SEDA: An Architecture for Well-Conditioned, Scalable Internet Services

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

Abstract

We propose a new design for highly concurrent Internet services, which we call the *staged event-driven architecture* (SEDA). SEDA is intended to support massive concurrency demands and simplify the construction of well-conditioned services. In SEDA, applications consist of a network of event-driven *stages* connected by explicit *queues*. This architecture allows services to be well-conditioned to load, preventing resources from being overcommitted when demand exceeds service capacity. SEDA makes use of a set of *dynamic resource controllers* to keep stages within their operating regime despite large fluctuations in load. We describe several control mechanisms for automatic tuning and load conditioning, including thread pool sizing, event batching, and adaptive load shedding. We present the SEDA design and an implementation of an Internet services platform based on this architecture. We evaluate the use of SEDA through two applications: a high-performance HTTP server and a packet router for the Gnutella peer-to-peer file sharing network. These results show that SEDA applications exhibit higher performance than traditional service designs, and are robust to huge variations in load.

1 Introduction

The Internet presents a computer systems problem of unprecedented scale: that of supporting millions of users demanding access to services that must be responsive, robust, and always available. The number of concurrent sessions and hits per day to Internet sites translates into an even higher number of I/O and network requests, placing enormous demands on underlying resources. Yahoo! receives over 1.2 billion page views daily [62], and AOL's Web caches service over 10 billion hits a day [2]. Moreover, Internet services experience huge variations in service load, with bursts coinciding with the times that the service has the most value. The well-documented "Slashdot Effect" shows that it is not uncommon to experience more than 100-fold increases in demand when a site becomes popular [58]. As the demand for Internet services grows, new system design techniques must be used to manage this load.

This systems challenge is magnified by three trends that increase the generality of services. First, services themselves are becoming more complex, with static content replaced by dynamic content that involves extensive computation and I/O. Second, service logic tends to change rapidly, which increases the complexity of engineering and deployment. Third, services are increasingly hosted on general-purpose facilities, rather than on platforms that are carefully engineered for a particular service. As these trends continue, we envision that a rich array of novel services will be authored and pushed into the infrastructure where they may become successful enough to scale to millions of users. Several investigations are addressing the high-level aspects of service authorship, including naming, lookup, composition, and versioning [16, 21, 22, 53, 55]. We focus here on the performance aspect of the problem: achieving robust performance on a wide range of services subject to huge variations in load, while preserving ease of authorship.

Replication is a key aspect of service scalability. Given a service instance that can sustain a certain level of performance, it must be replicated to sustain a many-fold increase in load. Scalable clusters are now widely used to obtain replication within a service site [18], and wide-area replication is increasingly employed for specific services, such as content distribution networks [1, 3, 19]. However, because the peak load may be orders of magnitude greater than the average, it is not practical to replicate most services to handle the maximum potential demand. Therefore, we expect large spikes in the load experienced by each node. Our goal is to develop a general framework for authoring highly concurrent and well-conditioned service instances that handle load gracefully.

Unfortunately, traditional operating system designs and widely promoted models of concurrency do not provide this graceful management of load. Commodity operating systems focus on providing maximal transparency by giving each process the abstraction of a virtual machine with its own CPU, memory, disk, and network. This goal is somewhat at odds with the needs of Internet services, which demand massive concurrency and extensive control over resource usage. Processes and threads are well-supported models of concurrent programming, but often entail high overhead in terms of context-switch time and memory footprint, which limits concurrency. Transparent resource virtualization prevents applications from making informed decisions, which are vital to managing excessive load.

Much work has focused on performance and robustness for specific services [4, 24, 44, 63]. However, with services becoming increasingly dynamic and flexible, this engineering burden becomes excessive. Few tools exist that aid the development of highly concurrent, well-conditioned services; our goal is to reduce this complexity by providing general-purpose mechanisms that aid software developers in obtaining these properties.

For more information as well as a source-code release of the software described in this paper, please see http://www.cs.berkeley.edu/~mdw/proj/seda/.

To appear in the Eighteeth Symposium on Operating Systems Principles (SOSP-18), Chateau Lake Louise, Canada, October 21-24, 2001.

We propose a new design framework for highly concurrent server applications, which we call the *staged event-driven architecture* (SEDA).[1] SEDA combines aspects of threads and event-based programming models to manage the concurrency, I/O, scheduling, and resource management needs of Internet services. In SEDA, applications are constructed as a network of *stages*, each with an associated *incoming event queue*. Each stage represents a robust building block that may be individually conditioned to load by thresholding or filtering its event queue. In addition, making event queues explicit allows applications to make informed scheduling and resource-management decisions, such as reordering, filtering, or aggregation of requests. SEDA makes use of *dynamic resource throttling* to control the resource allocation and scheduling of application components, allowing the system to adapt to overload conditions.

This paper describes the design, architecture, and implementation of a SEDA-based Internet services platform. This platform provides efficient, scalable I/O interfaces as well as several resource control mechanisms, including thread pool sizing and dynamic event scheduling. We evaluate the framework through two applications — a high-performance HTTP server and a packet router for the Gnutella peer-to-peer file-sharing network. We present performance and scalability results for these applications, demonstrating that SEDA achieves robustness over huge variations in load and outperforms other service designs. Our Java-based SEDA HTTP server outperforms two popular Web servers implemented in C, as described in Section 5.1. We argue that using SEDA, highly concurrent applications are easier to build, more efficient, and more robust to load. With the right set of interfaces, application designers can focus on application-specific logic, rather than the details of concurrency and resource management.

2 Background and Related Work

SEDA draws together two important lines of research: the use of thread-based concurrency models for ease of programming and event-based models for extensive concurrency. This section develops the lineage of this approach by outlining the key contributions and problems in the steps leading to the SEDA design.

Intuitively, a service is *well-conditioned* if it behaves like a simple pipeline, where the depth of the pipeline is determined by the path through the network and the processing stages within the service itself. As the offered load increases, the delivered throughput increases proportionally until the pipeline is full and the throughput saturates; additional load should not degrade throughput. Similarly, the response time exhibited by the service is roughly constant at light load, because it is dominated by the depth of the pipeline. As load approaches saturation, the queueing delay dominates. In the closed-loop scenario typical of many services, where each client waits for a response before delivering the next request, response time should increase linearly with the number of clients.

The key property of a well-conditioned service is *graceful degradation*: as offered load exceeds capacity, the service maintains high throughput with a linear response-time penalty that impacts all clients equally, or at least predictably according to some service-specific policy. Note that this is not the typical Web experience; rather, as load increases, throughput decreases and response time increases dramatically, creating the impression that the service has crashed.

2.1 Thread-based concurrency

The most commonly used design for server applications is the thread-per-request model, as embodied in RPC packages [52], Java Remote Method Invocation [54], and DCOM [37]. This model is well supported by modern languages and programming environments. In this model, shown in Figure 1, each accepted request consumes a thread to process it, with synchronization operations protecting shared resources. The operating system overlaps computation and I/O by transparently switching among threads.

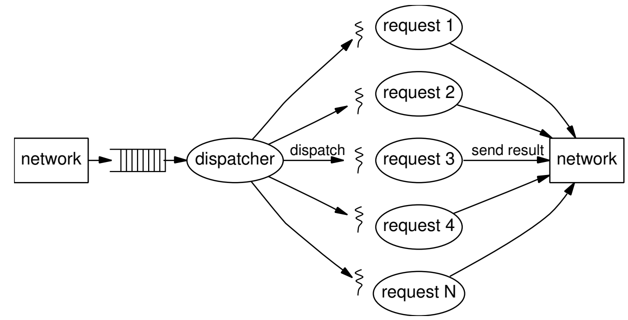


Figure 1: **Threaded server design:** *Each incoming request is dispatched to a separate thread, which processes the request and returns a result to the client. Edges represent control flow between components. Note that other I/O operations, such as disk access, are not shown here, but would be incorporated into each threads' request processing.*

Although relatively easy to program, the overheads associated with threading — including cache and TLB misses, scheduling overhead, and lock contention — can lead to serious performance degradation when the number of threads is large. As a concrete example, Figure 2 shows the performance of a simple threaded server as the number of threads increases. Although the effective thread limit would be large for general-purpose timesharing, it is not adequate for the tremendous concurrency requirements of an Internet service.

Threads and processes are primarily designed to support multiprogramming, and existing OSs strive to virtualize hardware resources in a way that is transparent to applications. Applications are rarely given the opportunity to participate in system-wide resource management decisions, or given indication of resource availability in order to adapt their behavior to changing conditions. Virtualization fundamentally hides the fact that resources are limited and shared [61].

A number of systems have attempted to remedy this problem by exposing more control to applications. Scheduler activations [5], application-specific handlers [59], and operating systems such as SPIN [11], Exokernel [28], and Nemesis [34] are all attempts to augment limited operating system interfaces by giving applications the ability to specialize the policy decisions made by the kernel. However, the design of these systems is still based on multiprogramming, as the focus continues to be on safe and efficient resource virtualization, rather than on graceful management and high concurrency.

