Overload Management as a Fundamental Service Design Primitive

Matt Welsh and David Culler
Computer Science Division
University of California, Berkeley
{mdw,culler}@cs.berkeley.edu

Abstract

This position paper makes the case that overload management should be a critical design goal for Internet-based systems and services. Few Internet service designs take overload into account, treating the problem as one of capacity planning rather than engineering the service to behave gracefully under extreme load. We argue that the right approach to overload management is to explicitly signal overload conditions to the application, allowing it to participate in resource management decisions. Furthermore, we claim that feedback-driven control, rather than static resource limits, should be the basis for detecting and controlling overload. We present a feedback-driven approach to overload control based on the staged event-driven architecture (SEDA) model for Internet service design. This approach makes use of adaptive admission controllers for meeting administrator-specified performance targets, such as 90th percentile response time. We demonstrate the use of these overload control mechanisms in two applications: a complex Web-based e-mail service, and a dynamic Web server benchmark.

1 Introduction

In this position paper we argue that overload prevention is a fundamental requirement for distributed systems and services connected to the Internet. Unfortunately, few systems have adequately addressed the management of extreme load, relying mainly on overprovisioning of resources (e.g., replication). However, given the enormous user population on the Internet, overprovisioning is infeasible as the peak load that a service experiences may be orders of magnitude greater than the average. The events of September 11, 2001 provided a poignant reminder of the inability of Internet services to scale: virtually every Internet news site was unavailable for several hours due to unprecedented demand [11]. The increasing prevalence of sophisticated denial-of-service attacks, launched simultaneously from thousands of unrelated machines, further underscores this problem.

Moreover, as our notion of Internet-based services expands to embrace a range of novel distributed systems, including global storage services [10, 20], peer-to-peer systems [6, 18], and sensor networks [5, 8], throwing more resources at the problem does not help: individual nodes in these large computing frameworks are not necessarily backed by massive data centers which can grow to meet capacity.

Despite the importance of load management, few systems directly address this problem, treating it as an issue of capacity planning rather than preparing in advance for (inevitable) overload. Web servers, clustered middle-tier systems, databases, directory services, and file servers all require some form of overload management, though few deployed systems are architected to take this problem into account. To a large extent, this is due to inadequate interfaces for resource management. Most operating systems adhere to the principle of *resource virtualization* to simplify application development. Unfortunately, this approach makes it difficult for applications to be aware of, or adapt to, real resource limitations [24]. For example, the UNIX *malloc* interface simply returns NULL when memory cannot be allocated; an application has no way to know whether a future *malloc* operation will fail, so adapting to memory pressure is nearly impossible.

The programming models used for Internet services generally fail to express resource constraints in a meaningful way. CORBA [15], RPC [21], Java RMI [22], and now .NET [19] all expose a programming model in which distributed components communicate mainly through remote procedure call, simplifying the harnessing of remote resources through a familiar programming abstraction. Unfortunately this abstraction makes no attempt at exposing resource limits or overload conditions to the participating applications. For example, Java RMI calls can throw a generic exception due to any type of failure, but there is typically little that an RMI application can do when this occurs: should the application fail, retry the operation, or invoke an alternate interface?

This problem is compounded when Internet services are constructed through composition of many distributed systems, as is the case with the emergent field of "Web services." Consider a Web service consisting of several independent components communicating through a common protocol such as SOAP. When one component becomes a resource bottleneck, the only overload management technique generally used is for the service to refuse additional TCP connections. While effectively shielding that service from load, other participants experience very long connection delays (e.g., due to TCP's exponential SYN retransmit backoff behavior), causing the bottleneck to propagate through the entire distributed application.

This paper outlines a framework for building Internet services that are inherently robust to load, using two simple techniques: dynamic resource management and fine-grained admission control. While these techniques have been explored elsewhere in the context of specific applications, we find that few Internet service programming models make them explicit. Our approach is based on a software architecture called the *staged event-driven architecture* (or SEDA), which decomposes an Internet service into a network of event-driven stages connected with explicit event queues. Load management in SEDA is accomplished by introducing a feedback loop that observes the behavior and performance of each stage, and applies resource control and admission control to effectively manage overload.

Our previous work on SEDA [25] focused primarily on the efficiency and scalability of this architecture with respect to traditional concurrent server designs. In this paper, we build on the SEDA framework by introducing adaptive overload control mechanisms and discussing the impact of overload control on the SEDA programming model. We report our experiences with several approaches for overload management in SEDA, and present initial results showing the effectiveness of these techniques in a complex Web-based email service.

