3 The *Replicate* pattern

The *Replicate* pattern is used to replicate a single stage across multiple physical resources, such as the nodes of a cluster. Replication allows performance bottlenecks to be overcome by

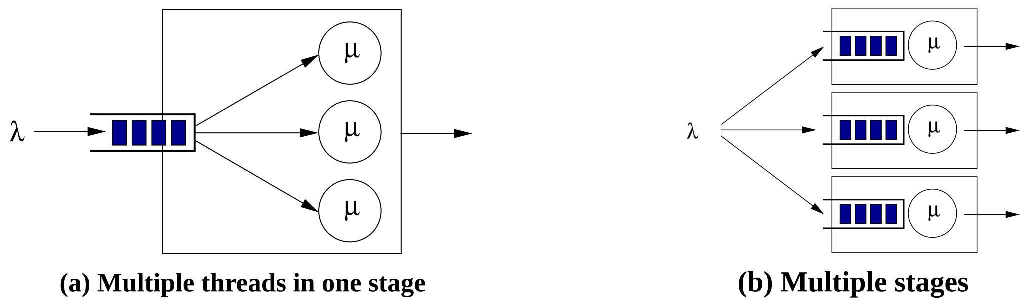


Figure 22: **Replicated queueing system:** *This figure shows two queueing systems with identical processing capacity but with different performance characteristics. (a) shows a single stage with multiple threads each processing requests from a single queue. (b) shows multiple single-threaded stages each with their own queue.*

harnessing additional resources within a service. Consider a case where an application has a single bottleneck stage that requires a large number of threads to meet offered load. If the thread limit m' is relatively low, it may be necessary to replicate the stage across multiple machines, rather than allocate additional threads to the single stage.

There is a disadvantage to replication, however: redirecting requests to separate queues can have a negative impact on response time compared to all requests waiting in the same queue. Figure 22 shows two queueing systems: one which consists of a stage with multiple threads processing requests from a single queue, and another which consists of multiple replicas of a single-threaded stage. Figure 23 illustrates the performance tradeoff between multithreading and replication as the number of threads or replicas varies. The replicated system exhibits higher response times as $\lambda \rightarrow \lambda_{\mathrm{max}}$, the maximum stable arrival rate.

This effect is due to the randomness of the exponential interarrival time and service time processes. Intuitively, allowing all requests to wait in the same queue for m servers allows requests to be directed to the next available server as soon as any server becomes available. Directing requests to per-server queues upon arrival gives requests less “flexibility” in terms of which server will process it.

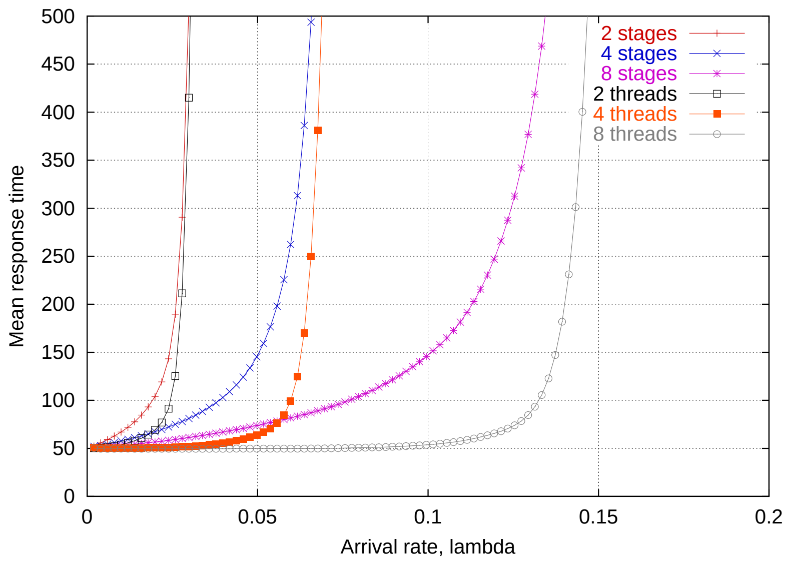


Figure 23: **Performance effect of replication:** *This figure shows the response time as a function of the arrival rate λ for two servers: (a) Two M/M/m stages in tandem, the first with an exponential service rate of $\mu_1 = 2.9$ and a single thread, and the second with an exponential service rate of $\mu_2 = 0.02$ and either 2, 4, or 8 threads; (b) A tree of stages consisting of a single stage at the root with μ_1 and a single thread as above, and with 2, or, or 8 stages at the leaves each with service rate μ_2 and a single thread. Requests have an equal probability of being routed to each of the leaf stages. As the figure shows, adding threads to a stage has roughly the same effect on the maximum stable arrival rate as replicating a single-threaded stage, though the replicated system exhibits higher response times under lighter load.*

5 Summary and limitations

In this chapter we have considered a formal performance model for SEDA-based services based on a queueing network of load-dependent service centers. Load dependence is characterized in terms of the number of pending requests, the number of threads, and the natural concurrency of each stage. While not completely descriptive of all of the factors that contribute to system performance, this model is still valuable as a tool for understanding the behavior of a SEDA application and the effect of using each of the design patterns for structuring services. For example, decomposing a stage using *Pipeline* has the benefit that in that the thread pool for each stage can be provisioned separately, though a system designer must ensure that each stage in the pipeline has enough threads to handle incoming load. Similarly, there is a clear performance tradeoff between replicating a stage and allocating additional resources to it, given the increased response times due to queueing effects in the replicated service.

The performance model and simulation results presented in this chapter have several inherent limitations. Our intent has been to develop the model as a guide to understanding the high-level performance aspects of the SEDA design, rather than to address the many details involved in modeling a complete system. Indeed, it is unlikely that existing queueing theory results are able to capture the full complexity of the SEDA platform as implemented in a realistic environment.

Our use of load-dependent Jackson networks assumes exponential interarrival time and service time distributions for each stage, which may not be realistic. Evidence suggests that characteristic service time distributions for Internet services are heavy-tailed [57], but few theoretical results exist for such systems. Use of alternative service time distributions would be straightforward in simulation, though not as readily amenable to analysis.

The model here treats each stage as an independent server and assumes that all resource contention across stages is characterized by the performance penalty ϕ as a function of the number of threads in the system. The model does not attempt to associate ϕ with specific forms of resource contention, such as competition for I/O, memory, or CPU resources. An interesting extension to the model would be to consider the effects of CPU scheduling and context-switch overhead on performance. BCMP networks [14] extend the product-form solution of Jackson to a broader class of networks, including those with round-robin and processor sharing service disciplines. However, BCMP networks do not address resource contention in general.

Finally, our model does not characterize the potential locality benefit obtained using the *Pipeline* and *Partition* design patterns. A straightforward extension of the model would allow the service rate $\mu(n)$ of each stage to scale as a function of the number of requests processed in each batch, and with the specificity of the stage's task processing code.

Chapter 5

Sandstorm: A SEDA-based Internet Services Prototype

To demonstrate the effectiveness of the SEDA architecture, we have implemented a prototype Internet services platform based on the SEDA design, called *Sandstorm*. Sandstorm makes the SEDA design concrete, providing a service construction framework that includes interfaces for implementing stages, connecting stages into a network, queue management, and admission control. Sandstorm implements a number of controllers that tune the resource allocation of each stage based on observed performance, and provides libraries for asynchronous socket and file I/O. In this chapter we detail the design and implementation of Sandstorm, discuss the application programming model that it provides, and provide an initial evaluation in terms of code size and basic performance numbers.

1 Sandstorm overview

Sandstorm is a Java-based runtime system based on the SEDA design. In some sense it can be considered an "operating system for services," although it is implemented entirely at user-level on top of a commodity OS. Sandstorm applications are constructed as a network of stages connected

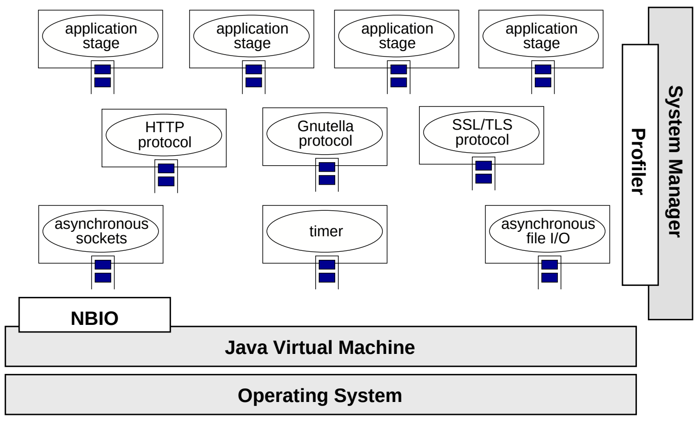


Figure 24: **Sandstorm architecture.** *Sandstorm is an implementation of SEDA in Java. The system provides a management interface for creating, destroying and configuring stages; a profiling and debugging interface; and several pre-built stages providing asynchronous I/O, timers, and various protocol libraries. Nonblocking socket I/O is provided with the* NBIO *library, which implements native code bindings to OS calls such as* poll(2) *and nonblocking sockets. Asynchronous file I/O is accomplished by wrapping blocking I/O calls in a stage.*

with explicit event queues; application developers provide the code for *event handlers* that embody the core event-processing logic within each stage. Sandstorm provides facilities for stage-graph construction, queue management, signals, and timers, as well as scalable, asynchronous network and disk I/O interfaces. In addition, Sandstorm provides built-in *resource controllers* that mitigate the resource demands that the application places on the system. In this way, a Sandstorm application consists of a set of (more or less) naive event handlers and the runtime system handles the details of load conditioning. The Sandstorm architecture is shown in Figure 24.

1.1 Use of the Java language

Sandstorm is implemented entirely in the Java programming language [51]. The decision to use Java instead of a “systems language” such as C or C++ was based on a desire to exploit the various features of the Java environment, including cross-platform portability, type safety, and automatic memory management. These features ultimately contribute to making services more robust: Java avoids a large class of common programming errors (including array bounds violations, type mismatching, and failure to reclaim memory).

Java has become a popular language for building Internet services, as exemplified by broad industry support for the Java 2 Enterprise Edition (J2EE) platform [123]. Until recently, the performance of Java fell far behind that of C and C++, primarily due to the lack of good compilation technology. However, modern Java Virtual Machines and Just-in-Time (JIT) compilers exhibit performance rivaling that of C. We believe that the safety and software-engineering benefits of Java outweigh the dwindling performance gap.

One serious drawback to using Java is the lack of support for nonblocking I/O, which is necessary to avoid the use of a thread per connection. Until the release of Java Development Kit (JDK) version 1.4 in early 2002, nonblocking I/O was not supported by the standard Java libraries. To overcome this limitation we implemented *NBIO*, a package that supports nonblocking socket I/O and several variants of event dispatching from the operating system [144]. NBIO is described in more detail in Section 5. JDK 1.4 now includes the *java.nio* package, which adopts much of the functionality of NBIO.

1.2 Event handler overview

A Sandstorm application designer implements an *event handler* for each stage in the service, which conforms to the *EventHandlerIF* interface. *EventHandlerIF* includes the two methods *handleEvent()* and *handleEvents(),* which take as arguments a single event or a batch of multiple events to process. The event handler is invoked by the Sandstorm runtime system whenever there are new events to process; the event handler itself does not determine when it is invoked or with what set of events. This allows the runtime system to control the execution of stages, the number and type of events passed to each stage, and so forth.

The event handler should process each event passed to it, and may optionally enqueue one or more outgoing events to other stages. Alternately, an event handler may drop, reorder, or delay events, but no mechanism is provided to reject events (that is, provide a signal to the originating stage that an event has been rejected) once they have been passed to the event handler. This is discussed in more detail in the next section. Because a stage may contain multiple threads, an event handler must synchronize access to any shared state within the stage (as well as across multiple stages).

2 Sandstorm design principles

Apart from the structure mandated by the SEDA architecture itself, Sandstorm embodies several other design principles in keeping with our goals of simplifying service construction. The most important aspect of Sandstorm's design is that of simplifying the application interface to afford the greatest amount of flexibility to the runtime system. Applications merely implement event handlers; they do not manage threads, scheduling, event queues, or most aspects of load conditioning. Of course, specific services may wish to extend these internal interfaces, but the goal has been to capture as much of the essential needs of a wide class of services in the underlying runtime.

2.1 Thread management

A fundamental design decision in Sandstorm is that the runtime, rather than the application, is responsible for creating, scheduling, and managing threads. This approach is motivated by two observations. First, thread management is difficult and error-prone, and requiring applications to manage their own threads results in undue complexity. Second, giving the runtime control over threads allows for a great deal of flexibility in terms of allocation and scheduling policies.

For this reason, threads in Sandstorm are managed by an internal *thread manager* that is responsible for allocating and scheduling threads across stages. Each stage is managed by one thread manager, but there may be multiple thread managers in the system. Care must be taken so that different thread managers do not interfere with one another (for example, having a thread manager monopolize resources); since thread managers are internal to the runtime system, this requirement is easy to meet.

Implementing different thread managers allows one to experiment with different thread allocation and scheduling policies without affecting application code. We have implemented several different thread managers in Sandstorm. The default thread manager maintains a private thread pool for each stage in the application; the size of each pool is managed by the *thread pool controller*, described in Section 4.1. An alternate thread manager allocates a single systemwide thread pool and schedules those threads across stages according to some algorithm (such as weighted round robin).

We have primarily made use of the thread-pool-per-stage thread manager in our experiments, because it is simple and manages each stage independently. Also, this approach relies upon the operating system to schedule threads, which we consider an advantage since the OS has more knowledge than the runtime system about resource usage, blocking, page faults, and other conditions that affect scheduling decisions. An interesting area for future work would be to consider alternate thread management policies for Sandstorm. The Scout [100] and Exokernel [70] operating systems, as well as the Click router [77], have explored scheduling policies in environments similar to Sandstorm, but have focused on specific applications (such as packet routing). It may be possible to apply similar techniques to Sandstorm.

2.2 Event queue management

The second design principle is that the runtime system is responsible for the number and rate of events processed by each stage's event handler, since these are factors that affect application performance and load conditioning. For example, if a stage is the source of a resource bottleneck, the runtime system could choose to defer its execution until enough resources are available for the stage to make progress. Likewise, the runtime can determine the ideal number of events to pass to each invocation of an event handler to optimize performance. This is the goal of the Sandstorm *batching controller*, described in Section 4.2.

In Sandstorm, applications are not responsible for dequeueing events from their associated event queue. Rather, each runtime-managed thread first dequeues a batch of events from the stage's queue, and then invokes the associated event handler with those events for processing. Event handlers may only perform an *enqueue* operation onto stage event queues; the dequeue interface is never exposed to applications.

2.3 Event processing requirements

The third design principle is that enqueue operations on a stage's event queue are synchronous: the queue either accepts events for (future) processing or rejects them immediately. A queue rejection causes the originator to receive an immediate exception or error code, indicating that the recipient stage is overloaded and that load conditioning should be performed.

This approach implies that event handlers cannot reject events passed to them; no mechanism is provided to signal to the originator of an event that the recipient is unable to process it. Rather, admission control must be used to perform early rejection of incoming work from a queue, causing the corresponding enqueue operation to fail. Note, however, that a stage may elect to drop or delay an event that has been passed to the event handler. This technique permits application-specific load shedding, but should only be used in specific situations, such as when it is safe to silently drop events within a stage. In all other cases admission control should be used to bound the incoming event queue.

In response to an enqueue rejection, a stage may wish to drop the event (if it is safe to do so), perform an alternate action (e.g., degrade service fidelity), adjust its own admission control parameters (e.g., to throttle the rate of incoming events into it), or block the event-handling thread until the downstream queue can accept the event. The latter case provides a simple mechanism for backpressure: if all of the threads within a stage are blocking on a downstream event queue, the queue for the blocked stage will eventually fill as well.

Class	**Description**
EventHandlerIF	Core logic for stage; implemented by applications
QueueElementIF	Empty interface for queue elements
StageIF	Handle to stage's name and event queue
ManagerIF	Global access to stages; stage creation
SingleThreadedEventHandlerIF	Indicates single-threaded stage
SinkIF	Enqueue end of event queue
EnqueuePredicateIF	Guards enqueue operations on event queue
ProfilableIF	Hook for system profiler

Figure 25: **Core Sandstorm classes.** *This table shows the core classes in the Sandstorm implementation of SEDA. Each of these interfaces is implemented either by the application designer (in the case of* EventHandlerIF*), or by the Sandstorm implementation itself. By defining system functionality in terms of these interfaces, it is straightforward for a service designer to provide an alternate implementation of various components of the runtime.*

3 Implementation details

In this section we discuss the Sandstorm implementation in detail, covering the interfaces for system management, stage graph construction, admission control, timers and signals, and profiling support. Figure 25 provides an overview of the core classes in the Sandstorm implementation. In Section 7 we describe an asynchronous HTTP protocol library as a concrete example of the use of the Sandstorm interfaces.

3.1 Queue elements and event handlers

QueueElementIF is an empty interface that must be implemented by any class that represents an event that can be placed onto a queue and passed to an event handler. As described earlier, the application logic for each stage is represented as an instance of *EventHandlerIF*, which provides four methods. *init()* and *destroy()* are invoked when the stage is initialized and destroyed, respectively; *handleEvent()* and *handleEvents()* are invoked with a single event, or a batch of events, to process.

Because a stage may be multithreaded, event handlers must be designed to be reentrant. This requirement implies that event handlers must synchronize access to any shared state within the stage, as well as across stages. An event handler may implement the optional interface *SingleThreadedEventHandlerIF*, which indicates to the runtime system that it is non-reentrant. In this case the runtime guarantees that only one thread will be executing the event handler at once.

3.2 Stage graph construction

In Sandstorm, connectivity between stages is determined entirely at runtime, and is driven by stages requesting a handle to other stages in the system through a system-provided management interface. This approach is in contrast to compile-time or configuration-time construction of the stage graph, which would allow the system to check certain properties of the graph statically (e.g., whether two connected stages agree on the event types passed between them). In Sandstorm, stages do not explicitly state the types of events that they generate and that they accept, though this functionality could be provided by a wrapper around the basic event handler API. Dynamic graph construction affords the greatest degree of flexibility: for example, using dynamic class loading in Java, new stages can be be introduced into the service at runtime.

Each stage has an associated *stage handle*, provided by the class *StageIF*. The stage handle contains methods that return the stage's name (a string) and a handle to its incoming event queue. *StageIF* also contains a *destroy()* method that can be used to clean up and remove a stage from the system. A stage refers to another stage by looking up its *StageIF* handle through the system management interface, *ManagerIF*, which maintains a mapping from stage names to stage handles.

When a stage is initialized, an instance of its event handler is created and the *init()* method invoked on it. *init()* takes a configuration parameter block as an argument, which provides pointers to the system manager, the stage's own stage handle, as well as initialization parameters in the form of name-value mappings. When Sandstorm is initialized, a configuration file is used to specify the set of initial stages and their initialization parameters. New stages can be created at runtime through the system manager. When a stage is destroyed, the *destroy()* method of its event handler is invoked before removing it from the system. Thereafter, any attempt to enqueue events to the stage will result in an exception indicating that the stage has been destroyed.

3.3 Queues and admission controllers

Because event handlers are not responsible for dequeueing events from their own event queue, only the enqueue interface is exposed to applications. This is represented by the general-purpose *SinkIF* interface, shown in Figure 26. Three basic enqueue mechanisms are supported: *enqueue()*, which enqueues a single event; *enqueue_many()*, which enqueues multiple events; and *enqueue_lossy()*, which enqueues an event but drops it if the event is rejected by the queue. In contrast, *enqueue()* and *enqueue_many()* throw an exception if the queue rejects new events, or if the corresponding stage has been destroyed. In general, applications should use the *enqueue()* or *enqueue_many()* interfaces, which provide an explicit rejection notification to which the stage must respond. *enqueue_lossy()* should only be used in cases where it is not important whether a given event is dropped.

Admission control in Sandstorm is implementing using *enqueue predicates*, which are short methods that are invoked whenever an enqueue operation is attempted on a queue. *EnqueuePredicateIF* is used to represent an enqueue predicate and supports one method: *accept()*, which takes a single event as an argument and returns a boolean value indicating whether the queue accepts the event or not.

Enqueue predicates should be fast and consume few resources, because they run in the context of the originating stage, and are on the critical path of enqueue operations. Enqueue predicates are typically used to implement simple queue-management policies, such as thresholding or rate control. A more complicated enqueue predicate might assign different priorities to incoming events, and reject low-priority events under heavy load.

The *SinkIF* methods *setEnqueuePredicate()* and *getEnqueuePredicate()* can be used to assign and retrieve the enqueue predicate for a given queue. Note that any stage can assign an enqueue predicate to any other stage; this design was intentional as it affords the greatest flexibility

```
public interface SinkIF {

  /* Enqueue the given event. */
  public void enqueue(QueueElementIF event)
    throws SinkException;

  /* Enqueue the given set of events. */
  public void enqueue_many(QueueElementIF events[])
     throws SinkException;

  /* Enqueue the given event; drop if the queue rejects. */
  public boolean enqueue_lossy(QueueElementIF event);

  /* Return the number of events in the queue. */
  public int size();

  /* Prepare a transactional enqueue;
   * return a transaction token.       */
  public Object enqueue_prepare(QueueElementIF events[])
     throws SinkException;

  /* Commit a previously prepared transactional enqueue. */
  public void enqueue_commit(Object transaction_token);

  /* Abort a previously prepared transactional enqueue. */
  public void enqueue_abort(Object transaction_token);

  /* Assign an enqueue predicate. */
  public void setEnqueuePredicate(EnqueuePredicateIF predicate);

  /* Retrieve the enqueue predicate, if any. */
  public EnqueuePredicateIF getEnqueuePredicate();

}
```

Figure 26: **The Sandstorm sink interface.** *This interface represents the enqueue end of a Sandstorm event queue. Methods are provided to enqueue events (with and without rejection notification), return the number of pending events, perform transactional enqueues, and manipulate the queue's admission control predicate.*

```
QueueElementIF events[];
SinkIF queue1, queue2;
Object token1, token2;

try {
  token1 = queue1.enqueue_prepare(events);
} catch (SinkException se1) {
  // Cannot enqueue
  return;
}

try {
  token2 = queue1.enqueue_prepare(events);
} catch (SinkException se2) {
  // Cannot enqueue -- must abort first enqueue
  queue1.enqueue_abort(key1);
}

// We now know that both queues accept the events
queue1.enqueue_commit(token1);
queue2.enqueue_commit(token2);
```

Figure 27: **Transactional enqueue example.** *This code excerpt demonstrates Sandstorm's transactional enqueue support, implementing an "all or nothing" enqueue operation across two queues.*

over placement and control of queue management policies.

Event queues also support *transactional enqueue* operations, which are provided using three additional methods in *SinkIF*. *enqueue_prepare()* takes an array of events as an argument and returns a transaction token if the queue accepts the events, throwing an exception otherwise. The transaction token is later passed as an argument to *enqueue_commit()* (causing the corresponding events to be actually enqueued) or *enqueue_abort()* (to abort the transactional enqueue).

Transactional enqueue is useful in a number of situations. For example, Figure 27 demonstrates the use of this interface to perform an "all or nothing" enqueue operation on two separate queues. Here, an admission control rejection from either queue causes the entire operation to be aborted. Another use of transactional enqueue is to perform a *split-phase* enqueue, in which a stage first tests whether a downstream queue can accept new events, and then performs the work necessary to generate those events. This is useful in cases where event generation requires considerable resources that the application wishes to avoid consuming if the events are to be rejected by the downstream queue. For example, a stage might perform an *enqueue_prepare()* for a “placeholder” event, filling in the event with generated data only if the event is accepted.