2.2 Bounded thread pools

To avoid the overuse of threads, a number of systems adopt a coarse form of load conditioning that serves to bound the size of the thread pool associated with a service. When the number of requests in the server exceeds some fixed limit, additional connections are not accepted. This approach is used by Web servers such as Apache [6], IIS [38], and Netscape Enterprise Server [42], as well as application servers such as BEA Weblogic [10] and IBM WebSphere [25]. By limiting the number of concurrent threads, the server can avoid throughput degradation, and the overall performance is more robust than the unconstrained thread-per-task model. However, this approach can introduce a great deal of *unfairness* to clients: when all server threads are busy or blocked, client requests queue up in the network for servicing. As we will show in Section 5.1, this can cause clients to experience arbitrarily large waiting times.

When each request is handled by a single thread, it is difficult to

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

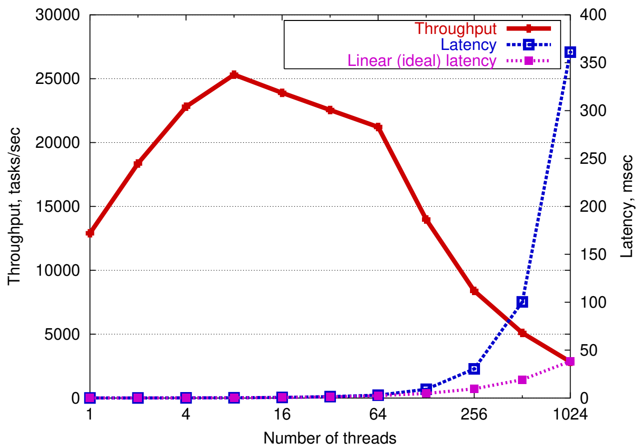


Figure 2: **Threaded server throughput degradation:** *This benchmark measures a simple threaded server which creates a single thread for each task in the pipeline. After receiving a task, each thread performs an 8 KB read from a disk file; all threads read from the same file, so the data is always in the buffer cache. Threads are pre-allocated in the server to eliminate thread startup overhead from the measurements, and tasks are generated internally to negate network effects. The server is implemented in C and is running on a 4-way 500 MHz Pentium III with 2 GB of memory under Linux 2.2.14. As the number of concurrent tasks increases, throughput increases until the number of threads grows large, after which throughput degrades substantially. Response time becomes unbounded as task queue lengths increase; for comparison, we have shown the ideal linear response time curve (note the log scale on the x axis).*

identify internal performance bottlenecks in order to perform tuning and load conditioning. Consider a simple threaded Web server in which some requests are inexpensive to process (e.g., cached static pages) and others are expensive (e.g., large pages not in the cache). With many concurrent requests, it is likely that the expensive requests could be the source of a performance bottleneck, for which it is desirable to perform load shedding. However, the server is unable to inspect the internal request stream to implement such a policy; all it knows is that the thread pool is saturated, and must arbitrarily reject work without knowledge of the source of the bottleneck.

Resource containers [7] and the concept of *paths* from the Scout operating system [41, 49] are two techniques that can be used to bound the resource usage of tasks in a server. These mechanisms apply vertical resource management to a set of software modules, allowing the resources for an entire data flow through the system to be managed as a unit. In the case of the bottleneck described above, limiting the resource usage of a given request would avoid degradation due to cache misses, but allow cache hits to proceed unabated.

2.3 Event-driven concurrency

The scalability limits of threads have led many developers to eschew them almost entirely and employ an event-driven approach to managing concurrency. In this approach, shown in Figure 3, a server consists of a small number of threads (typically one per CPU) that loop continuously, processing events of different types from a queue. Events may be generated by the operating system or internally by the application, and generally correspond to network and disk I/O readiness and completion notifications, timers, or other application-specific events. The event-driven approach implements the processing of each task as a finite state machine, where transitions between states in the FSM are triggered by events. In this way the server maintains its own continuation state for each task rather than relying upon a thread context.

The event-driven design is used by a number of systems, including

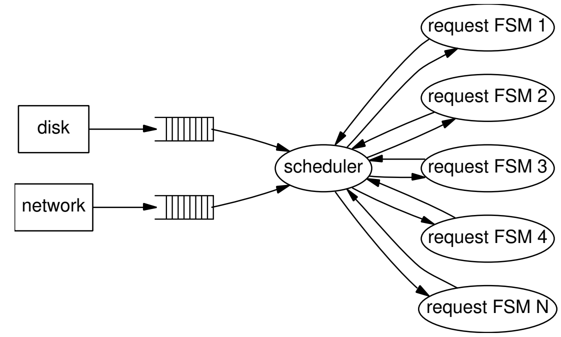


Figure 3: **Event-driven server design:** *This figure shows the flow of events through an event-driven server. The main thread processes incoming events from the network, disk, and other sources, and uses these to drive the execution of many finite state machines. Each FSM represents a single request or flow of execution through the system. The key source of complexity in this design is the event scheduler, which must control the execution of each FSM.*

the Flash [44], thttpd [4], Zeus [63], and JAWS [24] Web servers, and the Harvest [12] Web cache. In Flash, each component of the server responds to particular types of events, such as socket connections or filesystem accesses. The main server process is responsible for continually dispatching events to each of these components, which are implemented as library calls. Because certain I/O operations (in this case, filesystem access) do not have asynchronous interfaces, the main server process handles these events by dispatching them to *helper processes* via IPC. Helper processes issue (blocking) I/O requests and return an event to the main process upon completion. Harvest's structure is very similar: it is single-threaded and event-driven, with the exception of the FTP protocol, which is implemented by a separate process.

The tradeoffs between threaded and event-driven concurrency models have been studied extensively in the JAWS Web server [23, 24]. JAWS provides a framework for Web server construction allowing the concurrency model, protocol processing code, cached filesystem, and other components to be customized. Like SEDA, JAWS emphasizes the importance of adaptivity in service design, by facilitating both static and dynamic adaptations in the service framework. To our knowledge, JAWS has only been evaluated under light loads (less than 50 concurrent clients) and has not addressed the use of adaptivity for conditioning under heavy load.

Event-driven systems tend to be robust to load, with little degradation in throughput as offered load increases beyond saturation. Figure 4 shows the throughput achieved with an event-driven implementation of the service from Figure 2. As the number of tasks increases, the server throughput increases until the pipeline fills and the bottleneck (the CPU in this case) becomes saturated. If the number of tasks in the pipeline is increased further, excess tasks are absorbed in the server's event queue. The throughput remains constant across a huge range in load, with the latency of each task increasing linearly.

An important limitation of this model is that it assumes that event-handling threads do not block, and for this reason nonblocking I/O mechanisms must be employed. Although much prior work has investigated scalable I/O primitives [8, 9, 33, 46, 48], event-processing threads can block regardless of the I/O mechanisms used, due to interrupts, page faults, or garbage collection.

Event-driven design raises a number of additional challenges for the application developer. Scheduling and ordering of events is probably the most important concern: the application is responsible for deciding when to process each incoming event and in what order to process the FSMs for multiple flows. In order to balance fairness with low response time, the application must carefully multiplex the execution of multiple

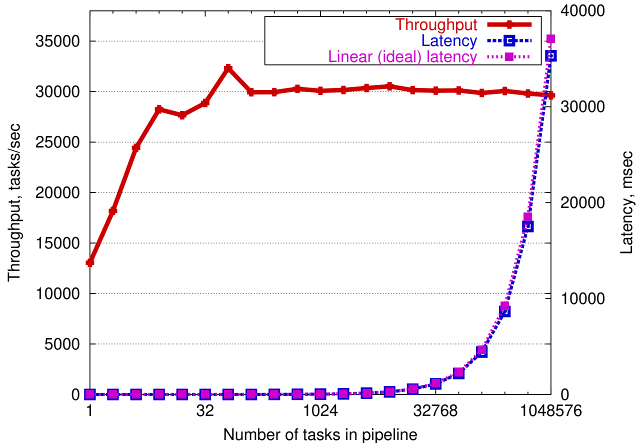


Figure 4: **Event-driven server throughput:** *This benchmark measures an event-driven version of the server from Figure 2. In this case, the server uses a single thread to process tasks, where each task reads 8 KB from a single disk file. Although the filesystem interface provided by the operating system used here (Linux 2.2.14) is blocking, because the disk data is always in the cache, this benchmark estimates the best possible performance from a nonblocking disk I/O layer. As the figure shows, throughput remains constant as the load is increased to a very large number of tasks (note the change in the horizontal axis scale from Figure 2), and response time is linear (note the log scale on the x axis).*

FSMs. The choice of an event scheduling algorithm is often tailored to the specific application, and introduction of new functionality may require the algorithm to be redesigned. Also, modularity is difficult to achieve, as the code implementing each state must be trusted not to block or consume a large number of resources that can stall the event-handling thread.

2.4 Structured event queues

Several variants on the standard event-driven design have been proposed to counter the problems outlined above. A common aspect of these designs is to structure an event-driven application using a set of event queues to improve code modularity and simplify application design.

