USENIX Association

Proceedings of USITS '03: 4th USENIX Symposium on Internet Technologies and Systems

Seattle, WA, USA
March 26–28, 2003

Adaptive Overload Control for Busy Internet Servers

Matt Welsh and David Culler
Intel Research, Berkeley
and University of California, Berkeley
{mdw,culler}@cs.berkeley.edu

Abstract

As Internet services become more popular and pervasive, a critical problem that arises is managing the performance of services under extreme overload. This paper presents a set of techniques for managing overload in complex, dynamic Internet services. These techniques are based on an adaptive admission control mechanism that attempts to bound the 90th-percentile response time of requests flowing through the service. This is accomplished by internally monitoring the performance of the service, which is decomposed into a set of event-driven *stages* connected with request queues. By controlling the rate at which each stage admits requests, the service can perform focused overload management, for example, by filtering only those requests that lead to resource bottlenecks. We present two extensions of this basic controller that provide class-based service differentiation as well as application-specific service degradation. We evaluate these mechanisms using a complex Web-based e-mail service that is subjected to a realistic user load, as well as a simpler Web server benchmark.

1 Introduction

Internet services have become a vital resource for many people. Internet-based e-mail, stock trading, driving directions, and even movie listings are often considered indispensable both for businesses and personal productivity. Web application hosting has opened up new demands for service performance and availability, with businesses relying on remotely-hosted services for accounting, human resources management, and other applications.

At the same time, Internet services are increasing in complexity and scale. Although much prior research has addressed the performance and scalability concerns of serving static Web pages [8, 28, 31], many modern services rely on dynamically-generated content, which requires significant amounts of computation and I/O to generate. It is not uncommon for a single Internet service request to involve several databases, application servers, and front-end Web servers. Unlike static content, dynamically-generated pages often cannot be cached or replicated for better performance, and the resource requirements for a given user load are very difficult to predict.

Moreover, Internet services are subject to enormous variations in demand, which in extreme cases can lead to *overload*. During overload conditions, the service's response times may grow to unacceptable levels, and exhaustion of resources may cause the service to behave erratically or even crash. The events of September 11, 2001 provided a poignant reminder of the inability of most Internet services to scale: many news sites worldwide were unavailable for several hours due to unprecedented demand. CNN.com experienced a request load 20 times greater than the expected *peak*, at one point exceeding 30,000 requests a second. Despite growing the size of the server farm, CNN was unable to handle the majority of requests to the site for almost 3 hours [23]. Many services rely on overprovisioning of server resources to handle spikes in demand. However, when a site is seriously overloaded, request rates can be orders of magnitude greater than the average, and it is clearly infeasible to overprovision a service to handle a 100-fold or 1000-fold increase in load.

Overload management is a critical requirement for Internet services, yet few services are designed with overload in mind. Often, services rely on the underlying operating system to manage resources, yet the OS typically does not have enough information about the service's resource requirements to effectively handle overload conditions. A common approach to overload control is to apply fixed (administrator-specified) resource limits, such as bounding the number of simultaneous socket connections or threads. However, it is difficult to determine the ideal resource limits under widely fluctuating loads; setting limits too low underutilizes resources, while setting them too high can lead to overload regardless. In addition, such resource limits do not have a direct relationship to client-perceived service performance.

We argue that Internet services should be designed from the ground up to detect and respond intelligently to overload conditions. In this paper, we present an architecture for Internet service design that makes overload management explicit in the programming model, providing services with the ability to perform fine-grained control of resources in response to heavy load. In this model, based on the *staged event-driven architecture* (SEDA) [40], services are constructed as a network

of event-driven *stages* connected with explicit request *queues*. By applying admission control to each queue, the flow of requests through the service can be controlled in a focused manner.

To achieve scalability and fault-tolerance, Internet services are typically replicated across a set of machines, which may be within a single data center or geographically distributed [16, 37]. Even if a service is scaled across many machines, individual nodes still experience huge variations in demand. This requires that effective overload control techniques be deployed at the per-node level, which is the focus of this paper.

Our previous work on SEDA [40] addressed the efficiency and scalability of the architecture, and an earlier position paper [39] made the case for overload management primitives in service design. This paper builds on this work by presenting an adaptive admission control mechanism within the SEDA framework that attempts to meet a 90th percentile response time target by filtering requests at each stage of a service. This mechanism is general enough to support class-based prioritization of requests (e.g., allowing certain users to obtain better service than others) as well as application-specific service degradation.

Several prior approaches to overload control in Internet services have been proposed, which we discuss in detail in Section 5. Many of these techniques rely on static resource limits [3, 36], apply only to simplistic, static Web page loads [2, 9], or have been studied only under simulation [10, 20]. In contrast, the techniques described in this paper allow services to adapt to changing loads, apply to complex, dynamic Internet services with widely varying resource demands, and have been implemented in a realistic application setting. We evaluate our overload control mechanisms using both a resource-intensive Web-based e-mail service and a simple Web server benchmark. Our results show that these adaptive overload control mechanisms are effective at controlling the response times of complex Internet services, and permit flexible prioritization and degradation policies to be implemented.

