A Design Framework for Highly Concurrent Systems

Matt Welsh, Steven D. Gribble, Eric A. Brewer, and David Culler
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720 USA
{mdw,gribble,brewer,culler}@cs.berkeley.edu

Abstract

Building highly concurrent systems, such as large-scale Internet services, requires managing many information flows at once and maintaining peak throughput when demand exceeds resource availability. In addition, any platform supporting Internet services must provide high availability and be able to cope with burstiness of load. Many approaches to building concurrent systems have been proposed, which generally fall into the two categories of threaded and event-driven programming. We propose that threads and events are actually on the ends of a design spectrum, and that the best implementation strategy for these applications is somewhere in between.

We present a general-purpose design framework for building highly concurrent systems, based on three design components — *tasks*, *queues*, and *thread pools* — which encapsulate the concurrency, performance, fault isolation, and software engineering benefits of both threads and events. We present a set of design patterns that can be applied to map an application onto an implementation using these components. In addition, we provide an analysis of several systems (including an Internet services platform and a highly available, distributed, persistent data store) constructed using our framework, demonstrating its benefit for building and reasoning about concurrent applications.

1 Introduction

Large Internet services must deal with concurrency at an unprecedented scale. 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 high concurrency, Internet services have three other properties which necessitate a fresh look at how these systems are designed: burstiness, continuous demand, and human-scale access latency. Burstiness of load is fundamental to the Internet; dealing with overload conditions must be designed into the system from the beginning. Internet services must also exhibit very high availability, with a downtime of no more than a few minutes a year. Finally, because access latencies for Internet services are at human scale and are limited by WAN and modem access times, an important engineering tradeoff to make is to optimize for high throughput rather than low latency.

Building highly concurrent systems is inherently difficult. Structuring code to achieve high throughput is not well-supported by existing programming models. While threads are a commonly used device for expressing concurrency, the high resource usage and scalability limits of many thread implementations has led many developers to prefer an event-driven approach. However, these event-driven systems are generally built from scratch for particular applications, and depend on mechanisms not well-supported by most languages and operating systems. In addition, using event-driven programming for concurrency can be more complex to develop and debug than threads.

That threads and events are best viewed as the opposite ends of a design spectrum; the key to developing highly concurrent systems is to operate in the *middle* of this spectrum. Event-driven techniques are useful for obtaining high concurrency, but when building real systems, threads are valuable (and in many cases required) for exploiting multiprocessor parallelism and dealing with blocking I/O mechanisms. Most developers are aware that this spectrum exists, by utilizing both thread and event-oriented approaches for concurrency. However, the dimensions of this spectrum are not currently well understood.

We propose a general-purpose design framework for building highly concurrent systems. The key idea behind our framework is to use event-driven programming for high throughput, but leverage threads (in limited quantities) for parallelism and ease of programming. In addition, our framework addresses the other requirements for these applications: high availability and maintenance of high throughput under load. The former is achieved by introducing fault boundaries between application components; the latter by conditioning the load placed on system resources.

This framework provides a means to reason about the structural and performance characteristics of the system as a whole. We analyze several different systems in terms of the framework, including a distributed persistent store and a scalable Internet services platform. This analysis demonstrates that our design framework provides a useful model for building and reasoning about concurrent systems.

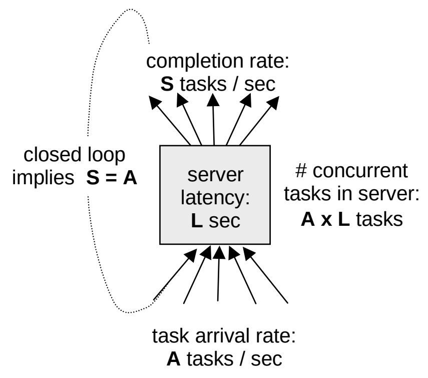

Figure 1: *Concurrent server model:* The server receives A tasks per second, handles each task with a latency of L seconds, and has a service response rate of S tasks per second. The system is closed loop: each service response causes another tasks to be injected into the server; thus, $S = A$ in steady state.

2 Motivation: Robust Throughput

To explore the space of concurrent programming styles, consider a hypothetical server (as illustrated in Figure 1) that receives A tasks per second from a number of clients, imposes a server-side delay of L seconds per task before returning a response, but overlaps as many tasks as possible. We denote the task completion rate of the server as S. A concrete example of such a server would be a web proxy cache; if a request to the cache misses, there is a large latency while the page is fetched from the authoritative server, but during that time the task doesn't consume CPU cycles. For each response that a client receives, it immediately issues another task to the server; this is therefore a closed-loop system.

There are two prevalent strategies for handling concurrency in modern systems: *threads* and *events*. 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. The alternative, *events*, allows programmers to manage concurrency explicitly by structuring code as a single-threaded handler that reacts to events (such as non-blocking I/O completions, application-specific messages, or timer events). We explore each of these in turn, and then formulate a robust hybrid design pattern, which leads to our general design framework.