The Click modular packet router [40] is one such example. In Click, packet processing stages are implemented by separate code modules with their own private state. Click is optimized to improve per-packet latency through the router, allowing a single thread to call directly through multiple packet-processing stages. This design is targeted at a specific application (routing) and a single thread services all event queues. Click makes the assumption that modules have bounded processing times, leading to a relatively static resource-management policies. Qie *et al.* [47] also describe techniques for scheduling and load conditioning in a software-based router; like SEDA, their design makes use of controllers to adjust runtime parameters dynamically based on load.

Gribble's Distributed Data Structures (DDS) [20] layer also makes use of a structured event-processing framework. In DDS, storage servers emulate asynchronous network and disk I/O interfaces by making use of fixed-size thread pools, and software components are composed using either explicit event queues or implicit upcalls. Work Crews [56] and the TSS/360 queue scanner [35] are other examples of systems that make use of structured event queues and limited numbers of threads to manage concurrency. In each of these systems, the use of an event queue decouples the execution of two components, which improves modularity and robustness.

StagedServer [31] is another system that makes use of modules communicating using explicit event queues. In this case, the goal is to maximize processor cache locality by carefully scheduling threads and events within each module. By aggregating the execution of multiple similar events within a queue, locality is enhanced, leading to greater performance.

Lauer and Needham's classic paper [32] discusses the merits of processes communicating via messages and contrasts this approach to that of "procedures," closely related to the threaded model described above. SEDA can be seen as an instance of the message-oriented model discussed there. The authors claim that the message-based and procedure-based models are duals of each other, and that any program implemented in one model can just as efficiently be implemented in the other. While we agree with this basic sentiment, this argument overlooks the complexity of building scalable general-purpose multithreading, as well as the inherent difficulties of adapting to load in a thread-based model, without an explicit request queue.

3 The Staged Event-Driven Architecture

In this section we propose a new software architecture, the *staged event-driven architecture* (SEDA), which is designed to enable high concurrency, load conditioning, and ease of engineering for Internet services. SEDA decomposes an application into a network of *stages* separated by *event queues* and introduces the notion of *dynamic resource controllers* to allow applications to adjust dynamically to changing load. An overview of the SEDA approach to service design is shown in Figure 5.

3.1 Goals

The primary goals for SEDA are as follows:

Support massive concurrency: To avoid performance degradation due to threads, SEDA makes use of event-driven execution wherever possible. This also requires that the system provide efficient and scalable I/O primitives.

Simplify the construction of well-conditioned services: To reduce the complexity of building Internet services, SEDA shields application programmers from many of the details of scheduling and resource management. The design also supports modular construction of these applications, and provides support for debugging and performance profiling.

Enable introspection: Applications should be able to analyze the request stream to adapt behavior to changing load conditions. For example, the system should be able to prioritize and filter requests to support degraded service under heavy load.

Support self-tuning resource management: Rather than mandate *a priori* knowledge of application resource requirements and client load characteristics, the system should adjust its resource management parameters dynamically to meet performance targets. For example, the number of threads allocated to a stage can be determined automatically based on perceived concurrency demands, rather than hard-coded by the programmer or administrator.

3.2 Stages as robust building blocks

The fundamental unit of processing within SEDA is the *stage*. A stage is a self-contained application component consisting of an *event handler*, an *incoming event queue*, and a *thread pool*, as depicted in Figure 6. Each stage is managed by a *controller* that affects scheduling and thread allocation, as described below. Stage threads 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 enqueuing them on the event queues of other stages.

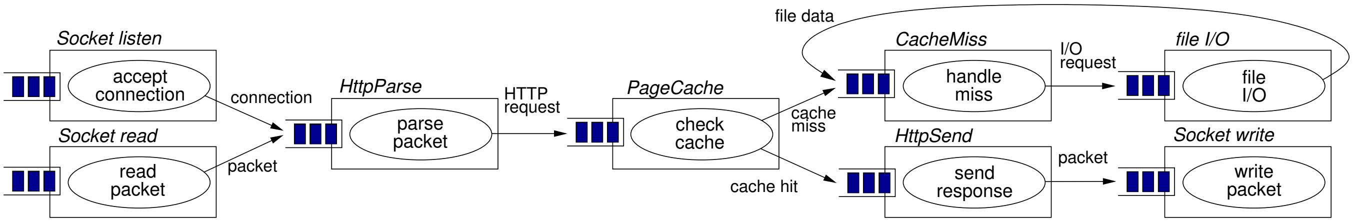


Figure 5: **Staged event-driven (SEDA) HTTP server:** *This is a structural representation of the SEDA-based Web server, described in detail in Section 5.1. The application is composed as a set of* stages *separated by* queues*. Edges represent the flow of events between stages. Each stage can be independently managed, and stages can be run in sequence or in parallel, or a combination of the two. The use of event queues allows each stage to be individually load-conditioned, for example, by thresholding its event queue. For simplicity, some event paths and stages have been elided from this figure.*

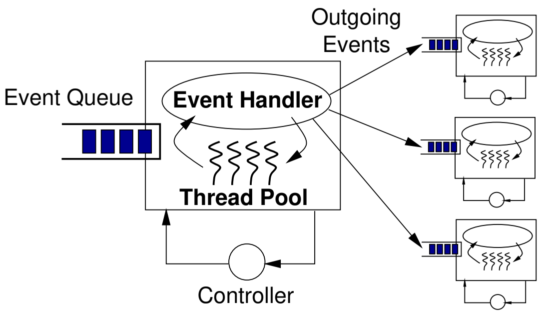
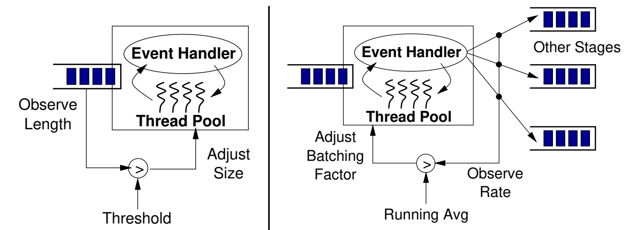


Figure 6: **A SEDA Stage:** *A stage consists of an* incoming event queue*, a* thread pool*, and an application-supplied* event handler*. The stage's operation is managed by the* controller*, which adjusts resource allocations and scheduling dynamically.*



(a) Thread pool controller **(b) Batching controller**

Figure 7: **SEDA resource controllers:** *Each stage has an associated* controller *that adjusts its resource allocation and behavior to keep the application within its operating regime. The* thread pool controller *adjusts the number of threads executing within the stage, and the* batching controller *adjusts the number of events processed by each iteration of the event handler.*

Threads are the basic concurrency mechanism within SEDA, yet their use is limited to a small number of threads per stage, rather than a single thread per task in the system. Moreover, the use of dynamic control (see Section 3.4) can automatically tune the number of threads allocated to each stage based on demand.[2] This design allows stages to run in sequence or in parallel, or a combination of the two, depending upon the characteristics of the thread system and scheduler. In this paper we assume preemptive, OS-supported threads in an SMP environment, although this choice is not fundamental to the SEDA design. For example, a thread system could be designed which is cognizant of the staged structure of the application and schedules threads accordingly. We return to this issue in Section 3.4.

The core logic for each stage is provided by the event handler, the input to which is a batch of multiple events. Event handlers do not have direct control over queue operations or threads. By separating core application logic from thread management and scheduling, the stage is able to control the execution of the event handler to implement various resource-management policies. For example, the number and ordering of events passed to the event handler can be controlled externally by the runtime environment. However, the application may also implement its own scheduling policy by filtering or reordering the event batch passed to it.

3.3 Applications as a network of stages

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

An important aspect of event queues in SEDA is that they may be *finite*: that is, an enqueue operation may fail if the queue wishes to reject new entries, say, because it has reached a threshold. Applications may make use of backpressure (by blocking on a full queue) or load shedding (by dropping events) when enqueue operations fail. Alternately, the application may wish to take some service-specific action, such as sending an error to the user, or performing an alternate function, such as providing degraded service.

Figure 5 illustrates the structure of a SEDA-based application, in this case the Haboob Web server described in Section 5.1. The application consists of a number of application-specific stages to process HTTP requests, implement a page cache, and so forth, as well as several generic stages provided by the runtime to support asynchronous I/O. These interfaces are described further in Section 4.

The introduction of a queue between stages decouples their execution by introducing an explicit control boundary. This model constrains the execution of a thread to a given stage, as a thread may only pass data across the control boundary by enqueuing an event. A basic question is whether two code modules should communicate by means of a queue, or directly through a subroutine call. Introducing a queue between two modules provides isolation, modularity, and independent load management, but may increase latency. For example, a third-party code module can be isolated in its own stage, allowing other stages to communicate with it through its event queue, rather than by calling it directly.

The SEDA design facilitates debugging and performance analysis of services, which has traditionally been a challenge for complex multithreaded servers. The decomposition of application code into stages and explicit event delivery mechanisms facilitates inspection; for example, a debugging tool can trace the flow of events through the system and visualize the interactions between stages. Because stages interact through

[2] Rather than allocating a separate thread pool per stage, it is possible to have multiple stages share the same thread pool. To simplify the discussion, we describe SEDA in terms of a private thread pool per stage. Note also that the number of stages in an application is typically much smaller than the number of threads that the system can support, so a separate thread pool per stage is reasonable.