2 The Staged Event-Driven Architecture

Our overload management techniques are based on the *staged event-driven architecture* (or SEDA), a model for designing Internet services that are inherently scalable and robust to load. In SEDA, applications are structured as a graph of event-driven *stages* connected with explicit *event queues*, as shown in Figure 1. We provide a brief overview of the architecture here; a more complete description and extensive performance results are given in [40].

2.1 SEDA Overview

SEDA is intended to support the massive concurrency demands of large-scale Internet services, as well as to exhibit good behavior under heavy load. Traditional server designs rely on processes or threads to capture the concurrency needs of the server: a common design is to devote a thread to each client connection. However, general-purpose threads are unable to scale to the large numbers required by busy Internet services [5, 17, 31].

The alternative to multithreading is event-driven concurrency, in which a small number of threads are used to process many simultaneous requests. However, event-driven server designs can often be very complex, requiring careful application-specific scheduling of request processing and I/O. This model also requires that application code never block, which is often difficult to achieve in practice. For example, garbage collection, page faults, or calls to legacy code can cause the application to block, leading to greatly reduced performance.

To counter the complexity of the standard event-driven approach, SEDA decomposes a service into a graph of *stages*, where each stage is an event-driven service component that performs some aspect of request processing. Each stage contains a small, dynamically-sized *thread pool* to drive its execution. Threads act as implicit continuations, automatically capturing the execution state across blocking operations; to avoid overusing threads, it is important that blocking operations be short or infrequent. SEDA provides nonblocking I/O primitives to eliminate the most common sources of long blocking operations.

Stages are connected with explicit *queues* that act as the execution boundary between stages, as well as a mechanism for controlling the flow of requests through the service. This design greatly reduces the complexity of managing concurrency, as each stage is responsible only for a subset of request processing, and stages are isolated from each other through composition with queues.

As shown in Figure 2, a stage consists of an *event handler*, an *incoming event queue*, and a dynamically-sized *thread pool*. 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. The stage's incoming event queue is guarded by an *admission controller* that accepts or rejects new requests for the stage. The overload control mechanisms described in this paper are based on adaptive admission control for each stage in a SEDA service.

Additionally, each stage is subject to dynamic *resource control*, which attempts to keep each stage within its ideal operating regime by tuning parameters of the stage's operation. For example, one such controller adjusts the number of threads executing within each stage based on an observation of the stage's offered load (incoming queue length) and performance (throughput). This approach frees the application programmer from

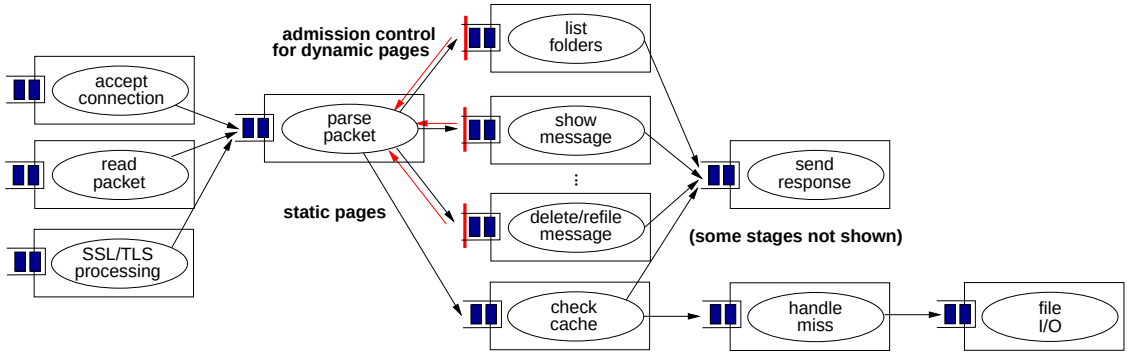

Figure 1: **Structure of the Arashi SEDA-based email service:** *This is a typical example of a SEDA-based Internet service, consisting of a network of stages connected with explicit event queues. Each stage is subject to adaptive resource management and admission control to prevent overload. Requests are read from the network and parsed by a the* read packet *and* parse packet *stages on the left. Each request is then passed to a stage that handles the particular request type, such as listing the user's mail folders. Static page requests are handled by a separate set of stages that maintain an in-memory cache. For simplicity, some event paths and stages have been elided from this figure.*

manually setting "knobs" that can have a serious impact on performance. More details on resource control in SEDA are given in [40].

2.2 Advantages of SEDA

While conceptually simple, the SEDA model has a number of desirable properties for overload management:

Exposure of the request stream: Event queues make the request stream within the service explicit, allowing the application (and the underlying runtime environment) to observe and control the performance of the system, e.g., through reordering or filtering of requests.

