The Staged Event-Driven Architecture for Highly-Concurrent Server Applications

Ph.D. Qualifying Examination Proposal

Matt Welsh

Computer Science Division
University of California, Berkeley
mdw@cs.berkeley.edu

Abstract

We propose a new design for highly-concurrent server applications such as Internet services. This design, the *staged event-driven architecture* (SEDA), is intended to support massive concurrency demands for a wide range of applications. In SEDA, applications are constructed as a set of event-driven *stages* separated by *queues*. This design allows services to be well-conditioned to load, preventing resources from being overcommitted when demand exceeds service capacity. Decomposing services into a set of stages enables modularity and code reuse, as well as the development of debugging tools for complex event-driven applications. We present the SEDA design, as well as *Sandstorm*, an Internet services platform based on this architecture. We evaluate the use of Sandstorm through two applications: A simple HTTP server benchmark and a packet router for the Gnutella peer-to-peer file sharing network.

1 Introduction

The Internet presents a systems problem of unprecedented scale: that of supporting millions of users demanding access to services which 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. Microsoft's web sites receive over 300 million hits with 4.1 million users a day; Lycos has over 82 million page views and more than a million users daily. As the demand for Internet services grows, as does their functionality, new system design techniques must be used to manage this load.

In addition to supporting high concurrency, Internet services must be well-conditioned to load. When the demand on a service exceeds its capacity, the service should not overcommit its resources or degrade in such a way that all clients suffer. As the number of Internet users continues to grow, load conditioning becomes an even more important aspect of service design. The peak load on an Internet service may be more than an order of magnitude greater than its average load; in this case overprovisioning of resources is generally infeasible.

Unfortunately, few tools exist that aid the development of highly-concurrent, well-conditioned services. Existing operating systems typically provide applications with the abstraction of a virtual machine with its own CPU, memory, disk, and network; the O/S multiplexes these virtual machines (which may be embodied as processes or threads) over real hardware. However, providing this level of abstraction entails a high overhead in terms of context switch time and memory footprint, thereby limiting concurrency. The use of process-based concurrency also makes resource management more challenging, as the operating system generally does not associate individual resource principles with each I/O flow through the system.

The use of event-driven programming techniques can avoid the scalability limits of processes and threads. However, such systems are generally built from scratch for particular applications, and depend on mechanisms not well-supported by most languages and operating systems. Subsequently, obtaining high performance requires that the application designer carefully manage event and thread scheduling, memory allocation, and I/O streams. It is unclear whether this design methodology yields a reusable, modular system that can support a range of different applications.

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

Hypothesis

This work proposes a new design 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 some sense the goal is to design and develop an "operating system for services." However, our intent is to implement this system on top of a commodity O/S, which will increase compatibility with existing software and ease the transition of applications to the new architecture.

The design of SEDA is based on three key design goals:

First, to *simplify the task of building complex, event-driven applications.* To avoid the scalability limits of threads, the SEDA execution model is based on event-driven programming techniques. To shield applications from the complexity of managing a large event-driven system, the underlying platform is responsible for managing the details of thread management, event scheduling, and I/O.

The second goal is to *enable load conditioning.* SEDA is structured to facilitate fine-grained, application-specific resource management. Incoming request queues are exposed to application modules, allowing them to drop, filter, or reorder requests during periods of heavy load. The underlying system can also make global resource management decisions without the intervention of the application.

Finally, we wish to *support a wide range of applications.* SEDA is designed with adequate generality to support a large class of server applications, including dynamic Web servers, peer-to-peer networking, and streaming media services. We plan to build and measure a range of applications to evaluate the flexibility of our design.

We claim that using SEDA, highly-concurrent applications will be easier to build, perform better, and will be more robust under load. With the right set of interfaces, application designers can focus on application-specific logic, rather than the details of concurrency and event-driven I/O. By controlling the scheduling and resource allocation of each application module, the system can adapt to overload conditions and prevent a runaway component from consuming too many resources. Exposing request queues allows the system to make informed scheduling decisions; for example, by prioritizing requests for cached, in-memory data over computationally expensive operations such as dynamic content generation.

In this proposal, we present the SEDA architecture, contrasting it to the dominant server designs in use today. We also present *Sandstorm*, an initial implementation of the SEDA design, and evaluate the system against several benchmark applications. These include a simple HTTP server as well as a peer-to-peer network application.

2 Motivation

Our work is motivated by four fundamental properties of Internet services: high concurrency, dynamic content, continuous availability demands, and robustness to load.

High Concurrency The growth in popularity and functionality of Internet services has been astounding. While the Web itself is getting bigger, with recent estimates anywhere between 1 billion [21] and 2.5 billion [36] unique documents, the number of users on the Web is also growing at a staggering rate. A recent study [13] found that there are over 127 million adult Internet users in the United States alone.

As a result, Internet applications must support unprecedented concurrency demands, and these demands will only increase over time. On an average day, Yahoo! serves 780 million pageviews, and delivers over 203 million messages through its e-mail and instant messenger services [48]. Internet traffic during 2000 U.S. presidential election was at an all-time high, with ABC News reporting over 27.1 million pageviews in one day, almost 3 times the

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

peak load that this site had ever received. Many news and information sites were reporting a load increase anywhere from 130% to 500% over their average [28].

Dynamic Content The early days of the Web were dominated by the delivery of static content, mainly in the form of HTML pages and images. More recently, dynamic, on-the-fly content generation has become more more widespread. This trend is reflected in the incorporation of dynamic content into the benchmarks used to evaluate Web server performance, such as SPECWeb99 [39].

Take for example a large "mega-site" such as Yahoo! [47], which provides many dynamic services under one roof, ranging from search engine to real-time chat to driving directions. In addition to consumer-oriented sites, specialized business-to-business applications, ranging from payroll and accounting to site hosting, are becoming prevalent. Accordingly, Dataquest projects that the worldwide application service provider market will reach $25 billion by 2004 [8].

Continuous Availability Internet services must exhibit very high availability, with a downtime of no more than a few minutes a year. Even so, there are many documented cases of Web sites crashing under heavy usage. Such popular sites as EBay [29], Excite@Home [17], and E*Trade [5] have had embarrassing outages during periods of high load. While some outages cause only minor annoyance, others can have a more serious impact: the E*Trade outage resulted in a class-action lawsuit against the online stock brokerage by angry customers. As more people begin to rely upon the Internet for managing financial accounts, paying bills, and even voting in elections, it is increasingly important that these services are robust to load and failure.

Robustness to Load Demand for Internet services can be extremely bursty, with the peak load being many times that of the average. As an example, Figure 1 shows the load on the U.S. Geological Survey Pasadena Field Office Web site after a large earthquake hit Southern California in October 1999. The load on the site increased almost 3 orders of magnitude over a period of just 10 minutes, causing the Web server's network link to saturate and its disk log to fill up [42].[2] The term "Slashdot effect" is often used to describe what happens when a site is hit by sudden, heavy load. This term refers to the technology news site `slashdot.org`, which is itself hugely popular and often brings down other less-resourceful sites when linking to them from its main page.

[2]We are indebted to Stan Schwarz for providing us with the web logs from this event.

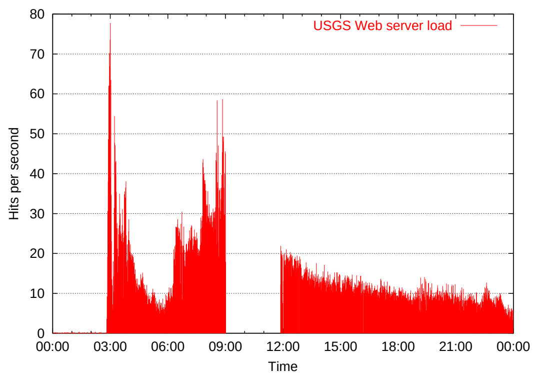


Figure 1: **The effect of sudden load on a Web server:** *This is a graph of the web server logs from the USGS Pasadena Field Office Web site after an earthquake registering 7.1 on the Richter scale hit Southern California on October 16, 1999. The load on the site increased almost 3 orders of magnitude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

One approach to dealing with heavy load is to overprovision. In the case of a Web site, the administrators simply buy enough Web server machines to handle the peak load that the site could experience, and load balance across them. However, overprovisioning is infeasible when the ratio of peak to average load is very high. This approach also neglects the cost issues which arise when scaling a site to a large "farm" of machines; the cost of two machines is no doubt much higher than twice the cost of one machine. It is also arguable that during times of heavy load are exactly when the service is needed the most. This implies that in addition to being adequately provisioned, services should be *well-conditioned to load.* That is, when the demand on a service exceeds its capacity, a service should not overcommit its resources or degrade in such a way that all clients suffer.