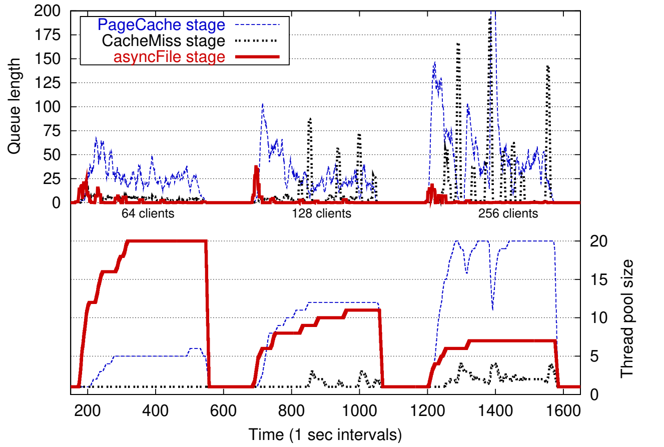


Figure 8: **SEDA thread pool controller:** *This graph shows the operation of the thread pool controller during a run of the Haboob Web server, described in Section 5.1. The controller adjusts the size of each stage's thread pool based on the length of the corresponding event queue. In this run, the queue length was sampled every 2 seconds and a thread was added to the pool if the queue exceeded 100 entries (with a maximum per-stage limit of 20 threads). Threads are removed from the pool when they are idle for more than 5 seconds. The* asyncFile *stage uses a controller threshold of 10 queue entries to exaggerate the controller's behavior.*

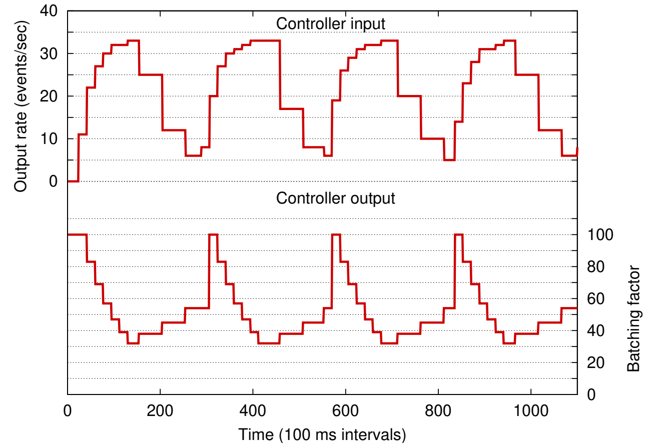


Figure 9: **SEDA batching controller:** *This graph shows the operation of the batching controller for a simple benchmark consisting of a single stage generating events at an oscillating rate. This causes the measured output rate of the stage to vary as shown in the top portion of the figure. While the output rate increases, the controller decreases the batching factor. When the output rate decreases, the controller increases the batching factor. The batching factor is reset to its maximum value after a sudden drop in the output rate.*

an event-dispatch protocol instead of a traditional API, it is straightforward to interpose proxy stages between components for debugging and performance profiling. Using this mechanism, our prototype of SEDA is capable of generating a graph depicting the set of application stages and their relationship. The prototype can also generate temporal visualizations of event queue lengths, memory usage, and other system properties that are valuable in understanding performance.

3.4 Dynamic resource controllers

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

We have implemented several resource controllers in SEDA, two of which are shown in Figure 7. The first is the *thread pool controller*, which adjusts the number of threads executing within each stage. The goal is to avoid allocating too many threads, but still have enough threads to meet the concurrency demands of the stage. The controller periodically samples the input queue and adds a thread when the queue length exceeds some threshold, up to a maximum number of threads per stage. Threads are removed from a stage when they are idle for a specified period of time. Figure 8 shows the effect of the thread pool controller operating within the Web server described in Section 5.1; the controller operation is discussed in more detail in Section 4.2.

The second is the *batching controller*, which adjusts the number of events processed by each invocation of the event handler within a stage (the *batching factor*). It has been observed [31] that processing many events at once increases throughput, as cache locality and task aggregation can be performed. However, a large batching factor can also increase response time. The controller attempts to trade off these effects by searching for the smallest batching factor that sustains high throughput. It operates by observing the output rate of events from a stage (by maintaining a moving average across many samples) and decreases the batching factor until throughput begins to degrade. If throughput degrades slightly, the batching factor is increased by a small amount. The controller responds to sudden drops in load by resetting the batching factor to its maximum value. Figure 9 shows the batching controller at work.

These mechanisms represent two simple examples of dynamic control in SEDA. It is possible to introduce more complex controllers into the system; for example, a controller might adjust thread pool sizes based on a global notion of stage priority, or to keep the number of threads in the entire system below some threshold. Another option is to adjust thread scheduling parameters based on the stage's progress, as proposed by Steere *et al.* [51]. The SEDA asynchronous sockets library, described in the next section, contains an optional controller that throttles the rate at which packets are read from the network. In Section 5.1 we describe an application-specific controller that adaptively sheds load to meet a response time target. SEDA's structure facilitates inspection and control of the underlying application, and a range of control strategies are possible in this model.

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

3.5 Sandstorm: A SEDA prototype

We have implemented a SEDA-based Internet services platform, called *Sandstorm*. Sandstorm is implemented entirely in Java, and makes use of a set of native libraries for nonblocking socket I/O (described in Section 4). Using the latest Java implementations, coupled with judicious use of Java's language features, we have found the software engineering and robustness benefits of using Java have more than outweighed the performance tradeoffs. For instance, we rely on Java's automated memory management to garbage-collect "expired" events as they pass through the system; this greatly simplifies the code as components are not responsible for tracking the lifetime of events. The performance gap between Java and statically compiled languages is also closing; in fact, our Java-based SEDA Web server outperforms two popular Web servers implemented in C, as described in Section 5.1.

In Sandstorm, each application module implements a simple event handler interface with a single method call, `handleEvents()`, which processes a batch of events pulled from the stage's incoming event queue. Applications do not create or manage threads; this is the responsibility of the runtime system and associated controllers. Sandstorm provides a thread manager interface that can be tailored to implement various thread allocation and scheduling policies; the version described here manages a pool of threads for each stage and relies upon the underlying OS for scheduling. Sandstorm provides APIs for naming, creating, and destroying stages, performing queue operations, controlling queue thresholds, as well as profiling and debugging. The socket and file I/O mechanisms described in the next section are provided as standard interfaces.

The Sandstorm runtime consists of 19934 lines of code with 7871 non-commenting source statements (NCSS). Of this, 3023 NCSS are devoted to the core runtime and 2566 to the I/O facilities.

4 Asynchronous I/O Primitives

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

4.1 Asynchronous socket I/O

The Sandstorm asynchronous socket (*asyncSocket*) layer provides an easy-to-use nonblocking sockets interface for services. Applications create instances of the classes *asyncClientSocket* and *asyncServerSocket* to initiate outgoing and incoming socket connections. When a connection is established, an *asyncConnection* object is pushed onto the event queue provided by the user (typically the queue associated with the requesting stage). Incoming packets are enqueued onto the user's event queue, and *asyncConnection* implements a queue interface onto which outgoing packets can be placed. Each outgoing packet may also have an associated event queue onto which a completion event is pushed when the packet is transmitted. Error and other notification events are passed to the user in a similar way.

Internally, the asyncSocket layer is implemented using three stages, which are shared across all sockets, as shown in Figure 10. *readStage* reads network packets and responds to user requests to initiate packet reading on a new socket. *writeStage* writes packets to the network and establishes new outgoing connections. *listenStage* accepts new TCP connections and responds to user requests to listen on a new port. Each operation on an *asyncConnection*, *asyncClientSocket*, or *asyncServerSocket* is converted into a request and placed onto the appropriate stage's request queue.