Focused, application-specific admission control: By applying fine-grained admission control to each stage, the system can avoid bottlenecks in a focused manner. For example, a stage that consumes many resources can be conditioned to load by throttling the rate at which events are admitted to just that stage, rather than refusing all new requests in a generic fashion. The application can provide its own admission control algorithms that are tailored for the particular service.

Modularity and performance isolation: Requiring stages to communicate through explicit event-passing allows each stage to be insulated from others in the system for purposes of code modularity and performance isolation.

2.3 Overload exposure and admission control

The goal of overload management is to prevent service performance from degrading in an uncontrolled fashion under heavy load, as a result of overcommitting resources. As a service approaches saturation, response times typically grow very large and throughput may degrade substantially. Under such conditions, 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. Note that rejecting requests is just one form of load shedding; several alternatives are discussed below.

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 service can limit the rate at which a 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; however, with this policy 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 and trigger rejection of incoming events when some performance threshold has been exceeded. Additionally, an admission controller could assign a cost to each event in the system, prioritizing 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.

This mechanism allows overload control to be performed within a service in response to measured resource bottlenecks; this is in contrast to "external" service control based on an *a priori* model of service capacity [2, 10]. Moreover, by performing admission control on a per-stage basis, overload response can be focused on those requests that lead to a bottleneck, and be customized for each type of request. This is as opposed to generic overload response mechanisms that fail to consider the nature of the request being processed [18, 19].

When the admission controller rejects a request, the corresponding enqueue operation fails, indicating to the

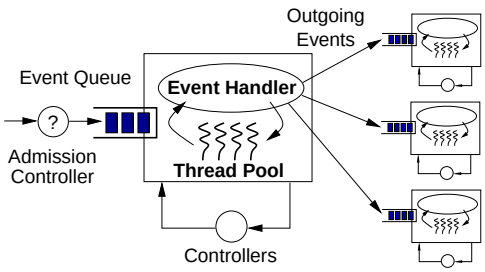

Figure 2: **A SEDA Stage:** *A stage consists of an* incoming event queue *with an associated* admission controller, *a* thread pool, *and an application-supplied* event handler. *The stage's operation is managed by a set of* controllers, *which dynamically adjust resource allocations and scheduling. The* admission controller *determines whether a given request is admitted to the queue.*

originating stage that there is a bottleneck in the system. The upstream stage is therefore responsible for reacting to these "overload signals" in some way. 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.

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, which depends greatly on the service logic, as described above. If the service has been replicated across multiple servers, the request can be redirected to another node, either internally or by sending an HTTP redirect message to the client. Services may also provide differentiated service by delaying certain requests in favor of others: an e-commerce site might give priority to requests from users about to complete an order. Another overload 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. In some applications, however, backpressure may be undesirable as it causes requests to queue up, possibly for long periods of time.

More generally, an application may *degrade service* in response to overload, allowing a larger number of requests to be processed 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 such 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 admission control primitive to signal overload to applications.

3 Overload Control in SEDA

In this section we present three particular overload control mechanisms that have been constructed using the stage-based admission control primitives described earlier. We begin with a motivation for the use of 90th-percentile response time as a client-based performance metric to drive overload control. We then discuss an adaptive admission control mechanism to meet a 90th-percentile response time target, and describe an extension that enables service differentiation across different classes of users. We also describe the use of application-specific service degradation in this framework.

3.1 Performance metrics

A variety of performance metrics have been studied in the context of overload management, including throughput and response time targets [9, 10], CPU utilization [2, 11, 12], and differentiated service metrics, such as the fraction of users in each class that meet a given performance target [20, 25]. In this paper, we focus on *90th-percentile response time* as a realistic and intuitive measure of client-perceived system performance. This metric has the benefit that it is both easy to reason about and captures the user's experience 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 network connection to the service and bears little relation to user-perceived performance).

In this context, the system administrator specifies a target value for the service's 90th-percentile response time. The target response time 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).

3.2 Response time controller design

The design of the per-stage overload controller in SEDA is shown in Figure 3. The controller consists of several components. A *monitor* measures response times for each request passing through a stage. Requests are tagged with the current time when they enter the service. At each stage S, the request's response time is calculated as the time it leaves S minus the time it entered the system. While this approach does not measure network effects, we expect that under overload the greatest contributor to perceived request latency will be intra-service

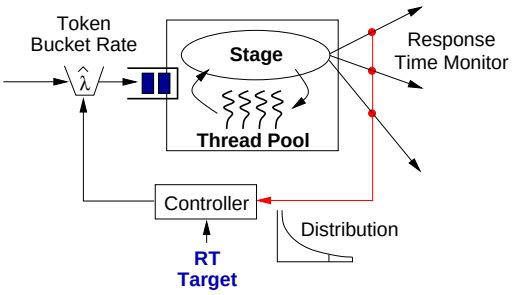

Figure 3: **Response time controller design:** *The controller observes a history of response times through the stage, and adjusts the rate at which the stage accepts new requests to meet an administrator-specified 90th-percentile response time target.*

Parameter	Description	Default value
nreq	# reqs before controller run	100
timeout	Time before controller run	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	Weight on additive increase	-0.1
$rate_{min}$	Minimum rate	0.05
$rate_{max}$	Maximum rate	5000.0

Figure 4: **Parameters used in the response time controller.**

response time.[1]

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 [33]. A wide range of alternate admission control policies are possible, including drop-tail FIFO or variants of random early detection (RED) [14].