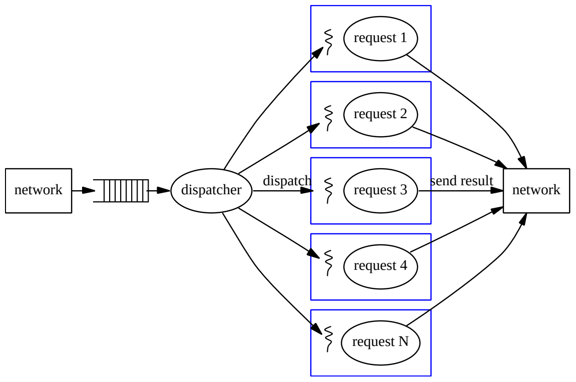


Figure 2: **Threaded server design:** *Each incoming request is dispatched to a separate thread, which processes the request and returns a result to the client. Edges represent control flow between components. Note that other I/O operations, such as disk access, are not shown here, but would be incorporated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

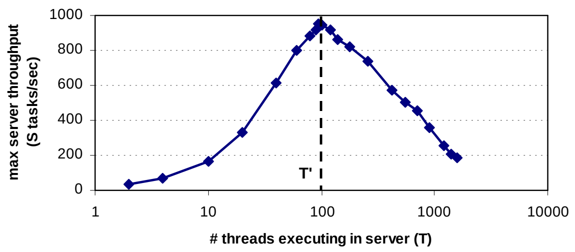


Figure 3: **Threaded server throughput degradation:** *This benchmark has a very fast client issuing many concurrent 150-byte tasks over a single TCP connection to a simple server which allocates one thread per task. Threads are pre-allocated in the server to eliminate thread startup overhead from the measurements. After receiving a task, each thread sleeps for* $L = 50$ *ms before sending a 150-byte response to the client. The server is implemented in Java and is running on a 167 MHz UltraSPARC running Solaris 5.6. As the number of concurrent threads* T *increases, throughput increases until* $T > T'$*, after which the throughput of the system degrades substantially.*

3 The Staged Event-Driven Architecture

We argue that these fundamental properties of Internet services demand a new approach to server software design. In this section we explore the space of server software architectures, focusing on the two dominant programming models: *threads* and *events*. We then propose a new architecture, the *staged event-driven architecture* (SEDA), which makes use of both of these models to address the needs of highly-concurrent services.

3.1 Thread-based concurrency

Most operating systems and languages support a thread-based concurrency model, in which each concurrent task flowing through the system is allocated its own thread of control. The O/S then multiplexes these threads over the real CPU, memory, and I/O devices. Threading allows programmers to write straight-line code and rely on the operating system to overlap computation and I/O by transparently switching across threads. This situation is depicted in Figure 2. However, thread programming presents a number of correctness and tuning challenges. Synchronization primitives (such as locks, mutexes, or condition variables) are a common source of bugs. Lock contention can cause serious performance degradation as the number of threads competing for a lock increases.

The most serious problem with threads is that they often entail a large overhead in terms of context-switch time and memory footprint. As the number of threads increases, this can lead to serious performance degradation. As an example, consider a simple server application which allocates one thread per task entering the system. Each task imposes a server-side delay of L seconds before returning a response to the client; L is meant to represent the processing time required to process a task, which may involve a combination of computation and disk I/O. There is typically a maximum number of threads T' that a given system can support, beyond which performance degradation occurs. Figure 3 shows the performance of such a server as the number of threads increases. In this figure, while the thread limit T' would be large for general-purpose timesharing, it would not be adequate for the tremendous concurrency requirements of an Internet service.

3.2 Event-based concurrency

The scalability limits of threads have led many developers to prefer an event-driven approach. In this design, a server consists of a small number of threads (typically one per CPU) which respond to events generated by the operating system or internally by the application. These events might in-

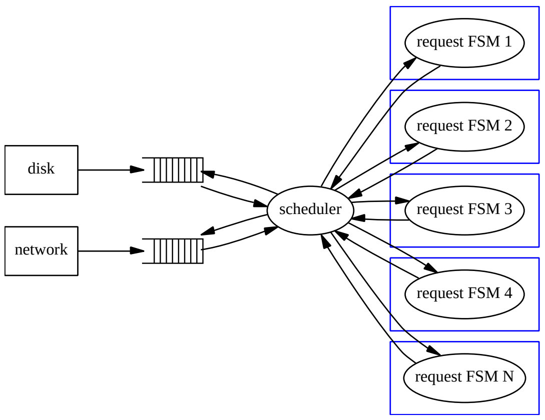


Figure 4: **Event-driven server design:** *This figure shows the flow of events through a monolithic event-driven server. The main thread processes incoming events from the network, disk, and other sources, and uses these to drive the execution of many finite state machines. Each FSM represents a single 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.*

clude disk and network I/O completions or timers. This model assumes that the event-handling threads do not block, and for this reason nonblocking I/O mechanisms are employed. However, event processing threads can block regardless of the I/O mechanisms used: page faults and garbage collection are common sources of thread suspension that are generally unavoidable.

The event-driven approach implements individual task flows through the system as finite state machines, rather than threads, as shown in Figure 4. Transitions between states in the FSM are triggered by events. Consider a simple event-driven Web server, which uses a single thread to manage many concurrent HTTP requests. Each request has its own state machine, depicted in Figure 5. The sequential flow of each request is no longer handled by a single thread; rather, one thread processes all concurrent requests in disjoint stages. This can make debugging difficult, as stack traces no longer represent the control flow for the processing of a particular task. Also, task state must be bundled into the task itself, rather than stored in local variables or on the stack as in a threaded system.

This "monolithic" event-driven design raises a number of additional challenges for the application developer. It is difficult to modularize such an application, as individual states are directly linked with others in the flow of execution. The code implementing each state must be trusted, in the sense that library calls into untrusted code (which may block or consume a large number of resources) can stall the event-handling thread.

Scheduling and ordering of events is probably the most important concern when using the pure event-driven approach. 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 FSMs. Also, the application must decide how often to service the network or disk devices in order to maintain high throughput. The choice of an event scheduling algorithm is often tailored to the specific application; introduction of new functionality may require the algorithm to be redesigned.

3.3 The Staged Event-Driven Architecture

We propose a new design, the *staged event-driven architecture* (SEDA), which is a variant on the event-driven approach described above. Our goal is to retain the performance and concurrency benefits of the event-driven model, but avoid the software engineering difficulties which arise.

SEDA makes use of a set of design patterns, first described in [46], which break the control flow through an event-driven system into a series of *stages* separated by *queues*. Each stage represents some set of states from the FSM in the monolithic event-driven design. The key difference is that each stage can now be considered an independent, contained entity with its own incoming event queue. Figure 6 depicts a simple HTTP server implementation using the SEDA design. Stages pull tasks from their incoming task queue, and dispatch tasks by pushing them onto the incoming queues of other stages. Note that the graph in Figure 6 closely resembles the original state machine from Figure 5: there is a close correlation between state transitions in the FSM and event dispatch operations in the SEDA implementation.

In SEDA, threads are used to drive the execution of stages. This design decouples event handling from thread allocation and scheduling: stages are not responsible for managing their own threads, rather, the underlying platform can choose a thread allocation and scheduling policy based on a number

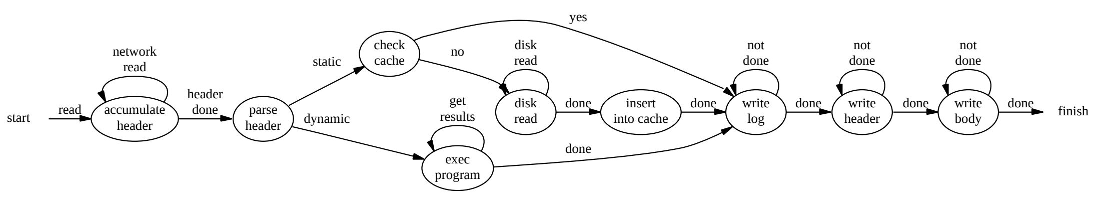


Figure 5: **Event-driven finite state machine:** *Each concurrent request in an event-driven HTTP server would use a finite state machine as shown here. Transitions between states are made by responding to external events, such as I/O readiness and completion events.*

of factors. If every stage in the application is non-blocking, then it is adequate to use one thread per CPU, and schedule those threads across stages in some order. For example, in an overload condition, stages which consume fewer resources could be given priority. Another approach is to delay the scheduling of a stage until it has accumulated enough work to amortize the startup cost of that work. An example of this is aggregating multiple disk accesses and performing them all at once.