Each asyncSocket stage services two separate event queues: a request queue from the user, and an I/O readiness/completion event queue

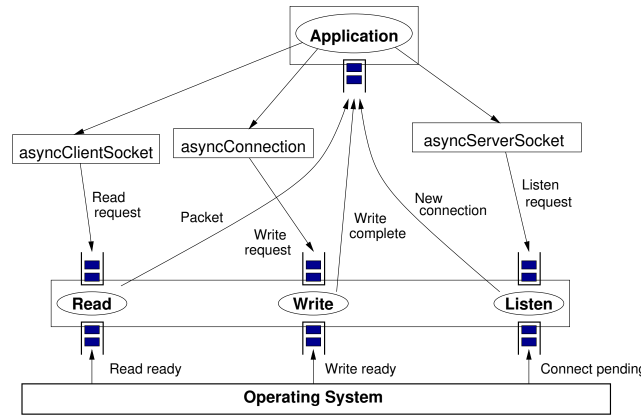


Figure 10: **SEDA-based asynchronous sockets layer:** *The Sandstorm sockets interface consists of three stages:* read*,* write*, and* listen*. The* read *stage responds to network I/O readiness events and reads data from sockets, pushing new packets to the application stage. The* write *stage accepts outgoing packets and schedules them for writing to the appropriate socket. It also establishes new outgoing socket connections. The* listen *stage accepts new TCP connections and pushes connection events to the application.*

from the operating system. The thread within each stage alternately services each queue, using a simple timeout mechanism to toggle between the two. The I/O event queue is implemented as a library that causes dequeue operations to invoke the appropriate OS call to retrieve I/O events. Our current implementation supports the standard UNIX *poll(2)* system call as well as the `/dev/poll` [46] interface for event delivery. A native library is used to provide nonblocking socket calls in Java [60]. To increase fairness across sockets, each stage *randomizes* the order in which it processes I/O events delivered by the operating system. This is necessary because the OS generally returns socket events in a fixed order (e.g., in increasing order by file descriptor).

readStage operates by performing a socket read whenever an I/O readiness event indicates that a socket has data available. It reads at most 16 KB into a pre-allocated buffer and enqueues the resulting packet onto the event queue provided by the user. In case of an I/O error (e.g., because the peer has closed the connection), the stage closes the socket and pushes an appropriate notification event to the user. Each socket read requires the allocation of a new packet buffer; while this can potentially cause a great deal of garbage collection overhead, we have not found this to be a performance issue. Note that because this system is implemented in Java, no explicit deallocation of expired packets is necessary. *readStage* also provides an optional rate controller that can throttle the rate at which packets are read from the network; this controller is useful for performing load shedding during overload conditions. The controller is implemented by calculating a moving average of the incoming packet rate and introducing artificial delays into the event-processing loop to achieve a certain rate target.

writeStage receives packet write requests from the user and enqueues them onto an internal queue associated with the particular socket. When the OS indicates that a socket is ready for writing, it attempts to write the next packet on that socket's outgoing queue. As described in Section 5.2, the socket queue may be thresholded to prevent "slow" sockets from consuming too many resources in the server.

To evaluate the performance of asyncSocket, we implemented a simple server application that accepts bursts of 8KB packets from a number of clients, responding with a single 32-byte ACK for each burst of 1000 packets. This somewhat artificial application is meant to stress the network layer and measure its scalability as the number of clients increases. Figure 11 shows the aggregate throughput of the server as

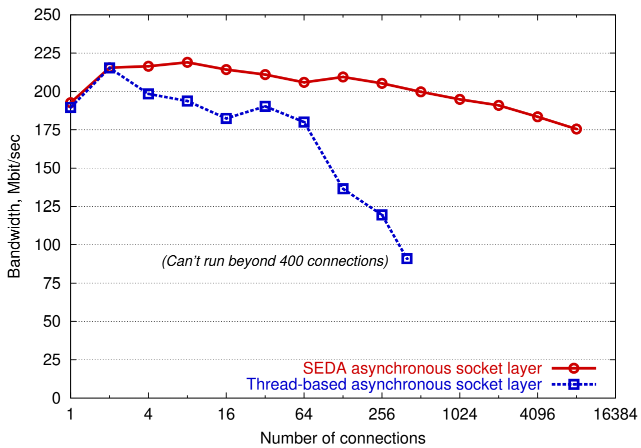


Figure 11: **Asynchronous sockets layer performance:** *This graph shows the performance of the SEDA-based asynchronous socket layer as a function of the number of simultaneous connections. Each client opens a connection to the server and issues bursts of 8KB packets; the server responds with a single 32-byte ACK for each burst of 1000 packets. All machines are connected via switched Gigabit Ethernet and are running Linux 2.2.14. The SEDA-based server makes use of nonblocking I/O primitives provided by the operating system. Performance is compared against a compatibility layer that makes use of blocking sockets and multiple threads to emulate asynchronous I/O. The thread-based layer was unable to accept more than 400 simultaneous connections, because the number of threads required would exceed the per-user thread limit in Linux.*

the number of clients increases from 1 to 8192. The server and client machines are all 4-way 500 MHz Pentium III systems interconnected using Gigabit Ethernet running Linux 2.2.14 and IBM JDK 1.3.

Two implementations of the socket layer are shown. The SEDA-based layer makes use of nonblocking I/O provided by the OS and the `/dev/poll` event-delivery mechanism [46]. This is compared against a compatibility layer that uses blocking sockets and a thread pool for emulating asynchronous I/O. This layer creates one thread per connection to process socket read events and a fixed-size pool of 120 threads to handle socket writes. This compatibility layer was originally developed to provide asynchronous I/O under Java, which does not provide this functionality directly.

The nonblocking implementation clearly outperforms the threaded version, which degrades rapidly as the number of connections increases. In fact, the threaded implementation crashes when receiving over 400 connections, as the number of threads required exceeds the per-user thread limit in Linux. The slight throughput degradation for the nonblocking layer is due in part to lack of scalability in the Linux network stack; even using the highly-optimized `/dev/poll` mechanism [46] for socket I/O event notification, as the number of sockets increases the overhead involved in polling readiness events from the operating system increases significantly [29].

4.2 Asynchronous file I/O

The Sandstorm file I/O (*asyncFile*) layer represents a very different design point than asyncSocket. Because the underlying operating system does not provide nonblocking file I/O primitives, we are forced to make use of blocking I/O and a bounded thread pool to implement this layer.[3] Users perform file I/O through an *asyncFile* object, which supports the familiar interfaces *read*, *write*, *seek*, *stat*, and *close*. Each of these operations translates into a request being placed on the asyncFile stage's event queue. asyncFile threads dequeue each request and perform the corresponding (blocking) I/O operation on the file. To ensure that multiple I/O requests on the same file are executed serially, only one thread may process events for a particular file at a time. When an I/O request completes, a corresponding completion event is enqueued onto the user's event queue.

The asyncFile stage is initialized with a single thread in its thread pool. The SEDA thread pool controller is responsible for dynamically adjusting the size of the thread pool based on observed concurrency demand. Figure 8 shows the thread pool controller at work during a run of the SEDA-based Web server described in Section 5.1. The run is broken into three periods, each corresponding to an increasing number of clients; note that client load is extremely bursty. As bursts of file accesses arrive, the controller adds threads to each stage's thread pool until saturating at a maximum of 20 threads. Between periods, there is no demand for I/O, and the thread pool shrinks. While the *PageCache* and *CacheMiss* stages require more threads with increasing client load, the number of threads needed to service file I/O actually decreases. This is because the underlying filesystem buffer cache is warming up, and is able to service disk requests more rapidly. The thread pool controller infers that fewer threads are needed to manage the disk concurrency, and avoids creating threads that are not needed.

5 Applications and Evaluation

In this section we present a performance and load-conditioning evaluation of two applications: *Haboob*,[4] a high-performance HTTP server; and a packet router for the Gnutella peer-to-peer file sharing network. Whereas Haboob typifies a "closed-loop" server in which clients issue requests and wait for responses, the Gnutella packet router is an example of an "open-loop" server in which the server performance does not act as a limiting factor on offered load.

5.1 Haboob: A high-performance HTTP server

Web servers form the archetypal component of scalable Internet services. Much prior work has investigated the engineering aspects of building high-performance HTTP servers, but little has been said about load conditioning, robustness, and ease of construction. One benefit of studying HTTP servers is that a variety of industry-standard benchmarks exist to measure their performance. We have chosen the load model from the SPECweb99 benchmark suite [50] as the basis for our measurements, with two important modifications. First, we measure only the performance of static Web page accesses (which constitute 70% of the SPECweb99 load mix). Second, we keep the Web page file set fixed at 3.31 GB of disk files, corresponding to a SPECweb99 target load of 1000 connections. Files range in size from 102 to 921600 bytes and are accessed using a Zipf-based request distribution mandated by SPECweb99. More details can be found in [50].

5.1.1 Haboob architecture

The overall structure of Haboob is shown in Figure 5. The server consists of 10 stages, 4 of which are devoted to asynchronous socket and disk I/O, as described in the previous section. The *HttpParse* stage is responsible for accepting new client connections and for HTTP protocol processing for incoming packets. The *HttpRecv* stage accepts HTTP connection and request events and passes them onto the *PageCache* stage (if they represent disk files) or generates responses directly (for dynamic pages generated to gather server statistics). *PageCache* implements an in-memory Web page cache implemented using a hashtable

[3] Patches providing nonblocking file I/O support are available for Linux, but are not yet part of standard distributions. Furthermore, these patches make use of a kernel-level thread pool to implement nonblocking file writes, over which SEDA would have no control.

[4] A *haboob* is a large dust storm occurring in the desert of Sudan.

indexed by URL, each entry of which contains a response packet consisting of an HTTP header and Web page payload. The *CacheMiss* stage is responsible for handling page cache misses, using the asynchronous file I/O layer to read in the contents of the requested page from disk. Finally, *HttpSend* sends responses to the client and handles some aspects of connection management and statistics gathering. An additional stage (not shown in the figure) generates dynamic Web pages from HTML templates with embedded code written in the Python scripting language [36]. This feature provides general-purpose server-side scripting, akin to Java Server Pages [26].