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.

[1]To avoid TCP's exponential backoff for initial SYN retransmission, our implementation of SEDA rapidly accepts new client connections. In cases where the number of incoming connections is extremely large, or initial SYNs are dropped by the network, our approach can be supplemented with some form of network latency estimation [30, 35] to obtain a more accurate response-time estimate.

In order to prevent 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: $-(err - c_i)adj_i$. The constant c_i is used to weight the rate increase such that when $err = c_i$ the rate adjustment is 0.

The parameters used in the implementation are summarized in Figure 4. These parameters were determined experimentally using a combination of microbenchmarks with artificial loads and real applications with realistic loads (e.g., the e-mail service described in the next section). 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. In particular, 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 is less critical. 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. It would be useful to apply concepts from control theory to aid in the tuning process, but this requires the development of complex models of system behavior. We discuss the role of control theoretic techniques in more detail in Section 5.3.

3.3 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 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,

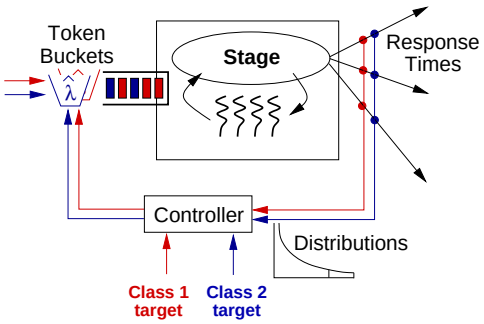

Figure 5: **Multiclass overload controller design:** *For each request class, the controller measures the 90th-percentile response time, and adjusts the rate at which the stage accepts new requests of each class. When overload is detected, the admission rate for lower-priority classes is reduced before that of higher-priority classes.*

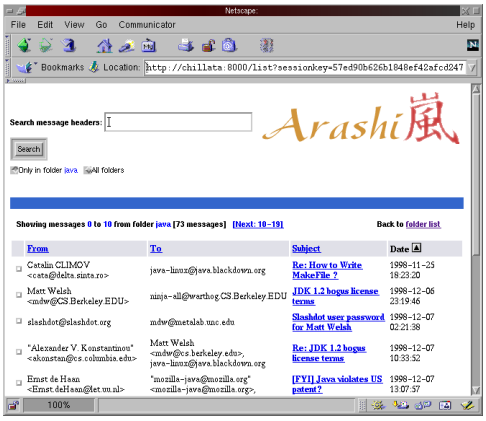

Figure 6: **Screenshot of the Arashi e-mail service:** *Arashi allows users to read e-mail through a Web browser interface. Many traditional e-mail reader features are implemented, including message search, folder view, sorting message lists by author, subject, or date fields, and so forth.*

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 3.2, 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 that of higher-priority classes. Admission rates are increased as in Section 3.2, 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 during the next iteration of the controller.

3.4 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 [1, 7, 16]. SEDA itself does not implement service degradation mechanisms, but rather signals overload to applications in a way that allows them to degrade if possible. Stages can obtain the current 90th-percentile response time measurement as well as enable or disable the stage's admission control mechanism. This allows an service to implement degradation by periodically sampling the current response time and comparing it to the 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.

4 Evaluation

In this section, we evaluate the SEDA overload control mechanisms using two applications: a complex Web-based e-mail service, and a Web server benchmark involving dynamic page generation that is capable of degrading service in response to overload.

4.1 Arashi: A SEDA-based e-mail service

We wish to study the behavior of the SEDA overload controllers in a highly dynamic environment, under a wide variation of user load and resource demands. We have developed the *Arashi*[2] Web-based e-mail service, which is akin to Hotmail and 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. A screenshot of the Arashi service is shown in Figure 6.