3.4 Timers and signals

Stages often wish to receive events at certain times, and Sandstorm provides a global timer interface for this purpose. Application code can register with the timer for events to be enqueued onto a particular queue at some time in the future. The timer is implemented with a single thread that maintains a list of pending timer events sorted by increasing time in the future. The timer thread drops timer events if rejected by the associated queue.

In addition to timers, Sandstorm provides a general-purpose signaling mechanism that can be used to propagate global state changes to one or more event queues. Stages can register with the systemwide signal manager to receive events of certain types. For instance, one signal type indicates that all stages in the Sandstorm configuration file have been created and initialized; this can be used as an initialization barrier. This mechanism could also be used to signal systemwide resource limits to applications, for example, informing stages when available memory is approaching some limit. We return to this idea in Chapter 8.

3.5 Profiling and debugging

Sandstorm includes a built-in profiler, which records information on memory usage, queue lengths, and stage relationships at runtime. The data generated by the profiler can be used to visualize the behavior and performance of the application; for example, a graph of queue lengths over time can help identify a bottleneck. Figure 42 in Chapter 6 is an example of such a graph.

The profiler is implemented as a single thread that periodically gathers statistics and writes the data to a log file, which can be visualized using standard tools such as *gnuplot*. The profiler normally gathers information on system memory usage, queue lengths, thread pool sizes, and various

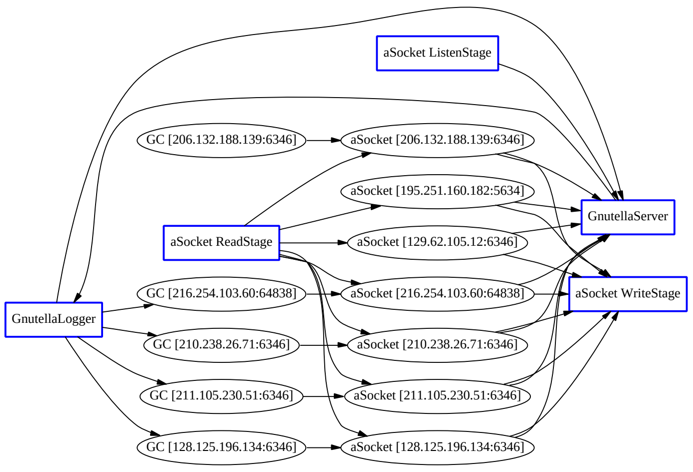
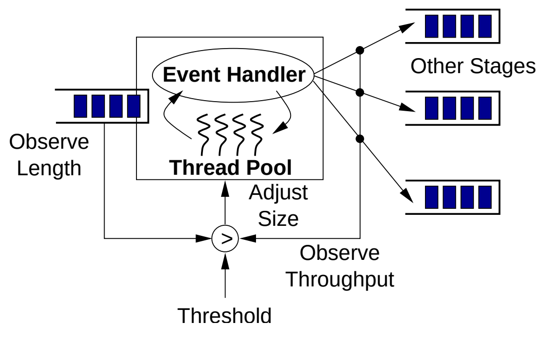

aSocket ListenStage
GC [206.132.188.139:6346]
aSocket [206.132.188.139:6346]
aSocket [195.251.160.182:5634]
GnutellaServer
aSocket ReadStage
aSocket [129.62.105.12:6346]
GC [216.254.103.60:64838]
aSocket [216.254.103.60:64838]
aSocket WriteStage
GnutellaLogger
GC [210.238.26.71:6346]
aSocket [210.238.26.71:6346]
GC [211.105.230.51:6346]
aSocket [211.105.230.51:6346]
GC [128.125.196.134:6346]
aSocket [128.125.196.134:6346]

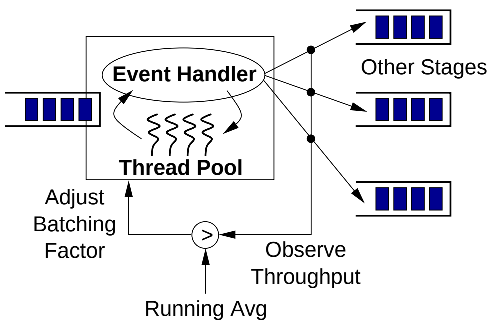
Figure 28: **Visualization of stage connectivity:** *This graph was automatically generated from profile data taken during a run of a Sandstorm-based Gnutella server, described in Chapter 6. In the graph, boxes represent stages, ovals represent classes through which events flow, and edges represent event propagation. The main application stage is* `GnutellaLogger`*, which makes use of* `GnutellaServer` *to manage connections to the Gnutella network. The intermediate nodes represent Gnutella packet-processing code and socket connections.*

controller parameters. Application code can supply additional statistics-gathering hooks by implementing the interface *ProfilableIF*, which supplies a single method: *profileSize()*, returning an integer. Data from application-supplied hooks is also included in the profile.

The profiler can also generate a graph of stage connectivity, based on a runtime trace of event flow. Figure 28 shows an automatically generated graph of stage connectivity within the Gnutella packet router described in Chapter 6. Because the Sandstorm stage graph is dynamic and stage handles may be directly passed between stages, the system must determine the stage graph at runtime. Stage connectivity is inferred by interposing a proxy interface on every event queue in the system. For each enqueue operation, the proxy records the stage performing the enqueue (determined using the current thread identifier) and the stage onto which the enqueue was performed (a property of the event queue). A count of the number of events crossing a particular arc of the stage graph is also recorded. The *graphviz* package [47] from AT&T Research is used to render the graph from the log file generated by the profiler.



(a) Thread pool controller



(b) Batching controller

Figure 29: **Sandstorm resource controllers:** *This figure depicts two of the resource controllers in Sandstorm that adjust the resource allocation and behavior of each stage to keep the application within its operating regime. The* thread pool controller *adjusts the number of threads executing within the stage based on perceived demand, and the* batching controller *adjusts the number of events processed by each iteration of the event handler to optimize throughput and response time.*

4 Resource controllers

An important aspect of the SEDA architecture is automatic management of the resource consumption of each stage, which attempts to keep the service within its ideal operating regime despite fluctuations in load. Sandstorm provides a number of dynamic resource controllers for this purpose. In this section we describe the implementation of two such controllers: the thread pool controller and the batching controller, both shown in Figure 29. In Chapter 7 we discuss the use of queue throttling controllers for overload management.

4.1 Thread pool controller

The Sandstorm thread pool controller adjusts the number of threads executing within each stage. The goal is to avoid allocating too many threads, but still have enough threads to meet the concurrency demands of the stage. The controller periodically samples the input queue (once per second by default) and adds a thread when the queue length exceeds some threshold (100 events by default). Threads are removed from a stage when they are idle for a specified period of time (5 seconds by default).

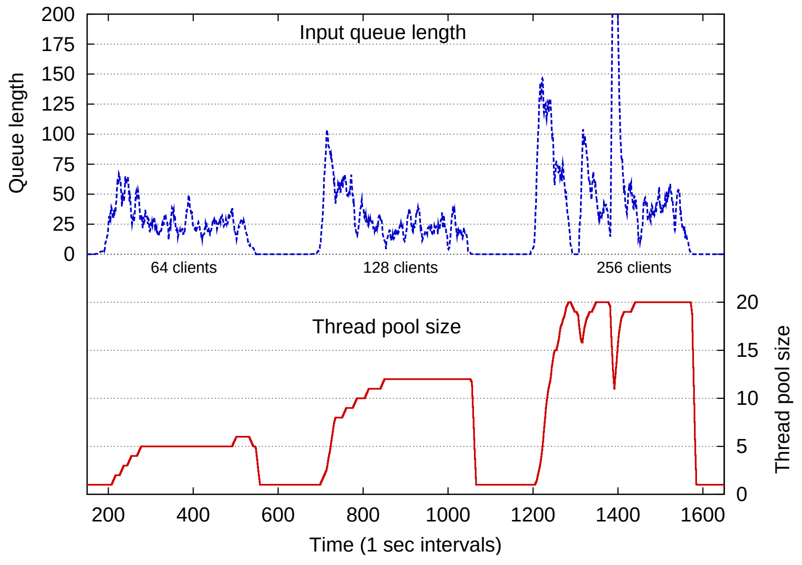
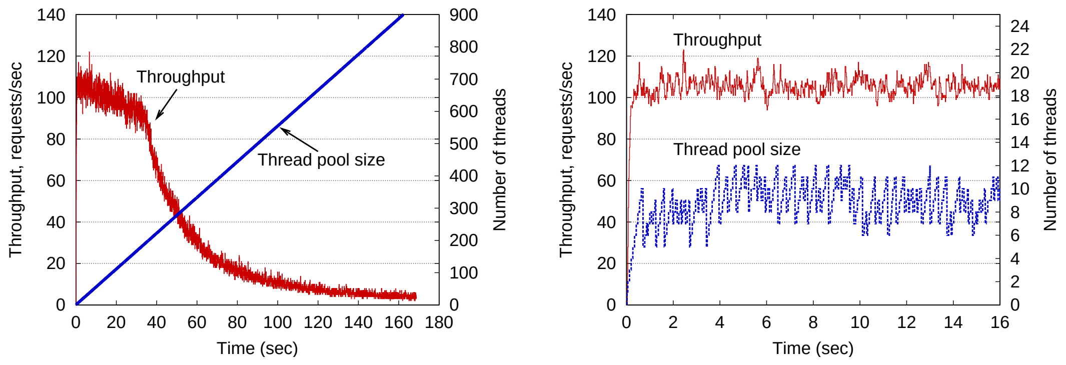


Figure 30: **Sandstorm thread pool controller:** *This graph shows the operation of the thread pool controller for one of the stages in the Haboob Web server, described in Chapter 6. The controller adjusts the size of each stage's thread pool based on the length of the corresponding event queue. In this run, the queue length was sampled every 2 seconds and a thread was added to the pool if the queue length exceeded 100 events. Here, a maximum per-stage limit of 20 threads was imposed. Threads are removed from the pool when they are idle for more than 5 seconds. The three bursts of activity correspond to an increasing number of clients accessing the Web server; as the figure shows, as the demand on the stage increases, so does the number of threads allocated to it. Likewise, as demand subsides, threads are removed from the pool.*

Figure 30 shows the thread pool controller at work during a run of the Sandstorm-based Web server described in Chapter 6. The run contains three bursts of activity, corresponding to an increasing number of clients accessing the Web server. Note as well that the client load is extremely bursty. As bursts of requests arrive, the controller adds threads to the stage's thread pool until saturating at an administrator-specified maximum of 20 threads. Between periods, there is no demand for service, and the thread pool shrinks.

Rather than imposing a fixed limit on the number of threads in a stage, the thread pool controller can automatically determine the maximum thread pool size, using mechanisms similar to multiprogramming level control in virtual memory systems [82]. The controller samples the throughput of each stage (requests processed per second), maintaining a weighted moving average



(a) Without thrashing detection **(b) With thrashing detection**

Figure 31: **Thread pool thrashing detection:** *This figure shows the operation of the thrashing detection mechanism in the Sandstorm thread pool controller, which serves to bound the maximum size of each stage's thread pool to avoid performance degradation. Here, a single stage is continuously overloaded, processing CPU-bound requests, some of which require entering a critical section. In (a), no thrashing detection is enabled, so the thread pool controller continues to add threads to the stage despite performance degradation. In (b), the thrashing detection mechanism maintains a small thread pool size that sustains high throughput.*

of the throughput samples. The controller also remembers an estimate of the highest throughput delivered by the stage, as well as the "optimal" maximum thread pool size that corresponds to that throughput.

The maximum thread pool size is determined in the following manner. If the current throughput is higher than the highest recorded throughput by some margin (20% by default), the controller records the current throughput and thread pool size as the new optimal values. If the current throughput is below the highest recorded throughput (20% by default), the controller sets the size of the thread pool to the previously recorded optimal value. At the same time, a random number of threads (between 0 and 4 threads by default) are removed from the pool. This is intended to avoid local minima where the current estimate of the optimal thread pool size is no longer accurate. By deliberately underprovisioning the stage, the thread pool controller will naturally allocate additional threads if needed based on the queue length threshold. If these threads are no longer needed, the number of threads in the pool will decay over time when the controller resets the thread pool size, or as threads go idle.

Figure 31 shows the operation of the thrashing detection mechanism during the run of a simple application consisting of a single stage that is continuously overloaded. The stage processes requests, each of which runs 100,000 iterations of a loop that generates random numbers. With 50% probability, each request causes the corresponding thread to enter into a critical section during the execution of the random-number-generation loop. Upon exiting the critical section the thread wakes up all other threads waiting on the condition variable, leading to "thundering herd" contention for the critical section. This application is meant to stress two scalability limits of threads, namely, the overhead of scheduling many CPU-bound threads and performance degradation due to lock contention and synchronization.

The stage is processing requests from an event queue that is always full, so the thread pool controller will always attempt to allocate additional threads to the stage. Figure 31(a) shows the throughput and number of threads allocated to the stage with thrashing detection disabled; as the number of threads grows to be very large the throughput drops significantly. Figure 31(b) shows that with thrashing detection enabled, the controller maintains a small thread pool size that sustains high throughput. The benchmark is being run on a machine with 4 CPUs, and 50% of the requests enter a critical section; accordingly, the thread pool controller maintains a pool size of about 8 threads.

Because thread pool control is local to each stage, there is a possibility that if a large number of threads are allocated to one stage, other stages could experience throughput degradation. Our approach to thrashing detection should avoid most of these cases: if a large number of threads are allocated to one stage, that stage's own thrashing detection should cause it to back off before it seriously impacts the performance of other stages.

4.2 Batching controller

As discussed previously, the runtime passes a *batch* of multiple events to the event handler upon invocation, rather than individual events. The goal of request batching is to improve throughput by increasing cache locality during event handler execution, or by allowing the application to amortize expensive operations over multiple requests. In addition, the event handler may be able to reorder or prioritize requests in a batch. We call the maximum size of a batch passed to an event handler the *batching factor*. Note that if the batching factor is larger than the current length of the event queue, the batch simply contains all of the pending events.

There is an inherent tradeoff in the use of batching, however. Intuitively, a large batching factor can improve throughput by maximizing the opportunities for multi-request processing and overhead amortization. However, a large batching factor can reduce parallelism. For example, if a stage has multiple threads, a large batching factor can cause one thread to receive all of the requests, leading to an imbalance in thread operation and missed opportunities for parallelism. Keeping the batching factor small can also reduce response time, as it reduces cases of head-of-line blocking where a single thread is processing all events in series, rather than allowing multiple threads to process events in parallel.

The Sandstorm batching controller attempts to trade off these effects by searching for the smallest batching factor that sustains high throughput. It operates by observing the throughput of each stage, maintaining an estimate of the highest throughput seen in a recent time window. The controller decreases the batching factor by a small multiplicative factor (20% by default) until the observed throughput falls below some margin of the best throughput estimate (10% by default). The controller then begins to increase the batching factor by a small factor (also 20%) until the throughput approaches the best estimate. The controller responds to a sudden drop in throughput by resetting the batching factor to its maximum value (1000 events by default).

Figure 32 shows the batching controller at work on a simple benchmark consisting of a single stage processing a continuous stream of events. In the benchmark, the stage is capable of amortizing a single expensive operation over the events in a batch, but has an artificially imposed optimal batching factor of 200 events. If the event handler receives more than 200 events, its performance degrades, and if it receives less than 200 events, there is less throughput gain from amortizing the expensive operation across the batch. In the figure, the controller reduces the batch-

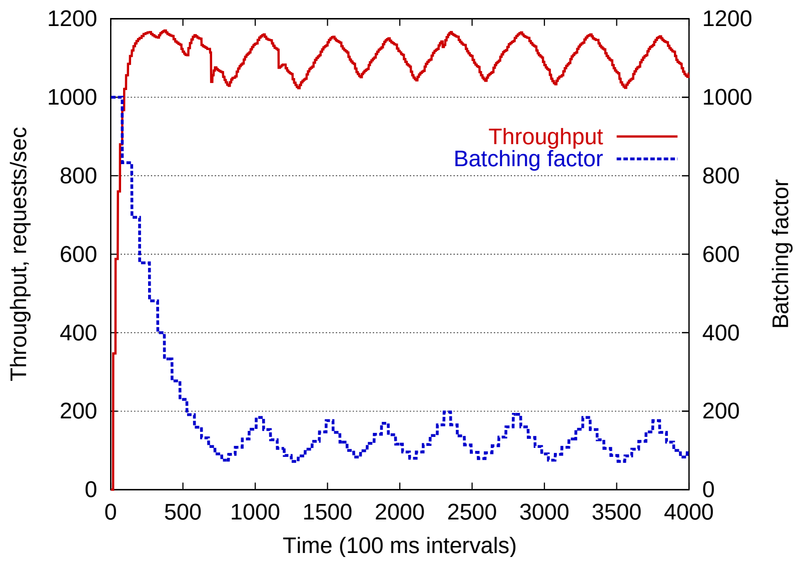


Figure 32: **Sandstorm batching controller:** *This graph shows the operation of the batching controller for a simple benchmark consisting of a single stage processing a continuous stream of events. The stage's event handler has an optimal batching factor of 200 events: the first event in each batch performs a complex operation (generating 10,000 random numbers), which is amortized across up to 199 other events in the same batch. If the batching factor is either less than or greater than 200 events, throughput will degrade as the complex operation must be performed more frequently. The controller first reduces the batching factor to just below the optimal value, and then increases the batching factor until it approaches the peak throughput.*

ing factor to just below the optimal value, and then increases the batching factor until it approaches the peak throughput. The controller does not damp it response, which explains the oscillations near the optimal batching factor.

4.3 Other controllers

These mechanisms represent two simple examples of dynamic control in Sandstorm. A wide range of potential controllers could be implemented within this framework. For example, the thread pool controller could be extended to support a notion of stage priority, in which higher-priority stages receive a greater number of threads than lower-priority stages. Another option is to adjust thread scheduling parameters based on the stage's progress, as proposed by Steere *et* *al.* [120]. In this case, the number of threads (or priorities of threads within a stage) could be tuned to maintain balanced rates of event production and consumption between each pair of connected stages. Note that our current thread pool controller approximates this operation: by adding threads to a stage with a queue backlog, stages that are underprovisioned for load are given higher priority. Sandstorm's structure facilitates inspection and control of the underlying application, and a range of control strategies are possible in this model.

5 NBIO: Nonblocking I/O facilities for Java

Java did not support nonblocking I/O interfaces until the introduction of JDK 1.4 in early 2002. Before this it was necessary to use blocking operations and threads for multiplexing across multiple I/O channels. As we have discussed, this approach does not scale well, with performance degrading considerably as the number of I/O channels grows large. To overcome this problem, we implemented *NBIO*, a library making use of native method calls to provide true nonblocking I/O in Java. NBIO has been released to the public and has seen widespread adoption by developers building scalable services in Java. NBIO has also influenced the design of the nonblocking I/O interfaces being standardized in JDK 1.4, and the author has participated on the expert group defining those interfaces.

NBIO extends the standard Java socket interfaces with new classes *NonblockingSocket* and *NonblockingServerSocket*, as well as nonblocking stream interfaces *NonblockingInputStream* and *NonblockingOutputStream*. All operations (*connect*, *accept*, *read*, *write*, etc.) on these classes are nonblocking. Native methods, written in C, are used to invoke the corresponding system calls for these operations; the standard Java Native Interface [124] is used to bind these methods to the Java classes.

To support I/O event notification across a set of nonblocking sockets, NBIO provides the class *SelectSet* that implements an interface similar to the UNIX *select()* system call. Applications register instances of the *SelectItem* class with a *SelectSet*, each of which consists of a socket as well as an event mask indicating the types of events that the application wishes to be notified of on that socket. Event types include read readiness, write readiness, accept readiness (for incoming connections on a server socket), and connection established (for an outgoing connection from a client socket). The method *SelectSet.select()* is used to test for I/O events on all of the registered *SelectItem*s, returning a list of pending events. *SelectSet.select()* either blocks indefinitely waiting for new events, or a timeout can be supplied as an optional argument.

Several *SelectSet* implementations are provided to make the best use of the underlying OS facilities. The default implementation uses the standard *poll()* system call found on most UNIX systems. An alternate implementation uses the `/dev/poll` mechanism [111] mechanism that demonstrates better scalability than *poll()* on some platforms. An alternate implementation for Windows 2000 systems is provided, using the *WSAAsyncSelect* interface.

NBIO does not yet provide nonblocking file I/O functionality. The reasons behind this are discussed in more detail in Section 6.2.

6 Asynchronous I/O primitives

To meet SEDA's goal of supporting high concurrency requires efficient, robust I/O interfaces. This section describes the implementation of asynchronous socket and file I/O primitives, using the SEDA concepts to structure these layers. We describe an asynchronous network socket layer that makes use of the NBIO package for nonblocking I/O, and an asynchronous file I/O layer that uses blocking OS calls and a thread pool to expose nonblocking behavior. Both of these layers are implemented as a set of Sandstorm stages that can be used by applications to provide fast asynchronous I/O.

6.1 Asynchronous socket I/O

While NBIO provides low-level nonblocking socket interfaces in Java, it is important to integrate this functionality into the Sandstorm programming model. For example, NBIO requires

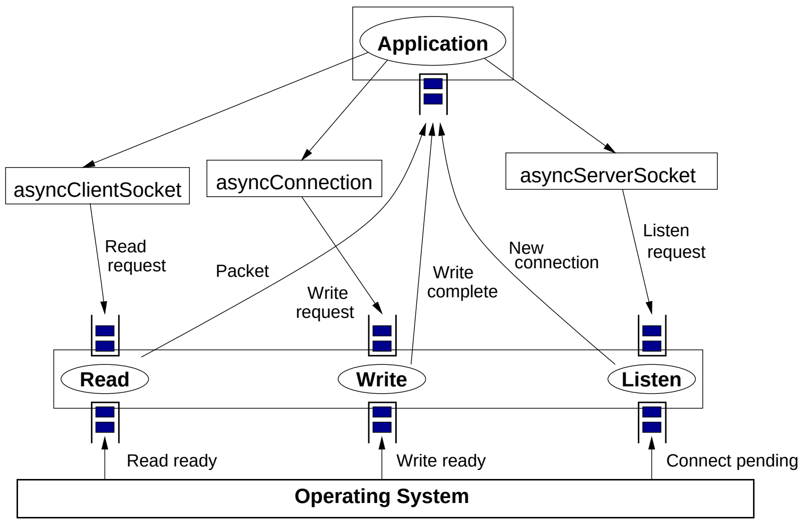


Figure 33: **SEDA-based asynchronous sockets layer:** *The Sandstorm sockets interface consists of three stages:* read, write, *and* listen. *The* read *stage responds to network I/O readiness events and reads data from sockets, pushing new packets to the application stage. The* write *stage accepts outgoing packets and schedules them for writing to the appropriate socket. It also establishes new outgoing socket connections. The* listen *stage accepts new TCP connections and pushes connection events to the application.*

that applications actively perform *select()* operations to receive pending I/O events; in contrast, Sandstorm stages are invoked passively when new events are ready for processing. In addition, NBIO requires that applications carefully schedule I/O across a large number of sockets; however, one of the goals of Sandstorm is to prevent applications from making complex scheduling decisions. In this section we describe the Sandstorm asynchronous sockets layer, *asyncSocketIO*, which is layered on top of NBIO and provides an easy-to-use interface. *asyncSocketIO* is also capable of using the *java.nio* nonblocking I/O support in JDK 1.4.