The page cache attempts to keep the cache size below a given threshold (set to 204800 KB for the measurements provided below). It aggressively recycles buffers on capacity misses, rather than allowing old buffers to be garbage-collected by the Java runtime; we have found this approach to yield a noticeable performance advantage. The cache stage makes use of application-specific event scheduling to increase performance. In particular, it implements shortest connection first (SCF) [15] scheduling, which reorders the request stream to send short cache entries before long ones, and prioritizes cache hits over misses. Because SCF is applied only to each set of events provided by the batching controller, starvation across requests is not an issue.

Constructing Haboob as a set of stages greatly increased the modularity of the design, as each stage embodies a robust, reusable software component that can be individually conditioned to load. We were able to test different implementations of the page cache without any modification to the rest of the code; the runtime simply instantiates a different stage in place of the original page cache. Likewise, another developer who had no prior knowledge of the Haboob structure was able to replace Haboob's use of the asynchronous file layer with an alternate filesystem interface with little effort. Not including the Sandstorm platform, the Web server code consists of only 3283 non-commenting source statements, with 676 NCSS devoted to the HTTP protocol processing library.

5.1.2 Benchmark configuration

For comparison, we present performance measurements from the popular Apache [6] Web server (version 1.3.14, as shipped with Linux Red Hat 6.2 systems) as well as the Flash [44] Web server from Rice University. Apache makes use of a fixed-size process pool of 150 processes; each process manages a single connection at a time, reading file data from disk and sending it to the client in 8 KB chunks, using blocking I/O operations. Flash uses an efficient event-driven design, with a single process handling most request-processing tasks. A set of helper processes perform (blocking) disk I/O, pathname resolution, and other operations. The maximum size of the Flash static page cache was set to 204800 KB, the same size as in Haboob. Both Apache and Flash are implemented in C, while Haboob is implemented in Java.

All measurements below were taken with the server running on a 4-way SMP 500 MHz Pentium III system with 2 GB of RAM and Linux 2.2.14. IBM JDK v1.3.0 was used as the Java platform. 32 machines of a similar configuration were used for load generation, with each client machine using a number of threads to simulate many actual clients. All machines are interconnected via switched Gigabit Ethernet. Although this configuration does not simulate wide-area network effects, our interest here is in the performance and stability of the server under heavy load.

The client load generator loops, continually requesting a Web page (using a distribution specified by the SPECweb99 suite), reading the result, and sleeping for a fixed time of 20 milliseconds before requesting the next page. To more closely simulate the connection behavior of clients in the wide area, each client closes the TCP connection after 5 HTTP requests, and reestablishes the connection before continuing. This value was chosen based on observations of HTTP traffic from [39].[5] All benchmarks were run with warm filesystem and Web page caches. Note that the file set size of 3.31 GB is much larger than physical memory, and the static page cache for Haboob and Flash was set to only 200 MB; therefore, these measurements include a large amount of disk I/O.

5.1.3 Performance analysis

Figure 12 shows the performance of Haboob compared with Apache and Flash in terms of aggregate throughput and response time. Also shown is the Jain fairness index [27] of the number of requests completed by each client. This metric is defined as

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

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

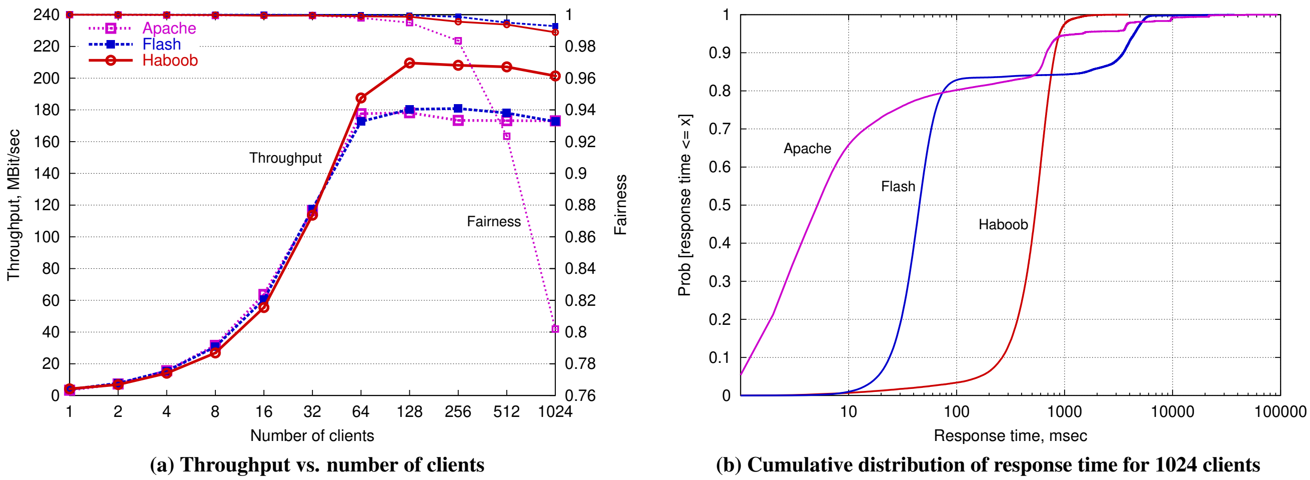
As Figure 12(a) shows, Haboob's throughput is stable as the number of clients increases, sustaining over 200 Mbps for 1024 clients. Flash and Apache also exhibit stable throughput, although slightly less than Haboob. This result might seem surprising, as we would expect the process-based Apache server to degrade in performance as the number of clients becomes large. Recall, however, that Apache accepts no more than 150 connections at any time, for which is not difficult to sustain high throughput using process-based concurrency. When the number of clients exceeds this amount, all other clients must wait for increasingly longer periods of time before being accepted into the system. Flash has a similar problem: it caps the number of simultaneous connections to 506, due to a limitation in the number of file descriptors that can be used with the *select()* system call. When the server is saturated, clients must wait for very long periods of time before establishing a connection.[6]

This effect is demonstrated in Figure 12(b), which shows the cumulative distribution of response times for each server with 1024 clients. Here, response time is defined as the total time for the server to respond to a given request, including time to establish a TCP connection if one has not already been made. Although all three servers have approximately the same *average* response times, the distribution is very different. Apache and Flash show a greater distribution of low response times than Haboob, but have very long tails, exceeding tens of seconds for a significant percentage of requests. Note that the use of the log scale in the figure underemphasizes the length of the tail. The maximum response time for Apache was over 93 seconds, and over 37 seconds for Flash. The long tail in the response times is caused by exponential backoff in the TCP retransmission timer for establishing a new connection, which under Linux can grow to be as large as 120 seconds.

With Apache, if a client is "lucky", its connection is accepted quickly and all of its requests are handled by a single server process. Moreover, each process is in competition with only 149 other processes, which is a manageable number on most systems. This explains the large number of low response times. However, if a client is "unlucky" it will have to wait for a server process to become available; TCP retransmit backoff means that this wait time can become very large. This unequal

[5] Note that most Web servers are configured to use a much higher limit on the number of HTTP requests per connection, which is unrealistic but provides improved benchmark results.

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



	256 clients				*1024 clients*			
Server	*Throughput*	*RT mean*	*RT max*	*Fairness*	*Throughput*	*RT mean*	*RT max*	*Fairness*
Apache	173.36 Mbps	143.91 ms	27953 ms	0.98	173.09 Mbps	475.47 ms	93691 ms	0.80
Flash	180.83 Mbps	141.39 ms	10803 ms	0.99	172.65 Mbps	665.32 ms	37388 ms	0.99
Haboob	208.09 Mbps	112.44 ms	1220 ms	0.99	201.42 Mbps	547.23 ms	3886 ms	0.98

Figure 12: **Haboob Web server performance:** *This figure shows the performance of the Haboob Web server compared to Apache and Flash. (a) shows the throughput of each server using a fileset of 3.31 GBytes as the number of clients increases from 1 to 1024. Also shown is the Jain fairness index delivered by each server. A fairness index of 1 indicates that the server is equally fair to all clients; smaller values indicate less fairness. (b) shows the cumulative distribution function of response times for 1024 clients. While Apache and Flash exhibit a high frequency of low response times, there is a heavy tail, with the maximum response time corresponding to several minutes.*

treatment of clients is reflected in the lower value of the fairness metric for Apache.

With Flash, all clients are accepted into the system very quickly, and are subject to queueing delays within the server. Low response times in Flash owe mainly to very efficient implementation, including a fast HTTP protocol processing library; we have performed few of these optimizations in Haboob. However, the fact that Flash accepts only 506 connections at once means that under heavy load TCP backoff becomes an issue, leading to a long tail on the response time distribution.

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

5.1.4 Adaptive load shedding

In this section, we evaluate the behavior of Haboob under overload, and demonstrate the use of an application-specific controller that attempts to keep response times below a given threshold through load shedding. In this benchmark, each client repeatedly requests a dynamic Web page that requires a significant amount of I/O and computation to generate. By subjecting each server to a large number of these "bottleneck" requests we can induce a heavier load than is generally possible when serving static Web pages.