Arashi is implemented using the *Sandstorm* platform, a SEDA-based Internet services framework implemented

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

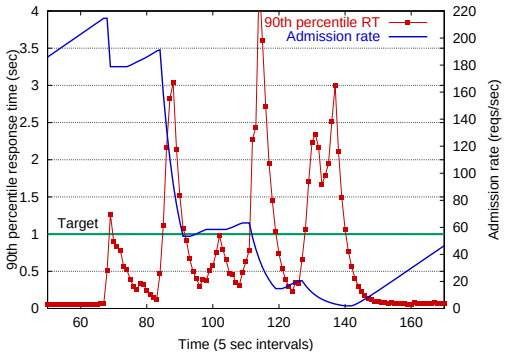

Figure 7: **Overload controller operation:** *This figure shows the operation of the SEDA overload controller for one of the stages in the Arashi e-mail service during a large load spike. A load spike of 1000 users enters the system at around $t = 70$ and leaves the system around $t = 150$. The response 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.*

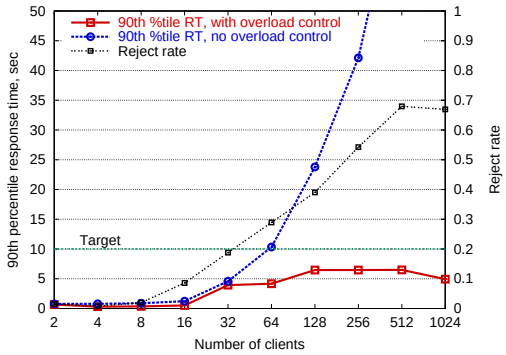

Figure 8: **Overload control in Arashi:** *This figure shows the 90th-percentile response time for the Arashi e-mail service with and without the overload controller enabled. The 90th-percentile response time target is 10 sec. Also shown is the fraction of rejected requests with overload control enabled. 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.*

in Java [40].[3] As shown in Figure 1, some number of stages are devoted to generic Web page processing, nonblocking network and file I/O, and maintaining a cache of recently-accessed static pages; in the Arashi service, there is only a single static Web object (the Arashi logo image). Arashi employs six stages to process dynamic page requests, with one stage assigned to each request type (show message, list folders, etc.). Each stage is implemented as a Java class that processes the corresponding request type, accesses e-mail data from a MySQL [27] database, and generates a customized HTML page in response. This design allows the admission control parameters for each request type to be tuned independently, which is desirable given the large variation in resource requirements across requests. For example, displaying a single message is a relatively lightweight operation, while listing the contents of an entire folder requires a significant number of database accesses.

The client load generator used in our experiments emulates a number of simultaneous users, each accessing a single mail folder consisting of between 1 and 12794 messages, the contents of which are based on actual e-mail archives. Emulated users access the service based on a simple Markovian model of user behavior derived from traces of the UC Berkeley Computer Science Division's IMAP server.[4] The inter-request think time is aggressively set to 20ms. When the service rejects a request from a user, the user waits for 5 sec before attempting to log into the service again. The Arashi service runs on a 1.2 GHz Pentium 4 machine running Linux 2.4.18, and the client load generators run on between 1 and 16 similar machines connected to the server with Gigabit Ethernet. Since we are only interested in the overload behavior of the server, WAN network effects are not incorporated into our evaluation.

4.2 Controller operation

Figure 7 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.

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

[3] Our earlier work [40] demonstrated that despite using Java, Sandstorm performance is competitive with systems implemented in C.

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

	16 users		*1024 users*	
Type	**No OLC**	**OLC**	**No OLC**	**OLC**
login	0.83 sec	0.59 sec	0.86 sec	3.84 sec
list folders	1.73 sec	0.57 sec	**365 sec**	5.75 sec
list msgs	2.37 sec	0.58 sec	**116 sec**	9.28 sec
show msg	0.70 sec	0.30 sec	**30.1 sec**	3.87 sec
delete	1.00 sec	0.28 sec	**11.3 sec**	6.85 sec
refile	1.00 sec	0.47 sec	**10.6 sec**	6.07 sec
search	8.17 sec	9.92 sec	**19.6 sec**	**18.1 sec**

Figure 9: **Breakdown of response times by request type:** *This table lists the 90th-percentile response time for each request type in the Arashi e-mail service for loads of 16 and 1024 users, both without overload control ("No OLC") and with overload control ("With OLC"). The response time target is 10 sec, and values in boldface exceeded the target. 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.*

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 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 4.3 and 4.4.

4.3 Overload control with increased user load

Figure 8 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 load increase.

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 9 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. 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.

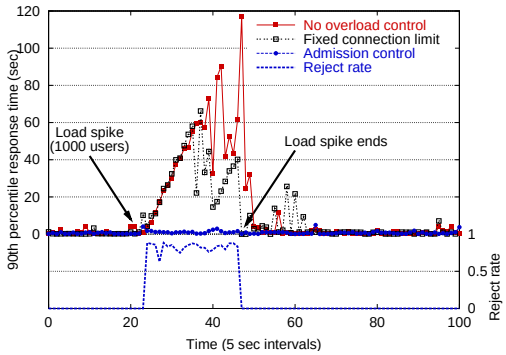

Figure 10: **Overload control under a massive load spike:** *This figure shows the 90th-percentile response time experienced by clients using the Arashi e-mail service under a massive load spike (from 3 users to 1000 users). Without overload control, response times grow without bound; with overload control (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.*