While the system as a whole is event-driven, stages may block internally (for example, by invoking a library routine or blocking I/O call), and use multiple threads for concurrency. The size of the blocking stage's thread pool should be chosen carefully to avoid performance degradation due to having too many threads, but also to obtain adequate concurrency.

Consider the static Web page cache of the HTTP server shown in Figure 6. Let us assume a fixed request arrival rate $\lambda = 1000$ requests per second, a cache miss frequency $p = 0.1$, and a cache miss latency of $L = 50$ ms. On average, $\lambda p = 100$ requests per second result in a miss. If we model the stage as a G/G/n queueing system with arrival rate λp, service time L, and n threads, then in order to service misses at a rate of λp, we need to devote $n = \lambda p L = 5$ threads to the cache miss stage [24].

Breaking event-handling code into stages allows those stages to be isolated from one another for the purposes of performance and resource management. By isolating the cache miss code into its own stage, the application can continue to process cache hits when a miss does occur, rather than blocking the entire request path. Introduction of a queue between stages decouples the execution of those stages, by introducing an explicit control boundary. Since a thread cannot cross over this boundary (it can only pass data across the boundary by enqueuing an event), it is possible to constrain the execution of threads to a given stage. In the example above, the static URL processing stage need not be concerned with whether the cache miss code blocks, since its own threads will not be affected.

SEDA has a number of advantages over the monolithic event-driven approach. First, this design allows stages to be developed and maintained individually. A SEDA-based application consists of a directed graph of interconnected stages; each stage can implemented as a separate code module in isolation from other stages. The operation of two stages can be composed by inserting a queue between them, thereby allowing events to pass from one to the other.

The second advantage is that the introduction of queues allows each stage to be individually conditioned to load. Backpressure can be implemented by having a queue reject new entries (e.g., by raising an error condition) when it becomes full. This is important as it allows excess load to be rejected by the system, rather than buffering an arbitrary amount of work. Alternately, a stage can drop, filter, or reorder incoming events in its queue to implement other load conditioning policies, such as prioritization.

Finally, the decomposition of a complex event-driven application into stages allows those stages to be individually replicated and distributed. This structure facilitates the use of shared-nothing clusters as a scalable platform for Internet services. Multiple copies of a stage can be executed on multiple cluster machines in order to remove a bottleneck from the system. Stage replication can also be used to implement fault tolerance: if one replica of a stage fails, the other can continue processing tasks. Assuming that stages do not share data objects in memory, event queues can be used to transparently distribute stages across multiple machines, by implementing a queue as a network pipe. This work does not focus on the replication, distribution, and

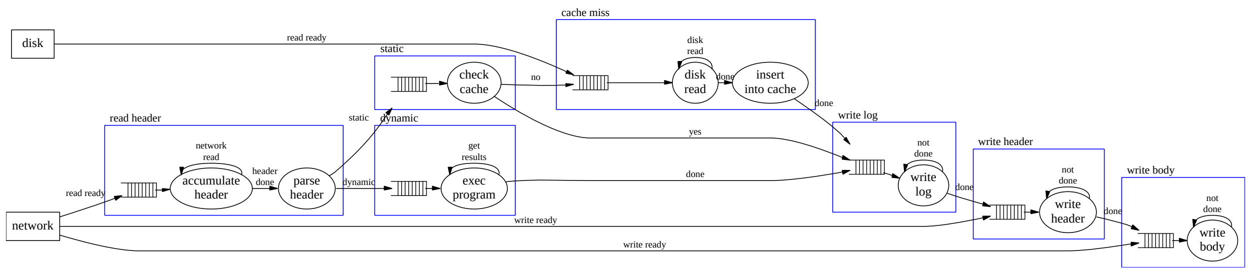


Figure 6: **Staged event-driven (SEDA) HTTP server:** *The states in the event-driven FSM have been broken into a set of* stages *separated by* queues. *Edges represent the flow of events between stages. Each stage can be independently managed, and stages can be run in sequence or in parallel, or a combination of the two. The use of event queues allows each stage to be individually load-conditioned, for example, by thresholding its event queue. For clarity, some event paths have been elided from this figure, such as disk and network I/O requests from the application.*

fault-tolerance aspects of SEDA; this will be discussed further in Section 6.

3.4 Research Issues

While SEDA provides a general framework for constructing scalable server applications, many research issues remain to be investigated.

Application structure There are many tradeoffs to consider when deciding how to break an application into a series of stages. The 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 also increases latency. As discussed above, a module which performs blocking operations can reside in its own stage for concurrency and performance reasons. More generally, any untrusted 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.

In this work we intend to develop an evaluation strategy for the mapping of application modules onto stages, and apply that strategy to applications constructed using the SEDA framework.

Thread allocation and scheduling We have discussed several alternatives for thread allocation and scheduling across stages, but the space of possible solutions is large. A major goal of this work is to evaluate different thread management policies within the SEDA model. In particular, we wish to explore the tradeoff between application-level and O/S-level thread scheduling. A SEDA application can implement its own scheduler by allocating a small number of threads and using them to drive stage execution directly. An alternative is to allocate a small thread pool for each stage, and have the operating system schedule those threads itself. While the former approach gives SEDA finer control over the use of threads, the latter makes use of the existing O/S scheduler and simplifies the system's design.

We are also interested in balancing the allocation of threads across stages, especially for stages which perform blocking operations. This can be thought of as a global optimization problem, where the system has some maximum feasible number of threads T' that we wish to allocate, either statically or dynamically, across a set of stages. As we will show in Section 5.2, dynamic thread allocation can be driven by inspection of queue lengths; if a stage's event queue reaches some threshold, it may be beneficial to increase the number of threads allocated to it.

Event scheduling In addition to inter-stage scheduling using threads, each stage may implement its own intra-stage event scheduling policy. While FIFO is the most straightforward approach to event queue processing, other policies might valuable, especially during periods of heavy load. For example, a stage may wish to reorder incoming events to process them in Shortest Remaining Processing Time (SRPT) order; this technique has been shown to be effective for certain Web server loads [18]. Alternately, a stage may wish to aggregate multiple requests which share common processing or data requirements; the database technique of multi-query optimization [37] is one example of this approach.

We believe that a key benefit of the SEDA design is the exposure of event queues to application stages. We plan to investigate the impact of different event scheduling policies on overall application performance, as well as its interaction to the load conditioning aspects of the system (discussed below).

General purpose load conditioning Perhaps the most complex and least-understood aspect of developing scalable servers is how to condition them to load. The most straightforward approach is to perform early rejection of work when offered load exceeds system capacity; this approach is similar to that used network congestion avoidance schemes such as random early detection [10]. However, given a complex application, this may not be the most efficient policy. For example, it may be the case that a single stage is responsible for much of the resource usage on the system, and that it would suffice to throttle that stage alone.

Another question to consider is what behavior the system should exhibit when overloaded: should incoming requests be rejected at random or according to some other policy? A heavily-loaded stock trading site may wish to reject requests for quotes, but allow requests for stock orders to proceed. SEDA allows stages to make these determinations independently, enabling a large class of flexible load conditioning schemes.

An effective approach to load conditioning is to threshold each stage's incoming event queue. When a stage attempts to enqueue new work on a clogged stage, an error condition will be raised. Backpressure can be implemented by propagating these "queue full" messages backwards along the event path. Alternately, the thread scheduler could detect a clogged stage and refuse to schedule stages upstream from it.

Queue thresholding does not address all aspects of load conditioning, however. Consider a stage which processes events very rapidly, but allocates a large block of memory for each event. Although no stage may ever become clogged, memory pressure generated by this stage alone will lead to system overload, rather than a combination of other factors (such as CPU time and I/O bandwidth). The challenge in this case is to detect the resource utilization of each stage to avoid the overload condition. Various systems have addressed this issue, including resource containers [1] and the Scout [38] operating system. We intend to evaluate whether these approaches can be applied to SEDA.