In *asyncSocketIO*, applications create instances of the classes *asyncClientSocket* and *asyncServerSocket* to initiate outgoing and incoming socket connections. When a connection is established, an *asyncConnection* object is pushed onto an event queue provided by the user (typically the queue associated with the requesting stage). Incoming packets are enqueued onto the user's event queue, and *asyncConnection* implements a queue interface onto which outgoing packets can be placed. Each outgoing packet may also have an associated event queue onto which a completion event is pushed when the packet is transmitted. Error and other notification events are passed to the user in a similar way.

Internally, the *asyncSocketIO* layer is implemented using three stages, which are shared across all sockets, as shown in Figure 33. *ReadStage* reads network packets and responds to user requests to initiate packet reading on a new socket. *WriteStage* writes packets to the network and establishes new outgoing connections. *ListenStage* accepts new TCP connections and responds to user requests to listen on a new port. Each operation on an *asyncConnection*, *asyncClientSocket*, or *asyncServerSocket* is converted into a corresponding request for one of the internal stages and placed on that stages's request queue. The *asyncSocketIO* stages are managed by their own thread manager that allocates a single thread to each stage.

Each of these stages must respond to read- and write-readiness events from the operating system and use these to drive socket processing, potentially across a large number of sockets. At the same time, each stage must respond to requests from the user, such as for creation of a new socket. Therefore, each stage services two separate event queues: a request queue from the user, and an I/O readiness/completion event queue from the operating system. The thread within each stage alternately services each queue, using a simple timeout mechanism to toggle between the two. The I/O event queue is implemented as a library that causes dequeue operations to invoke the underlying NBIO *SelectSet.select()* call to retrieve I/O events. To ensure fairness across sockets, each stage *randomizes* the order in which it processes I/O events delivered by the operating system. This is necessary because the OS generally returns socket events in a fixed order (e.g., in increasing order by file descriptor).

Each connected socket has an associated *SockState* object which is maintained by *ReadStage* and *WriteStage*. Because the operation of these two stages is mostly independent, little synchronization is required. The one exception arises due to closing sockets: sockets may be closed either by the user (by enqueueing a close request onto the appropriate *asyncConnection*), or externally, due to the peer shutting down the connection or an error condition.

ReadStage operates by performing a socket read whenever an I/O readiness event indicates that a socket has data available. It reads at most 16 KB into a pre-allocated buffer and enqueues the resulting packet onto the event queue provided by the user. In case of an I/O error (e.g., because the peer has closed the connection), the stage closes the socket and pushes an appropriate notification event to the user. Each socket read requires the allocation of a new packet buffer; while this can potentially cause a great deal of garbage collection overhead, we have not found this to be a performance issue. Note that because this system is implemented in Java, no explicit deallocation of expired packets is necessary.

ReadStage also provides an optional rate controller that can throttle the rate at which packets are read from the network; this controller is useful for performing load shedding during overload conditions. The controller is implemented by calculating a moving average of the incoming packet rate and introducing artificial delays into the event-processing loop to achieve a certain rate target.

WriteStage receives packet write requests from the user and enqueues them onto an internal queue associated with the corresponding socket. When the OS indicates that a socket is ready for writing, it attempts to write the next packet on that socket's outgoing queue. As described in Section 2, the socket queue may have an associated threshold to prevent a large backlog of outgoing sockets pending transmission. This can be a problem if a socket is connected to a slow peer, such as a modem link. In a blocking sockets interface, an attempt to write data to such a connection would simply block the caller. However, when using nonblocking I/O the rate of socket I/O and the rate of outgoing packet generation are decoupled, necessitating a limitation on the length of the outgoing packet queue.

6.2 Asynchronous file I/O

In contrast to *asyncSocketIO*, which makes use of nonblocking I/O primitives from the OS, the Sandstorm asynchronous file I/O (*asyncFileIO*) layer does not have true nonblocking OS primitives available to it. This is mainly due to the lack of nonblocking file support in commodity operating systems; moreover, the various implementations that do exist do not share common interfaces or semantics. For example, the POSIX.4 AIO [117] implementation in Linux provides nonblocking file reads, but uses an in-kernel thread pool for handling writes. Also, AIO does not provide a straightforward mechanism for testing I/O readiness across a set of files. Rather than conform to imperfect underlying APIs, we have instead opted to use the SEDA concept of "wrapping" blocking operations in a stage, making use of blocking I/O primitives and a tunable thread pool to expose an asynchronous file interface despite internal blocking behavior.

Users perform file I/O through an *asyncFile* object that supports the familiar interfaces *read*, *write*, *seek*, *stat*, and *close*. The *asyncFileIO* layer is implemented using a single stage that contains a dynamically sized thread pool, identical in nature to application-level stages. Each operation on an *asyncFile* causes a request to be placed onto this stage's event queue. Threads within the stage dequeue each request and perform the corresponding (blocking) I/O operation on the file. To ensure that multiple I/O requests on the same file are executed serially, only one thread may process events for a particular file at a time. This is an example of *stage-specific event scheduling*—the *asyncFileIO* threads are dispatched in a way that avoids contention for individual files.

When an I/O request completes, a corresponding completion event is enqueued onto the user's event queue. The *asyncFileIO* stage is initialized with a single thread in its thread pool, and the Sandstorm thread pool controller is responsible for dynamically adjusting the size of the thread pool based on observed concurrency demand.

7 Design example: HTTP protocol library

To illustrate the use of the Sandstorm interfaces, in this section we briefly describe the design of an asynchronous HTTP protocol library that makes use of the *asyncSocketIO* primitives. This library provides HTTP server-side protocol functionality that can be readily adopted for use in a Web server or other service; the Haboob Web server described in Chapter 6 is based on this library.

The core class in this package is *httpServer*, which implements a stage that accepts HTTP connections and performs protocol processing. An application stage can create an instance of *httpServer* to listen on a given port for HTTP connections. The *httpServer* stage creates the server socket and receives incoming connections on its event queue. For each connection, *httpServer* creates an instance of *httpConnection* and enqueues it to the user. *httpConnection* implements the *SinkIF* interface, so it can be used to send HTTP responses back to the corresponding client or close the connection.

Every data packet received by *httpServer* is passed through a per-connection state machine that performs protocol processing; this is implemented as the class *httpPacketReader*. *httpPacketReader* accumulates raw data from the underlying *asyncConnection* and parses it, building up the components of the HTTP request piecewise, including the request line, request header, and request body [42]. To simplify protocol processing, Sandstorm provides a utility class called *asyncSocketInputStream*. This class takes multiple packets received on an *asyncConnection* and allows them to be viewed as a contiguous stream of bytes, providing compatibility with standard Java text-processing libraries. When a complete HTTP request has been read and parsed, *httpPacketReader* enqueues an *httpRequest* object onto the user's event queue. The *httpRequest* provides access to the request header fields, URL, and body, as well as a pointer to the corresponding *httpConnection*. The user stage can then perform whatever processing is necessary to handle the *httpRequest*.

To send a response to the client, the user creates an instance of *httpResponse*, which has a number of subclasses corresponding to the various response types in HTTP: *OK*, *not found*, *bad request*, *service unavailable*, and *internal server error*. The response object contains the response header and payload, represented as separate buffers. The user then enqueues the *httpResponse* onto the desired *httpConnection*, which extracts the data buffers from the response and enqueues them onto the corresponding *asyncConnection*.

This basic approach to protocol design using SEDA is general enough to capture a range of protocols. We have also developed an asynchronous Transport Layer Security (TLS) [38] library that extends the *asyncSocketIO* interfaces, providing transparent authentication and encryption for socket connections.[1] Its design is very similar to that of the HTTP library, but uses additional stages for performing encryption and decryption as well as negotiating key exchange. The library uses PureTLS [33] and Cryptix [132] for the basic TLS functionality. Because the TLS library subclasses *asyncSocketIO*, any code (such as the HTTP library) making use of *asyncSocketIO* can substitute TLS-enabled sockets with very few changes.

8 Evaluation

In this section we provide an initial evaluation of the Sandstorm framework in terms of code complexity, microbenchmark results, and scalability of the *asyncSocketIO* layer. A more detailed evaluation based on application results is given in Chapter 6.

8.1 Code size and complexity

Figure 34 shows a breakdown of the code size for each of the Java packages in the Sandstorm implementation. The number of classes, methods, and non-commenting source statements (NCSS) [109] are shown. The latter is a better indication of code size than source code lines, which vary greatly with indentation and coding style. The figure shows that Sandstorm is relatively small, with a total of only 10299 NCSS. The bulk of the complexity lies in the NBIO primitives and

[1] Dennis Chi developed the asynchronous TLS library.

Package	**Description**	**Classes**	**Methods**	**NCSS**
`nbio`	*Nonblocking I/O primitives*	14	182	756
`api`	*basic API*	28	71	167
`api.internal`	*internal APIs*	3	13	22
`main`	*initialization*	4	37	591
`core`	*core classes*	13	104	906
`internal`	*internal implementation*	20	130	1644
`asyncSocketIO`	*asynchronous sockets*	32	141	1375
`asyncFileIO`	*asynchronous files*	20	78	493
`http`	*HTTP protocol*	13	88	787
`Gnutella`	*Gnutella protocol*	14	113	878
`util`	*various utilities*	3	20	281
Total		**191**	**1099**	**10299**

Figure 34: **Sandstorm code size.** *This table shows the number of classes, methods, and non-commenting source statements (NCSS) in each package of the Sandstorm implementation.*

asynchronous sockets library.

8.2 Sandstorm microbenchmarks

The basic queue implementation in Sandstorm consists of a linked list as well as two monitors: one for synchronizing changes to the list, and another used as a condition variable to wake up a sleeping thread blocked on an empty queue. The basic queue operations are very efficient: on a 930 MHz Pentium III system running Linux 2.2.14 and IBM JDK 1.3, an enqueue operation costs 0.5 μsec without admission control; and 0.7 μsec with a simple thresholding admission controller (enqueue predicate). Nonblocking dequeue costs 0.2 μsec and a blocking dequeue costs 0.9 μsec (due to the extra synchronization involved).

In addition, we measured the overhead of events traversing multiple stages, including the thread context switch overhead across stages. Here we assume each stage has its own thread pool with a single thread. The benchmark generates a ring of stages and measures the average time for a single event to propagate through the ring. On the system described above, the stage crossing overhead averaged 5.1 μsec per stage for 10 stages. This is a single-processor system so there is no opportunity for parallelism. The stage crossing overhead increases linearly as the number of

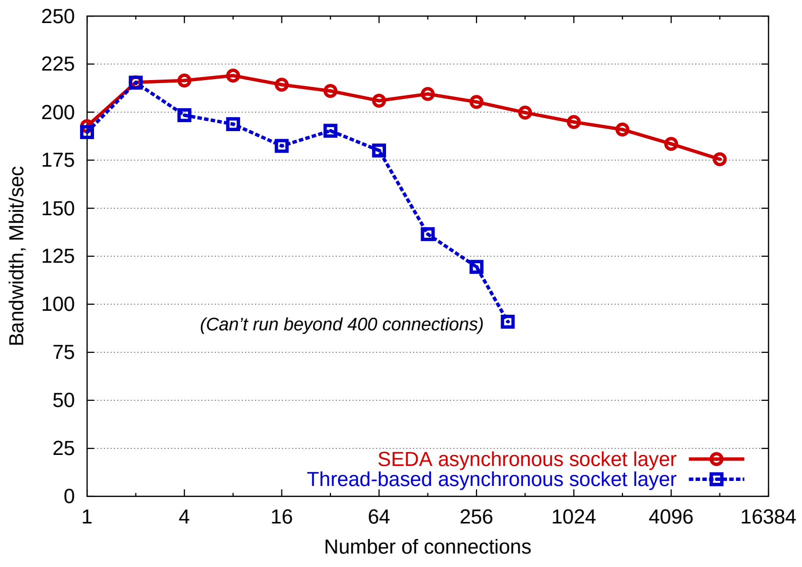


Figure 35: **Asynchronous sockets layer performance:** *This graph shows the performance of the SEDA-based asynchronous socket layer as a function of the number of simultaneous connections. Each client opens a connection to the server and issues bursts of 8KB packets; the server responds with a single 32-byte ACK for each burst of 1000 packets. All machines are connected via switched Gigabit Ethernet and are running Linux 2.2.14. The SEDA-based server makes use of nonblocking I/O primitives provided by the operating system. Performance is compared against a compatibility layer that makes use of blocking sockets and multiple threads to emulate asynchronous I/O. The thread-based layer was unable to accept more than 400 simultaneous connections, because the number of threads required would exceed the per-user thread limit in this version of Linux.*

stages increases, due to increased overheads of thread context switching and lower locality. With 100 stages the overhead increases to 40 μsec per stage. Note, however, that with batching enabled this context switch overhead is amortized across all events in a batch.

8.3 Asynchronous sockets performance

To evaluate the performance of *asyncSocketIO*, we implemented a simple server application that accepts bursts of 8KB packets from a number of clients, responding with a single 32-byte ACK for each burst of 1000 packets. This somewhat artificial application is meant to stress the network layer and measure its scalability as the number of clients increases. Figure 35 shows the aggregate throughput of the server as the number of clients increases from 1 to 8192. The server and client machines are all 4-way 500 MHz Pentium III systems interconnected using Gigabit Ethernet running Linux 2.2.14 and IBM JDK 1.3.

Two implementations of the socket layer are shown. The SEDA-based layer makes use of nonblocking I/O provided by the OS and the `/dev/poll` event-delivery mechanism [111]. This is compared against a compatibility layer that uses blocking sockets and a thread pool for emulating asynchronous I/O. This layer creates one thread per connection to process socket read events and a fixed-size pool of 120 threads to handle socket writes. This compatibility layer was originally developed to provide asynchronous I/O under Java before the development of the NBIO package.

The nonblocking implementation clearly outperforms the threaded version, which degrades rapidly as the number of connections increases. In fact, the threaded implementation crashes when receiving over 400 connections, as the number of threads required exceeds the per-user thread limit in this version of Linux. The slight throughput degradation for the nonblocking layer is due in part to lack of scalability in the Linux network stack; even using the highly optimized `/dev/poll` mechanism [111] for socket I/O event notification, as the number of sockets increases the overhead involved in polling readiness events from the operating system increases significantly [74].

9 Summary

Our prototype of a SEDA-based Internet services platform, Sandstorm, is intended to simplify the development of scalable services in several ways. By isolating application logic into event handlers that are unable to affect thread allocation, scheduling, and queue management, the runtime is given a great deal of control over the execution of service components, allowing developers to focus on application-specific details. Sandstorm's thread pool and batching controller mechanisms shield service developers from the complexity of performance tuning; the controllers automatically determine ideal runtime parameters based on observed behavior. Scalable, robust I/O primitives are provided through the *asyncSocketIO* and *asyncFileIO* layers, which extract general-purpose functionality into a clean, reusable set of interfaces.

It is important not to confuse the SEDA architecture with the Sandstorm platform implementation. Many of the design choices made in Sandstorm—such as thread management, OS-driven scheduling, and dynamic stage graph construction—are not inherent to the SEDA model, and alternate designs could be investigated. We believe that Sandstorm embodies the right balance of flexibility and ease-of-programming for a wide class of services, a claim we investigate in the next chapter.

Chapter 6

Application Evaluation

This chapter presents a detailed performance evaluation of three significant applications developed using the SEDA architecture: *Haboob*, a high-performance HTTP server; a packet router for the Gnutella peer-to-peer file sharing network; and *Arashi*, a Web-based e-mail service. These three applications typify a broad class of Internet services, and each represents a different set of design and performance goals.

In Haboob, the focus is on achieving high bandwidth and robust performance for a workload consisting of static Web pages; Haboob puts the SEDA model "on the map" with respect to prior work on static Web server design. The Gnutella packet router is used to demonstrate robustness under real-world loads: we ran our router for several days connected to the live Gnutella network. The Gnutella router is also an example of an "open loop" server in which server performance does not act as a limiting factor on offered load. Arashi is intended to demonstrate complex service dynamics through heavyweight, on-the-fly content generation: e-mail is retrieved from a back-end database, and a Python-based scripting language is used to format each Web page response. In Chapter 7, Arashi is used to evaluate several overload control mechanisms in SEDA.

Our goal is to evaluate the SEDA design through complex, realistic applications under extreme load situations. The three applications presented here are meant to be complete and usable, which forces us to address some of the more challenging questions of service design in the SEDA space.

1 Haboob: A high-performance HTTP server

Web servers form the archetypal component of scalable Internet services. Much prior work has investigated the engineering aspects of building high-performance HTTP servers, but little has been said about load conditioning, robustness, and ease of construction. Here, we describe *Haboob*,[1] a full-featured Web server capable of hosting both static and dynamic pages, built using the Sandstorm framework.

One benefit of studying HTTP servers is that a variety of industry-standard benchmarks exist to measure their performance, and a great deal of prior results have been published describing performance tradeoffs in HTTP server design. In this section we present the Haboob architecture as well as a detailed performance analysis, using a modified version of the SPECweb99 [119] Web server benchmark. We compare Haboob's performance to that of the Apache [8] and Flash [106] Web servers, showing that Haboob achieves about 10% higher throughput than these other servers, while exhibiting scalable performance as load increases.

1.1 Haboob architecture

The overall structure of Haboob is shown in Figure 36. The server consists of 11 essential stages, as well as additional stages for dynamic page processing as described below. As described in the previous chapter, three stages are used for asynchronous sockets, one for asynchronous file I/O, and three for asynchronous SSL/TLS protocol processing. The *HttpParse* stage is responsible for accepting new client connections and for HTTP protocol processing for incoming packets. The *HttpRecv* stage accepts HTTP connection and request events and dispatches them to one of several stages based on the URL and type of request. For static Web pages, the request is passed to the

[1] *Haboob* is an Arabic word, describing a type of sand storm occurring in the desert of the Sudan.

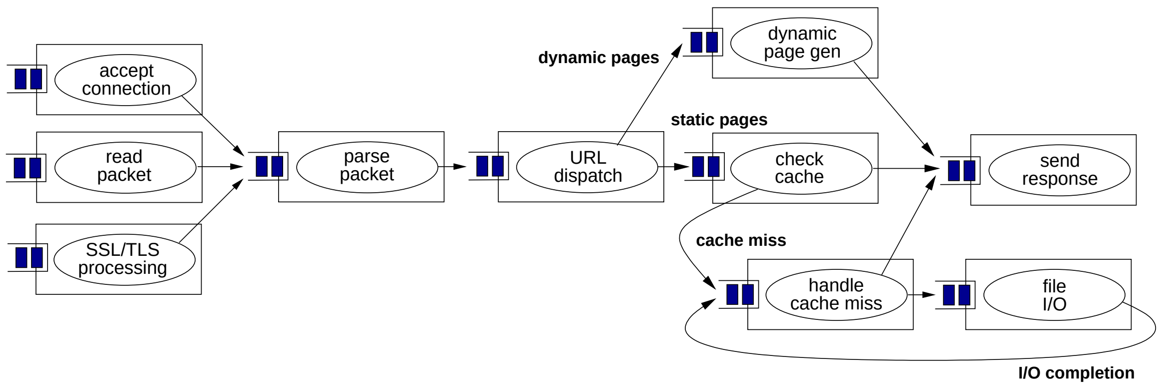


Figure 36: **Haboob HTTP server architecture:** *This is a structural representation of the SEDA-based Web server,* Haboob*. The server consists of a graph of stages for processing both static and dynamic HTTP requests. The server maintains a cache of recently accessed static pages, and a Python-based scripting language (*PyTeC*) is used to process dynamic pages. For simplicity, some event paths and stages have been elided from this figure.*

PageCache stage. For dynamic Web pages, the request is dispatched to an appropriate handler stage based on the URL; the service designer can introduce new dynamic page handler stages as needed.

PageCache implements an in-memory Web page cache implemented using a hashtable indexed by URL, each entry of which contains a response packet consisting of an HTTP header and Web page payload. The *CacheMiss* stage is responsible for handling page cache misses, using the asynchronous file I/O layer to read in the contents of the requested page from disk. File I/O completion events are passed back to the *CacheMiss* stage for insertion into the cache. When *PageCache*, *CacheMiss*, or a dynamic page handler stage have prepared an HTTP response for the client, the corresponding data is enqueued for transmission by the asynchronous sockets layer.

The page cache attempts to keep the cache size below an administrator-specified threshold (set to 204,800 KB for the measurements provided below). It aggressively recycles buffers on capacity misses, rather than allowing old buffers to be garbage-collected by the Java runtime; we have found this approach to yield a noticeable performance advantage. The cache stage makes use of application-specific event scheduling to increase performance. In particular, it implements shortest connection first (SCF) [35] scheduling, which reorders the request stream to send short cache entries before long ones, and prioritizes cache hits over misses. Because SCF is applied only to each batch of requests at a time, starvation across requests is not an issue.

1.2 Design rationale

A number of factors contributed to the decision to structure Haboob as described here. We have already discussed the rationale behind the design of the asynchronous sockets, file, and TLS/SSL layers in the previous chapter. The primary design consideration in Haboob was code modularity: it is straightforward to delineate the various stages of the service according to function. For example, HTTP request parsing is a generic function that is largely independent of the specific service layered above it. The URL dispatch stage, *HttpRecv*, is specific to Haboob and must distinguish between static and dynamic requests. One could imagine a range of URL dispatch mechanisms, for example, based on information such as client IP address or cookies; by isolating dispatch within its own stage, other aspects of the service are unaffected by the choice of dispatch policy. The code modularity of the staged design allowed us to test different implementations of the page cache without any modification to the rest of the code; the runtime simply instantiates a different stage in place of the original page cache. Likewise, another developer who had no prior knowledge of the Haboob structure was able to replace Haboob's use of the asynchronous file layer with an alternate filesystem interface with little effort.

Other structuring decisions were intended to introduce load conditioning points within the request processing flow. For example, isolating cache miss handling into its own stage allows the length of the cache miss queue to signal overload conditions. Each dynamic request handler is contained within its own stage, allowing admission control to be performed for each request type; this is the basis for overload management in the Arashi e-mail service, described in detail in Chapter 7.

Finally, other aspects of Haboob's design were motivated by performance considerations. It is expected that cache hits will generally outnumber cache misses, so it is critical to keep the cache hit handling stage as lean as possible and allow multithreading within the stage to exploit potential parallelism. An earlier incarnation of Haboob used a separate stage to perform HTTP response formatting; this code was inlined into the "send response" operation (an instance of the *Combine* design pattern) to reduce response latency. Since response formatting is a fast operation that consumes few resources, there was little need to isolate this code within its own stage. Conversely, performing cache miss handling in a separate stage allows that stage to be independently conditioned to load, and any increased latency due to queueing and context-switch overhead is negligible given the cost of filesystem access.

As with all SEDA-based services, Haboob's design represents the use of the design patterns described in Chapter 3. Conceptually, one could transform a thread-per-request Web server (such as Apache) into Haboob by repeated application of these design patterns. For example, blocking file and network I/O operations are converted to split-phase operations through the use of the *Pipeline* pattern, resulting in the Sandstorm asynchronous I/O layers. *Pipeline* is also used to introduce code modularity, for instance, as described above with the *HttpRecv* URL dispatch stage. The *Combine* pattern is used when it is natural for a single stage to perform multiple aspects of request processing, as in the case of inlining the HTTP reply formatting code. Finally, the *Partition* pattern is used to separate static and dynamic page processing into different stages. Of the patterns, only *Replicate* is not represented, since Haboob runs in a single address space on one machine.

1.3 Benchmark configuration

We have chosen the load model from the SPECweb99 benchmark suite [119] as the basis for our measurements. SPECweb99 consists of both static and dynamic workload models based on traffic from several popular Web sites. In the static workload, clients access files according to a Zipf-based request distribution; files range in size from 102 to 921600 bytes. The dynamic workload consists of a variety of request types that measure common dynamic server functionality including the use of POST, GET, CGI scripts, and cookies.