4.4 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 10 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. Figure 10 shows the client response times when overload control is disabled and a limit of 128 simultaneous connections is imposed on the server. As the figure shows, this approach leads to large response times overall. During this benchmark run, over 561 of the 1000 clients ex-

	Per-stage AC		*Single-stage AC*	
Type	**90th RT**	**Rejected**	**90th RT**	**Rejected**
login	2.07 sec	44.3%	1.00 sec	18.8%
list folders	8.11 sec	59.6%	3.97 sec	59.6%
list msgs	8.04 sec	47.1%	6.20 sec	53.7%
show msg	3.90 sec	23.1%	2.04 sec	49.1%
delete	4.86 sec	11.3%	3.26 sec	51.4%
refile	4.60 sec	10.4%	2.12 sec	54.7%
search	**22.2 sec**	0%	**18.9 sec**	53.3%

Figure 11: **Comparison of per-stage versus single-stage admission control:** *This table shows the 90th-percentile response time and reject rate by request type for a load of 128 users on the Arashi service. The response time target is 10 sec, and times shown in boldface exceeded the performance target. With per-stage admission control, the rejection rate is tuned based on the overhead of each request type. For single-stage admission control, all requests experience approproximately the same rejection rate.*

perienced connection timeout errors.

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.

Applying admission control to each stage in the Arashi service allows the admission rate to be separately tuned for each type of request. An alternative policy would be to use a single admission controller that filters all incoming requests, regardless of type. Under such a policy, a small number of expensive requests can cause the admission controller to reject many unrelated requests from the system. Figure 11 compares these two policies, showing the admission rate and 90th-percentile response time by request type for a load of 128 users. As the figure shows, using a single admission controller is much more aggressive in terms of the overall rejection rate, leading to lower response times overall. However, the policy does not discriminate between infrequent, expensive requests and more common, less expensive requests.

4.5 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 only when the lowest service quality still leads to overload.

We evaluate the use of service degradation through a simple Web server benchmark that incorporates a continuous "quality knob" that can be tuned to trade performance for service fidelity. 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.

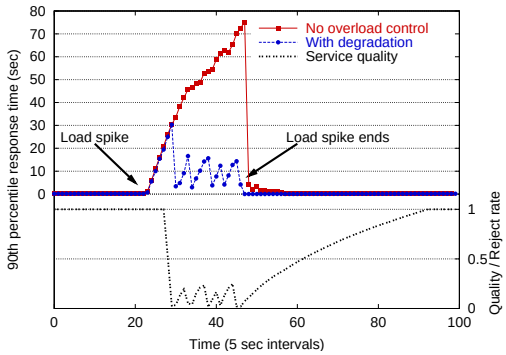

Figure 12: **Effect of service degradation:** *This figure shows the 90th-percentile response time experienced by clients accessing a simple service consisting of a single bottleneck stage. The stage is capable of degrading the quality of service delivered to clients in order to meet response time demands. 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.*

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

Figure 12 shows the effect of service degradation under an extreme load spike, and Figure 13 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.

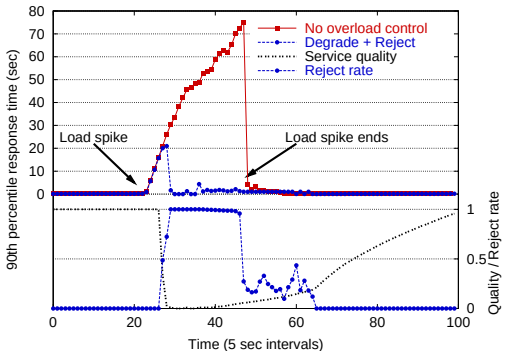

Figure 13: **Service degradation combined with admission control:** *This figure shows the effect of service degradation combined with admission control. The experiment is identical to that in Figure 12, except that the bottleneck stage re-enables admission control when the service quality is at its 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.*

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.

4.6 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 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.

Figure 14 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 no preferential treatment is given to the high priority requests.

In Figure 15, 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 14, though the lower-priority class is penalized with a higher rejection rate. The average reject rate is 87.9% for the low-priority requests, and 48.8% for the high-priority requests. This is compared to 55.5% and 57.6%, respectively, when no service differentiation is performed. Note 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.