Debugging As discussed earlier, few tools exist for understanding and debugging a complex event-driven system. We hope that the structure of SEDA-based applications will be more amenable to this kind of analysis. The decomposition of application code into stages and explicit event delivery mechanisms should facilitate inspection. For example, a debugging tool could trace the flow of events through the system and visualize the interactions between stages. As discussed in Section 4, our early 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 which are valuable in understanding the behavior of applications.

4 Prototype Design and Evaluation

We have implemented a prototype of an Internet services platform which makes use of the staged event-driven architecture. This prototype, called *Sandstorm*, has evolved rapidly from a bare-bones system to a general-purpose platform for hosting highly-concurrent applications. In this section we describe the Sandstorm system, and provide a performance analysis of its basic concurrency and I/O features. In Section 5 we present an evaluation of several simple applications built using the platform.

4.1 Sandstorm

Figure 7 shows an overview of the Sandstorm architecture. Sandstorm is based directly on the SEDA design, and is implemented in Java. A Sandstorm application consists of a set of stages connected by queues. Each stage consists of two parts: an *event handler*, which is the core application-level code for processing events, and a *stage wrapper*, which is responsible for creating and managing event queues. A set of stages is controlled a *thread manager*, which is responsible for allocating and scheduling threads across those stages.

Applications are not responsible for creating queues or managing threads; only the event handler interface is exposed to application code. This

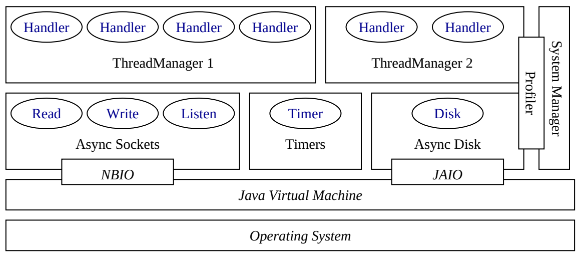


Figure 7: **Sandstorm architecture:** *A Sandstorm application is implemented as a set of* stages*, the execution of which is controlled by* thread managers*. Thread managers allocate and schedule threads across each stage according to some policy. Each stage has an associated* event handler*, represented by ovals in the figure, which is the core application logic for processing events within that stage. Sandstorm provides an asynchronous socket interface over* NBIO*, which is a set of nonblocking I/O abstractions for Java. Applications register and receive timer events through the* timer *stage. The Sandstorm asynchronous disk layer is still under development, and is based on a Java wrapper to the POSIX AIO interfaces.*

```
public void handleEvent(QueueElementIF elem);
public void handleEvents(QueueElementIF elems[]);
public void init(ConfigDataIF config)
  throws Exception;
public void destroy() throws Exception;
```

Figure 8: **Sandstorm event handler interface:** *This is the set of methods which a Sandstorm event handler must implement.* `handleEvent` *takes a single event as input and processes it;* `handleEvents` *takes a batch of events, allowing the event handler to perform its own cross-event scheduling.* `init` *and* `destroy` *are used for initialization and cleanup of an event handler.*

interface is shown in Figure 8 and consists of four methods. `handleEvent` takes a single event (represented by a `QueueElementIF`) and processes it. `handleEvents` takes a batch of events and processes them in any order; it may also drop, filter, or reorder the events. This is the basic mechanism by which applications implement intra-stage event scheduling. `init` and `destroy` are used for event handler initialization and cleanup.

Initialization When an event handler is initialized it is given a handle to the *system manager*, which provides various functions such as stage creation and lookup. When a stage is created it is given a unique name in the system, represented by a string. An event handler may obtain a handle to the queue for any other stage by performing a lookup through the system manager. The system manager also allows stages to be created and destroyed at runtime.

Profiling Sandstorm includes a built-in profiler, which records information on memory usage, queue lengths, and stage relationships at runtime. The data generated by the profiler can be used to visualize the behavior and performance of the application; for example, a graph of queue lengths over time can help identify a bottleneck (Figure 14 is an example of such a graph). The profiler can also generate a graph of stage connectivity, based on a runtime trace of event flow. Figure 9 shows an automatically-generated graph of a simple Gnutella server running on Sandstorm; the *graphviz* package [12] from AT&T Research is used to render the graph.

Thread Managers The thread manager interface is an integral part of Sandstorm's design. This interface allows stages to be registered and deregistered with a given thread manager implementation. Implementing a new thread manager allows one to experiment with different thread allocation and scheduling policies without affecting application code.

Sandstorm provides two thread manager implementations. `TPPTM` (*thread-per-processor*) allocates one thread per processor, and schedules those threads across stages in a round-robin fashion. Of course, many variations on this simple approach are possible. The second thread manager implementation is `TPSTM` (*thread-per-stage*), which allocates one thread for each incoming event queue for each stage. Each thread performs a blocking dequeue operation on its queue, and invokes the corresponding event handler's `handleEvents` method when events become available.

`TPPTM` performs application-level thread scheduling, in the sense that the ordering of stage processing (in this case round-robin) is determined by the thread manager itself. `TPSTM`, on the other hand, relies on the operating system to schedule stages: threads may be suspended when the perform a blocking dequeue operation on their event queue, and enqueuing an event onto a queue makes a thread runnable. Thread scheduling in `TPSTM` is therefore

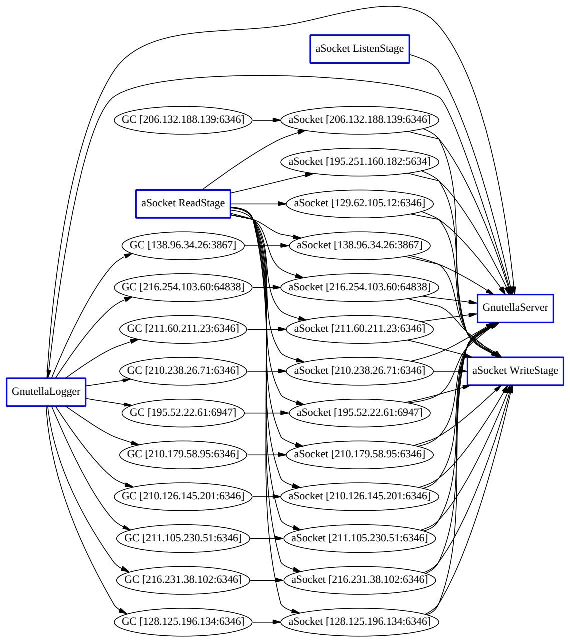

aSocket ListenStage
GC [206.132.188.139:6346]
aSocket [206.132.188.139:6346]
aSocket [195.251.160.182:5634]
aSocket ReadStage
aSocket [129.62.105.12:6346]
GC [138.96.34.26:3867]
aSocket [138.96.34.26:3867]
GC [216.254.103.60:64838]
aSocket [216.254.103.60:64838]
GnutellaServer
GC [211.60.211.23:6346]
aSocket [211.60.211.23:6346]
GC [210.238.26.71:6346]
aSocket [210.238.26.71:6346]
aSocket WriteStage
GnutellaLogger
GC [195.52.22.61:6947]
aSocket [195.52.22.61:6947]
GC [210.179.58.95:6346]
aSocket [210.179.58.95:6346]
GC [210.126.145.201:6346]
aSocket [210.126.145.201:6346]
GC [211.105.230.51:6346]
aSocket [211.105.230.51:6346]
GC [216.231.38.102:6346]
aSocket [216.231.38.102:6346]
GC [128.125.196.134:6346]
aSocket [128.125.196.134:6346]

Figure 9: **Visualization of application stages:** *This graph was automatically generated from profile data taken during a run of a Sandstorm-based Gnutella server, described in Section 5.2. In the graph, boxes represent stages, and ovals represent library classes through which events flow. Edges indicate event propagation. The main application stage is* `GnutellaLogger`*, which makes use of* `GnutellaServer` *to manage connections to the Gnutella network. The intermediate nodes represent Gnutella packet-processing code and socket connections.*

driven by the flow of events in the system, while `TPPTM` must waste cycles by polling across queues, unaware of which stages may have pending events. An important question for this work will be understanding the tradeoffs between different thread scheduling approaches.

Timers Sandstorm includes a general-purpose timer facility, allowing a stage to register an event which should be delivered at some time in the future. This is implemented as a stage which accepts timer request events, and uses a dedicated thread manager to fire those events at the appropriate time.

4.2 I/O Interfaces

An important aspect of Sandstorm's design is its I/O layers, providing asynchronous network and disk interfaces for applications. These two layers are designed as a set of stages which accept I/O requests and propagate I/O completion events to the application.