Rather than use the standard SPECweb99 load generator, we have implemented our own variant that collects more detailed performance statistics than the original, and allows us to tweak various aspects of the generated load to study the behavior of each measured server. We have also made two important modifications to the SPECweb99 benchmark. First, rather than scale the size of the Web page file set with the number of simulated clients, we keep the file set fixed at 3.31 GB, which corresponds to a SPECweb99 target load of 1000 connections. The rationale here is that a server provisioned for a heavy traffic load should still perform well with a small number of clients; under the SPECweb99 rules, a lightly loaded server would only be responsible for hosting a fraction of the content. Secondly, we measure only the performance of static Web pages, which constitute 70% of the SPECweb99 load mix. The rationale here is that it is very difficult to conduct an apples-to-apples performance study of multiple Web servers using dynamic scripting. This is because scripting performance depends greatly on factors such as the language that scripts are implemented in, the script interface to the Web server, and so forth. These factors differ greatly across the three servers measured here, and a significant engineering effort would be required to eliminate these differences.

For comparison, we present performance measurements from the popular Apache [8] Web server (version 1.3.14, as shipped with Linux Red Hat 6.2 systems) as well as the Flash [106] Web server from Rice University. Apache is the canonical example of a process-based Web server, where each client connection is handled entirely by a single process. Flash, on the other hand, is a typical event-driven Web server that has been highly tuned to maximize concurrency.

Apache uses a dynamically sized process pool that grows and shrinks as the number of concurrent client connections varies. Each process accepts a single client connection and handles all processing for it, reading file data from disk and sending it to the client in 8 KB chunks, using blocking I/O. The size of the process pool therefore bounds the number of connections that are simultaneously accepted by the server; the maximum process pool sized used in our measurements is 150, which we found to yield the best performance of several configurations tested. Increasing the process pool further either yielded no performance advantage or led to throughput degradation under heavy load.

Flash uses an efficient event-driven design, with a single process handling most request-processing tasks. The process executes in a loop, calling *select()* to retrieve I/O events from the operating system, and dispatching each event to a handler function that processes it. Because non-blocking file I/O is not provided by most operating systems, Flash uses a set of helper processes to perform blocking disk I/O operations, not unlike the design of the Sandstorm asynchronous file I/O layer. Helper processes are also used for pathname conversions, listing the contents of directories, and dynamic page generation. Helper processes communicate with the main process using UNIX domain sockets, which entails relatively high overheads as the operating system must be involved in interprocess communication. Flash maintains a static page cache that was configured to a maximum size of 204,800 KB, the same size as in Haboob. Both Apache and Flash are implemented in C, while Haboob is implemented in Java.

The client load generator developed for our experiments is also implemented in Java and make use of multiple threads to simulate many clients on a single machine. Each client thread runs in a loop, establishing a connection to the server and requesting a Web page according to the distribution specified by our variant on the SPECweb99 benchmark rules. After receiving the result, the client waits 20 milliseconds before requesting the next page. HTTP/1.1 persistent connections are used, allowing multiple HTTP requests to be issued (one at a time) on a single TCP connection. Each benchmark run for a particular client load lasted 500 seconds.

To more closely simulate the connection behavior of clients in the wide area, each client closes its TCP connection to the server after 5 HTTP requests, and reestablishes the connection before continuing. This value was chosen based on observations of HTTP traffic from [98]. Note that most Web servers are configured to use a much higher limit on the number of HTTP requests per connection, which is unrealistic but provides improved benchmark results.

All measurements below were taken with the server running on a 4-way SMP 500 MHz

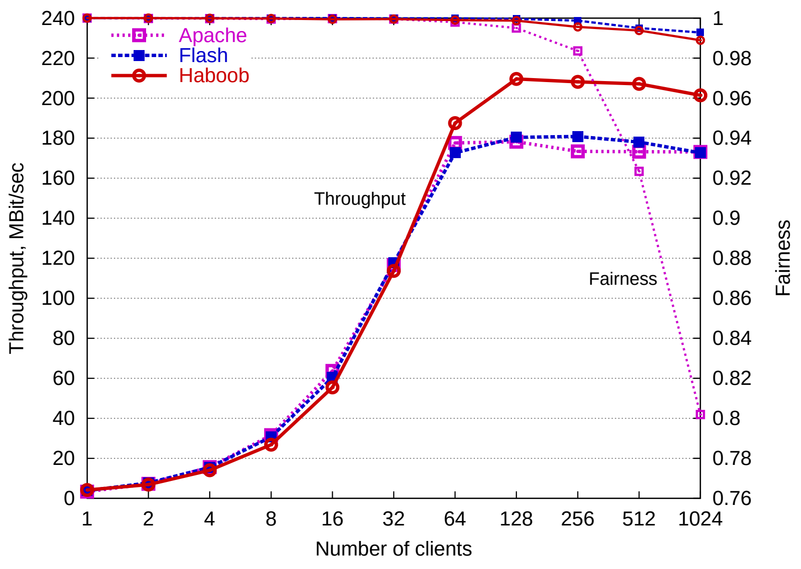


Figure 37: **Haboob Web server throughput:** *This figure shows the throughput of the Haboob Web server compared to Apache and Flash. From 1 to 1024 clients are accessing a fileset of 3.31 GBytes with a think time of 20 ms between requests. Haboob achieves roughly 10% higher throughput than Apache and Flash, and all three servers maintain high throughput despite increasing load. Also shown is the Jain fairness index delivered by each server. A fairness index of 1 indicates that the server is equally fair to all clients; smaller values indicate less fairness. The Haboob and Flash servers yield very high fairness under increasing loads, while Apache's fairness degrades considerably due to its failure to rapidly accept incoming TCP connections when saturated.*

Pentium III system with 2 GB of RAM and Linux 2.2.14. IBM JDK v1.3.0 was used as the Java platform. Thirty-two machines of a similar configuration were used for load generation. All machines were interconnected using switched Gigabit Ethernet. Although this configuration does not simulate wide-area network effects, our interest here is in the performance and stability of the server under heavy load. All benchmarks were run with warm filesystem and Web page caches. Note that the file set size of 3.31 GB is much larger than the physical memory of the server, and the static page cache for Haboob and Flash was set to only 200 MB; therefore, these measurements include a large amount of disk I/O.

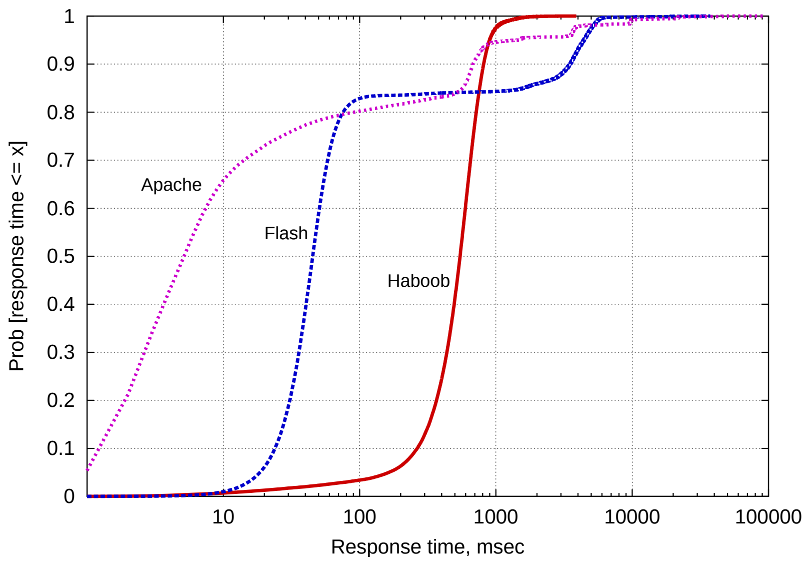


Figure 38: **Web server response times with 1024 clients:** *This figure shows the cumulative response time distribution for Haboob, Apache, and Flash with 1024 clients. While Apache and Flash exhibit a high frequency of low response times, there is a heavy tail, with the maximum response time corresponding to several minutes. This is due to exponential backoff in the TCP SYN retransmit timer: Apache accepts only 150 connections, and Flash accepts only 506, despite 1024 clients requesting service. Note the log scale on the horizontal axis.*

1.4 Performance analysis

Figure 37 shows the throughput of Haboob compared with Apache and Flash with between 1 and 1024 clients. Also shown is the Jain fairness index [68] of the number of requests completed by each client. This metric is defined as

$$f(x) = \frac{(\sum x_i)^2}{N \sum x_i^2}$$

where x_i is the number of requests for each of N clients. A fairness index of 1 indicates that the server is equally fair to all clients; smaller values indicate less fairness. Intuitively, if k out of N clients receive an equal share of service, and the other $N - k$ clients receive no service, the Jain fairness index is equal to k/N.

As Figure 37 shows, Haboob's throughput is stable as the number of clients increases, sustaining over 200 Mbps for 1024 clients. Flash and Apache also exhibit stable throughput, although

	64 clients			
Server	*Throughput*	*RT mean*	*RT max*	*Fairness*
Apache	177.56 Mbps	16.61 ms	2285 ms	0.99
Flash	172.65 Mbps	18.58 ms	12470 ms	0.99
Haboob	187.52 Mbps	15.08 ms	1872 ms	0.99
	1024 clients			
Apache	173.09 Mbps	475.47 ms	93691 ms	0.80
Flash	172.65 Mbps	665.32 ms	37388 ms	0.99
Haboob	201.42 Mbps	547.23 ms	3886 ms	0.98

Figure 39: **Haboob Web server performance summary:** *This table summarizes the performance of the Haboob, Apache, and Flash Web servers for 64 clients and 1024 clients. Note that the average response time for each server is nearly identical, though the maximum response times vary widely.*

slightly less than Haboob. This result might seem surprising, as we would expect the process-based Apache server to degrade in performance as the number of clients becomes large. Recall, however, that Apache accepts no more than 150 connections at any time, for which is not difficult to sustain high throughput using process-based concurrency. When the number of clients exceeds this amount, all other clients must wait for increasingly longer periods of time before being accepted into the system. Flash has a similar problem: it caps the number of simultaneous connections to 506, due to a limitation in the number of file descriptors that can be used with the *select()* system call. When the server is saturated, clients must wait for very long periods of time before establishing a connection.[2]

This effect is demonstrated in Figure 38, which shows the cumulative distribution of response times for each server with 1024 clients. Here, response time is defined as the total time for the server to respond to a given request, including time to establish a TCP connection if one has not already been made. Although all three servers have approximately the same *average* response times, the distribution is very different. Apache and Flash exhibit a greater fraction of low response times than Haboob, but have very long tails, exceeding tens of seconds for a significant percentage of requests. Note that the use of the log scale in the figure underemphasizes the length of the tail.

[2]It is worth noting that both Apache and Flash were very sensitive to the benchmark configuration, and our testing revealed several bugs leading to seriously degraded performance under certain conditions. For example, Apache's throughput drops considerably if the server, rather than the client, closes the HTTP connection. The results presented here represent the most optimistic results from these servers.

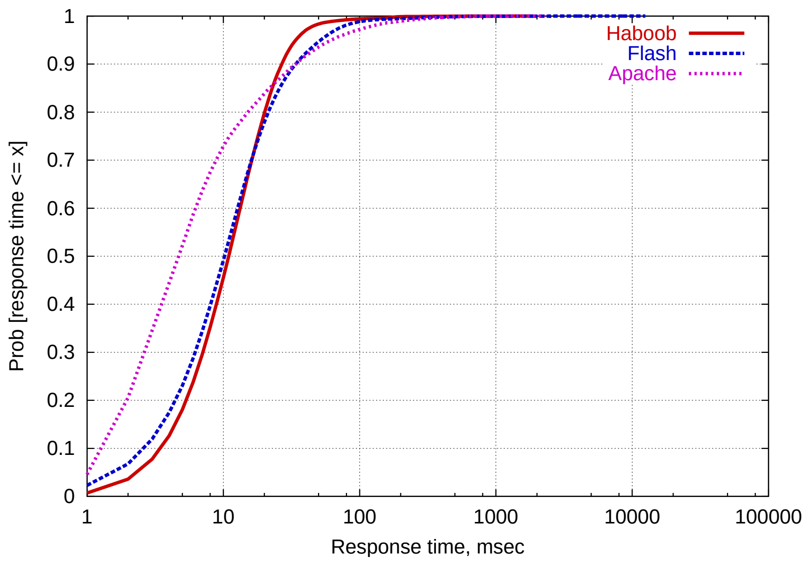


Figure 40: **Web server response times with 64 clients:** *This figure shows the cumulative response time distribution for Haboob, Apache, and Flash with 64 clients. Under light load, all three servers exhibit roughly identical response time distributions. This is in contrast to Figure 38, which shows that Apache and Flash exhibit a great deal of unfairness to clients when heavily loaded. Note the log scale on the horizontal axis.*

The maximum response time for Apache was over 93 seconds, and over 37 seconds for Flash. For comparison, Figure 40 shows the response time distribution for 64 clients, a considerably lighter load. In this case, all three servers exhibit roughly the same response time distribution.

The long tail in the response times is caused by exponential backoff in the TCP retransmission timer for establishing a new connection, which under Linux can grow to be as large as 120 seconds. With Apache, if a client is “lucky”, its connection is accepted quickly and all of its requests are handled by a single server process. Moreover, each process is in competition with only 149 other processes, which is a manageable number on most systems. This explains the large number of low response times. However, if a client is “unlucky” it will have to wait for a server process to become available; TCP retransmit backoff means that this wait time can become very large. This unequal treatment of clients is reflected in the lower value of the fairness metric for Apache.

With Flash, all clients are accepted into the system very quickly, and are subject to queueing delays within the server. Low response times in Flash owe mainly to very efficient implementation, including a fast HTTP protocol processing library; we have not implemented these optimizations in Haboob. However, the fact that Flash accepts only 506 connections at once means that under heavy load TCP backoff becomes an issue, leading to a long tail on the response time distribution. Flash's fairness metric remains high because it *eventually* services (nearly) all requests despite running out of file descriptors in the case of 512 and 1024 clients; that is, the benchmark run is not long enough to cause clients not to receive responses. We would expect the fairness metric to drop if higher loads were placed on the server.

In contrast, Haboob exhibits a great degree of fairness to clients when heavily loaded. The mean response time was 547 ms, with a maximum of 3.8 sec. This is in keeping with our goal of graceful degradation—when the server is overloaded, it should not unfairly penalize waiting requests with arbitrary wait times. Haboob rapidly accepts new client connections and allows requests to queue up within the application, where they are serviced fairly as they pass between stages. Because of this, the load is visible to the service, allowing various load conditioning policies to be applied. For example, to provide differentiated service, it is necessary to efficiently accept connections for inspection. The tradeoff here is between low average response time versus low variance in response time. In Haboob, we have opted for the latter.

2 Gnutella packet router

We chose to implement a Gnutella packet router to demonstrate the use of SEDA for non-traditional Internet services. The Gnutella router represents a very different style of service from an HTTP server: that of routing packets between participants in a peer-to-peer file sharing network. Services like Gnutella are increasing in importance as novel distributed applications are developed to take advantage of the well-connectedness of hosts across the wide area. The peer-to-peer model has been adopted by several distributed storage systems such as Freenet [32], OceanStore [79], and Intermemory [29].

Gnutella [49] allows a user to search for and download files from other Gnutella users. The protocol is entirely decentralized; nodes running the Gnutella client form an ad-hoc multihop routing network layered over TCP/IP, and nodes communicate by forwarding received messages to their neighbors. Gnutella nodes tend to connect to several (typically four or more) other nodes at once, and the initial discovery of nodes on the network is accomplished through a well-known host. There are five message types in Gnutella: *ping* is used to discover other nodes on the network; *pong* is a response to a ping; *query* is used to search for files being served by other Gnutella hosts; *queryhits* is a response to a query; and *push* is used to allow clients to download files through a firewall. The packet router is responsible for broadcasting received *ping* and *query* messages to all other neighbors, and routing *pong*, *queryhits*, and *push* messages along the path of the corresponding *ping* or *query* message. Details on the message formats and routing protocol can be found in [49].

2.1 Architecture

The SEDA-based Gnutella packet router is implemented using 3 stages, in addition to those of the asynchronous sockets layer. The *GnutellaServer* stage accepts TCP connections and parses packets, passing complete packet events to the *GnutellaRouter* stage. *GnutellaRouter* performs actual packet routing, sending each received packet to the appropriate outgoing connection(s), and maintains routing tables. *GnutellaCatcher* is a helper stage used to join the Gnutella network by contacting a well-known site to receive a list of hosts to connect to. It attempts to maintain at least 4 simultaneous connections to the network, in addition to any connections established by other Internet clients.

Joining the "live" Gnutella network and routing packets allows us to test SEDA in a real-world environment, as well as to measure the traffic passing through the router. During one 37-hour run, the router processed 24.8 million packets (with an average of 179 packets/sec) and received 72,396 connections from other hosts on the network, with an average of 12 simultaneous connections at any given time. We have measured the router as being capable of sustaining over 20,000 packets a second.

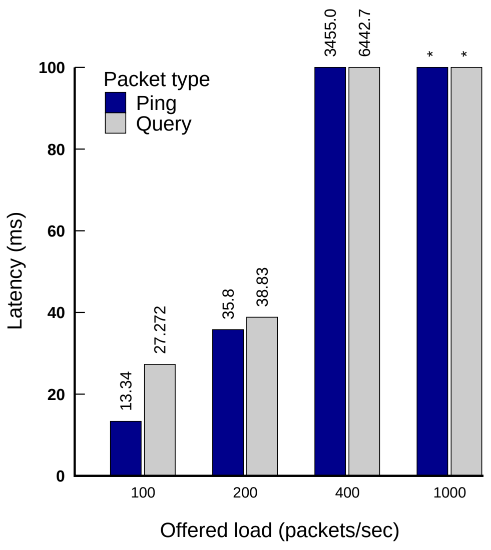
2.2 Protection from slow sockets

Our original packet router prototype exhibited an interesting bug: after several hours of correctly routing packets through the network, the server would crash after running out of memory. Using the Sandstorm profiling tools and observing the various stage queue lengths allowed us to easily detect the source of the problem: a large number of outgoing packets were queueing up for certain client connections, causing the queue length (and hence memory usage) to become unbounded. We have measured the average packet size of Gnutella messages to be approximately 32 bytes; a packet rate of just 115 packets per second can saturate a 28.8-kilobit modem link, still commonly in use by many users of the Gnutella software.

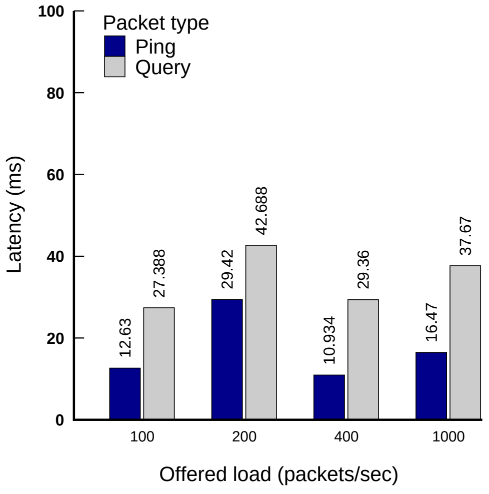
The solution in this case was to impose a threshold on the outgoing packet queue for each socket, and close connections that exceed their threshold. This solution is acceptable because Gnutella clients automatically discover and connect to multiple hosts on the network; the redundancy across network nodes means that clients need not depend upon a particular host to remain connected to the network.

2.3 Load conditioning behavior

To evaluate the use of Sandstorm's resource controllers for load conditioning, we introduced a deliberate bottleneck into the Gnutella router, in which every query message induces a servicing delay of 20 ms. This is accomplished by having the application event handler sleep for 20 ms when a query packet is received. We implemented a load-generation client that connects to the server and generates streams of packets according to a distribution approximating that of real Gnutella traffic. In our Gnutella traffic model, query messages constitute 15% of the generated packets. With a single thread performing packet routing, it is clear that as the number of packets flowing into the server increases, this delay will cause large backlogs for other messages.



(a) Using single thread



(b) Using thread pool controller

Figure 41: **Gnutella packet router latency:** *These graphs show the average latency of ping and query packets passing through the Gnutella packet router with increasing incoming packet rates. Query packets (15% of the packet mix) induce an artificial server-side delay of 20 ms. (a) shows the latency with a single thread processing packets. Note that the latency increases dramatically as the offered load exceeds server capacity; at 1000 packets/sec, the server ran out of memory before a latency measurement could be taken. (b) shows the latency with the thread pool controller enabled. Note that for 100 and 200 packets/sec, no threads were added to the application stage, since the event queue never reached its threshold value. This explains the higher packet latencies compared to 400 and 1000 packets/sec, for which 2 threads were added to the stage.*

Figure 41(a) shows the average latencies for ping and query packets passing through the server with an offered load increasing from 100 to 1000 packets/sec. Both the client and server machines use the same configuration as in the HTTP server benchmarks (see page 125). Packet latencies increase dramatically when the offered load exceeds the server's capacity. In the case of 1000 packets/sec, the server crashed before a latency measurement could be taken. This is because the enormous backlog of incoming packets caused the server to run out of memory.

A number of load conditioning policies could be employed by the router to deal with such an overload situation. A simple policy would be to threshold each stage's incoming event queue and drop packets when the threshold has been exceeded. Alternately, an approach similar to that used in Random Early Detection (RED) congestion avoidance schemes [43] could be used, where packets

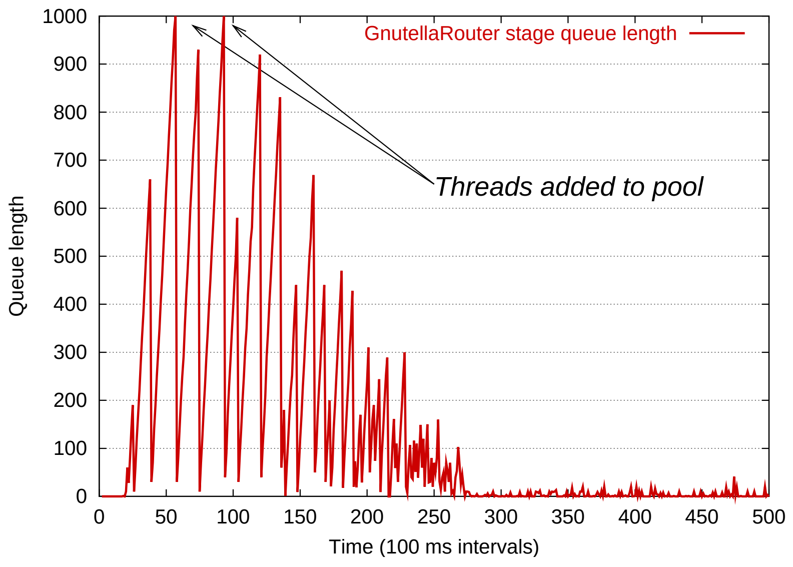


Figure 42: **Thread pool controller operation in the Gnutella packet router:** *This figure shows the queue length of the* GnutellaRouter *stage over time for a load of 1000 packets/sec, with the thread pool controller active. As the figure shows, the controller added a thread to the stage at each of the two points, which alleviated the overload situation.*

are dropped probabilistically based on the length of the input queue. Although these policies cause many packets to be dropped during overload, due to the lossy nature of Gnutella network traffic this may be an acceptable solution. An alternate policy would be to admit all packets into the system, but have the application event handler filter out query packets, which are the source of the overload. Yet another policy would be to bound the rate at which the *asyncSocketIO* layer reads incoming packets from the network.