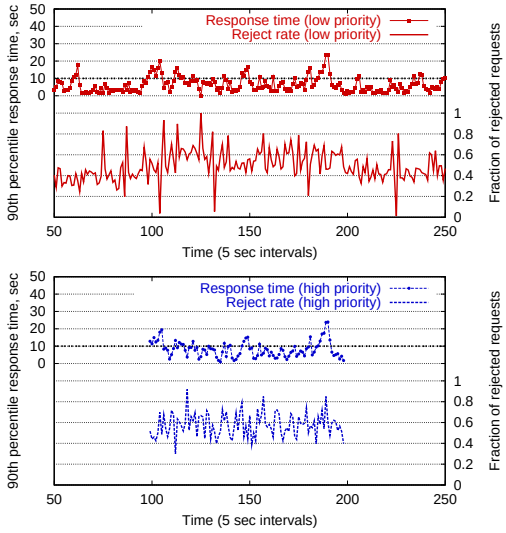

Figure 14: **Multiclass experiment without service differentiation:** *This figure shows the operation of the overload control mechanism in Arashi with two classes of 128 users each accessing the service. The high-priority users begin accessing the service at time $t = 100$ and leave at $t = 200$. No 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.*

5 Related Work

In this section we survey prior work in Internet service overload control, discussing previous approaches as they relate to four broad categories: resource containment, admission control, control-theoretic approaches, and service degradation. In [38] we present a more thorough overview of related work.

5.1 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

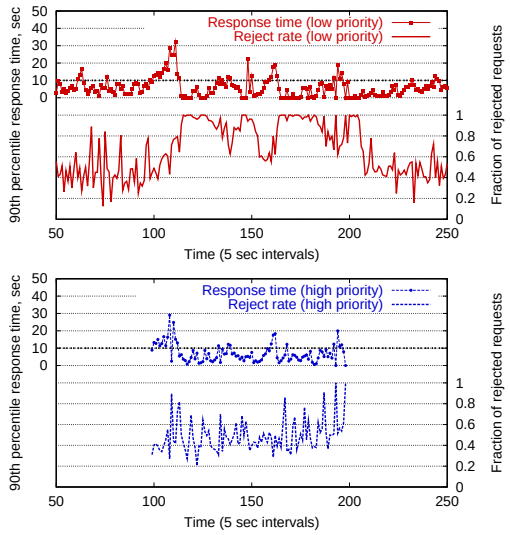

Figure 15: **Multiclass service differentiation:** *This figure shows the operation of the multiclass overload control mechanism in Arashi with two classes of 128 users each. Service differentiation between the two classes is enabled and the 90th-percentile response time target for each class is 10 sec. 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, while high-priority users are able to receive a greater fraction of service.*

entity (say, the system administrator) imposes a limit on the resource usage of a process or application. 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 [4]. There are two serious problems with this approach. First, it is based on a static thread or connection limit, which does not directly correspond to user-perceived performance. Service response time depends on many factors such as user load, the length of time a given connection is active, and the type of request (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. As described earlier, this wait time can grow to be very long, due to TCP's exponential backoff on SYN retransmissions.

Zeus [41] and thttpd [3] 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 [24]. 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.

5.2 Admission control

The use of admission control as an overload management technique has been explored by several systems. Many of the proposed techniques are based on fixed policies, such as bounding the maximum request rate of requests to some constant value. Another common aspect of these approaches is that they often reject incoming work to a service by refusing to accept new client TCP connections.

Iyer *et al.* [18] describe a simple admission control mechanism based on bounding the length of the Web server request queue. This work analyzes various settings for the queue abatement and discard thresholds, though does not specify how these thresholds should be set to meet any given performance target. Cherkasova and Phaal [11] present *session-based* admission control, driven by a CPU utiliization threshold, which performs an admission decision when a new session arrives from a user, rather than for individual requests or connections. Such a policy would be straightforward to implement in SEDA.

Voigt *et al.*[36] 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. Another traffic-shaping approach is described in [19], which drives the selection of incoming packet rates based on an observation of system load, such as CPU utilization and memory usage. Web2K [6] brings several of these ideas together in a Web server "front-end" that performs admission control based on the length of the request queue; as in [18], the issue of determining appropriate queue thresholds is not addressed. Lazy Receiver Processing [13] prevents the TCP/IP receive path from overloading the system; this technique could be coupled with SEDA's overload controllers in extreme cases where incoming request rates are very high. Qie *et al.*[34] introduce resource limitations within the Flash [31] Web serber, primarily as a protection against denial-of-service attacks, though the idea could be extended to overload control.

Several other admission control mechanisms have been presented, though often only in simulation or for simplistic workloads (e.g., static Web pages). PAC-

ERS [10] attempts to limit the number of admitted requests based on expected server capacity, but this paper deals with a simplistic simulated service where request processing time is linear in the size of the requested Web page. A related paper allocates requests to Apache server processes to minimize per-class response time bounds [9]. This paper is unclear on implementation details, and the proposed technique silently drops requests if delay bounds are exceeded, rather than explicitly notifying clients of overload. Kanodia and Knightly [20] develop an admission control mechanism based on *service envelopes*, a modelling technique used to characterize the traffic of multiple flows over a shared link. The admission controller attempts to meet response-time bounds for multiple classes of service requests, but again is only studied under a simple simulation of Web server behavior.