2 The Need for Dynamic Overload Management

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. Various kinds of resource limits are used: bounding the number of processes or threads within a server is a common technique, as is limiting the number of client socket connections to the service. Both of these approaches have the fundamental problem that it is generally not possible to know what the ideal resource limits should be. Setting the limit too low underutilizes resources, while setting the limit too high can lead to oversaturation and serious performance degradation under overload. Refusing to accept additional TCP connections under heavy load is inadvisable as it causes clients to retransmit the initial SYN packet with exponential backoff, leading to very long response times [25]. This approach is also too coarse-grained in the sense that even a single client can consume all of the resources in the system; imposing process or connection limits does not solve the more general resource management issue.

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. Though resource allocations may change over time, the system does not typically use any feedback on application performance when determining allocations. One important exception is feedback-driven scheduling [13], in which application performance is used to tune scheduling parameters.

Reservation- and share-based resource limits have been explored in depth by systems such as Scout [14], Nemesis [12], Resource Containers [3], and Cluster Reserves [2]. These techniques work well for real-time and multimedia applications, which have relatively static resource demands that can be expressed as straightforward, fixed limits. 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.

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 performance, and applies dynamic control to manage resources. Several systems have explored the use of dynamic overload management in Internet services. Voigt *et al.* [23] and Jamjoom [9] present approaches enabling *service differentiation* in busy Internet servers: the basic idea is to adjust the priority or admission control parameters for each class of requests to yield higher performance for more important requests. In [23], the kernel adjusts process priorities to meet per-class response time targets. When the system is overloaded, processes are blocked and eventually new connections are refused. In [9], per-class admission control is performed by traffic shaping the incoming SYN queue for new connections. The latter technique is limited to classification by client IP address, while the former rapidly accepts incoming TCP connections permitting classification by HTTP header information.

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. Rather, overload management is provided as an OS function with generic load shedding techniques (e.g., blocking processes or rejecting connections) rather than application-specific service degradation. Also, these mechanisms are "wrapped around" existing applications rather than pushing overload control into the application design, where we argue it belongs.

3 SEDA: Making Overload Management Explicit

We have been experimenting with a new software design, the *staged event-driven architecture* (or SEDA), which is designed to provide adequate primitives for managing load in busy Internet services. 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 is given in [25].

3.1 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 [4, 7, 16]. SEDA makes use of efficient event-driven concurrency, in which a small number of threads are used to process many simultane-

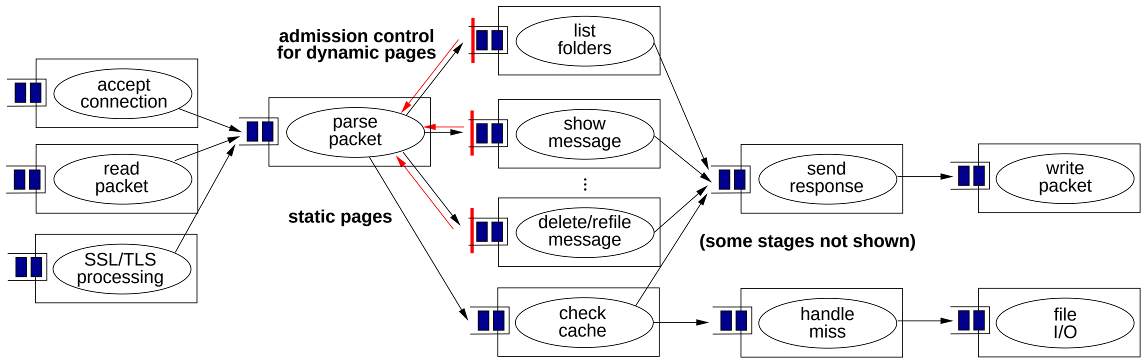


Figure 1: **Structure of the Arashi SEDA-based email service:** *The service consists of a network of stages connected with explicit event queues, coupled with adaptive resource and admission control to prevent overload. For simplicity, some event paths and stages have been elided from this figure.*

ous requests. This requires that request-processing operations be nonblocking, avoiding the need to devote a large number of threads to blocking I/O operations.