An alternate approach is to make use of SEDA's resource controllers to overcome the bottleneck automatically. In this approach, the thread pool controller adds threads to the *GnutellaRouter* stage when it detects that additional concurrency is required. Figure 41(b) shows the average latencies in the Gnutella router with the SEDA thread pool controller enabled and an incoming packet load of 1000 packets/sec. As shown in Figure 42, the controller adds 2 threads to the *GnutellaRouter* thread pool, allowing the server to handle the increasing load despite the bottleneck. Note that the number of threads required matches the theoretical value obtained from Little's law (see

Chapter 4): If we model the stage as a queueing system with n threads, an average packet arrival rate of λ, a query packet frequency of p, and a query servicing delay of L seconds, then the number of threads needed to maintain a completion rate of λ is $n = \lambda pL = (1000)(0.15)(20 \text{ ms}) = 3$ threads.

3 Arashi: A dynamic Web-based e-mail service

Our results with the Haboob Web server and Gnutella packet router mainly serve to demonstrate the scalability of the SEDA framework under a range of loads. However, another axis that we wish to investigate deals with mechanisms for managing overload, especially in a highly dynamic environment. To explore this aspect of SEDA, we have developed the *Arashi* Web-based e-mail service.[3] Arashi is akin to Hotmail or Yahoo! Mail, allowing users to access e-mail through a Web browser interface with various functions: managing e-mail folders, deleting and refiling messages, searching for messages, and so forth. Arashi uses the Haboob Web server as its core, but includes significant new functionality to support dynamic page generation, SSL/TLS protocol connectivity, and database access.

3.1 Arashi architecture

Arashi is built as a SEDA application consisting of 17 stages, as shown in Figure 43. Apart from the 11 stages provided by Sandstorm and Haboob, Arashi employs six additional stages to process dynamic page requests. These dynamic pages are each driven by a script that accesses e-mail data from a back-end MySQL [102] database and generates a customized HTML page in response. This design mimics standard "three-tier" Web applications that consist of a Web server front-end, a database back-end, and an *application server* middle tier that performs service-specific scripting. In this case, Arashi acts as both the Web server and the middle tier.

A typical user view of the Arashi service is shown in Figure 44. To use Arashi, a user first logs into the service with a username and password. The user's list of message folders is then

[3]*Arashi* is the Japanese word for *storm*.

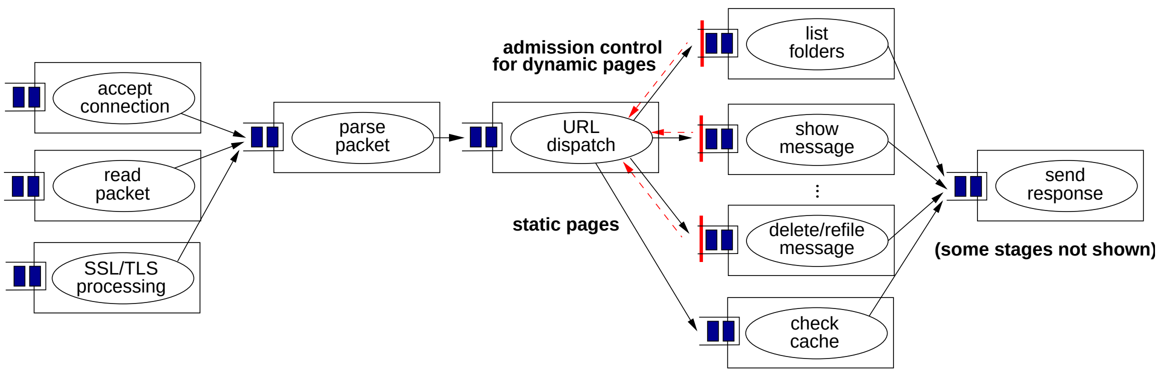
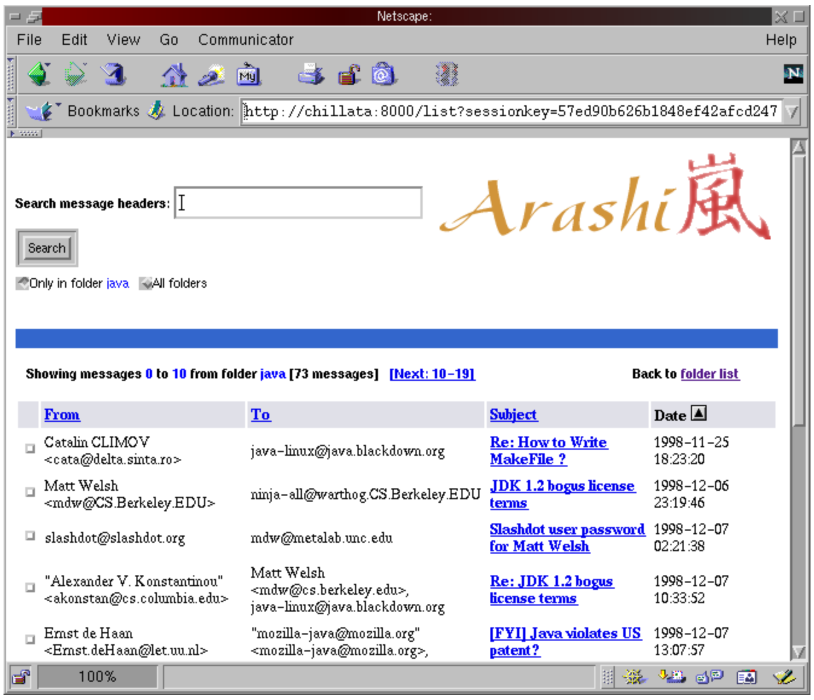


Figure 43: **Architecture of the Arashi e-mail service:** *Arashi is based on the Haboob Web server, and shares many stages with it. The* HttpRecv *stage directs requests either to the Haboob page cache (for static pages) or to one of several dynamic page stages. Dynamic pages are implemented in* PyTeC, *a Python-based scripting language. Each request type is handled by a separate stage, and these stages are subject to admission control to prevent overload.*

displayed. Clicking on a folder lists the first 10 messages in the folder; the user can navigate through the messages with next/previous page links. Clicking on a message shows the message headers and contents, with all embedded URLs transformed into clickable links. On each page the user has the option of deleting or refiling a message (either the currently displayed message or those selected from the list view with checkboxes). Each page also has a search box that performs a substring match on all message headers either in the currently selected folder or across all folders.

There are nine user request types: login, folder list, message list, show message, delete message, refile message, search, create folder, and delete folder. Each request type is processed by its own stage according to the URL of the request; for example, the URL prefix `/folderlist` indicates a folder list operation. In this case, when a user accesses the list of mail folders, the `/folderlist` stage retrieves the list of folders from the database, and generates an HTML table with one row per folder, translating each internal folder ID to its user-assigned folder name. Isolating each request type into its own stage allows overload control to be applied on a per-request-type basis, as we will see in Chapter 7. Request handling for folder creation/deletion and message refile/deletion are coalesced into a single stage.


Netscape:
File Edit View Go Communicator Help
Bookmarks Location: http://chillata:8000/list?sessionkey=57ed90b626b1848ef42afcd247
Arashi 嵐
Search
100%

Search message headers:

Only in folder java All folders

Showing messages 0 to 10 from folder java [73 messages] [Next: 10-19] Back to folder list

From	To	Subject	Date
Catalin CLIMOV <cata@delta.sinta.ro>	java-linux@java.blackdown.org	Re: How to Write MakeFile ?	1998-11-25 18:23:20
Matt Welsh <mdw@CS.Berkeley.EDU>	ninja-all@warthog.CS.Berkeley.EDU	JDK 1.2 bogus license terms	1998-12-06 23:19:46
slashdot@slashdot.org	mdw@metalab.unc.edu	Slashdot user password for Matt Welsh	1998-12-07 02:21:38
"Alexander V. Konstantinou" <akonstan@cs.columbia.edu>	Matt Welsh <mdw@cs.berkeley.edu>, java-linux@java.blackdown.org	Re: JDK 1.2 bogus license terms	1998-12-07 10:33:52
Ernst de Haan <Ernst.deHaan@let.uu.nl>	"mozilla-java@mozilla.org" <mozilla-java@mozilla.org>,	[FYI] Java violates US patent?	1998-12-07 13:07:57

Figure 44: **Screenshot of the Arashi e-mail service:** *Arashi allows users to read e-mail through a Web browser interface. Many traditional e-mail reader features are implemented, including message search, folder view, sorting message lists by author, subject, or date fields, and so forth.*

User authentication is performed through a simple username/password scheme when the user initially logs into the Arashi service. The TLS protocol may also be used to secure the client connection to the server. Upon logging in, the service generates a unique 64-bit *session key* that is automatically appended to each link in the HTML code generated by the service. The session key is used to authenticate subsequent requests to the service, avoiding the overhead of password exchange for each request. The session key also prevents the leakage of password information from the service. For example, should the user click on an external link from an e-mail message, the HTTP *Referer* field would contain only the session key, a nonce that cannot be used by a subsequent user to gain access to the service. This authentication model closely resembles that used by a wide range of e-commerce sites.

3.2 PyTeC service construction language

To simplify the development of the Arashi service logic, we developed a new scripting language called *PyTeC*, which is based on Python [91].[4] PyTeC is similar to Java Server Pages [126] and PHP [134], allowing application logic to be embedded within HTML templates. The service author writes an HTML page with embedded Python commands; these commands can perform functions such as database access or calls to library routines implemented in Python or Java. Python makes an excellent service scripting language, as it supports a wealth of text-based operations such as regular expressions.

The PyTeC compiler first generates a Python source file from each PyTeC (HTML plus Python code) source file; this Python code is then compiled to Java bytecode using Jython [133]. The resulting Java class implements a generic dynamic page handler interface supported by Haboob, and is wrapped in its own stage. Therefore, each PyTeC script is compiled into a Sandstorm stage that can be directly linked into the service. The great majority of the Arashi service logic is implemented in PyTeC, and only a few support routines were implemented in Java. Figure 45 shows an example of the PyTeC code for displaying an e-mail message in Arashi.

3.3 Database connection pooling

Establishing connections to the back-end database is typically an expensive operation involving instantiation of several classes, opening a socket, and authenticating the Arashi application to the database server. Arashi's PyTeC components make use of database connection pooling to avoid the overhead of creating and destroying database connection contexts for each request. This is accomplished through a simple mechanism whereby a PyTeC component creates a *DBConnection* object, which does one of two things: either allocates a connection from a pool, or establishes a new connection to the database if the pool is empty. When the component is finished with the connection, it is returned to the pool for another component to use. In the current implementation,

[4]The PyTeC language was primarily designed and implemented by Eric Wagner.

```
<html><body>
<a href="about.html"><img src="arashilogotrans.gif"></a>
<%include ARASHI-HEADER.pyh>

<%pytec
# ’req’ is the user request passed to this page, and
# _session is the session data, extracted from the request
# by including ARASHI-HEADER.pyh above

userid = _session.getValue("userid")
msgid = req.getQuery("msgid")
if not Arashi.validate\_message(userid,msgid):
  return error("User "+userid+" not allowed to access message "+msgid)

# Fetch the message contents from the database
conn = DBConnection()
results = conn.execute("select * from msgtable,foldertable
  where msgtable.msgid="+msgid+" and foldertable.msgid=msgtable.msgid")

while results.next():
  folderid = results.getString("folderid")
  folder = Arashi.folderid_to_name(folderid)

  # makeURL() generates a URL with the user’s session key appended
  folderurl = _session.makeURL("/list") + "&folderid="+folderid
  print ’<b>Folder:</b> <a href="’+furl+’">’+folder+’</a></font><br>’

  # make_safe escapes HTML special characters
  to = Arashi.make_safe(results.getString("header_to"))
  print "<b>To:</b>  "+to+"<br>"
  froms = Arashi.make_safe(results.getString("header_from"))
  print "<b>From:</b>  "+froms+"<br>"
  subject = Arashi.make_safe(results.getString("header_subject"))
  print "<b>Subject:</b>  "+subject+"<br>"
  date = Arashi.make_safe(results.getString("date"))
  print "<b>Date:</b>  "+date+"<br></td>"
  body = results.getBytes("body")

  # mail is stored in Base64 format in the database;
  # url_subst adds an HTML link to any URL that appears in the body
  # of the message.
  bodyStr = Arashi.url_subst(Arashi.base64_decode(body))
  if (bodyStr != None):
    print ’<p><pre>’+bodyStr+’</pre><p>’

conn.close()
%>
</body></html>
```

Figure 45: **Simplified PyTeC source code for an Arashi request:** *This figure shows the source code for an Arashi “message display” operation. The code consists of Python embedded in HTML, which performs database access operations as well as calls to Python and Java libraries.*

next state →	*login*	*list folders*	*list msgs*	*show msg*	*delete*	*refile*	*search*
from: *login*	—	1.0	—	—	—	—	—
from: *list folders*	—	—	1.0	—	—	—	—
from: *list msgs*	0.27	—	—	0.68	0.02	0.02	0.01
from: *show msg*	0.03	—	0.58	—	0.18	0.20	0.01
from: *search*	0.27	—	—	0.68	0.02	0.02	0.01

Figure 46: **State transition probabilities for the Arashi client load generator:** *This table shows the probability of an emulated user transitioning from a given state (listed in the left column) to a new state (in the top row). These probabilities are based on traces from the Berkeley departmental IMAP server. The set of possible transitions out of a given state is limited by the request types that a user can generate from that state; for example, after listing the set of folders, the only possible transition is to select a new folder. For the* delete *and* refile *states, the user always transitions to the previous page visited.*

idle connections are not removed from the pool, but a simple garbage-collection mechanism would be easy to implement.

3.4 Benchmark configuration

Arashi was designed primarily to evaluate the overload control mechanisms presented in Chapter 7, so we are not concerned strictly with peak performance, but rather the shape of the performance curve under increasing load. By running the Arashi service on the same machine as the database engine, it is easy to drive Arashi into overload; here we present basic performance measurements that demonstrate this fact. In Chapter 7 we present a family of mechanisms for managing overload in Arashi.

The Arashi client load generator uses Sandstorm's nonblocking sockets interface to make a varying number of simultaneous connections to the server, with each connection corresponding to one client. Each emulated user has a single mail folder consisting of between 1 and 12794 messages, the contents of which are derived from the author's own e-mail archives. The client load generator parses the HTML content returned by the Arashi service and scans it for specially marked tags indicating potential links to follow from each page. Emulated users access the service based on a simple Markovian model of user behavior derived from traces of the UC Berkeley Computer Science

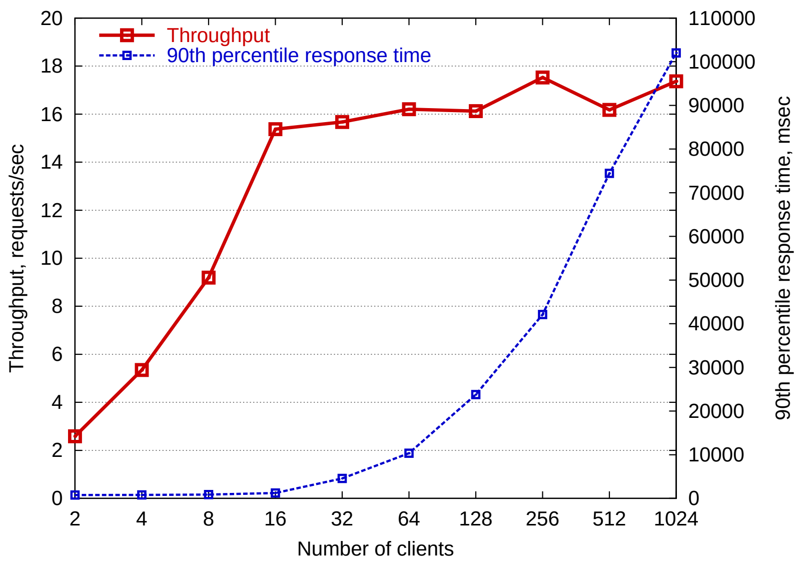


Figure 47: **Arashi server performance:** *This figure shows the throughput (in terms of requests per second) and 90th-percentile response time for the Arashi e-mail service as a function of increasing user load. As the figure shows, a modest load is able to saturate the service, leading to increasingly large response times as load increases beyond the saturation point. Throughput does not degrade as user load is increased beyond saturation.*

Division's IMAP server.[5] Figure 46 shows the state transition probabilities made by emulated users. Each of the states (login, list folders, and so forth) correspond to each type of request that a user can generate. Note that creation and deletion of folders is not included in this benchmark.

For each request, the client load generator records the response time as well as the time required to establish a TCP connection to the service if one was not already made. The overall throughput (in terms of requests per second and megabits/sec) is also recorded. As in the Haboob measurements, the inter-request think time is 20ms.

3.5 Throughput and response time measurements

Figure 47 shows the throughput and 90th-percentile response time of the Arashi service as load increases from 1 to 1024 users. With a modest load of 128 users, the service becomes

[5]We are indebted to Steve Czerwinski for providing the IMAP trace data.

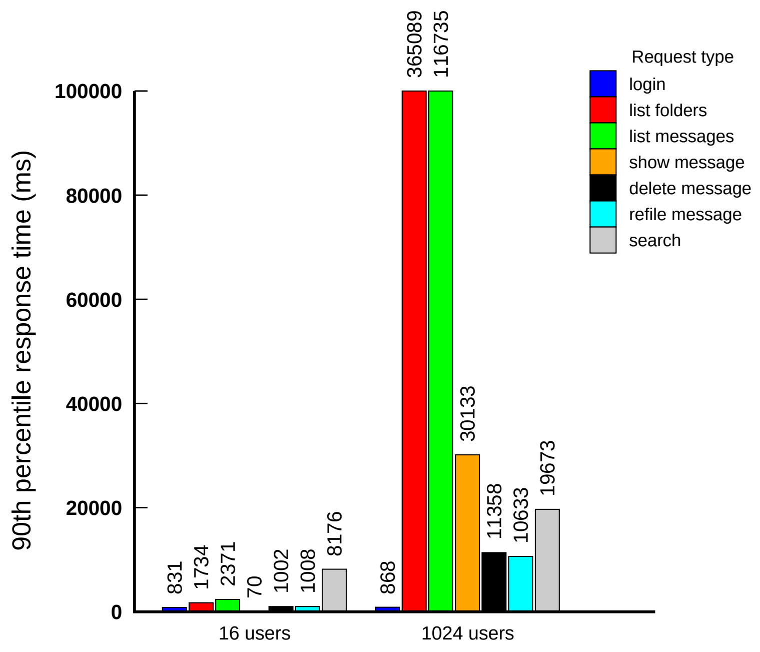


Figure 48: **Response time based on request type:** *This figure shows the 90th-percentile response time for each request type in the Arashi e-mail service for loads of 16 and 1024 users. The response time depends heavily on the complexity of the user request; while login requests are processed quickly, searching messages for a small string is very resource-intensive.*

saturated, leading to increasingly large response times as load continues to increase. Arashi exhibits well-conditioned performance as load increases, with throughput saturating at a load of around 16 clients. Also, throughput does not degrade as the user load is increased beyond saturation.

In Arashi, the primary performance bottleneck is the MySQL database engine, as well as the dynamic page generation code that performs database access and HTML rendering. As Figure 48 shows, the response time depends greatly on the type of user request. As load increases, listing the set of folders and the contents of a particular folder exhibit much higher response times than other requests. This is for two reasons: first, these are the most popular operations, and second, they are relatively complex. For example, listing the contents of a folder involves a number of database accesses to fetch the headers of each message in the folder. Listing the folders themselves requires several database operations to map from internal folder ID to the user-assigned folder name.

We have intentionally not made any effort to optimize Arashi to improve performance, as this service is designed to serve as a vehicle for exploring overload management techniques in Chapter 7. If this service were intended for actual deployment, a fair amount of performance gain could be obtained using standard techniques such as caching the HTML pages and database results produced by each PyTeC script.

4 Summary

This chapter has presented a design and performance study of three interesting applications built using the SEDA model: a high-performance Web server, a Gnutella packet router, and a complex e-mail service. Through these applications, we have demonstrated that the SEDA model enables the construction of services that scale to very heavy user loads, and behave gracefully as load exceeds service capacity.

A critical aspect of the Haboob Web server design is that it rapidly accepts all TCP connections to avoid the response-time penalty of SYN retransmit backoff. This approach also yields a great deal of fairness to clients, as all requests are given equal opportunity to be admitted to and processed by the system. The Gnutella packet router demonstrated the use of thread pool management to overcome performance bottlenecks, as well as the use of outgoing socket thresholding to avoid memory leaks due to connections from wide-area clients. Finally, the Arashi e-mail service represents a complex application with dynamic Web page generation, database access, and a wide variation in resource demands for each request.

The results here serve mainly to highlight the scalability of the SEDA platform and demonstrate uses of the programming model. In the next chapter we explore the other primary goal of SEDA: to enable flexible management of overload conditions.

Chapter 7

Adaptive Overload Management

Previous chapters have focused on the efficiency and scalability of the SEDA architecture with respect to current server designs, and have discussed the use of dynamic control of system resources for load conditioning and automatic tuning. In this chapter we concentrate on the use of adaptive, per-stage admission control as a means for managing extreme overload.

We begin by discussing the role of feedback-driven control for overload management, contrasting this approach to more traditional resource control mechanisms, such as static resource containment and prioritization. We also discuss some of the difficulties of applying classic control theory to dynamic Internet services, including the lack of good system models and nonlinear dynamics.

Next, we develop an adaptive admission control mechanism that attempts to bound the 90th-percentile response time of requests flowing through a SEDA application. We extend this technique to incorporate application-specific service degradation as well as service differentiation across multiple request or user classes. These techniques are evaluated using two applications: the Arashi e-mail service described in the previous chapter, and a Web server benchmark involving dynamic page generation. These results show that feedback-driven admission control is an effective way to manage overload in SEDA-based services.

1 Background

Chapter 2 discussed various approaches to overload management in Internet services, many of which focused on static resource containment, in which *a priori* resource limits are imposed on an application or service to avoid overcommitment. As discussed previously, these approaches are inflexible in that they can underutilize resources (if limits are set too low) or lead to overload (if limits are set too high). Also, these mechanisms are typically concerned with resource allocation of coarse-grained entities such as processes or sessions. In Internet services, it is more important to meet a statistical performance target across a large number of requests; this model does not map well onto techniques such as process-based resource containment or scheduling.

We argue that the right approach to overload management in Internet services is feedback-driven control, in which the system actively observes its behavior and applies dynamic control to manage resources [146]. As discussed in Chapter 2, several systems have explored the use of closed-loop overload management in Internet services, such as limiting incoming packet rates [69, 138] or allocating server processes to requests to meet performance targets [89]. These mechanisms are approaching the kind of overload management techniques we would like to see in Internet services, yet they are inflexible in that the application itself is not designed to manage overload.

Our approach builds upon these ideas by introducing per-stage adaptive admission control into the SEDA framework. Applications are explicitly signaled of overload conditions through enqueue failures. Moreover, applications are given the responsibility to react to overload conditions in some way, such as by shedding load or degrading the quality of service. In this way, SEDA makes overload management a first-class application design primitive, rather than an operating system function with generic load-shedding policies.

1.1 Performance metrics

A variety of performance metrics have been studied in the context of overload management, including throughput and response time targets [27, 28], CPU utilization [3, 30, 37], and differentiated service metrics, such as the fraction of users in each class that meet a given performance target [72, 89]. We focus on *90th-percentile response time* as a realistic and intuitive measure of client-perceived system performance, defined as follows: if the 90th percentile response time is t, then 90% of the requests experience a response time equal to or less than t. This metric has the benefit that it is both easy to reason about and captures administrators' (and users') intuition of Internet service performance. This is as opposed to average or maximum response time (which fail to represent the "shape" of a response time curve), or throughput (which depends greatly on the user's connection to the service and has little relationship with user-perceived performance).