For each request, the server performs several iterations of a loop that opens a file, reads data from it, and generates a sum of the data. After this processing the server returns an 8 KB response to the client. In Apache, this was implemented as a Perl module that runs in the context of an Apache server process. In Flash, the provided "fast CGI" interface was used, which creates a number of (persistent) server processes to handle dynamic requests. When a CGI request is made, an idle server process is used to handle the request, or a new process created if none are idle. In Haboob, the bottleneck was implemented as a separate stage, allowing the number of threads devoted to the stage processing to be determined by the thread pool controller, and the use of thresholding on the stage's incoming event queue to reject excess load. Because of differences between these three implementations, the amount of work performed by the dynamic page generation was calibrated to cause a server-side delay of 40 ms per request.

Figure 13 shows the cumulative distribution of response times for the three servers with 1024 clients. Apache and Haboob each exhibit large response times, but for different reasons. Apache's response time tail is caused by TCP retransmit backoff, as explained above, and with only 150 concurrent processes the queueing delay within the server is minimal. In Haboob, up to 1024 concurrent requests are queued within the server at any given time, leading to a large queueing delay at the bottleneck. Flash's apparent low response time is due to a bug in its CGI processing code, which causes it to close connections prematurely when it is unable to fork a new CGI process. With 1024 clients, there may be up to 1024 CGI processes in the system at one time; along with the other Flash processes, this exceeds the per-user process limit in Linux. When the fork fails, Flash closes the client connection immediately, without returning any response (even an error message) to the client. In this run, over 74% of requests resulted in a prematurely closed connection.

This mechanism suggests an effective way to bound the response time of requests in the server: namely, to adaptively shed load when the server detects that it is overloaded. To demonstrate this idea, we constructed an application-specific controller within the bottleneck stage, which observes the average response time of requests passing through the stage. When the response time exceeds a threshold of 5 sec, the controller exponentially reduces the stage's queue threshold. When the

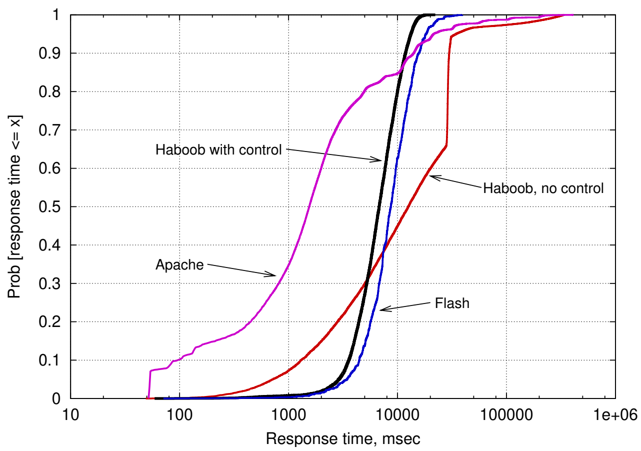


Figure 13: **Response time controller:** *This graph shows the effect of an application-specific controller that sheds load in order to keep response times below a target value. Here, 1024 clients are repeatedly requesting a dynamic Web page that requires both I/O and computation to generate. Apache and Haboob (with no control) process all such requests, leading to large response times. Flash rejects a large number of requests due to a bug in its CGI processing code; clients are never informed that the server is busy. With the response-time controller enabled, Haboob rejects requests with an error message when the average response time exceeds a threshold of 5 sec.*

response time is below the threshold, the controller increases the queue threshold by a fixed amount. When the *HttpRecv* stage is unable to enqueue a new request onto the bottleneck stage's event queue (because the queue threshold has been exceeded), an error message is returned to the client. Note that this is just one example of a load shedding policy; alternatives would be to send an HTTP redirect to another node in a server farm, or to provide degraded service.

Figure 13 shows the cumulative response time distribution with the response-time controller enabled. In this case, the controller effectively reduces the response time of requests through the server, with 90% of requests exhibiting a response time below 11.8 sec, and a maximum response time of only 22.1 sec. In this run, 98% of requests were rejected from the server due to queue thresholding. Note that this controller is unable to *guarantee* a response time below the target value, since bursts occurring when the queue threshold is high can induce spikes in the response time experienced by clients.

5.2 Gnutella packet router

We chose to implement a Gnutella packet router to demonstrate the use of SEDA for non-traditional Internet services. The Gnutella router represents a very different style of service from an HTTP server: that of routing packets between participants in a peer-to-peer file sharing network. Services like Gnutella are increasing in importance as novel distributed applications are developed to take advantage of the well-connectedness of hosts across the wide area. The peer-to-peer model has been adopted by several distributed storage systems such as Freenet [14], OceanStore [30], and Intermemory [13].

Gnutella [19] allows a user to search for and download files from other Gnutella users. The protocol is entirely decentralized; nodes running the Gnutella client form an ad-hoc multihop routing network layered over TCP/IP, and nodes communicate by forwarding received messages to their neighbors. Gnutella nodes tend to connect to several (typically four or more) other nodes at once, and the initial discovery of nodes on the network is accomplished through a well-known host. There are five message types in Gnutella: *ping* is used to discover other nodes on the network; *pong* is a response to a ping; *query* is used to search for files being served by other Gnutella hosts; *queryhits* is a response to a query; and *push* is used to allow clients to download files through a firewall. The packet router is responsible for broadcasting received *ping* and *query* messages to all other neighbors, and routing *pong*, *queryhits*, and *push* messages along the path of the corresponding *ping* or *query* message. Details on the message formats and routing protocol can be found in [19].

5.2.1 Architecture

The SEDA-based Gnutella packet router is implemented using 3 stages, in addition to those of the asynchronous socket I/O layer. The code consists of 1294 non-commenting source statements, of which 880 NCSS are devoted to Gnutella protocol processing. The *GnutellaServer* stage accepts TCP connections and processes packets, passing packet events to the *GnutellaRouter* stage, which performs actual packet routing and maintenance of routing tables. *GnutellaCatcher* is a helper stage used to join the Gnutella network by contacting a well-known site to receive a list of hosts to connect to. It attempts to maintain at least 4 simultaneous connections to the network, in addition to any connections established by other wide-area clients. Joining the "live" Gnutella network and routing packets allows us to test SEDA in a real-world environment, as well as to measure the traffic passing through the router. During one 37-hour run, the router processed 24.8 million packets (with an average of 179 packets/sec) and received 72,396 connections from other hosts on the network, with an average of 12 simultaneous connections at any given time. The router is capable of sustaining over 20,000 packets a second.

5.2.2 Protection from slow sockets

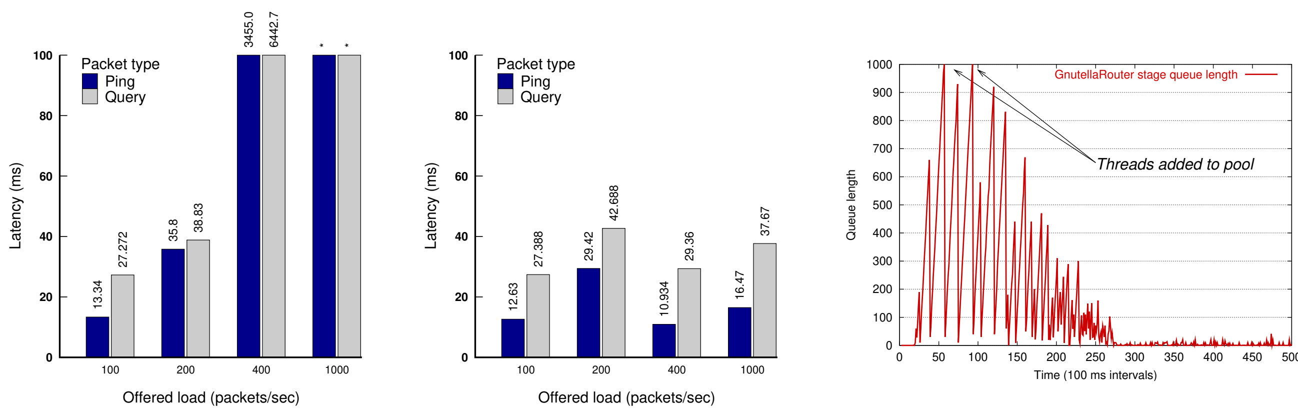
Our original packet router prototype exhibited an interesting memory leak: after several hours of correctly routing packets through the network, the server would crash after running out of memory. Observing the various stage queue lengths allowed us to easily detect the source of the problem: a large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

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

5.2.3 Load conditioning behavior

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

Figure 14(a) shows the average latencies for ping and query packets passing through the server with an offered load increasing from 100 to 1000 packets/sec. Both the client and server machines use the same configuration as in the HTTP server benchmarks. Packet latencies increase dramatically when the offered load exceeds the server's capacity.