4.2.1 Asynchronous sockets interface

Sandstorm provides applications with an asynchronous network sockets interface, allowing a stage to obtain a handle to a socket object and request a connection to a remote host and TCP port. When the connection is established, a connection object is enqueued onto the stage's event queue. The application may then enqueue data to be written to the connection. When data is read from the socket, a buffer object is enqueued onto the stage's incoming event queue. Applications may also create a server socket, which accepts new connections, placing connection objects on the application event queue when they arrive.

This interface is implemented as a set of three event handlers, *read*, *write*, and *listen*, which are responsible for reading socket data, writing socket data, and listening for incoming connections, respectively. Each handler has two incoming event queues: an application request queue and an I/O queue. The application request queue is used when applications push request events to the socket layer, to establish connections or write data. The I/O queue contains events indicating I/O completion and readiness for a set of sockets.

Sandstorm's socket layer makes use of *NBIO* [44], a Java library providing native code wrappers to O/S-level nonblocking I/O and event delivery mechanisms, such as the UNIX `poll` system call. This interface is necessary as the standard Java libraries do not provide nonblocking I/O primitives.

4.2.2 Asynchronous disk interface

The asynchronous disk layer for Sandstorm is still under development.[3] The current design is based on a Java wrapper to the POSIX.4 [11] AIO interfaces. As an interim solution, it is possible to design an asynchronous disk I/O stage using blocking I/O and a thread pool. This is the approach used by Gribble's distributed data structure storage "bricks" [15].

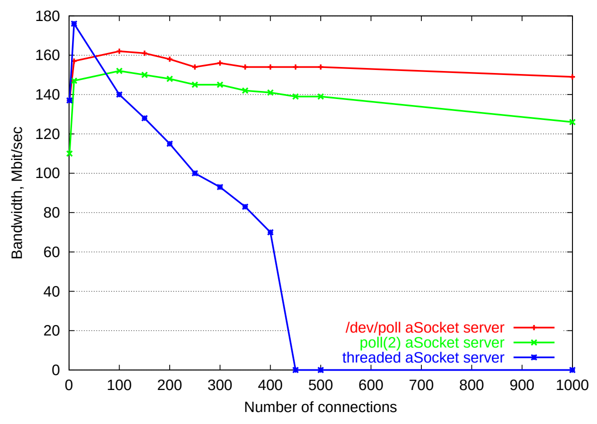


Figure 10: **Asynchronous socket performance:** *This graph shows the performance of Sandstorm's asynchronous socket layer as a function of the number of simultaneous connections. Each client opens a connection to the server and issues bursts of 1000 8Kb packets; the server responds with a single 32-byte ACK for each burst. Three implementations of of the socket layer are shown: two make use of the NBIO nonblocking I/O interface, implemented using either the* `/dev/poll` *mechanism or* `poll` *system call for event delivery from the O/S. The third implementation uses a pair of threads for each socket to emulate nonblocking behavior over blocking I/O primitives.*

4.2.3 Performance analysis

To evaluate the performance of Sandstorm's asynchronous sockets layer, we implemented a simple server application which accepts bursts of 1000 8-kilobyte packets, responding with a single 32-byte ACK for each burst. This somewhat artificial application is meant to stress the network layer and measure its scalability as the number of clients increases. Figure 10 shows the aggregate throughput of the server as the number of client connections increases from 1 to 1000. The server and client machines were all 4-way 500 MHz Pentium III systems, interconnected using Gigabit Ethernet, and running Linux 2.2.5 with IBM JDK 1.1.8.

Three implementations of the socket layer are shown. The first two make use of the NBIO layer for nonblocking I/O, and the third emulates asynchronous I/O over blocking socket primitives by making use of a thread pool. The NBIO-based implementations use only 3 threads: one each for read, write, and listen stages. The threaded server allocates one thread for reading from each connection, and uses a thread pool with up to 120 threads to process write requests.

Two variants of NBIO are used: one makes use of the `poll` system call for event delivery from the operating system, and the other uses `/dev/poll`. The main difference between the two interfaces is in the overhead to test for events across many I/O streams. `poll` has known scalability problems when the number of file descriptors it considers is large [3, 23]; `/dev/poll` is an alternate interface which amortizes the cost of declaring event interest by the application. We make use of the University of Michigan's `/dev/poll` patches for the Linux kernel [34].

The results show that the NBIO-based implementations clearly outperform the threaded implementation, 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 exceeds the per-user thread limit in Linux (currently set to 512). The `/dev/poll` server sustains higher throughput than the `poll`-based server; this is due to the scalability limits of `poll` as the number of file descriptors becomes very large.

5 Application Analysis

In this section we describe the implementation and analysis of two applications built using Sandstorm: a simple HTTP server, and a Gnutella packet router. This focus of this section is on evaluating the performance and behavior of these applications under heavy load. The functionality of each application is quite rudimentary but serves to demonstrate the SEDA design in a "real world" scenario.

5.1 Simple HTTP Benchmark

We implemented a simple HTTP benchmark server in Sandstorm, consisting of a single stage which responds to each HTTP request with a static 8 kilobyte page from an in-memory cache. The server makes use of HTTP/1.1 persistent connections, which allows multiple HTTP requests to be issued over a single TCP connection. In this case, the server processes 100 HTTP requests on each connection before closing it down.

The clients are simple load generators which spin in a tight loop, opening an HTTP connection to

[3]Most of this work is being done by James Hendricks, a computer science undergraduate at Berkeley.

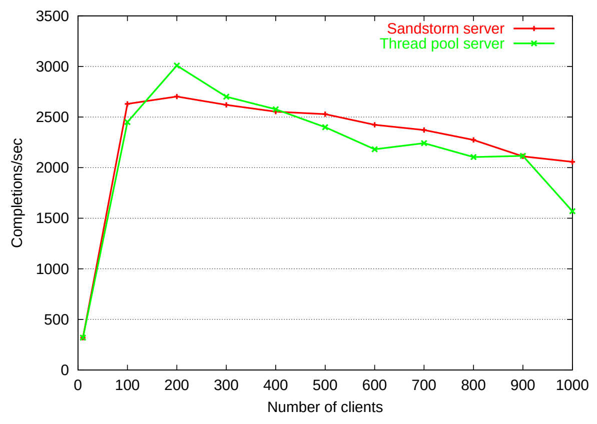


Figure 11: **Simple HTTP Server Throughput:** *This graph shows the performance of a SEDA-based HTTP server implemented using Sandstorm, and a thread pool based server using blocking I/O. The Sandstorm server consists of a single application stage with one thread, while the threaded server has a fixed pool of 150 threads.*

the server and issuing requests, sleeping for 20 ms after receiving a response. When the server closes the connection the client immediately attempts to reestablish. Although this traffic is somewhat unrealistic, the goal is to drive the server into overload with only a modest number of clients. The server and client machines are in the same configuration as described in Section 4.2.3.

To evaluate Sandstorm against the traditional threaded approach to server construction, we implemented the same benchmark using blocking socket I/O and a fixed-size thread pool of 150 threads. Each thread spins in a loop which accepts a connection, processes 100 HTTP requests, and closes the connection. This design closely resembles that of the popular Apache [40] Web server. Both server implementations are in Java.

Figure 11 shows the aggregate throughput of both servers as the number of clients is scaled from 1 to 1000. Both servers obtain good aggregate performance even when the server is very loaded. However, this result is misleading, since it ignores the fact that the thread pool server can only serve 150 clients at any given time, potentially causing the wait time for other clients to be large.

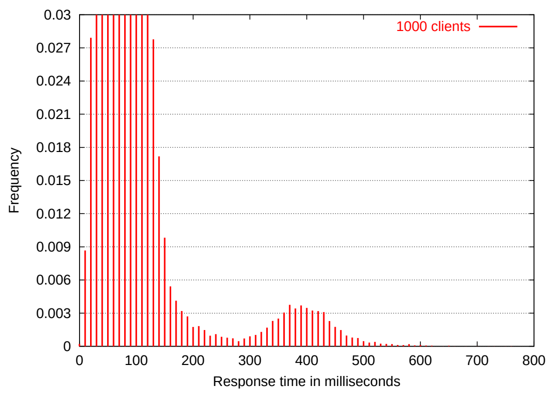
Figure 12 shows histograms of the response time for each server when loaded with 1000 clients. Here, response time only includes the wait time for a client to receive a response for an HTTP request; it does not include the wait time to establish a TCP connection to the server. Figure 12(a) shows the response time for the Sandstorm server, which has a median response time of 1105 ms and a maximum of 15344 ms. The response times are clearly concentrated at the low end of the scale, with a secondary mode around 400 ms. Figure 12(b) shows the response time for the thread pool server, which has a median response time of 4570 ms and a maximum of 190766 ms, which is over 3 minutes. This response time is probably unacceptable to most users. Note the modes at 3000 ms intervals; these continue to the right of the graph. These are due to the default TCP retransmission timeout under Linux, which is set at 3000 ms.