In this context, the system administrator specifies a target value for the 90th-percentile response time exhibited by requests flowing through the service.

The target value may be parameterized by relative utility of the requests, for example, based on request type or user classification. An example might be to specify a lower response time target for requests from users with more items in their shopping cart. Our current implementation, discussed below, allows separate response time targets to be specified for each stage in the service, as well as for different classes of users (based on IP address, request header information, or HTTP cookies).

1.2 Overload exposure through admission control

As discussed earlier, each stage in a SEDA application has an associated admission controller that guards access to the event queue for that stage. The admission controller (implemented as an enqueue predicate) is invoked upon each enqueue operation on a stage and may either accept or reject the given request. Numerous admission control strategies are possible, such as simple thresholding, rate limiting, or class-based prioritization. Additionally, the application may specify its own admission control policy if it has special knowledge that can drive the load conditioning decision.

When the admission controller rejects a request, the corresponding enqueue operation fails, indicating to the originating stage that there is a bottleneck in the system. Applications are therefore responsible for reacting to these “overload signals” in some way. More specifically, overload management is the responsibility of the upstream stage when an enqueue rejection occurs.

A wide range of overload management policies are available to the application. The simplest response is to block until the downstream stage can accept the request. This leads to backpressure, since blocked threads in a stage cause its incoming queue to fill, triggering overload response upstream. However, backpressure may be undesirable as it causes requests to queue up, possibly for long periods of time. Under high loads, queueing backlogged requests may also consume an excessive amount of memory. Another overload response is to drop the request. Depending on the application, this might involve sending an error message to the client or using an HTTP redirect message to bounce the request to another node in a server farm.

More generally, an application may *degrade service* in response to overload, allowing a larger number of requests to be processed albeit at lower quality. Examples include delivering lower-fidelity content (e.g., reduced-resolution image files) or performing alternate actions that consume fewer resources per request. Whether or not degradation is feasible depends greatly on the nature of the service. The SEDA framework itself is agnostic as to the precise degradation mechanism employed—it simply provides the adaptive admission control primitive to signal overload to applications.

2 Overload control mechanisms

In this section we describe three overload control mechanisms for SEDA: adaptive admission control for bounding 90th-percentile response times, service degradation, and class-based service differentiation.

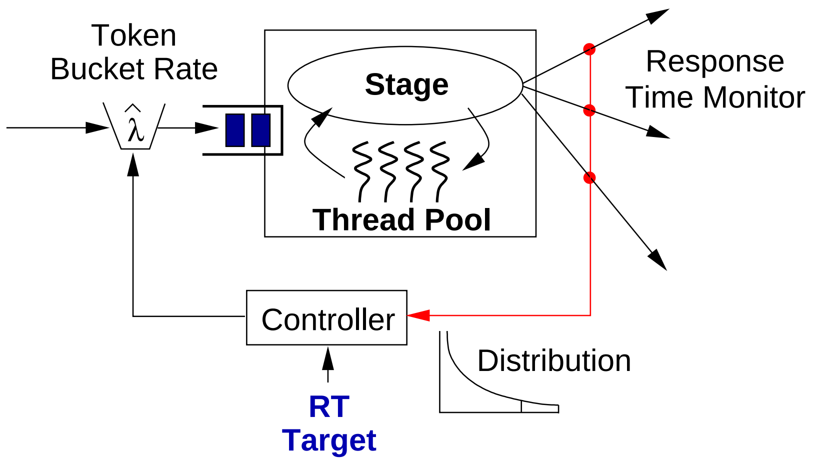


Figure 49: **Response time controller design:** *The controller observes a history of response times through the stage, and adjusts the rate at which the stage accepts new requests to meet an administrator-specified 90th-percentile response time target.*

2.1 Response time controller design

The design of the per-stage overload controller in SEDA is shown in Figure 49. The controller consists of several components. A *monitor* measures response times for each request passing through a stage. The measured 90th-percentile response time over some interval is passed to the *controller* that adjusts the *admission control parameters* based on the administrator-supplied response-time *target*. In the current design, the controller adjusts the rate at which new requests are admitted into the stage's queue by adjusting the rate at which new tokens are generated in a token bucket traffic shaper [110].

The basic overload control algorithm makes use of additive-increase/multiplicative-decrease tuning of the token bucket rate based on the current observation of the 90th-percentile response time. The controller is invoked by the stage's event-processing thread after some number of requests (*nreq*) has been processed. The controller also runs after a set interval (*timeout*) to allow the rate to be adjusted when the processing rate is low.

The controller records up to *nreq* response-time samples and calculates the 90th-percentile sample *samp* by sorting the samples and taking the $(0.9 \times nreq)$-th sample. In order to prevent

Parameter	**Description**	**Default value**
target	90th-percentile RT target	Set by administrator
nreq	# requests before controller executed	100
timeout	Time before controller executed	1 sec
α	EWMA filter constant	0.7
err_i	% error to trigger increase	-0.5
err_d	% error to trigger decrease	0.0
adj_i	Additive rate increase	2.0
adj_d	Multiplicative rate decrease	1.2
c_i	Constant weight on additive increase	-0.1
$rate_{min}$	Minimum rate	0.05
$rate_{max}$	Maximum rate	5000.0

Figure 50: **Parameters used in the response time controller.**

sudden spikes in the response time sample from causing large reactions in the controller, the 90th-percentile response time estimate is smoothed using an exponentially weighted moving average with parameter α:

$$cur = \alpha \cdot cur + (1 - \alpha) \cdot samp$$

The controller then calculates the error between the current response time measurement and the target:

$$err = \frac{cur - target}{target}$$

If $err > err_d$, the token bucket rate is reduced by a multiplicative factor adj_d. If $err < err_i$, the rate is increased by an additive factor proportional to the error: $adj_i \times -(err - c_i)$. The constant c_i is used to weight the rate increase such that when $err = c_i$ then the rate adjustment is 0.

The parameters used in the implementation are summarized in Figure 50. These parameters were determined experimentally using a combination of microbenchmarks with artificial loads and real applications with realistic loads (e.g., the Arashi e-mail service). In most cases the controller algorithm and parameters were tuned by running test loads against a service and observing the behavior of the controller in terms of measured response times and the corresponding admission rate.

These parameters have been observed to work well across a range of applications, however, there are no guarantees that they are optimal. We expect that the behavior of the controller is sensitive to the setting of the smoothing filter constant α, as well as the rate increase adj_i and decrease adj_d. The setting of the other parameters has less of an effect on controller behavior. The main goal of tuning is allow the controller to react quickly to increases in response time, while not being so conservative that an excessive number of requests are rejected.

An important problem for future investigation is the tuning (perhaps automated) of controller parameters in this environment. As discussed earlier, it would be useful to apply concepts from control theory to aid in the tuning process, but this requires realistic models of system behavior.

2.2 Service degradation

Another approach to overload management is to allow applications to degrade the quality of delivered service in order to admit a larger number of requests while maintaining a response-time target [2, 24, 45]. SEDA itself does not implement service degradation mechanisms, but rather signals overload to applications in a way that allows them to degrade if possible. SEDA allows application code to obtain the current 90th-percentile response time measurement from the overload controller, as well as to enable or disable the admission control mechanism. This allows an application to implement degradation by periodically sampling the current response time and comparing it to the administrator-specified target. If service degradation is ineffective (say, because the load is too high to support even the lowest quality setting), the stage can re-enable admission control to cause requests to be rejected.

2.3 Class-based differentiation

By prioritizing requests from certain users over others, a SEDA application can implement various policies related to class-based service level agreements (SLAs). A common example is to prioritize requests from “gold” customers, who might pay more money for the privilege, or to give

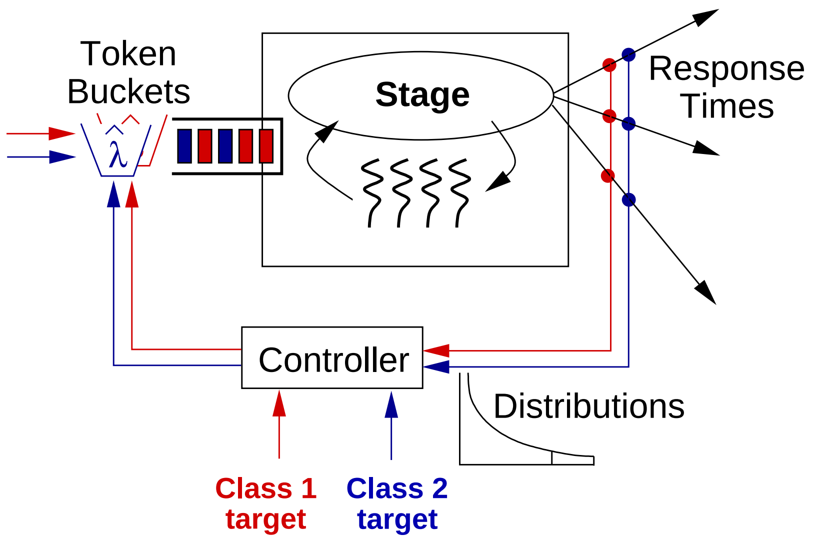


Figure 51: **Multiclass overload controller design:** *For each request class, the controller measures the 90th-percentile response time, and adjusts the rate at which the stage accepts new requests of each class. When overload is detected, the admission rate for lower-priority classes is reduced before that of higher-priority classes.*

better service to customers with items in their shopping cart.

Various approaches to class-based differentiation are possible in SEDA. One option would be to segregate request processing for each class into its own set of stages, in effect partitioning the service's stage graph into separate flows for each class. In this way, stages for higher-priority classes could be given higher priority, e.g., by increasing scheduling priorities or allocating additional threads. Another option is to process all classes of requests in the same set of stages, but make the admission control mechanism aware of each class, for example, by rejecting a larger fraction of lower-class requests than higher-class requests. This is the approach taken here.

The multiclass response time control algorithm is identical to that presented in Section 2.1, with several small modifications. Incoming requests are assigned an integer *class* that is derived from application-specific properties of the request, such as IP address or HTTP cookie information. A separate instance of the response time controller is used for each class c, with independent response time targets $target^c$. Likewise, the queue admission controller maintains a separate token bucket for each class.

For class c, if $err^c > err^c_d$, then the token bucket rate of all classes *lower than* c is reduced by a multiplicative factor $adjlo_d$ (with default value 10). If the rate of all lower classes is already equal to $rate_{min}$ then a counter lc^c is incremented; when $lc^c \geq lc_{thresh}$ (default value 20), then the rate for class c is reduced by adj_d as described above. In this way the controller aggressively reduces the rate of lower-priority classes before higher-priority classes. Admission rates are increased as in Section 2.1, except that whenever a higher-priority class exceeds its response time target, all lower-priority classes are flagged to prevent their admission rates from being increased.

3 Evaluation

We evaluate the SEDA overload control mechanisms using two applications: the Arashi e-mail service described in the previous chapter, and a Web server benchmark involving dynamic page generation that is capable of degrading service in response to overload.

Recall that the Arashi service processes nine different request types, which are handled by six separate stages (several of the request types are handled by the same stage). These stages are the bottleneck in the system as they perform database access and HTML page generation; the other stages are relatively lightweight. When the admission controller rejects a request, the HTTP processing stage sends an error message to the client indicating that the service is busy. The client records the error and waits for 5 seconds before attempting to log in to the service again.

3.1 Controller operation

Figure 52 demonstrates the operation of the overload controller, showing the 90th-percentile response time measurement and token bucket admission rate for one of the stages in the Arashi service (in this case, for the “list folders” request type). Here, the stage is being subjected to a very heavy load spike of 1000 users, causing response times to increase dramatically. Recall from Figure 47 that Arashi saturates at a load of 16 users; the offered load is therefore 62

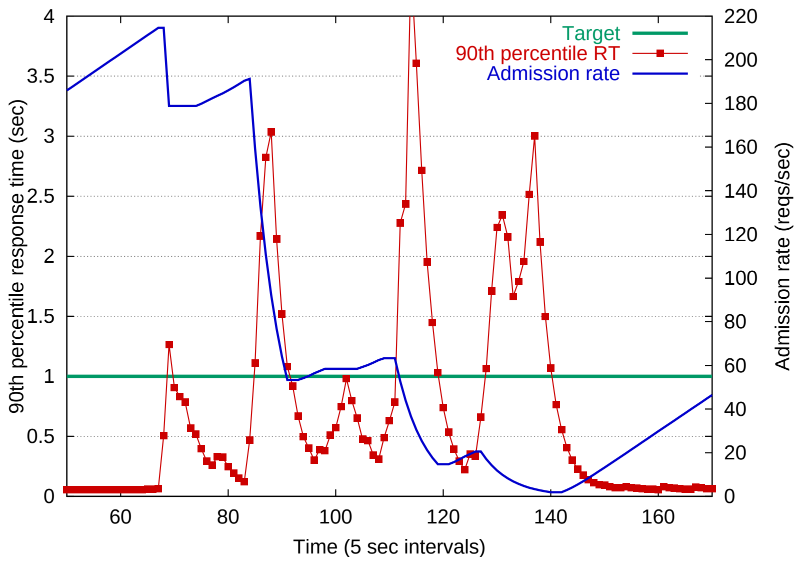


Figure 52: **Overload controller operation:** *This figure shows the operation of the SEDA overload controller for one of the stages in the Arashi e-mail service during a large load spike. A load spike of 1000 users enters the system at around $t = 70$ and leaves the system around $t = 150$. The response time target is set to 1 sec. The overload controller responds to a spike in response time by exponentially decreasing the admission rate of the stage. Likewise, when the measured response time is below the target, the admission rate is increased slowly. Notice the slight increase in the admission rate around $t = 100$; this is an example of the proportional increase of the admission rate based on the error between the response time measurement and the target. The spikes in the measured response time are caused by bursts of requests entering the stage, as well as resource contention across stages.*

times the capacity of the service.

As the figure shows, the controller responds to a spike in the response time by exponentially decreasing the token bucket rate. When the response time is below the target, the rate is increased slowly. Despite several overshoots of the response time target, the controller is very effective at keeping the response time near the target. The response time spikes are explained by two factors. First, the request load is extremely bursty due to the realistic nature of the client load generator. Second, because all stages share the same back-end database, requests for other stages (not shown in the figure) may cause resource contention that affects the response time of the "list folders" stage. Note, however, that the largest response time spike is only about 4 seconds, which is

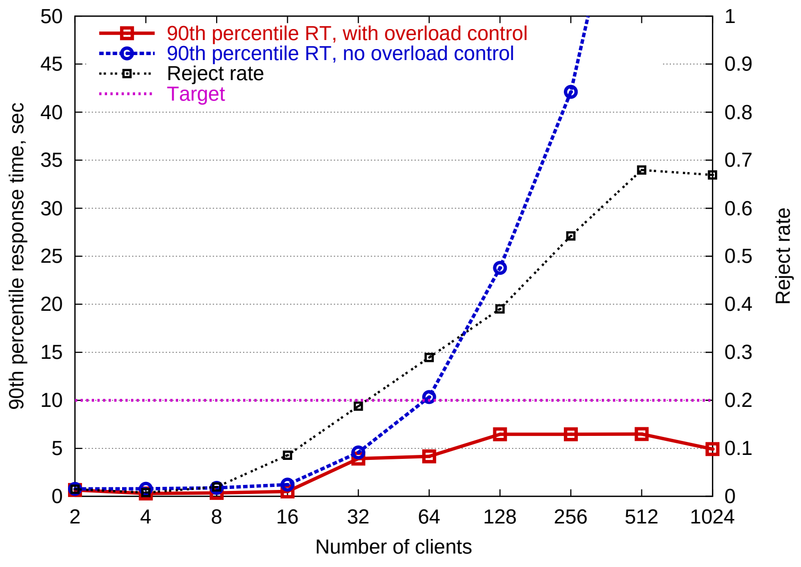


Figure 53: **Overload control in Arashi:** *This figure shows the 90th-percentile response time for the Arashi e-mail service with and without the overload controller enabled. The 90th-percentile response time target is 10 sec. Also shown is the fraction of rejected requests with overload control enabled. Note that the overload controller is operating independently on each request type, though this figure shows the 90th-percentile response time and reject rate averaged across all requests. As the figure shows, the overload control mechanism is effective at meeting the response time target despite a many-fold increase in load.*

not too serious given a response time target of 1 second. With no admission control, response times grow without bound, as we will show in Sections 3.2 and 3.3.

3.2 Overload control with increased user load

Figure 53 shows the 90th-percentile response time of the Arashi service, as a function of the user load, both with and without the per-stage overload controller enabled. Also shown is the fraction of overall requests that are rejected by the overload controller. The 90th-percentile response time target is set to 10 sec. For each data point, the corresponding number of simulated clients load the system for about 15 minutes, and response-time distributions are collected after an initial warm-up period of about 20 seconds. As the figure shows, the overload control mechanism is effective at meeting the response time target despite a many-fold increase in load, up to 1024 users.

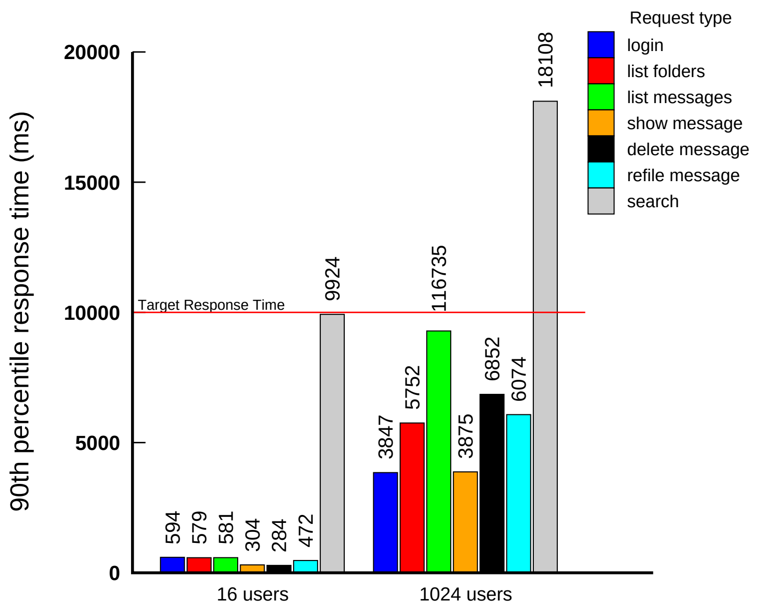


Figure 54: **Per-request-type response times with overload control:** *This figure shows the 90th-percentile response time for each request type in the Arashi e-mail service for loads of 16 and 1024 users, with the overload controller enabled using a response time target of 10 sec. Although request types exhibit a widely varying degree of complexity, the controller is effective at meeting the response time target for each type. With 1024 users, the performance target is exceeded for* search *requests, due to their relative infrequency. Compare these values with Figure 48 in Chapter 6, which shows response times without overload control enabled.*

Recall that the overload controller is operating on each request type separately, though this figure shows the 90th-percentile response time and reject rate across *all* requests. Figure 54 breaks the response times down according to request type, showing that the overload controller is able to meet the performance target for each request type individually. Compare this figure to Figure 48 in Chapter 6, which shows response times without overload control enabled. With 1024 users, the performance target is exceeded for *search* requests. This is mainly due to their relative infrequency: search requests are very uncommon, comprising less than 1% of the request load. The controller for the *search* stage is therefore unable to react as quickly to arrivals of this request type.

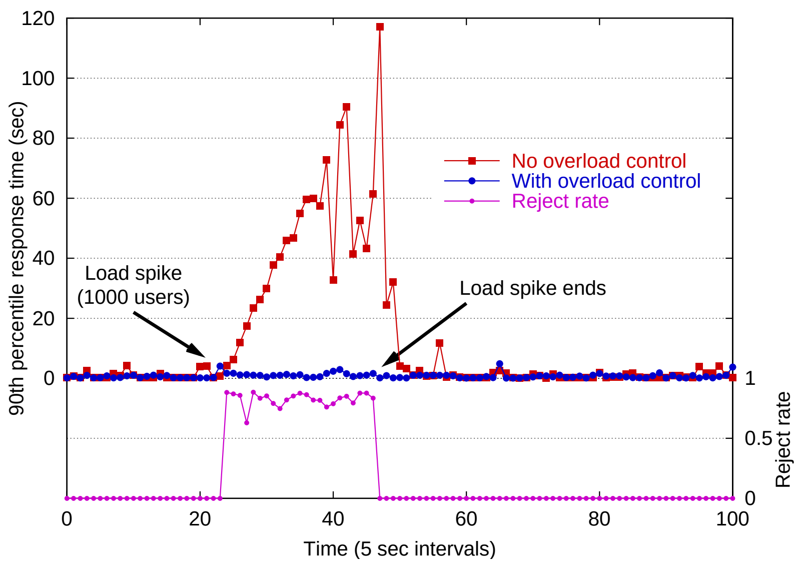


Figure 55: **Overload control under a massive load spike:** *This figure shows the 90th-percentile response time experienced by clients using the Arashi e-mail service under a massive load spike (from 3 users to 1000 users). Without overload control, response times grow without bound; with overload control (using a 90th-percentile response time target of 1 second), there is a small increase during load but response times quickly stabilize. The lower portion of the figure shows the fraction of requests rejected by the overload controller.*

3.3 Overload control under a massive load spike

The previous section evaluated the overload controller under a steadily increasing user load, representing a slow increase in user population over time. We are also interested in evaluating the effectiveness of the overload controller under a sudden load spike. In this scenario, we start with a base load of 3 users accessing the Arashi service, and suddenly increase the load to 1000 users. This is meant to model a "flash crowd" in which a large number of users access the service all at once.

Figure 55 shows the performance of the overload controller in this situation. Without overload control, there is an enormous increase in response times during the load spike, making the service effectively unusable for all users. With overload control and a 90th-percentile response time target of 1 second, about 70-80% of requests are rejected during the spike, but response times for admitted requests are kept very low.

Our feedback-driven approach to overload control is in contrast to the common technique of limiting the number of client TCP connections to the service, which does not actively monitor response times (a small number of clients could cause a large response time spike), nor give users any indication of overload. In fact, refusing TCP connections has a negative impact on user-perceived response time, as the client's TCP stack transparently retries connection requests with exponential backoff.

We claim that giving 20% of the users good service and 80% of the users some indication that the site is overloaded is better than giving *all* users unacceptable service. However, this comes down to a question of what policy a service wants to adopt for managing heavy load. Recall that the service need not reject requests outright—it could redirect them to another server, degrade service, or perform an alternate action. The SEDA design allows a wide range of policies to be implemented: in the next section we look at degrading service as an alternate response to overload.

3.4 Service degradation experiments

As discussed previously, SEDA applications can respond to overload by degrading the fidelity of the service offered to clients. This technique can be combined with admission control, for example, by rejecting requests when the lowest service quality still leads to overload.

It is difficult to demonstrate the effect of service degradation in Arashi, because there are few interesting opportunities for reduced quality of service. If an e-mail service is overloaded, there is not much that the service can do besides rejecting or redirecting requests—it is not meaningful to retrieve e-mail at a degraded quality level. Though it may be possible to reduce the expense of HTML rendering (e.g., by removing inlined images and advertisements), HTML rendering is not the bottleneck in Arashi.