2.1 Threaded Servers

A simple threaded implementation of this server (Figure 2) uses a single, dedicated thread to service the network, and hands off incoming tasks to individual task-handling threads, which step through all of the stages of processing that task. One handler thread is created per task. An optimization of this simple scheme creates a pool of several threads in advance and dispatches tasks to threads from this pool, thereby amortizing the high cost of thread creation and destruction. In steady state, the number of threads T that execute concurrently in the server is $S \times L$. As the per-task latency increases, there is a corresponding increase in the number of concurrent threads needed to absorb this latency while maintaining a fixed throughput, and likewise the number of threads scales linearly with throughput for fixed latency.

Threads have become the dominant form of expressing concurrency. Thread support is standardized across most operating systems, and is so well-established that it is incorporated in modern languages, such as Java [9]. Programmers are comfortable coding in the sequential programming style of threads and tools are relatively mature. In addition, threads allow applications to scale with the number of processors in an SMP system, as the operating system can schedule threads to execute concurrently on separate processors.

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.

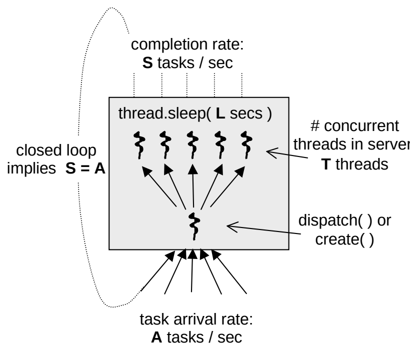

Figure 2: *Threaded server:* For each task that arrives at the server, a thread is either dispatched from a statically created pool, or a new thread is created to handle the task. At any given time, there are a total of T threads executing concurrently, where $T = A \times L$.

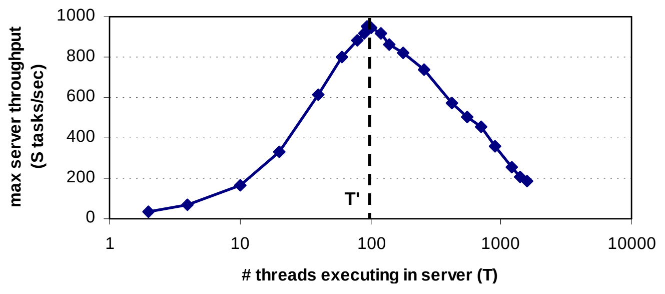

Figure 3: *Threaded server throughput degradation:* This benchmark has a very fast client issuing many concurrent 150-byte tasks over a single TCP connection to a threaded server as in Figure 2 with $L = 50$ms on a 167 MHz UltraSPARC running Solaris 5.6. The arrival rate determines the number of concurrent threads; sufficient threads are preallocated for the load. As the number of concurrent threads T increases, throughput increases until $T \geq T'$, after which the throughput of the system degrades substantially.

Regardless of how well the threaded server is crafted, as the number of threads in a system grows, operating system overhead (scheduling and aggregate memory footprint) increases, leading to a decrease in the overall performance of the system. There is typically a maximum number of threads T' that a given system can support, beyond which performance degradation occurs. This phenomenon is demonstrated clearly in Figure 3. 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.

2.2 Event-Driven Servers

An event-driven implementation of this server uses a single thread and non-blocking interfaces to I/O subsystems or timer utilities to "juggle" between the concurrent tasks, as shown in Figure 4. Event-driven systems are typically structured as a program thread that loops continuously, processing events of different types from a queue. This thread either blocks on or polls the queue to wait for new events.

Event-driven programming has its own set of inherent challenges. The sequential flow of each task is no longer handled by a single thread; rather, one thread processes all tasks 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. Event packages are not standardized, and there are few debugging tools for event-driven programmers. However, event-driven programming avoids many of the bugs associated with synchronization, such as race conditions and deadlocks.

Events generally cannot take advantage of SMP systems for performance, unless multiple event-processing threads are used. Also, 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. In addition, it may be impossible for all application code be non-blocking; often, standard library components and third-party code export blocking interfaces. In such cases, threads are valuable in that they provide a mechanism for obtaining concurrency over these blocking interfaces.

Event-driven systems tend to be robust to load, with little degradation in throughput as offered load increases beyond that which the system can deliver. If the handling of events and bundling of task state is efficient, the peak throughput can be high. Figure 5 shows the throughput achieved on an event-driven implementation of the network service from Figure 4 as a function of the load. The throughput exceeds that of the threaded server, but more importantly does not degrade with increased concurrency. 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, the excess tasks are absorbed in the queues of the system, either in the main event queue of the server, or in the network

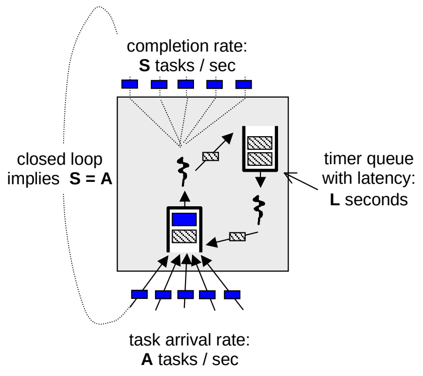

Figure 4: *Event-driven server:* Each task that arrives at the server is placed in a main event queue. The dedicated thread serving this queue sets an L second timer per task; the timer is implemented as queue which is processed by another thread. When a timer fires, a timer event is placed in the main event queue, causing the main server thread to generate a response.