These results highlight the importance of using multiple metrics to analyze server applications. While the thread pool server initially appears to perform as well as the SEDA-based server, aggregate throughput is not the only measure by which a server can be evaluated. In this case, response time measurements indicate that the threaded server is providing very poor service to clients. The SPECweb99 benchmark [39] uses a related metric for performance: the number of simultaneous connections that sustain a given bandwidth.

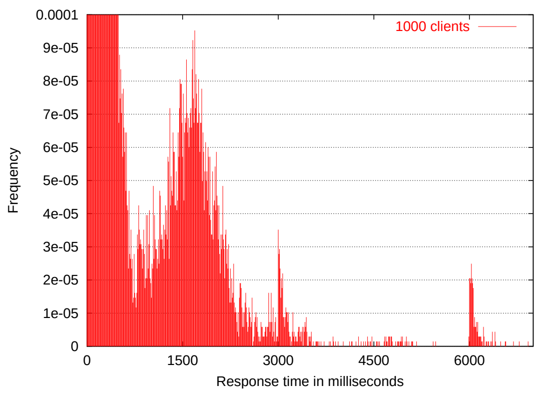
5.2 Gnutella Server

We chose to implement a Gnutella server in Sandstorm in order to explore a different class of applications from the standard client-server model, as typified by HTTP servers. Gnutella [14] is a peer-to-peer based file sharing application that allows a user running a Gnutella client to search for and download files from other Gnutella users. Nodes running the Gnutella client form an ad-hoc multihop routing network layered over TCP/IP; nodes communicate by forwarding received messages to their neighbors. Gnutella nodes tend to connect to several (four or more) other nodes at once. The initial discovery of nodes on the network can be accomplished through a number of means. One site, `gnutellahosts.com`, provides a well-known server which clients can connect to to receive a list of other Gnutella hosts.

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. Details on the message formats and routing protocol can be found in [14].



(a) Sandstorm server



(b) Thread pool server

Figure 12: **HTTP server response-time histograms:** *These graphs show histograms of the response times for the two HTTP server benchmarks when loaded with 1000 clients. The axes ranges for both histograms have been chosen to highlight their interesting features. (a) shows the response time histogram for the Sandstorm-based server; the median response time is 1105 ms with a maximum of 15344 ms. (b) shows the response time histogram for the thread pool-based server; the median response time is 4570 ms with a maximum of 190766 ms.*

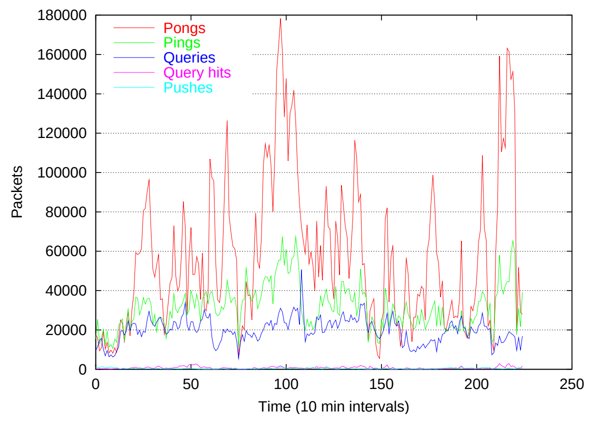


Figure 13: **Gnutella network packet trace:** *This graph shows a trace of Gnutella network activity taken over a 37-hour period using a SEDA-based Gnutella packet router. The router logged message frequencies for each type (ping, pong, query, queryhits, and push).*

The Sandstorm Gnutella server is implemented as 3 separate stages. The *GnutellaServer* stage accepts TCP connections from the Gnutella network and processes packets, passing packet events to the *GnutellaLogger* stage. *GnutellaLogger* is the main application stage and implements packet routing and logging. *GnutellaCatcher* is a helper stage used to initiate an initial connection to the Gnutella network; it does this by contacting the `gnutellahosts.com` service described earlier. The Sandstorm Gnutella server does not host any files, but simply acts as an intermediate node in the network, routing packets between other Gnutella clients. Join the "live" Gnutella network and routing packets allows us to test Sandstorm in a real-world environment, as well as sample the traffic passing through the router.

5.2.1 Packet tracing

Figure 13 shows a trace of Gnutella network activity recorded over a 37-hour period using the Sandstorm-based Gnutella packet router. Over this period of time, the server received 72396 connections from other (possibly non-unique) Gnutella network hosts, with an average of 12 connections maintained at any given time. The data shows that *pong* messages comprise the majority of network traffic, totalling more than 50% of all packets. It also shows that network traffic is extremely bursty and does not appear to follow clear diurnal patterns.

Obtaining this trace revealed a number of interesting challenges. The first is that many messages received by our server were improperly formatted, probably due to buggy Gnutella client implementations. Our original packet-processing code assumed correctly-formatted messages, and crashed when a bad message was received; this code was redesigned to detect and drop bad messages. The second challenge had to do with a memory leak which would cause the server to crash after a few hours of operation. Tracking down the source of this problem led to the development of Sandstorm's profiler, which permits the visualization of event queue lengths over time.

Such a visualization is shown in Figure 14(a), which shows the Java heap growing until it reaches its maximum size, causing the JVM to crash. The cause is that the packet router is queueing up outgoing packets for a number of connections which are not reading those packets quickly enough. This may be a result of the network link to those clients being saturated. 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. In Figure 14(a) the server was processing over 624 packets/second.

The solution was to implement thresholding on the outgoing packet queues for each socket. When a queue exceeds its threshold, the application closes the corresponding connection. Figure 14(b) shows the resulting profile when using a queue length threshold of 500. As the graph shows, several connections exceed the threshold and are terminated. The Java heap remained under 1639 Kb for the duration of this run.

5.2.2 Performance and load conditioning

To evaluate the performance aspects of the Gnutella server, we implemented a simple load-generation client which connects to the server and generates streams of packets according to a distribution mimicking that of real Gnutella traffic, obtained from a trace of Gnutella network activity. The client issues bursts of n packets with an inter-burst delay of d seconds; the load on the packet router can be increased by choosing appropriate values for n and d. Both the client and server machines use the same configuration as in the HTTP server benchmarks presented earlier.

In order to demonstrate load conditioning, we introduced a deliberate bottleneck into the server where 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. In our Gnutella traffic model, query messages comprise 15% of the generated packets. The application is using Sandstorm's thread pool-per-stage (`TPSTM`) thread manager with one thread per stage; it is clear that as the number of packets flowing into the server increases, this delay will cause large backlogs for other messages.

Figure 15(a) shows the average latencies for ping and query packets passing through the server with a burst size of $n = 10$ packets and an inter-burst delay d ranging from 100 ms to 10 ms. Packet latencies increase dramatically when the offered load exceeds the server's capacity. In the case of $d = 10$ ms, the server crashed before a latency measurement could be taken. Figure 16(a) shows the output of Sandstorm's queue length profiler in this case; the cause of the crash is the large number of packets in the incoming event queue for the `GnutellaLogger` stage, causing the Java heap to exceed its maximum size.

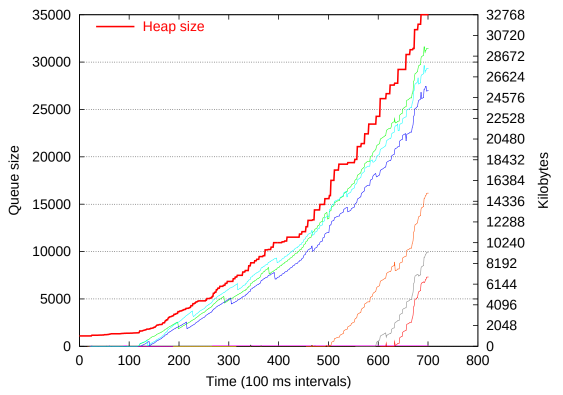
A simple way to avoid this problem is to threshold the incoming event queue for the bottleneck stage. While this approach effectively limits the size of the Java heap, it causes a large number of packets to be dropped when the queue is full. Alternately, incoming packets could be queued up in the network layer, by refusing to process socket read events; this would eventually cause the client's outgoing packet queue to fill as the TCP flow-control mechanism would prevent further packet transmissions by the client.