Therefore, we evaluate service degradation using a simple Web server that responds to each request with a dynamically generated HTML page that requires significant resources to gener-

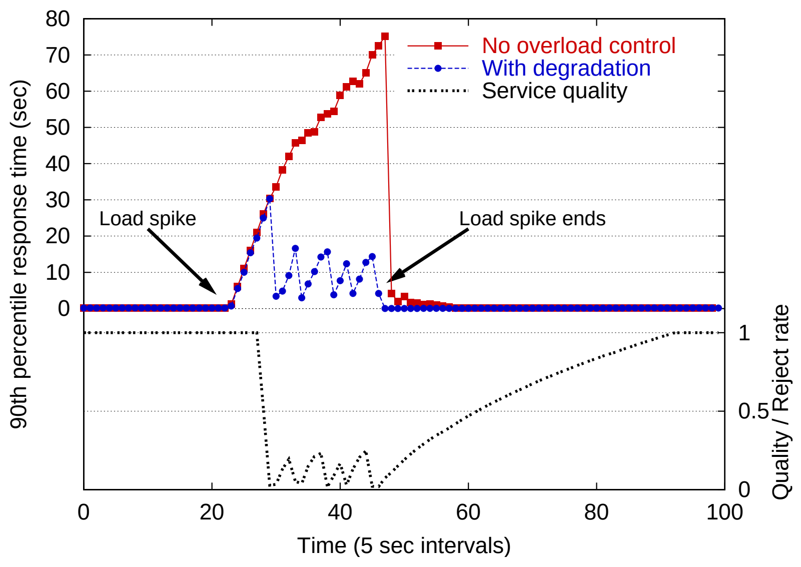


Figure 56: **Effect of service degradation:** *This figure shows the 90th-percentile response time experienced by clients accessing a simple service consisting of a single bottleneck stage. The stage is capable of degrading the quality of service delivered to clients in order to meet response time demands. The 90th-percentile response time target is set to 5 seconds. Without service degradation, response times grow very large under a load spike of 1000 users. With service degradation, response times are greatly reduced, oscillating near the target performance level.*

ate. A single stage acts as a CPU-bound bottleneck in this service; for each request, the stage reads a varying amount of data from a file, computes checksums on the file data, and produces a dynamically generated HTML page in response. The stage has an associated *quality factor* that controls the amount of data read from the file and the number of checksums computed. By reducing the quality factor, the stage consumes fewer CPU resources, but provides "lower quality" service to clients.

Using the overload control interfaces in SEDA, the stage monitors its own 90th-percentile response time and reduces the quality factor when it is over the administrator-specified limit. Likewise, the quality factor is increased slowly when the response time is below the limit. Service degradation may be performed either independently or in conjunction with the response-time admission controller described above. If degradation is used alone, then under overload all clients are given service but at a reduced quality level. In extreme cases, however, the lowest quality setting may still lead to very large response times. The stage optionally re-enables the admission controller

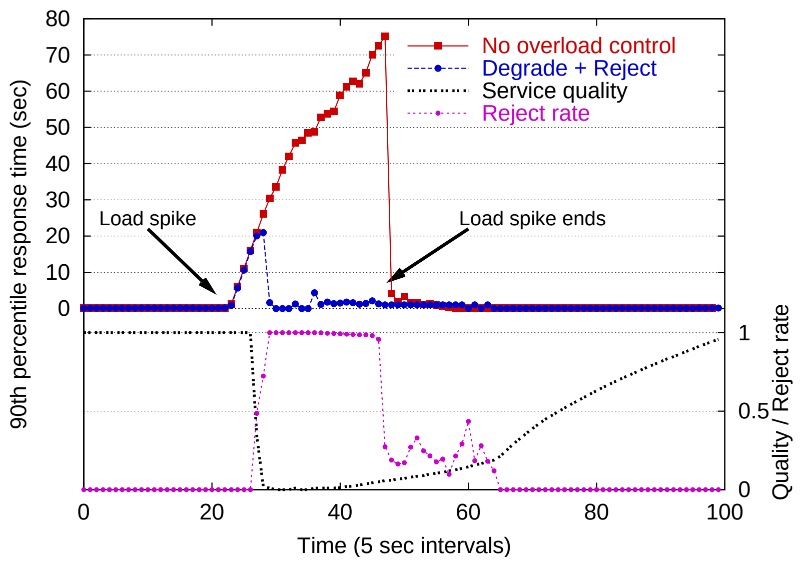


Figure 57: **Service degradation combined with admission control:** *This figure shows the effect of service degradation combined with admission control. The experiment is identical to that in Figure 56, except that the bottleneck stage re-enables admission control when the service quality is at its lowest level. In contrast to the use of service degradation alone, degradation coupled with admission control is much more effective at meeting the response time target.*

when the quality factor is at its lowest setting and response times continue to exceed the target.

Figure 56 shows the effect of service degradation under an extreme load spike, and Figure 57 shows the use of service degradation coupled with admission control. As these figures show, service degradation alone does a fair job of managing overload, though re-enabling the admission controller under heavy load is much more effective. Note that when admission control is used, a very large fraction (99%) of the requests are rejected; this is due to the extreme nature of the load spike and the inability of the bottleneck stage to meet the performance target, even at a degraded level of service.

3.5 Service differentiation

Finally, we evaluate the use of multiclass service differentiation, in which requests from lower-priority users are rejected before those from higher-priority users. In these experiments, we

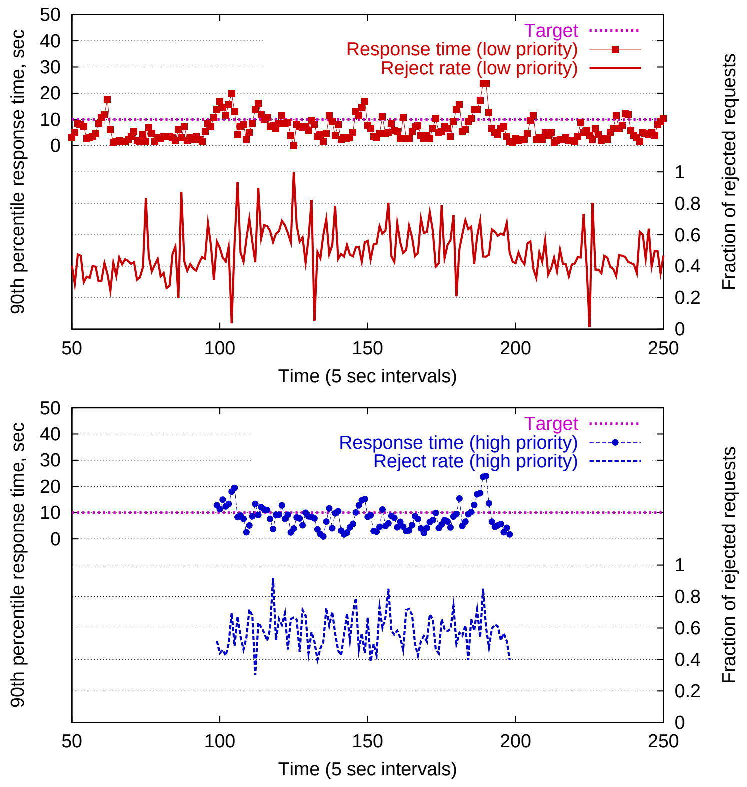


Figure 58: **Multiclass experiment without service differentiation:** *This figure shows the operation of the overload control mechanism in Arashi with two classes of 128 users each accessing the service. The high-priority users begin accessing the service at time $t = 100$ and leave at $t = 200$. No service differentiation is used, so all users are treated as belonging to the same class. The 90th-percentile response time target is set to 10 sec. The controller is able to maintain response times near the target, though no preferential treatment is given to higher-priority users as they exhibit an identical frequency of rejected requests.*

deal with two user classes, each with a 90th-percentile response time target of 10 sec, generating load against the Arashi e-mail service. Each experiment begins with a base load of 128 users from the lower-priority class. At a certain point during the run, 128 users from the higher-priority class also start accessing the service, and leave after some time. The user class is determined by a field in the HTTP request header; the implementation is general enough to support class assignment based on client IP address, HTTP cookies, or other information.

Recall that the multiclass admission controller operates by performing per-stage admis-

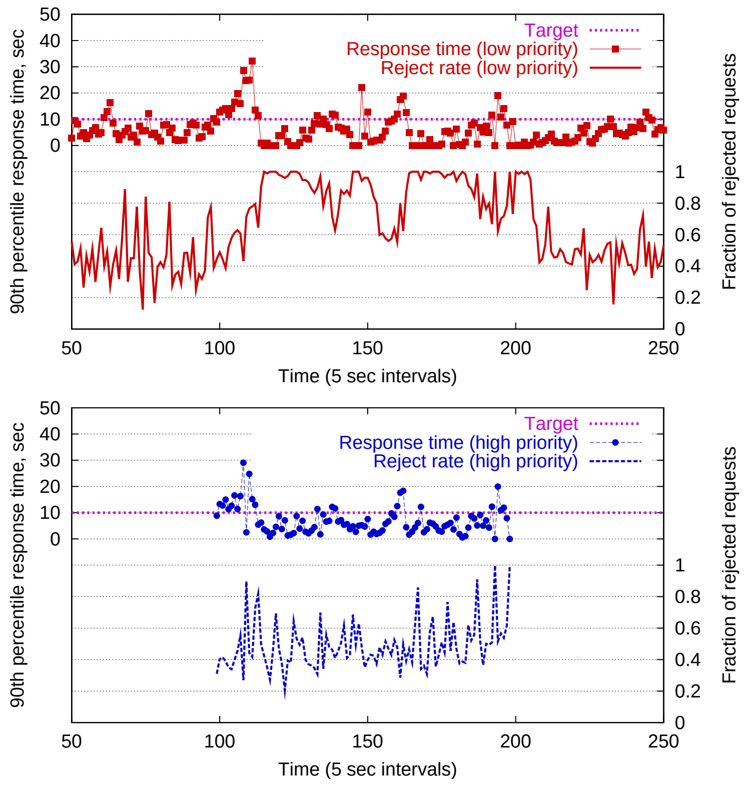


Figure 59: **Multiclass service differentiation:** *This figure shows the operation of the multiclass overload control mechanism in Arashi with two classes of 128 users each. Service differentiation between the two classes is enabled and the 90th-percentile response time target for each class is 10 sec. The high-priority users begin accessing the service at time* $t = 100$ *and leave at* $t = 200$*. As the figure shows, when the high-priority users become active, there is an initial load spike that is compensated for by penalizing the admission rate of the low-priority users. Overall the low-priority users receive a large number of rejections than high-priority users.*

sion control with a separate token bucket rate for each class of users, and the admission rate for lower-class requests is reduced before that for the higher-class requests. This scheme can be used with any number of user classes though we focus here on just two classes.

Figure 58 shows the performance of the multiclass overload controller without service differentiation enabled: all users are effectively treated as belonging to the same class. As the figure shows, the controller is able to maintain response times near the target, though the fraction of rejected requests is identical for the two classes of users; no preferential treatment is given to the high priority requests.

In Figure 59, service differentiation is enabled, causing requests from lower-priority users to be rejected more frequently than higher-priority users. As the figure demonstrates, while both user classes are active, the overall rejection rate for higher-priority users is slightly lower than that in Figure 58, though the lower-priority class is penalized with a higher rejection rate. Note also that the initial load spike (around $t = 100$) when the high priority users become active is somewhat more severe with service differentiation enabled. This is because the controller is initially attempting to reject only low-priority requests, due to the lag threshold (lc_{thresh}) for triggering admission rate reduction for high-priority requests.

4 Summary

In this chapter we have addressed the issue of adaptive overload management in SEDA, which is one of the most important aspects of the SEDA model over previous approaches to service design. We argue that the right way to address overload management is not through static resource limits or share-based allocation, but rather through feedback-driven control that actively observes and adapts the system's behavior under load. We also discussed the use of 90th-percentile response time as a metric for measuring overload.

We have designed and evaluated three overload control mechanisms for SEDA: adaptive, per-stage admission control; application-specific service degradation; and multiclass service differentiation. These mechanisms operate primarily by adapting the rate at which each stage in the system accepts requests, signaling overload to upstream stages through queue rejections. In the service degradation case, the application itself is responsible for monitoring response times and reacting to load, though it is capable of enabling and disabling the runtime-supplied admission controller as well. The evaluation of these control mechanisms, using both the complex Arashi e-mail service and a simpler dynamic Web server benchmark, show that these techniques are effective for managing load with increasing user populations as well as under a massive load spike.

Chapter 8

Lessons and Future Work

In this chapter we distill our experiences with the SEDA model and reflect on lessons learned about the architecture, discussing some of the strengths and weaknesses of the SEDA design. We also outline several areas for future work.

1 Reflections on the SEDA programming model

The SEDA design arose primarily out of a desire to simplify the development of systems making use of event-driven concurrency. Much of the initial work on SEDA was motivated by experiences with the Ninja I/O core, used by Gribble's Distributed Data Structures (DDS) [52] layer and initial versions of the vSpace [139] cluster-based service platform. As described in Chapter 2, the I/O core is a framework for event-driven concurrency based on the use of thread pools and software components composed using both direct upcalls and queues.

The most serious issue with the Ninja I/O core is that it made no distinction between upcalls (in which one component directly invokes another) and queues (in which one component enqueues an event to be processed asynchronously by another). Moreover, the upcall/enqueue interface was not permitted to reject events passed across the interface; components were therefore required to accept all new work for processing. This approach raises two important challenges for an application designer: first, whether (and how) to use upcalls or queues when composing two software components; and second, how to design components to be aware of whether an upcall or a queue is being used to integrate it into the application.

The first problem exists in a similar form in the SEDA model, in that application designers must decide how to decompose request processing into a graph of stages connected with queues. The design patterns presented in Chapter 3 are intended to serve as guidelines to address these questions. The second problem is more subtle: what is the responsibility of components in an event-driven application with respect to concurrency and load management? In the I/O core, when one component invokes another, it has no way of knowing in advance whether the operation will be synchronous (and hence stall the caller until it is complete) or asynchronous (returning immediately after an enqueue). The distinction is important for managing concurrency and resource bottlenecks.

In SEDA, there is no explicit upcall interface: two software components that wish to be composed through a direct method call simply invoke one another through a function call. Likewise, SEDA makes the use of queues explicit, thereby clarifying the semantics of stage composition across the queue boundary: stages operate asynchronously with respect to one another, and queues are permitted to reject new entries to manage load. The system designer must explicitly consider the possibility of queue rejection within the stage graph.

1.1 Simplification of event-driven design

In addition to making queue boundaries explicit, SEDA simplifies event-driven system design in several other ways. Application programmers are not responsible for allocation or scheduling of threads; rather, this functionality is provided by the underlying runtime system. Using dynamic control to tune thread pool sizes automatically avoids error-prone manual configuration, which can have a serious effect on performance. Also, the rich initialization and configuration API in the Sandstorm framework makes it relatively straightforward to develop stages in isolation from one another and “drop them into” an existing application.

The most important simplification introduced by SEDA is the explicit decomposition of event-processing code into stages. By having stages communicate through an explicit event-passing interface across bounded queues, the problem of managing the control structure of a complex event-driven application is greatly reduced. In the monolithic event-driven design, one constructs a single event loop that must explicitly schedule event-processing components, as well as be aware of the resource requirements of each component. In SEDA, concurrency management is limited to each individual stage, scheduling is performed by the underlying thread system, and resource bottlenecks are managed through admission control.

Code modularization in SEDA is achieved through the natural encapsulation of event-processing code within stages. In this model, composition is accomplished through asynchronous event exchange, rather than though the more traditional approach of function-call APIs. The SEDA model achieves better containment of resources within a stage, as stages are permitted to perform admission control, and the performance of a stage (e.g., its throughput and response time) are evident. Staged pipelining also permits straightforward interpositioning of code along a event-flow path. Composition of components is less rigid than a traditional API: a stage may simply pass through or ignore events that it does not understand.

SEDA's use of dynamic resource and overload control greatly relieves programmer burden when designing well-conditioned services. In the monolithic event-driven design, programmers must make a number of decisions about event scheduling, such as the number and placement of threads in the application, what order to process pending events, and so forth. SEDA obviates the need for this type of careful resource planning: thread allocation decisions are made by the runtime in response to load, and scheduling is performed by the underlying operating system. In addition, SEDA's overload control primitives make it relatively straightforward to ensure that a service will behave well under excessive demand. A service designer need only attach an overload controller to an appropriate set of stages to cause the runtime to perform admission control.

1.2 Disadvantages of SEDA

The SEDA model has several disadvantages that became clear as we gained more experience with the design. We believe that most of these issues could be addressed through some simple modifications to the programming model.

In the current design, events arriving at a stage are assumed to be independent: separate threads may process any two events, and hence in any order. However, if a stage has constraints on the order in which events can be processed, it must impose that ordering itself, for example by maintaining an internal list of incoming events that is synchronized across threads within a stage. We have provided a utility class that implements cross-event ordering, though ideally the SEDA programming model would allow stages to express their ordering constraints, if any, and the runtime would obey those constraints when dispatching events to threads.

In the Sandstorm implementation, stages have no way of knowing or controlling how many threads are operating within it. This precludes several optimizations, such as avoiding the use of locks when there is only one thread running in the stage. A stage may be declared as single-threaded at implementation time, but this determination is static: a single-threaded stage may *never* have multiple threads running within it. It would be useful to allow a stage to indicate to the runtime that it wishes to run in single-threaded mode for some period of time.

Likewise, it is sometimes desirable for a multithreaded stage to operate as though it were multiple single-threaded stages running in parallel, allowing each thread within the stage to operate entirely on private state. Sandstorm provides no interfaces for this, so programmers typically maintain a “pool” of private state objects that threads may claim when an event handler is invoked. A straightforward modification to Sandstorm would allow the creation of partitioned multi-threaded stages.

Although SEDA is intended to simplify concurrency issues in an event-driven application, it is still possible for state to be shared across stages as well as within a stage. Though we encourage the avoidance of shared state, for any reasonably complex application this need will undoubtedly arise. Currently, application developers must make use of standard synchronization primitives (mutexes, semaphores, etc.) when controlling access to shared state. An alternative would allow one stage to affect the scheduling properties of another stage to implement various mutual-exclusion properties. For example, threads could be scheduled such that no two threads that operate on a shared data structure are active at one time. It is unclear whether the complexity of this approach would outweigh the potential performance gains from avoiding locks.

2 Future Work

A wide range of open questions remain in the Internet service design space. We feel that the most important issues have to do with robustness and management of heavy load, rather than raw performance, which has been much of the research community's focus up to this point. Some of the interesting research challenges raised by the SEDA design are outlined below.

2.1 Directions in dynamic resource control

We have argued that measurement and control is the key to resource management and overload protection in busy Internet services. This is in contrast to long-standing approaches based on resource containment, which assign fixed resources to each task (such as a process, thread, or server request) in the system, and strive to contain the resources consumed by each task. Although these techniques have been used with some success [138], containment typically mandates an *a priori* assignment of resources to each task, limiting the range of applicable load-conditioning policies. Rather, we argue that dynamic resource control, coupled with application-specific adaptation in the face of overload, is the right way to approach load conditioning.

Introducing feedback as a mechanism for overload control raises a number of questions. For example, how should controller parameters be tuned? We have relied mainly on a heuristic approach to controller design, though more formal, control-theoretic techniques are possible [108]. Control theory provides a valuable framework for designing and evaluating feedback-driven systems, though many of the traditional techniques rely upon good mathematical models of system behavior, which are often unavailable for complex software systems. Capturing the performance and resource needs of real applications through detailed models is an important research direction if control-theoretic techniques are to be employed more widely.

The interaction between multiple levels of control in the system is also largely unexplored. For example, there is a subtle interplay between queue admission control and tuning per-stage thread pool sizes, given that one technique attempts to keep queue lengths low by scaling the number of threads, while another is attempting to meet performance targets through rejecting incoming queue entries. A simple approach is to disable admission control until the thread pool has reached its maximum size, though a more general approach to controller composition is needed.

2.2 Generalizing overload management

Our approach to overload management is based on adaptive admission control using "external" observations of stage performance. This approach uses no knowledge of the actual resource-consumption patterns of stages in an application, but is based on the implicit connection between request admission and performance. However, this does not directly capture all of the relevant factors that can drive a system into overload. For example, a memory-intensive stage (or a stage with a memory leak) can lead to VM thrashing with even a very low request-admission rate. Likewise, resource sharing across stages can cause one stage to affect the performance of another, leading to misdirected admission-control decisions.

A better approach to overload control would be informed by actual resource usage, allowing the system to make directed overload control decisions. By carefully accounting for resource usage on a per-stage basis, the system could throttle the operation of only those stages that are causing overload. Likewise, applications could degrade their service fidelity based on the resource availability: very different degradation decisions might be made if CPU were a bottleneck than if memory were limited. We have investigated one step in this direction, a system-wide resource monitor capable of signaling stages when resource usage (e.g., memory availability or CPU utilization) meets certain conditions. In this model, stages receive system-wide overload signals and use the information to voluntarily reduce their resource consumption.

2.3 Towards a service-oriented operating system

Although SEDA facilitates the construction of well-conditioned services over commodity operating systems, the SEDA model suggests new directions for OS design. An interesting aspect of Sandstorm is that it is purely a “user level” mechanism, acting as a resource management middleware sitting between applications and the underlying operating system. However, if given the opportunity to design an OS for scalable Internet services, many interesting ideas could be investigated, such as scalable I/O primitives, SEDA-aware thread scheduling, and application-specific resource management.

Designing an operating system specifically tailored for Internet services would shift the focus from multiprogramming workloads to supporting massive flows of requests from a large number of simultaneous clients. The I/O system could be structured around scalable, nonblocking primitives, and issues with supporting legacy (blocking) interfaces could be sidestepped. Questions about provisioning system resources would be made in favor of large degrees of scalability. For example, most default Linux installations support a small number of file descriptors per process, on the assumption that few users will need more than a handful. In a server-oriented operating system these limits would be based entirely on available resources, rather than protecting the system from runaway processes.

We envision an OS that supports the SEDA execution model directly, and provides applications with greater control over scheduling and resource usage. This approach is similar to that found in various research systems [7, 16, 70, 87] that enable application-specific resource management. Even more radically, a SEDA-based operating system need not be designed to allow multiple applications to share resources transparently. Internet services are highly specialized and are not intended to share the machine with other applications: it is generally undesirable for, say, a Web server to run on the same machine as a database engine (not to mention a scientific computation or a word processor). Although the OS may enforce protection (to prevent one stage from corrupting the state of the kernel or another stage), the system need not virtualize resources in a way that masks their availability from applications.

The standard UNIX scheduling algorithm, based on multilevel feedback priority queues, has served us well. However, it would be interesting to investigate the optimal thread scheduling policy for SEDA-based applications. A primary goal would be to ensure that the flow of requests through stages is in balance; in the current design, this is accomplished by adding threads to stages which appear to have increased concurrency demands, though this could also be effected through scheduling. The StagedServer [80] approach of scheduling to maximize cache locality could be incorporated as well. Under overload, the scheduler might give priority to stages that will release resources, preventing requests from propagating through the system until enough resources are available to complete their processing.

2.4 Using SEDA in a cluster environment

This dissertation has dealt primarily with concurrency and load management within a single node, though large-scale Internet services are typically constructed using workstation clusters to achieve incremental scalability and fault tolerance [45, 52]. An interesting direction for future work would be to explore the application of SEDA in a clustered environment. In this model, individual stages can be replicated across cluster nodes for high availability as well as to scale in response to load. Event queues can be implemented as network sockets, allowing inter-stage communication to be transparent regardless of whether two stages are on the same node or on different nodes.

The Ninja vSpace system [139] has explored several aspects of this design space. In vSpace, individual cluster nodes use the SEDA design to manage concurrency and resources locally.