(a) Using single thread **(b) Using thread pool controller** **(c) Queue length profile**

Figure 14: **Gnutella packet router latency:** *These graphs show the average latency of ping and query packets passing through the Gnutella packet router with increasing incoming packet rates. Query packets (15% of the packet mix) induce a server-side delay of 20 ms. (a) shows the latency with a single thread processing packets. Note that the latency increases dramatically as the offered load exceeds server capacity; at 1000 packets/sec, the server ran out of memory before a latency measurement could be taken. (b) shows the latency with the thread pool controller enabled. Note that for 100 and 200 packets/sec, no threads were added to the application stage, since the event queue never reached its threshold value. This explains the higher packet latencies compared to 400 and 1000 packets/sec, for which 2 threads were added to the stage. (c) shows the* GnutellaRouter *queue length over time for a load of 1000 packets/sec, with the thread pool controller active. The controller added a thread to the stage at each of the two points indicated.*

In the case of 1000 packets/sec, the server crashed (due to running out of memory for buffering incoming packets) before a latency measurement could be taken.

At this point, several load conditioning policies may be employed. A simple policy would be to threshold each stage's incoming event queue and drop packets when the threshold has been exceeded. Alternately, an approach similar to that used in Random Early Detection (RED) congestion avoidance schemes [17] could be used, where packets are dropped probabilistically based on the length of the input queue. Although these policies cause many packets to be dropped during overload, due to the lossy nature of Gnutella network traffic this may be an acceptable solution. An alternate policy would be admit all packets into the system, but have the application event handler filter out query packets (which are the source of the overload). Yet another policy would be to make use of the *asyncSocket* input rate controller to bound the incoming rate of packets into the system.

An alternate approach is to make use of SEDA's resource controllers to overcome the bottleneck automatically. In this approach, the thread pool controller adds threads to the *GnutellaRouter* stage when it detects that additional concurrency is required; this mechanism is similar to dynamic worker allocation in the cluster-based TACC [18] system. Figure 14(b) shows the average latencies in the Gnutella router with the SEDA thread pool controller enabled. As shown in Figure 14(c), 2 threads were added to the *GnutellaRouter* thread pool, allowing the server to handle the increasing packet loads despite the bottleneck. This number matches the theoretical value obtained from Little's result: If we model the stage as a queueing system with n threads, an average packet arrival rate of λ, a query packet frequency of p, and a query servicing delay of L seconds, then the number of threads needed to maintain a completion rate of λ is $n = \lambda p L = (1000)(0.15)(20\text{ ms}) =$ 3 threads.

6 Discussion and Conclusion

Internet services give rise to a new set of systems design requirements, as massive concurrency must be provided in a robust, easy-to-program manner that gracefully handles vast variations in load. SEDA is a step towards establishing design principles for this regime. In this paper we have presented the SEDA design and execution model, introducing the notion of stages connected by explicit event queues. SEDA makes use of a set of dynamic controllers to manage the resource usage and scheduling of each stage; we have described several controllers, including two that manage thread allocation across stages and the degree of batching used internally by a stage. We have also presented an analysis of two efficient asynchronous I/O components, as well as two applications built using the SEDA design, showing that SEDA exhibits good performance and robust behavior under load.

The SEDA model opens up new questions in the Internet services design space. The use of explicit event queues and dynamic resource controllers raise the potential for novel scheduling and resource-management algorithms specifically tuned for services. As future work we plan to implement a generalized flow-control scheme for communication between stages; in this scheme, each event requires a certain number of credits to enqueue onto the target stage's event queue. By assigning a variable number of credits to each event, interesting load-conditioning policies can be implemented.

We believe that measurement and control is the key to resource management and overload protection in busy Internet services. This is in contrast to long-standing approaches based on resource containment, which assign fixed resources to each task (such as a process, thread, or server request) in the system, and strive to contain the resources consumed by each task. Although these techniques have been used with some success in providing differentiated service within Internet services [57], containment typically mandates an *a priori* assignment of resources to each task, limiting the range of applicable load-conditioning policies. Rather, we argue that dynamic resource control, coupled with application-specific adaptation in the face of overload, is the right way to approach load conditioning.

Two new challenges arise when control is considered as the basis for resource management. The first is detecting overload conditions: many variables can affect the delivered performance of a service, and determining that the service is in fact overloaded, as well as the cause, is an interesting problem. The second is determining the appropriate control strategy to counter overload. We plan several improvements to the resource controllers in our current implementation, as well as new controllers that optimize for alternate metrics. For example, to reduce resource consumption, it may be desirable to prioritize stages that free resources over those that consume them. Under SEDA, the body of work on control systems [43, 45] can be brought to bear on service resource management, and we have only scratched the surface of the potential for this technique.

A common concern about event-driven concurrency models is ease of programming. Modern languages and programming tools support the development and debugging of threaded applications, and many developers believe that event-driven programming is inherently more difficult. The fact that most event-driven server applications are often quite complex and somewhat *ad hoc* in their design perpetuates this view. In our experience, programming in the SEDA model is easier than both multithreaded application design and the traditional event-driven model. When threads are isolated to a single stage, issues such as thread synchronization and race conditions are more straightforward to manage. Message-oriented communication between stages establishes explicit orderings; in the traditional event-driven design it is much more difficult to trace the flow of events through the system. We view SEDA as an ideal middle ground between threaded and event-driven designs, and further exploration of the programming model is an important direction for future work.

While SEDA facilitates the construction of well-conditioned services over commodity operating systems, the SEDA model suggests new directions for OS design. We envision an OS that supports the SEDA execution model directly, and provides applications with greater control over scheduling and resource usage. This approach is similar to that found in various research systems [5, 11, 28, 34] that enable application-specific resource management. Even more radically, a SEDA-based operating system need not be designed to allow multiple applications to share resources transparently. Internet services are highly specialized and are not designed to share the machine with other applications: it is generally undesirable for, say, a Web server to run on the same machine as a database engine (not to mention a scientific computation or a word processor). Although the OS may enforce protection (to prevent one stage from corrupting the state of the kernel or another stage), the system need not virtualize resources in a way that masks their availability from applications.

Acknowledgments

This research was supported by the Defense Advanced Research Projects Agency (grants DABT63-98-C-0038 and N66001-99-2-8913), the National Science Foundation (grant EIA-9802069), Intel Corporation, Nortel Networks, and Royal Philips Electronics. Matt Welsh was supported by a National Science Foundation Graduate Fellowship. We would like to thank Steve Gribble, Joe Hellerstein, and Marti Hearst for their valuable input on this paper. Eric Fraser, Matt Massie, Albert Goto, and Philip Buonadonna provided support for the Berkeley Millennium cluster used to obtain performance measurements. Eric Wagner provided the server-side scripting functionality in the Haboob Web server. We are especially indebted to our shepherd, Andrew Myers, and the anonymous reviewers for their helpful comments.

References

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

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

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

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

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

[6] Apache Software Foundation. The Apache web server. `http://www.apache.org`.

[7] 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.

[8] G. Banga and J. C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proc. 1998 Annual Usenix Technical Conference*, New Orleans, LA, June 1998.

[9] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[26] S. M. Inc. Java Server Pages API. `http://java.sun.com/products/jsp`.

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

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

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

[30] 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 *Proc. Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.

[31] J. Larus and M. Parkes. Using cohort scheduling to enhance server performance. Technical Report MSR-TR-2001-39, Microsoft Research, March 2001.

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

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

[34] 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.

[35] A. S. Lett and W. L. Konigsford. TSS/360: A time-shared operating system. In *Proc. Fall Joint Computer Conference, Volume 33, Part 1*, pages 15–28, 1968.

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

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

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

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

[40] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click modular router. In *Proc. 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 217–231, Kiawah Island, South Carolina, December 1999.

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

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

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

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

[45] 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.

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

[47] X. Qie, A. Bavier, L. Peterson, and S. Karlin. Scheduling computations on a software-based router. In *Proc. SIGMETRICS 2001*, June 2001.

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

[49] O. Spatscheck and L. Petersen. Defending against denial of service attacks in Scout. In *Proc. 3rd Symposium on Operating Systems Design and Implementation*, February 1999.

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

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

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

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

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

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

[56] M. Vandevoorde and E. Roberts. Work crews: An abstraction for controlling parallelism. Technical Report Research Report 42, Digital Equipment Corporation Systems Research Center, February 1988.

[57] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded Web servers. In *Proc. 2001 USENIX Annual Technical Conference*, Boston, June 2001.

[58] L. A. Wald and S. Schwarz. The 1999 Southern California Seismic Network Bulletin. *Seismological Research Letters*, 71(4), July/August 2000.

[59] D. A. Wallach, D. R. Engler, and M. F. Kaashoek. ASHs: Application-specific handlers for high-performance messaging. In *Proc. ACM SIGCOMM '96 Conference: Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 40–52, Stanford, California, August 1996.

[60] M. Welsh. NBIO: Nonblocking I/O for Java. http://www.cs.berkeley.edu/~mdw/proj/java-nbio.

[61] M. Welsh and D. Culler. Virtualization considered harmful: OS design directions for well-conditioned services. In *Proc. 8th Workshop on Hot Topics in Operating Systems (HotOS VIII)*, Schloss Elmau, Germany, May 2001.

[62] Yahoo! Inc. Yahoo! reports Second Quarter 2001 financial results. http://docs.yahoo.com/docs/pr/release794.html.

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