5.2.3 Thread pool sizing

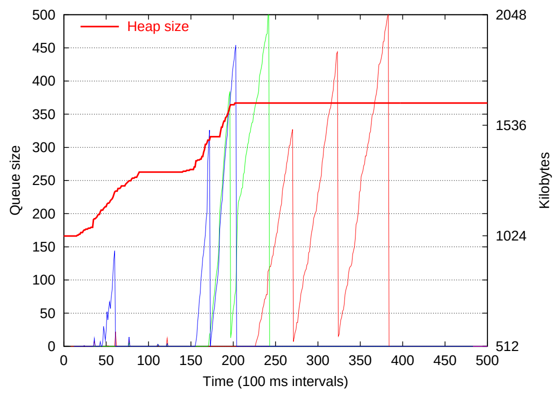
In this case, the bottleneck is caused by the application stage sleeping when it receives a query packet; however, we have allocated just one thread to the stage. As discussed in Section 3.3, the SEDA design allows multiple threads to be allocated to blocking stages, up to some reasonable limit. We have implemented this feature in Sandstorm as a *thread pool governor* within the thread pool-per-stage (`TPSTM`) thread manager. The governor dynamically adjusts the number of threads allocated to each stage based on an observation of event queue lengths.

Each event queue is assigned a threshold (in this case, the threshold is 1000). The governor samples the event queue lengths every s seconds, where s is currently set to 2; if a stage's event queue has reached its threshold, the governor adds one thread to that stage's pool, up to some maximum value (currently set to 10 threads per pool). In this way, locally optimal thread pool sizes are determined at runtime, rather than defined *a priori* by the system designer.

Figure 16(b) shows the Sandstorm profile of the Gnutella server with the governor enabled, using a queue threshold of 1000, $n = 10$, and $d = 10$ ms. In this case, two threads were added to the `GnutellaLogger` thread pool, corresponding to the



(a) No queue threshold



(b) Queue threshold = 500

Figure 14: **Queue-length profile of the Sandstorm Gnutella server:** *These graphs were generated from profile data produced by Sandstorm's profiler, and show queue lengths over time for each component of the Gnutella server application. The thick line represents the Java heap size, and the other lines represent write-queue lengths for each socket connection to the server. For clarity the full graph key has been suppressed. (a) shows the queue-length profile for the server with no socket write queue thresholding; as outgoing packets queue up for each connection, the heap grows in size until reaching its limit (set here to 32 megabytes), causing the JVM to crash. (b) shows the profile when a write queue threshold of 500 is used. When a socket exceeds this threshold, the connection is closed.*

points where the queue length reached its threshold value. As the figure shows, as threads are added to the stage, the queue length is reduced. Figure 15(b) shows the average message latencies through the server with the governor enabled.

While this approach is somewhat simplistic, it demonstrates the value of Sandstorm's thread manager abstraction. Applications need not be aware of the existence of the thread pool governor, and in fact the programming model does not expose this behavior to applications. As future work we intend to investigate other approaches to thread pool management, using more sophisticated control techniques.

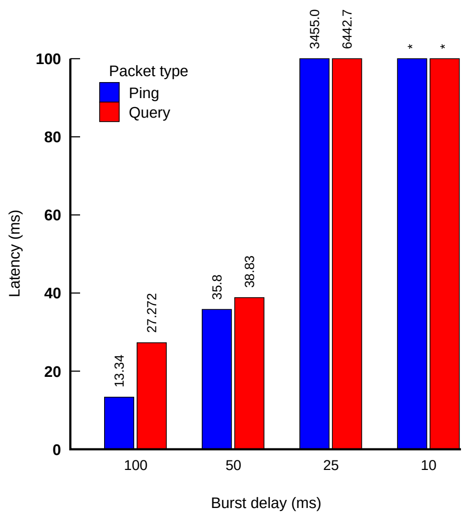
6 Research Methodology and Timeline

Evaluating the success of this research will be based on three factors. First, we will conduct a performance and load analysis of several applications built using the SEDA architecture, making use of the Sandstorm platform. Second, we will measure the effectiveness of the architecture in supporting other research projects at Berkeley, in particular the Ninja and OceanStore projects. Finally, we will release our software to the public, and measure the impact of Sandstorm and SEDA in other academic and industrial settings.

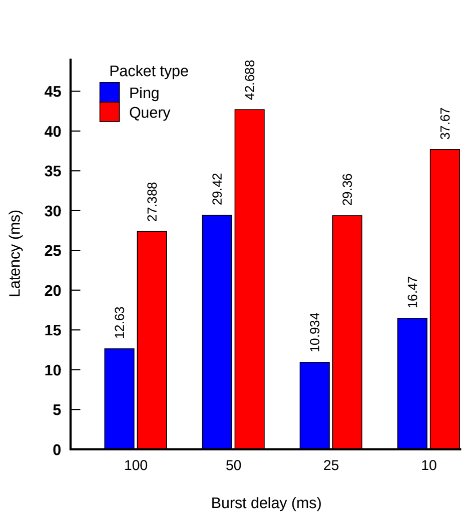
We will implement a number of interesting applications in Sandstorm, and demonstrate the use of the SEDA design to obtain high performance and good behavior under heavy load. This will include traditional applications, such as a dynamic Web server capable of supporting industry-standard benchmarks such as TPC-W [41] and SPECweb99 [39]. It will also include novel applications, such as a search engine for digital music, based on our previous work [45] in this area. Our goal is to show that the SEDA approach makes these applications easier to build, more robust to load, and more efficient.

We also intend to make the Sandstorm software available for other researchers at Berkeley, and encourage them to make use of it. The Ninja [16] project is already in the early stages of transitioning its clustered Internet services platform to Sandstorm. The OceanStore [25] project is currently developing a global storage system based on Sandstorm's I/O layer, and we plan to make the rest of the system available to them soon. By understanding the needs of other research projects which require a highly-concurrent server platform, Sandstorm will evolve into a more robust, flexible system.

We have already made a public release of Sandstorm's *NBIO* layer on the World Wide Web [44], and will release the rest of the system once it

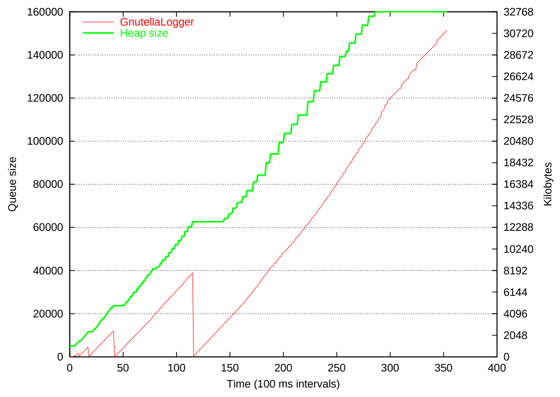


(a) Using single thread

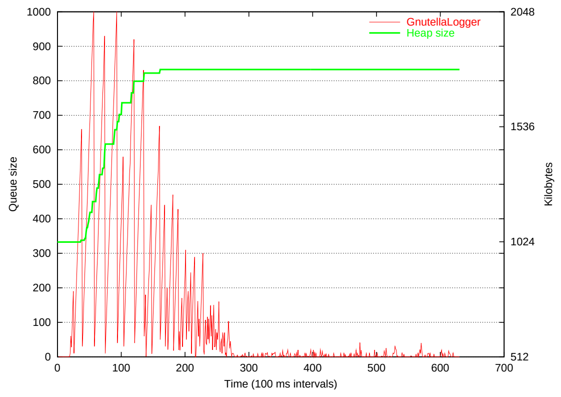


(b) Using thread governor

Figure 15: **Gnutella packet router latency:** *These graphs show the average latency of ping and query packets passing through the Gnutella packet router with a burst size of 10 packets and an inter-burst delay as shown on the x-axis. Query packets 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; with a burst delay of 10 ms, the server crashed by running out of memory before a latency measurement could be taken. (b) shows the latency with the Sandstorm thread governor enabled. Note that for* $d = 100$ *ms and* 50 *ms, no threads were added to the application stage, since the event queue never reached its threshold value. This explains the higher packet latencies over the* $d = 25$ *and* 10 *ms cases, for which 2 threads were added to the stage.*



(a) Using single thread



(b) Using thread governor

Figure 16: **Profile of bottlenecked Gnutella server:** *These graphs show the Sandstorm profile of the queue lengths within the Gnutella server with a query processing delay of 20 ms, burst size 10, and inter-burst delay of 10 ms. (a) shows the profile with a single thread servicing the* `GnutellaLogger` *stage, indicating that this stage is clearly a bottleneck. In this case the JVM crashed after the heap size reached the maximum value of 32 megabytes. (b) shows the profile with Sandstorm's thread governor enabled, which dynamically adds threads to the stage when its event queue reaches a threshold of 1000 entries. Here, the governor added two threads to the stage, effectively eliminating the bottleneck.*

has gone through additional testing and use within Berkeley. By allowing others to make direct use of our software, and supporting them in their efforts to do so, we will understand the impact of the SEDA design on other systems. NBIO has already influenced the design of a new set of I/O primitives being designed for a future release of the Java Development Kit by Sun Microsystems.