5.3 Control-theoretic approaches

Control theory [29] provides a formal framework for reasoning about the behavior of dynamic systems and feedback-driven control. A number of control-theoretic approaches to performance management of real systems have been described [26, 32], and several of these have focused on overload control for Internet services.

Abdelzaher and Lu [2] describe an admission control scheme that attempts to maintain a CPU utilization target using a proportional-integral (PI) controller and a simplistic linear model of server performance. Apart from ignoring caching, resource contention, and a host of other effects, this model is limited to static Web page accesses. An alternative approach, described in [25], allocates server processes to each class of pending connections to obtain a *relative delay* between user classes. Diao *et al.* [12] describe a control-based mechanism for tuning Apache server parameters (the number of server processes and the per-connection idle timeout) to meet given CPU and memory utilization targets. Recall that in Apache, reducing the number of server processes leads to increased likelihood of stalling incoming connections; although this technique effectively protects server resources from oversaturation, it results in poor client-perceived 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 greatest difficulties 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. The systems described here all make use of linear models, which may not be accurate in describing systems with widely varying loads and resource requirements. 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 recent work has attempted to apply control-theoretic concepts to this problem as well [21, 22].

5.4 Service degradation

A number of systems have explored the use of service degradation to manage heavy load. The most straightforward type of degradation is to reduce the quality of static Web content, such as by reducing the resolution or compression quality of images delivered to clients [1, 7, 16]. 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 was the approach taken by CNN.com on September 11, 2001; in response to overload, CNN replaced its front page with simple HTML page that that could be transmitted in a single Ethernet packet [23]. This technique was implemented manually, though a better approach would be to degrade service gracefully and automatically in response to load.

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 [15] 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. 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.

6 Future Work and Conclusions

We have argued that measurement and control are the keys to resource management and overload protection

in busy Internet services. This is in contrast to long-standing approaches based on resource containment, which typically mandate an *a priori* assignment of resources to each request, limiting the range of applicable load-conditioning policies. Still, introducing feedback as a mechanism for overload control raises a number of questions, such as how controller parameters should be tuned. We have relied mainly on a heuristic approach to controller design, though more formal, control-theoretic techniques are possible [32]. 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 use of per-stage admission control allows the service to carefully control the flow of requests through the system, for example, by only rejecting those requests that lead to a bottleneck resource. The downside of this approach is that a request may be rejected late in the processing pipeline, after it has consumed significant resources in upstream stages. There are several ways to address this problem. Ideally, request classification and load shedding can be performed early; in Arashi, requests are classified in the first stage after they have been read from the network. Another approach is for a stage's overload controller to affect the admission-control policy of upstream stages, causing requests to be dropped before encountering a bottleneck. This paper has focused on stage-local reactions to overload, though a global approach is also feasible in the SEDA framework.

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. 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 even with a very low request-admission rate. One direction for future work is to inform the overload control mechanism with more direct measurements of per-stage resource consumption. 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.

We have presented an approach to overload control for dynamic Internet services, based on adaptive, per-stage admission control. In this approach, the system actively observes application performance and tunes the admission rate of each stage to attempt to meet a 90th-percentile response time target. We have presented extensions of this approach that perform class-based service differentiation as well as application-specific service degradation. The evaluation of these control mechanisms, using both the complex Arashi e-mail service and a simpler dynamic Web server benchmark, show that they are effective for managing load with increasing user populations as well as under massive load spikes.

Software Availability

The SEDA software, related papers, and other documentation are available for download from `http://www.cs.berkeley.edu/~mdw/proj/seda`.

References

[1] 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.

[2] 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.

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

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

[5] 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.

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

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

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A network subsystem architecture for server systems. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, 1996.

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

[15] 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.

[16] 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.

[17] S. D. Gribble, E. A. Brewer, J. M. 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.

[18] 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.

[19] 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.

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

[21] 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.

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

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

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

[25] 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.

[26] 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.

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

[28] National Laboratory for Applied Network Research. The Squid Internet object cache. `http://www.squid-cache.org`.

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

[30] D. P. Olshefski, J. Nieh, and D. Agrawal. Inferring client response time at the Web server. In *Proceedings of SIGMETRICS 2002*, Marina Del Rey, California, June 2002.

[31] 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.

[32] 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.

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

[34] X. Qie, R. Pang, and L. Peterson. Defensive programming: Using an annotation toolkit to build DoS-resistant software. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002)*, December 2002.

[35] R. Rajamony and M. Elnozahy. Measuring client-perceived response times on the WWW. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2001.

[36] 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.

[37] 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.

[38] M. Welsh. *An Architecture for Highly Concurrent, Well-Conditioned Internet Services*. PhD thesis, UC Berkeley, August 2002.

[39] 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.

[40] 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.

[41] Zeus Technology. Zeus Web Server. `http://www.zeus.co.uk/products/ws/`.