Event-driven server designs can often be very complex, requiring application-specific request scheduling, often resulting in labyrinthine application code. Also, the requirement the request-processing logic never block is difficult to achieve in practice, especially if legacy code is incorporated into the service. To counter the complexity of the monolithic event-driven approach, SEDA decomposes a service into a graph of stages, where each stage is internally event-driven and contains a dynamically-sized thread pool to drive its execution. This allows event processing within a stage (and across stages) to proceed in parallel, and permits application code to block for short periods of time. Additionally, the complexity of managing concurrency is significantly reduced, as each stage is responsible only for a subset of request processing, and stages are isolated from others through composition with queues.

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.

- **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.

In SEDA, 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 manually setting "knobs" that can have a serious impact on performance. More details on resource control in SEDA are given in [25].

Each stage has an associated admission controller that guards access to the event queue for that stage. The admission controller 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. The simplest response is to block until the downstream stage can accept the request, which leads to backpressure within the graph of stages. Another response is to drop the request, possibly sending an error message to the client or using the HTTP redirect mechanism to bounce the request to another server. More generally, SEDA applications can *degrade service* in response to overload, such as delivering lower-quality content or choosing to consume fewer resources per request. The key is that the architecture is explicit about signaling overload conditions and allows the application to participate in load management decisions.

4 Adaptive Admission Control in SEDA

The use of per-stage admission control in SEDA allows applications to be conditioned to load in a focused manner. For example, the flow of requests into stage that are the source of a resource bottleneck can be throttled. In this section we describe an adaptive, per-stage admission control technique that attempts to bound the 90th percentile response time of requests flowing

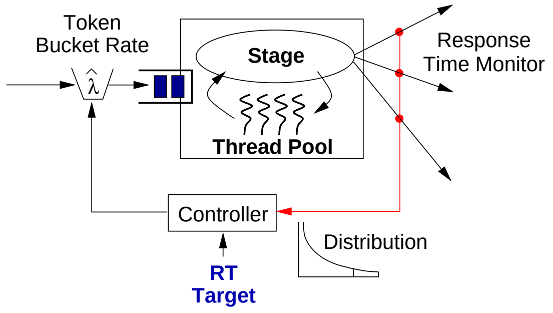


Figure 2: **Response time controller design:** *The controller observes a history of response times through the stage, and adjusts the rate at which the stage accepts new requests to meet an administrator-specified 90th-percentile response time target.*

through the graph of stages. We also discuss the use of service degradation and differentiation across multiple classes of requests.

4.1 Performance metrics

A variety of performance metrics have been studied in the context of overload management, including throughput and response time targets, differentiated service (e.g., the fraction of users in each class that meet a given performance target), and so forth. We focus here 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 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 would specify 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 each request, 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 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).

4.2 Response time controller design

The design of the per-stage overload controller in SEDA is shown in Figure 2. 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* which 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.

Due to space limitations we present a brief overview of the overload controller implementation. 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 overload controller is implemented as a function invoked by the stage's event-processing thread after some number of requests has been processed. This implies that the overload controller will not run when the token bucket rate is low; the algorithm therefore "times out" and performs a recalculation of the 90th percentile response time after a certain interval. When the 90th percentile response time estimate is above a high-water mark (e.g., 10% above the administrator-specified target), the token bucket rate is reduced by a multiplicative factor (e.g., dividing the admission rate by 2). When the estimate is below a low-water mark, the token bucket rate is increased by a small additive factor. The rate increase is proportional to the difference between the current response time estimate and the target; a larger error leads to a greater increase in the admission rate.

4.3 Service degradation and differentiation

In addition to the basic response-time controller, we have implemented mechanisms permitting applications to degrade service under load, as well as to provide differentiated levels of service based on the type of request or user class. A complete discussion of these mechanisms is beyond the scope of this paper, though we touch on them briefly here.

Rather than rejecting requests, SEDA applications may degrade the quality of delivered service in order to admit a larger number of requests given a response-time target. 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 estimate from the overload controller, as well as to enable or disable the admission control mechanism for a given stage. This allows an application to implement degradation by periodically sampling the current response time estimate 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.

Likewise, by prioritizing requests from certain users over others, a SEDA application can implement various policies related to class-based service level agreements. A common example is to give better service to requests from "gold" customers (who might pay more money for the offered service). Building on the basic response time controller described above, We have implemented service differentiation in SEDA using a controller that more aggressively rejects lower-class requests than higher-class requests when a stage is overloaded.

5 Evaluation

In this section we evaluate the overload controllers using two SEDA applications: a Web-based interface to email, and a Web server benchmark that is able to degrade service under heavy load.

