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

1 Introduction

In this 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 completely unavailable for several hours due to unprecedented demand [9]. 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 [8, 14], peer-to-peer systems [4, 12], and sensor networks [3, 6], 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. 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 [18]. 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 [5], RPC [15], Java RMI [16], and now .NET [13] 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 (in part 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 which observes the behavior and performance of each stage, and applies resource control and admission control to effectively manage overload.

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 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 [19]. 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 static resource containment is that typified by a variety of real-time and multimedia systems. In this approach, resource limits are typically expressed as proportions 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 static resource guarantee upon entering the system, and are forcibly terminated if resource limits are exceeded.

This approach has been explored in depth by systems such as Scout [11], Nemesis [10], Resource Containers [2], and Cluster Reserves [1]. This technique works 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.

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.* [17] and Jamjoom [7] 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 [17], 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 [7], 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 a 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* [19]. While conceptually simple, this model has a number of desirable properties for overload management:

- **Exposing the request stream:** Event queues make the request stream within the service explicit, allowing the application (and the underlying runtime) 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 both resource control and admission control. Resource controllers attempt to keep each stage within its ideal operating regime by dynamically tuning parameters of the stage's operation, such as the number of threads executing within the stage. This approach frees the application programmer from manually setting "knobs" that can have a serious impact on performance. More details on dynamic resource control in SEDA are given in [19].

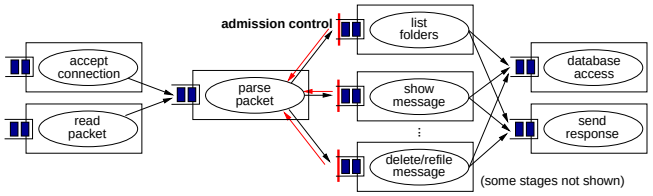

Figure 1: **Structure of the Arashi SEDA-based email service:** *The service consists of a network of stages connected with explicit event queues, coupled with adaptive admission control to prevent overload. For simplicity, some event paths and stages have been elided from this figure.*

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. 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 to the application and allows the application to participate in load management decisions.

Here we describe our experiences with admission control techniques in SEDA to manage overload. The basic approach we take is for the system administrator to specify an external performance target (such as throughput or response time requirement), and to apply per-stage admission control to attempt to meet that target.

3.1 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 which 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 easy to reason about and captures administrators' (and users') intuition of Internet service performance. This is as opposed to mean 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).

3.2 Overload 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*. Several admission control mechanisms are available; here we present results using token-bucket traffic shaping. In this scenario the controller adjusts the rate at

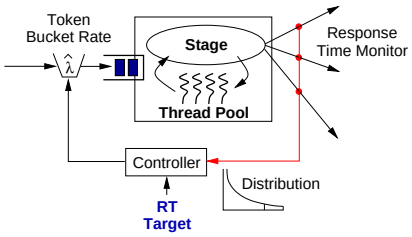

Figure 2: **Response time controller design:** *The controller observes a history of response times through the stage, and adjusts the rate at which the stage accepts new requests to meet an administrator-specified 90th-percentile response time target.*

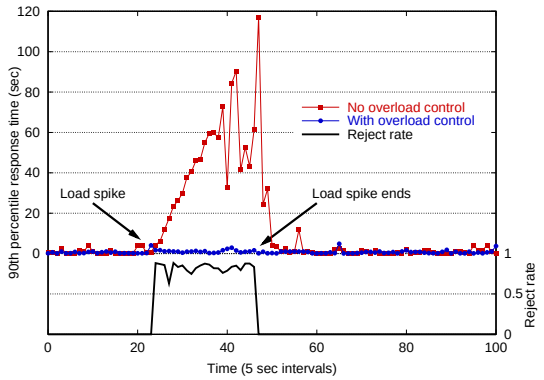

Figure 3: **Overload control under a massive load spike:** *This figure shows the 90th percentile response time experienced by clients using the Arashi e-mail service under a massive load spike (from 10 users to 1000 users). Without overload control, response times grow without bound; with overload control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

which the token bucket generates new tokens, effectively bounding the rate at which the stage accepts new requests.

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. Due to space limitations we have elided certain details of the implementation.

3.3 Evaluation

We evaluate our controller design through a complex, Web-based email application called *Arashi*. Arashi 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 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 which 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 70-80% of requests during the spike. Without overload control, there is an enormous increase in response times during the load spike.

This is in contrast to the common approach of limiting the number of client TCP connections to the service: this approach does not actively monitor response times (a small number of clients could cause a large response time spike), nor does it 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.

Instead of sending error messages to clients, the overload control mechanism could have responded in other ways, such as by degrading the service quality (e.g., limiting the number of e-mail messages displayed per page, or removing inlined images), or redirecting requests to other nodes in a server farm. Likewise, variants on the response time admission control metric could be used, such as class-based prioritization. For example, the client IP address could be used to assign a class to each request, allowing the system to deliver different levels of service to each class. While space limitations prevent us from providing further details, this example is strongly representative of the benefits of the SEDA architecture: exposing overload to applications, automatically shedding excess load, and incorporating overload management as a primitive in the service construction framework.

4 Conclusions

We argue that it is critically important to address the problem of overload from a systems services design perspective, rather than through *ad hoc* approaches lashed onto existing systems. We claim that feedback and dynamic control are the right ways to approach overload management, rather than static resource partitioning or simplistic mechanisms such as prioritization. The SEDA architecture makes it possible to build software which is inherently resilient to load, by exposing per-stage adaptive admission control directly to the programming model. Our initial results with this design, as well as considerable scalability and robustness measurements presented elsewhere [19], support the claim that the SEDA approach is an effective way to build robust Internet services.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] O. M. Group. The common object request broker: Architecture and specification, revision 2.3, June 1999.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

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

[12] 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.

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

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

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

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

[17] 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.

[18] 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)*.

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