Services are decomposed into a set of *clones*; each clone is a software module (essentially a stage) that can be replicated across cluster nodes. A *connection manager* is used to load-balance the service by directing incoming requests to an appropriate clone. Shared state across clones is stored in a cluster-wide distributed hash table [52].

vSpace raises a number of interesting issues in distributing a SEDA stage graph across cluster nodes. To achieve fault tolerance, it is necessary to retransmit events that are dispatched over the network should the receiving node fail. This implies that the receiving node send an acknowledgment when it has completed processing an event (or has passed the event on to another stage), to inform the sender that retransmission is no longer necessary. Distribution therefore induces a "request-reply" event-passing model, rather than the unidirectional event-passing model used in the single-node case. In a single-node service, retransmission and acknowledgments are unnecessary if the unit of service failure is the entire node (as is the case in our Sandstorm prototype).

Stage distribution also raises questions about the placement of stages within a cluster. For increased locality it is generally desirable to co-locate the stages for a given request-processing path (for example, a single client session) on the same machine. Likewise, by allowing certain stages to be replicated more than others, a coarse form of resource prioritization can be achieved. vSpace uses the notion of *partitions* to logically and physically separate multiple request processing paths within the service. A partition is used as the unit of resource affinity (e.g., binding a set of client sessions to a set of cluster nodes) and for prioritization (e.g., allowing larger number of resources to be devoted to higher-priority requests).

Chapter 9

Conclusions

This thesis has addressed a critical issue in the design and deployment of services on the Internet: that of implementing massively scalable services that are robust to huge variations in load. We have argued that traditional programming models, based on thread-driven and event-driven concurrency, fail to address the scalability and overload-management needs for large-scale Internet services. This domain gives rise to a new set of systems design requirements that are less evident in systems targeted at low levels of concurrency and load.

Our primary goal has been to develop a system architecture that simplifies the development of highly concurrent and well-conditioned services. In the Staged Event-Driven Architecture (SEDA), services are constructed as a network of event-driven stages connected with explicit event queues. Event-driven concurrency is used within each stage, thereby facilitating a high degree of scalability. Dynamic resource control and per-stage control are used to ensure that the service stays within its ideal operating regime despite large fluctuations in load.

We have presented the SEDA architecture as well as a set of design patterns that can be used to map an application onto a structure of stages and queues. These patterns are meant to make the abstract process of service design more concrete by encapsulating critical tradeoffs in terms of performance, load management, and code modularity. In addition, we have developed a queue-theoretic performance model for SEDA that describes the performance effects of each of these design patterns, and serves as a guide for system designers to understand the performance of the resulting system.

Our prototype of SEDA, called Sandstorm, is implemented in Java and makes use of native code bindings to provide nonblocking I/O primitives. We have found Java to be an excellent language for Internet service development, which is greatly facilitated by strong typing, built-in threading, and automatic memory management. Despite expectations to the contrary, we have found that Java has not had a negative impact on the performance and scalability of services developed using the Sandstorm framework.

We have developed several dynamic resource control mechanisms that tune the number of threads within each stage, as well as control the degree of event batching used internally by a stage. We have also presented a family of overload control mechanisms for SEDA, based on adaptive, per-stage admission control that attempts to meet a 90th percentile response time target. These mechanisms are capable of shedding load from the system under overload, or alternately allow applications to degrade the fidelity of delivered service to allow more requests to be processed at reduced quality. Likewise, differentiated service can be implemented by prioritizing requests from one user or request class over another.

We have evaluated the SEDA architecture through several applications, including Haboob, a high-performance Web server; a packet router for the Gnutella peer-to-peer file-sharing network; and Arashi, an e-mail service making use of dynamic scripting and database access. Our measurements show that services developed using the SEDA model exhibit good performance and extremely robust behavior under heavy load. The SEDA overload control mechanisms are effective at meeting performance targets for complex services, even under extreme load spikes.

We have argued that it is critically important to address the problem of overload from an Internet service design perspective, rather than through *ad hoc* approaches lashed onto existing systems. Rather than static resource partitioning or prioritization, we claim that the most effective way to approach overload management is to use feedback and dynamic control. This approach is more flexible, less prone to underutilization of resources, and avoids the use of static “knobs” that can be difficult for a system administrator to tune. In our approach, the administrator specifies only high-level performance targets which are met by feedback-driven controllers.

We also argue that it is necessary to expose overload to the application, rather than hiding load management decisions in an underlying OS or runtime system. Application awareness of load conditions allows the service to make informed resource management decisions, such as gracefully degrading the quality of service. In the SEDA model, overload is exposed to applications through explicit signals in the form of cross-stage enqueue failures.

The staged event-driven architecture represents a new design point for large-scale Internet services that must support massive concurrency and unexpected changes in load. SEDA brings together aspects of threading, event-driven concurrency, dynamic resource management, and adaptive overload control into a coherent framework. Our experience and evaluation of this design demonstrate that the SEDA approach is an effective way to build robust Internet services.

Bibliography

[1] TreadMarks: Distributed shared memory on standard workstations and operating systems.

[2] T. F. Abdelzaher and N. Bhatti. Adaptive content delivery for Web server QoS. In *Proceedings of International Workshop on Quality of Service*, London, June 1999.

[3] T. F. Abdelzaher and C. Lu. Modeling and performance control of Internet servers. In *Invited Paper, 39th IEEE Conference on Decision and Control*, Sydney, Australia, December 2000.

[4] Acme Labs. thttpd: Tiny/Turbo/Throttling HTTP Server. `http://www.acme.com/software/thttpd/`.

[5] Akamai, Inc. `http://www.akamai.com/`.

[6] America Online. Press Data Points. `http://corp.aol.com/press/press_datapoints.html`.

[7] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[8] Apache Software Foundation. The Apache Web server. `http://www.apache.org`.

[9] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster reserves: A mechanism for resource management in cluster-based network servers. In *Proceedings of the ACM SIGMETRICS*

Conference on Measurement and Modeling of Computer Systems, Santa Clara, CA, June 2000.

[10] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[11] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI '99)*, February 1999.

[12] G. Banga and J. C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proceedings of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, June 1998.

[13] G. Banga, J. C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999.

[14] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *Journal of the ACM*, 22(2):248–260, April 1975.

[15] BEA Systems. BEA WebLogic. `http://www.beasys.com/products/weblogic/`.

[16] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15)*, 1995.

[17] P. Bhoj, S. Ramanathan, and S. Singhal. Web2K: Bringing QoS to Web Servers. Technical Report HPL-2000-61, HP Labs, May 2000.

[18] Bloomberg News. E*Trade hit by class-action suit, *CNET News.com*, February 9, 1999. `http://news.cnet.com/news/0-1007-200-338547.html`.

[19] J. Borland. Net video not yet ready for prime time, *CNET news.com*. `http://news.cnet.com/news/0-1004-200-338361.html`, February 1999.

[20] L. Breslau, S. Jamin, and S. Shenker. Comments on the performance of measurement-based admission control algorithms. In *Proceedings of IEEE INFOCOM 2000*, Tel Aviv, Israel, March 2000.

[21] L. Breslau, E. W. Knightly, S. Shenker, I. Stoica, and H. Zhang. Endpoint admission control: Architectural issues and performance. In *Proceedings of ACM SIGCOMM 2000*, Stockholm, Sweeden, October 2000.

[22] British Broadcasting Corporation. Net surge for news sites. `http://news.bbc.co.uk/hi/english/sci/tech/newsid_1538000/1538149.stm`, September 2001.

[23] N. C. Burnett, J. Bent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Exploiting gray-box knowledge of buffer-cache contents. In *Proceedings of the 2002 USENIX Annual Technical Conference (USENIX '02)*, Monterey, CA, June 2002.

[24] S. Chandra, C. S. Ellis, and A. Vahdat. Differentiated multimedia Web services using quality aware transcoding. In *Proceedings of IEEE INFOCOM 2000*, March 2000.

[25] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical Internet object cache. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 153–163, January 1996.

[26] B. Chen and R. Morris. Flexible control of parallelism in a multiprocessor PC router. In *Proceedings of the 2001 USENIX Annual Technical Conference (USENIX '01)*, Boston, June 2001.

[27] H. Chen and P. Mohapatra. Session-based overload control in QoS-aware Web servers. In *Proceedings of IEEE INFOCOM 2002*, New York, June 2002.

[28] X. Chen, H. Chen, and P. Mohapatra. An admission control scheme for predictable server response time for Web accesses. In *Proceedings of the 10th World Wide Web Conference*, Hong Kong, May 2001.

[29] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti, and P. Yianilos. A prototype implementation of archival Intermemory. In *Proceedings of the Fourth ACM Conference on Digital Libraries (DL '99)*, Berkeley, CA, 1999.

[30] L. Cherkasova and P. Phaal. Session based admission control: A mechanism for improving the performance of an overloaded Web server. Technical Report HPL-98-119, HP Labs, June 1998.

[31] D. D. Clark. The structuring of systems using upcalls. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, December 1985.

[32] I. Clarke, O. Sandberg, B. Wiley, , and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system in designing privacy enhancing technologies. In *Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, 2000.

[33] Claymore Systems, Inc. PureTLS. `http://www.rtfm.com/puretls/`.

[34] CNN. 20 million Americans see Starr's report on Internet. `http://www.cnn.com/TECH/computing/9809/13/internet.starr/`, September 1998.

[35] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in Web servers. In *Proceedings of the 1999 USENIX Symposium on Internet Technologies and Systems (USITS '99)*, October 1999.

[36] M. L. Dertouzos. The future of computing. *Scientific American*, July 1999.

[37] Y. Diao, N. Gandhi, J. Hellerstein, S. Parekh, and D. Tilbury. Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache Web server. In *Proceedings of the Network Operations and Management Symposium 2002*, Florence, Italy, April 2002.

[38] T. Dierks and C. Allen. The TLS Protocol Version 1.0. Internet Network Working Group RFC2246, January 1999.

[39] Digital Island, Inc. `http://www.digitalisland.com/`.

[40] P. Druschel and L. Peterson. Fbufs: A high bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, 1993.

[41] M. Fayad and D. C. Schmidt. Object-oriented application frameworks. *Communications of the ACM, Special Issue on Object-Oriented Application Frameworks*, 40(10), October 1997.

[42] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Internet Network Working Group RFC2616, June 1999.

[43] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[44] A. Fox and E. A. Brewer. Harvest, yield and scalable tolerant systems. In *Proceedings of the 1999 Workshop on Hot Topics in Operating Systems*, Rio Rico, Arizona, March 1999.

[45] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, October 1997.

[46] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[47] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering, *Software – Practice and Experience*. `http://www.research.att.com/sw/tools/graphviz/`.

[48] Garner Group, Inc. Press Release. `http://www.gartner.com/public/static/aboutgg/pressrel/pr20001030a.html`, October 2000.

[49] Gnutella. `http://gnutella.wego.com`.

[50] S. Goldstein, K. Schauser, and D. Culler. Enabling primiives for compiling parallel languages. In *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, Rochester, NY, May 1995.

[51] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.

[52] S. Gribble, E. Brewer, J. Hellerstein, and D. Culler. Scalable, distributed data structures for Internet service construction. In *Proceedings of the Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, October 2000.

[53] S. Gribble, M. Welsh, R. von Behren, E. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R. Katz, Z. Mao, S. Ross, and B. Zhao. The Ninja architecture for robust Internet-scale systems and services. *Computer Networks*, June 2000. Special Issue on Pervasive Computing.

[54] S. D. Gribble. *A Design Framework and a Scalable Storage Platform to Simplify Internet Service Construction*. PhD thesis, UC Berkeley, September 2000.

[55] E. Hansen. Email outage takes toll on Excite@Home, *CNET News.com*, June 28, 2000. `http://news.cnet.com/news/0-1005-200-2167721.html`.

[56] M. Harchol-Balter. Performance modeling and design of computer systems. Unpublished lecture notes, `http://www-2.cs.cmu.edu/~harchol/Perfclass/class.html`.

[57] M. Harchol-Balter, M. Crovella, and S. Park. The case for SRPT scheduling in Web servers. Technical Report MIT-LCR-TR-767, MIT, October 1998.

[58] K. Harty and D. Cheriton. Application controlled physical memory using external page cache management, October 1992.

[59] Hewlett-Packard Corporation. e-speak Open Services Platform. `http://www.e-speak.net/`.

[60] J. Hu, S. Mungee, and D. Schmidt. Techniques for developing and measuring high-performance Web servers over ATM networks. In *Proceedings of INFOCOM '98*, March/April 1998.

[61] J. C. Hu, I. Pyarali, and D. C. Schmidt. High performance Web servers on Windows NT: Design and performance. In *Proceedings of the USENIX Windows NT Workshop 1997*, August 1997.

[62] IBM Corporation. IBM WebSphere Application Server. `http://www-4.ibm.com/software/webservers/`.

[63] Inktomi Corporation. Inktomi search engine. `http://www.inktomi.com/products/portal/search/`.

[64] Inktomi Corporation. Web surpasses one billion documents. `http://www.inktomi.com/new/press/2000/billion.html`, January 2000.

[65] R. Iyer, V. Tewari, and K. Kant. Overload control mechanisms for Web servers. In *Workshop on Performance and QoS of Next Generation Networks*, Nagoya, Japan, November 2000.

[66] J. Jackon. Jobshop-like queueing systems. *Management Science*, 10(1):131–142, October 1963.

[67] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, Inc., New York, 1991.

[68] R. Jain, D. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical Report TR-301, DEC Research, September 1984.

[69] H. Jamjoom, J. Reumann, and K. G. Shin. QGuard: Protecting Internet servers from overload. Technical Report CSE-TR-427-00, University of Michigan Department of Computer Science and Engineering, 2000.

[70] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, October 1997.

[71] M. F. Kaashoek, D. R. Engler, G. R. Ganger, and D. A. Wallach. Server operating systems. In *Proceedings of the 1996 SIGOPS European Workshop*, September 1996.

[72] V. Kanodia and E. Knightly. Multi-class latency-bounded Web services. In *Proceedings of IEEE/IFIP IWQoS 2000*, Pittsburgh, PA, June 2000.

[73] D. Katabi, M. Handley, and C. Rohrs. Internet congestion control for future high bandwidth-delay product environments. In *Proceedings of ACM SIGCOMM 2002*, Pittsburgh, PA, August 2002.

[74] D. Kegel. The C10K problem. `http://www.kegel.com/c10k.html`.

[75] S. Keshav. A control-theoretic approach to flow control. In *Proceedings of ACM SIGCOMM 1991*, September 1991.

[76] L. Kleinrock. *Queueing Systems, Volume 1: Theory*. John Wiley and Sons, New York, 1975.

[77] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[78] R. Konrad. Napster among fastest-growing Net technologies, *CNET news.com*. `http://news.com.com/2100-1023-246648.html`, October 2000.

[79] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.

[80] J. Larus and M. Parkes. Using cohort scheduling to enhance server performance. In *Proceedings of the 2002 USENIX Annual Technical Conference (USENIX '02)*, Monterey, CA, June 2002.

[81] H. Lauer and R. Needham. On the duality of operating system structures. In *Proceedings of the Second International Symposium on Operating Systems*, IRIA, October 1978.

[82] S. S. Lavenberg. *Computer Performance Modeling Handbook*. Academic Press, 1983.

[83] W. LeFebvre. CNN.com: Facing a world crisis. Invited talk at USENIX LISA'01, December 2001.

[84] J. Lemon. FreeBSD kernel event queue patch. `http://www.flugsvamp.com/~jlemon/fbsd/`.

[85] R. Lemos. Antivirus firms team to thwart DoS attacks, *CNET News.com*. `http://news.cnet.com/news/0-1003-200-6931389.html`, August 2001.

[86] R. Lemos. Web worm targets White House, *CNET News.com*. `http://news.com.com/2100-1001-270272.html`, July 2001.

[87] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14:1280–1297, September 1996.

[88] K. Li and S. Jamin. A measurement-based admission-controlled Web server. In *Proceedings of IEEE Infocom 2000*, Tel-Aviv, Israel, March 2000.

[89] C. Lu, T. Abdelzaher, J. Stankovic, and S. Son. A feedback control approach for guaranteeing relative delays in Web servers. In *IEEE Real-Time Technology and Applications Symposium*, Taipei, Taiwan, June 2001.

[90] C. Lu, J. Stankovic, G. Tao, and S. Son. Design and evaluation of a feedback control EDF algorithm. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, Phoenix, Arizona, December 1999.

[91] M. Lutz. *Programming Python*. O'Reilly and Associates, March 2001.

[92] F. Manjoo. Net traffic at all-time high, *WIRED News*, November 8, 2000. `http://www.wired.com/news/business/0,1367,40043,00.html`.

[93] D. McNamee and K. Armstrong. Extending the Mach external pager interface to accommodate user-level page replacement policies. In *Proceedings of the USENIX Mach Symposium*, 1990.

[94] K. McNaughton. Is eBay too popular?, *CNET News.com*, March 1, 1999. `http://news.cnet.com/news/0-1007-200-339371.html`.

[95] Microsoft Corporation. DCOM Technical Overview. `http://msdn.microsoft.com/library/backgrnd/html/msdn_dcomtec.htm`.

[96] Microsoft Corporation. IIS 5.0 Overview. `http://www.microsoft.com/windows2000/library/howitworks/iis/iis5techove%rview.asp`.

[97] J. Mogul. Operating systems support for busy Internet services. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, May 1995.

[98] J. C. Mogul. The case for persistent-connection HTTP. In *Proceedings of ACM SIGCOMM'95*, October 1995.

[99] E. Mohr, D. Kranx, and R. Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, 1991.

[100] D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation 1996*, October 1996.

[101] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Proceedings of the Fourteenth International Conference on Distributed Computing Systems*, pages 56–65, Poznan, Poland, June 1994.

[102] MySQL AB. MySQL. `http://www.mysql.com`.

[103] Netscape Corporation. Netscape Enterprise Server. `http://home.netscape.com/enterprise/v3.6/index.html`.

[104] Nielsen//NetRatings. Nielsen//NetRatings finds strong global growth in monthly Internet sessions and time spent online between April 2001 and April 2002. `http://www.nielsen-netratings.com/pr/pr_020610_global.pdf`, June 2002.

[105] K. Ogata. *Modern Control Engineering*. Prentice Hall, 1997.

[106] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the 1999 USENIX Annual Technical Conference*, June 1999.

[107] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*, February 1999.

[108] S. Parekh, N. Gandhi, J. L. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management*, Seattle, WA, May 2001.

[109] R. Park. Software size measurement: A framework for counting source statements. Technical Report CMU/SEI-92-TR-020, Carnegie-Mellon Software Engineering Institute, September 1992.

[110] C. Partridge. *Gigabit Networking*. Addison-Wesley, 1993.

[111] N. Provos and C. Lever. Scalable network I/O in Linux. Technical Report CITI-TR-00-4, University of Michigan Center for Information Technology Integration, May 2000.

[112] S. H. Rodrigues, T. E. Anderson, and D. E. Culler. High-performance local area communication with fast sockets. In *Proceedings of the 1997 USENIX Annual Technical Conference*, 1997.

[113] M. Russinovich. Inside I/O completion ports. `http://www.sysinternals.com/comport.htm`.

[114] A. T. Saracevic. Quantifying the Internet, *San Francisco Examiner*, November 5, 2000. `http://www.sfgate.com/cgi-bin/article.cgi?file=/examiner/hotnews/storie%s/05/Binternetsun.dtl`.

[115] D. C. Schmidt, R. E. Johnson, and M. Fayad. Software patterns. *Communications of the ACM, Special Issue on Patterns and Pattern Languages*, 39(10), October 1996.

[116] B. Schroeder and M. Harchol-Balter. Web servers under overload: How scheduling can help. Technical Report CMU-CS-02-143, Carnegie-Mellon University, June 2002.

[117] SGI, Inc. POSIX Asynchronous I/O. `http://oss.sgi.com/projects/kaio/`.

[118] E. G. Sirer, P. Pardyak, and B. N. Bershad. Strands: An efficient and extensible thread management architecture. Technical Report UW-CSE-97-09-01, University of Washington, September 1997.

[119] Standard Performance Evaluation Corporation. The SPECweb99 benchmark. `http://www.spec.org/osg/web99/`.

[120] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 145–158, 1999.

[121] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.

[122] Sun Microsystems Inc. Enterprise Java Beans Technology. `http://java.sun.com/products/ejb/`.

[123] Sun Microsystems Inc. Java 2 Enterprise Edition. `http://java.sun.com/j2ee`.

[124] Sun Microsystems Inc. Java Native Interface Specification. `http://java.sun.com/products/jdk/1.2/docs/guide/jni/index.html`.

[125] Sun Microsystems, Inc. Java Remote Method Invocation. `http://java.sun.com/products/jdk/rmi/`.

[126] Sun Microsystems Inc. Java Server Pages API. `http://java.sun.com/products/jsp`.

[127] Sun Microsystems Inc. Java Servlet API. `http://java.sun.com/products/servlet/index.html`.

[128] Sun Microsystems Inc. Jini Connection Technology. `http://www.sun.com/jini/`.

[129] Sun Microsystems Inc. RPC: Remote Procedure Call Protocol Specification Version 2. Internet Network Working Group RFC1057, June 1988.

[130] Superior Court of Santa Clara County, California. Cooper v. E*Trade Group, Inc. Case No. CV770328, `http://www.cybersecuritieslaw.com/lawsuits/primary_sources/cooper_v_etr%ade.htm`, November 1997.

[131] K. Taura and A. Yonezawa. Fine-grain multithreading with minimal compiler support: A cost effective approach to implementing efficient multithreading languages. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Las Vegas, Nevada, June 1997.

[132] The Cryptix Foundation Limited. Cryptix. `http://www.cryptix.org`.

[133] The Jython Project. `http://www.jython.org`.

[134] The PHP Group. PHP: Hypertext Preprocessor. `http://www.php.net`.

[135] S. Vajracharya and D. Chavarrá-Miranda. Asynchronous resource management. In *Proceedings of the International Parallel and Distributed Processing Symposium*, San Francisco, California, April 2001.

[136] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

[137] M. Vandevoorde and E. Roberts. Work crews: An abstraction for controlling parallelism. Technical Report Research Report 42, Digital Equipment Corporation Systems Research Center, February 1988.

[138] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded Web servers. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, June 2001.

[139] J. R. von Behren, E. Brewer, N. Borisov, M. Chen, M. Welsh, J. MacDonald, J. Lau, S. Gribble, and D. Culler. Ninja: A framework for network services. In *Proceedings of the 2002 USENIX Annual Technical Conference*, Monterey, California, June 2002.

[140] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th Annual ACM Symposium on Operating Systems Principles*, December 1995.

[141] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrating communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[142] L. A. Wald and S. Schwarz. The 1999 Southern California Seismic Network Bulletin. *Seismological Research Letters*, 71(4), July/August 2000.

[143] D. A. Wallach, D. R. Engler, and M. F. Kaashoek. ASHs: Application-specific handlers for high-performance messaging. In *Proceedings of the ACM SIGCOMM '96 Conference: Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 40–52, Stanford, California, August 1996.

[144] M. Welsh. NBIO: Nonblocking I/O for Java. `http://www.cs.berkeley.edu/~mdw/proj/java-nbio`.

[145] M. Welsh and D. Culler. Virtualization considered harmful: OS design directions for well-conditioned services. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS VIII)*, Schloss Elmau, Germany, May 2001.

[146] M. Welsh and D. Culler. Overload management as a fundamental service design primitive. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[147] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

[148] J. Wilcox. Users say MSN outage continues, *ZDNet News*. `http://zdnet.com.com/2100-11-269529.html`, July 2001.

[149] Yahoo! Inc. `http://www.yahoo.com`.

[150] Yahoo! Inc. Yahoo! Reports Fourth Quarter, Year End 2001 Financial Results. `http://docs.yahoo.com/docs/pr/4q01pr.html`.

[151] Zeus Technology. Zeus Web Server. `http://www.zeus.co.uk/products/ws/`.