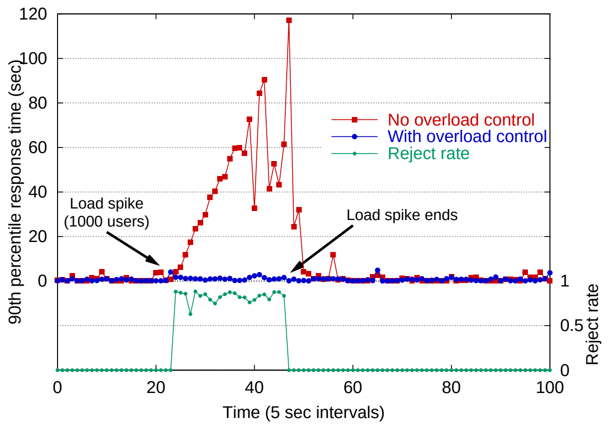


Figure 3: **Overload control under a massive load spike:** *This figure shows the 90th percentile response time experienced by clients using the Arashi e-mail service under a massive load spike (from 10 users to 1000 users). Without overload control, response times grow without bound; with overload control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

5.1 Arashi: A SEDA-based e-mail service

Arashi is a complex, Web-based email application that is akin to Hotmail or Yahoo! Mail, allowing users to access email through a Web browser interface with various functions: managing email folders, deleting and refiling messages, searching for messages, and so forth. Arashi (shown in Figure 1) is built as a SEDA application consisting of 16 stages, each stage handling some aspect of request processing such as HTTP parsing, disk I/O, or dynamic page generation. Email is stored in a MySQL database which runs on the same machine as the Arashi SEDA service; in this way, Arashi's admission control mechanisms effectively condition load on the database. Simulated clients generate load against the Arashi service using a realistic request distribution based on traces from the Berkeley departmental IMAP server.

Response-time-driven overload control is applied to each of the six stages that perform dynamic page processing. These stages are the bottleneck in the system as they perform database access and HTML page generation; the other stages are relatively lightweight. Each stage corresponds to one type of user request (login, listing folders, listing messages, showing a message, deleting/refiling messages and folders, and searching message headers). 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 login to the service again.

Figure 3 shows the performance of the overload controller under a massive load spike on the Arashi e-mail service. A base load of 10 users is rapidly accessing the service when a "flash crowd" of 1000 additional users arrive. Without overload control, client-measured response times grow to be very large. The overload controller maintains a 90th percentile response time target of 1 second, rejecting about 70% to 80% of requests during the spike. Without overload control, there is an enormous increase in response times during the load spike, making the service effectively unusable for all users.

This is in contrast to the common approach 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 once more 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, using per-stage admission control as a load management primitive.

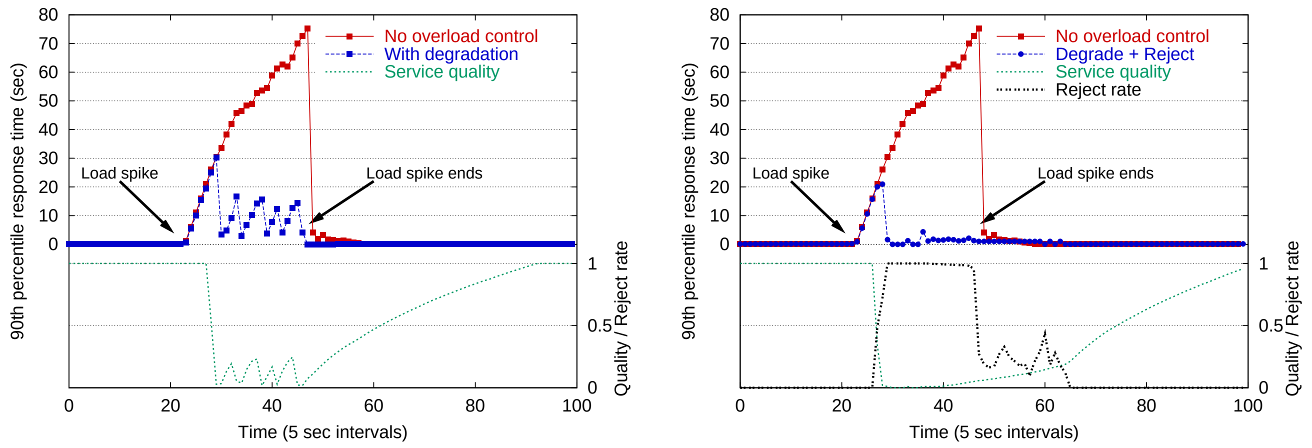
5.2 Service degradation experiments

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.