6.1 Outside of the Scope of this Work

This proposal focuses on the performance and load-conditioning aspects of single-node server applications constructed using the SEDA model. A natural extension to this model is a programming framework for massively scalable, cluster-based services. However, we do *not* intend to address replication, fault tolerance, or cluster programming models within this work. To facilitate certain scalability measurements, we may build a trivial extension of Sandstorm for clusters, but most of the interesting aspects of this problem will not be addressed. Our goal is to develop SEDA as a high-performance, single-node platform which can form the basis for an eventual cluster-based system.

6.2 Timeline

Phase 1 (0-6 months)

- Continue to develop the Sandstorm prototype system, paying particular attention to aspects of resource management, load conditioning, and thread pool scaling.
- Complete the implementation of Sandstorm's asynchronous disk I/O layer.
- Develop a dynamic HTTP server in Sandstorm capable of running industry-standard benchmarks. Analyze the performance and scaling behavior of this application under realistic loads.

Phase 2 (6-12 months)

- Develop a second application, such as a Gnutella "crawler" coupled with the music similarity search engine described in [45]. Use the application to drive investigation of other aspects of the SEDA design, such as alternate thread and event scheduling algorithms.
- Work with Ninja and OceanStore project members to encourage adoption of Sandstorm. Release Sandstorm code to the public.

Phase 3 (12-18 months)

- Incorporate feedback from other researchers into a revision of the Sandstorm software. Evaluate impact of Sandstorm and SEDA on other applications.
- Continue to develop tools for programming in the SEDA model, including debugging and visualization tools.
- Write thesis and graduate.

7 Related Work

The most relevant prior work is derived from the body of literature on the design of high-performance Web servers. The Flash web server [32] and the Harvest web cache [6] are based on an asynchronous, event-driven model which closely resembles the SEDA architecture. In Flash, each component of the web 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. Both Flash and Harvest typify the "monolithic" event-driven architecture described earlier. SEDA generalizes this approach by targeting a general-purpose framework for highly-concurrent applications.

The JAWS web server [19] combines an event-driven concurrency mechanism with a high-level programming construct, the *Proactor pattern* [20], which is intended to simplify the development of event-driven applications. Like SEDA, JAWS decouples event-handling mechanisms from thread management; unlike SEDA, JAWS does not expose event queues or permit application-specific load conditioning.

Other systems have addressed aspects of the design approach taken by SEDA. StagedServer [26] is a platform for server applications which makes direct use of the SEDA design, by breaking application components into stages separated by queues. In StagedServer, the motivation is to improve cache locality by scheduling stages in a way which avoids preemption. Sandstorm's thread-per-CPU thread manager implements a similar algorithm, although we have yet to investigate the cache locality benefits of this approach. In the case of SEDA, the focus is on high concurrency, load conditioning, and ease of use. We view SEDA and StagedServer as two sides of the same coin, and intend to draw on the lessons learned from the StagedServer design.

The Click modular packet router [30] uses a software architecture which is similar to our framework; packet processing stages are implemented by separate code modules with their own private state. Click modules communicate using either queues or function calls. Click is optimized to improve per-packet latency through the router, allowing a single thread to call directly through multiple stages. In SEDA, threads are isolated to their own stage for reasons of safety and load conditioning.

The Scout operating system [31] is also based on a design analogous to SEDA. In Scout, applications consist of a set of modules composed into a *path*. Scout uses the path abstraction to implement vertical resource management and integrated layer processing, applying the mechanism primarily to the implementation of multimedia network protocols. Like Click, Scout threads may call through multiple stages. Scout's resource management model [38] is similar to that proposed by resource containers [1], which allow the resources for an entire data flow through the system to be managed as a unit. We intend to apply similar techniques in SEDA to provide more complex load conditioning algorithms.

Much prior work has investigated scalable I/O primitives for server applications. We intend to build upon mechanisms for scalable network and disk I/O [2, 22, 43, 33, 9] and I/O event delivery [3, 27, 34, 35], incorporating these primitives into implementations of SEDA. Sandstorm's asynchronous sockets layer makes use of the `/dev/poll` event delivery mechanism as first described by [3].

The vast body of literature in scheduling algorithms is relevant to the selection of a thread and event-scheduling policy within SEDA. Crovella *et. al.* [7] and Harchol-Balter *et. al.* [18] investigated the use of shortest-connection-first and shortest-remaining-processing-time scheduling in Web servers; these mechanisms could easily be implemented in SEDA through event queue reordering. Bender *et. al.* [4] discuss alternate metrics for measuring Web server performance, and evaluate several task scheduling policies within their framework. These metrics could be useful within the context of SEDA-based applications.

8 Conclusion

The staged event-driven architecture is designed to make highly-concurrent applications more efficient, easier to build, and more robust. Our initial prototype of a SEDA-based Internet services platform, Sandstorm, demonstrates good performance and scalability, as well as adequate flexibility to support different types of applications. We have explored some of the features of the SEDA design through two simple applications, and have presented initial performance and load conditioning results. Much remains to be explored within the space of highly-concurrent server designs. Moving forward, we intend to build larger, more realistic applications, and use them to drive investigations into thread and event scheduling, load conditioning mechanisms, I/O scalability, and debugging tools within the SEDA environment.

References

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

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

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

[4] M. Bender, S. Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1998.

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

[6] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical internet object cache. In *Proceedings of the 1996*

Usenix Annual Technical Conference, pages 153–163, January 1996.

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

[8] Dataquest, Inc. Press Release. `http://www.gartner.com/dq/static/about/press/pr-b09252000.html`, September 2000.

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

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

[11] B. Gallmeister. *POSIX.4: Programming for the Real World*. O'Reilly and Associates, 1995.

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

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

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

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

[16] S. D. Gribble, M. Welsh, R. von Behren, E. A. 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.

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

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

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

[20] J. C. Hu, I. Pyarali, and D. C. Schmidt. Applying the Proactor pattern to high-performance Web servers. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems*, October 1998.

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

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

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

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

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

[26] J. Larus. Enhancing server performance with StagedServer. `http://www.research.microsoft.com/~larus/Talks/StagedServer.ppt`, October 2000.

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

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

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

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

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

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

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

[34] 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.

[35] M. Russinovich. Inside I/O Completion Ports. `http://www.sysinternals.com/comport.htm`.

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

[37] T. K. Sellis. Multiple-query optimization. *TODS*, 13(1):23–52, 1988.

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

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

[40] The Apache Web Server. `http://www.apache.org`.

[41] Transaction Processing Performance Council. TPC-W benchmark specification. `http://www.tpc.org/wspec.html`.

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

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

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

[45] M. Welsh, N. Borisov, J. Hill, R. von Behren, and A. Woo. Querying large collections of music for similarity. Technical Report UCB/CSD-00-1096, U.C. Berkeley Computer Science Division, November 1999.

[46] M. Welsh, S. D. Gribble, E. A. Brewer, and D. Culler. A design framework for highly concurrent systems. Technical Report UCB/CSD-00-1108, U.C. Berkeley Computer Science Division, April 2000.

[47] Yahoo! `http://www.yahoo.com`.

[48] Yahoo! Inc. Press Release. `http://docs.yahoo.com/docs/pr/3q00pr.html`, October 2000.