To demonstrate the use of service degradation in SEDA, we use a simple Web server that responds to each request with a dynamically-generated HTML page that requires significant resources to generate. A single stage acts as a 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 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 may optionally re-enable the admission controller when the quality factor is at its lowest setting and response times continue to exceed the target.

Figure 4 shows the effect of service degradation under an extreme load spike, both with and without the aid of admission control. As the figure shows, 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



(a) Using service degradation **(b) Using service degradation and admission control**

Figure 4: **Effect of service degradation vs. admission control:** *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 reducing the quality of service delivered to clients in order to meet response time demands. Here, the 90th percentile response time target is set to 5 seconds. With no degradation or admission control, response times grow large under a massive load spike of 1000 users. Service degradation alone does a fair job of meeting the response time target under overload, though service degradation coupled with admission control is much more effective.*

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.

6 Conclusions and Future Directions

This position paper has 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 right 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 overload conditions allows the service to make informed resource management decisions, such as 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. Our initial results with this design, as well as considerable scalability and robustness measurements presented elsewhere [25], support the claim that the SEDA approach is an effective way to build robust Internet services.

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.

User-level versus kernel-level load management: An interesting aspect of SEDA 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.

Design and tuning of control mechanisms: 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 [17]. 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. The interaction between multiple levels of control in the system — for example, the interplay between queue admission control and tuning per-stage thread pool sizes — is also largely unexplored.

Composition rules for service components: The presence of an expressive interface for overload management raises the larger question of how to use this mechanism across large applications. Within a SEDA application, stages may independently shed load by rejecting enqueue operations; stages must therefore be defensive with respect to generating load for downstream stages. An interesting design issue arises with respect to composing stages each with their own load-shedding policy. Can any two stages be directly composed, or do some forms of overload control preclude direct communication between stages? Further research might yield a set of composition rules that mandate the interactions between admission-controlled service components.

Overload management as part of the Internet infrastructure: Finally, we hope that future distributed systems developers consider overload as an important design consideration, rather than as only a question of resource provisioning. While current distributed programming models, such as RPC, fail to expose overload, novel application domains such as overlay networks [1] and peer-to-peer systems [10] have the opportunity to rethink this approach. Doing so will hopefully lead to an Internet infrastructure that is more robust to extreme variance of demand.

References

[1] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 2001)*, Banff, Canada, October 2001.

[2] 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.

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

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

[5] A. Cerpa and D. Estrin. ASCENT: Adaptive self-configuring sensor networks topologies. In *Proceedings of the Twenty First International Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002)*, New York, NY, June 2002.

[6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th Symposium on Operating Systems Princples (SOSP-18)*, Chateau Lake Louise, Canada, October 2001.

[7] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for Internet service construction. In *Proc. Fourth Symposium on Operating System Design and Implementation (OSDI 2000)*, October 2000.

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. In *Proceedings of ASPLOS 2000*, Cambridge, MA, November 2000.

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

[10] 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 *Proceeedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.

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

[12] 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.

[13] 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.

[14] D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. In *Proc. OSDI '96*, October 1996.

[15] Open Management Group. The Common Object Request Broker: Architecture and specification, revision 2.3, June 1999.

[16] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proc. 1999 Annual Usenix Technical Conference*, June 1999.

[17] 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 *Proc. IFIP/IEEE International Symposium on Integrated Network Management*, Seattle, WA, May 2001.

[18] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM 2001*, San Diego, CA, August 2001.

[19] J. Richter. *Applied Microsoft .NET Framework Programming*. Microsoft Press, 2002.

[20] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th Symposium on Operating Systems Princples (SOSP-18)*, Chateau Lake Louise, Canada, October 2001.

[21] Sun Microsystems. RPC: Remote Procedure Call Protocol Specification Version 2. Internet Network Working Group RFC1057, June 1988.

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

[23] 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, MA, June 2001.

[24] M. Welsh and D. Culler. Virtualization considered harmful: OS design directions for well-conditioned services. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII)*.

[25] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the 18th Symposium on Operating Systems Princples (SOSP-18)*, Chateau Lake Louise, Canada, October 2001.