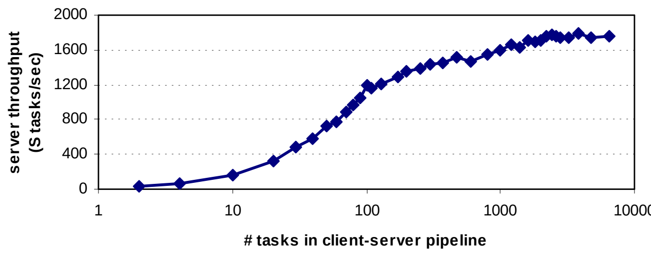

Figure 5: *Event-driven server throughput:* Using the same benchmark setup as in Figure 3, this figure shows the event-driven server's throughput as a function of the number of tasks in the pipeline. The event-driven server has one thread receiving all tasks, and another thread handling timer events. The throughput flattens in excess of that of the threaded server as the system saturates, and the throughput does not degrade with increased concurrent load.

stack queues associated with the client/server transport connection. The throughput of the server remains constant in such a situation, although the latency of each task increases.

Concurrency is explicit in the event-driven approach and associated directly with queues. With multiple tasks at various stages exposed in one or more queues, programmers can make use of application-specific knowledge to reorder event processing for prioritization or efficiency reasons.

2.3 The Thread and Event Spectrum

Although highly concurrent services provide new fuel for the debate over threads and events, a key observation is that the design space for concurrency isn't limited to these two points. Rather, there is a spectrum between these extremes and it is possible to build hybrid systems that strategically exploit properties of both. The goal is to obtain the benefit of threads, but to limit their number so as to prevent performance degradation. The question to be asked is how many threads are needed, and whether that number can fit within the operating range of the system.

The simplest example of a hybrid thread and event server is shown in Figure 6. It limits the number of concurrent threads running in the system to no more than T threads in a preallocated thread pool, and buffers incoming tasks in an event queue from which the thread pool feeds. Each task is handled by a single thread using standard blocking interfaces and developed using familiar tools. Excess concurrency is absorbed by the queue, and thus the response time increases with load but throughput does not degrade.

This hybrid approach makes it possible to determine whether a given implementation is sufficient to meet a target efficiency and throughput. Returning to our example service, the concurrency demand on the system is $A \times L$, which is serviced by the T threads in the pool. Within the operating regime where $A \times L \leq T \leq T'$, the hybrid server performs as well as an event-driven server, as shown in Figure 7(a). However, if $A \times L > T'$, then it will be impossible to service the concurrency demand without creating more than T' threads, as shown in Figure 7(b). If the size of the thread pool exceeds T', throughput degrades regardless of whether the thread pool is as large as the concurrency load. Therefore, T should be set to never exceed T', and if $A > T/L$, then the excess tasks will accumulate in the task queue, which absorbs bursts but increases the latency to process each task.

Figure 7(c) shows the performance of a microbenchmark implementation of this hybrid server for various values of L, demonstrating that as L increases, the hybrid system will be unable to meet the concurrency demand $A \times L$. Note that because tasks are buffered by the incoming queue, throughput will not degrade as long as the size of the thread pool is chosen to be less than T', as shown in Figure 7(d).

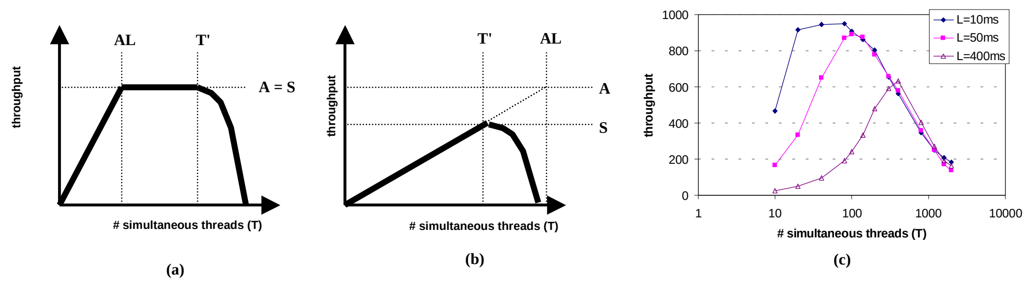

Figure 7: *Throughput of the hybrid event and thread system:* (a) and (b) illustrate the theoretical performance of the hybrid server, where T' is larger or smaller than the concurrency demand $A \times L$. (c) shows measurements of the benchmark presented in Figure 3, augmented by placing a queue in front of the thread pool, for different values of L and T. (d) shows the throughput of the hybrid server when $T = T'$, which is the optimal operating point of the server. Here, $L = 50$ms. The middle plateau in (d) corresponds to the point where the pipeline has filled and convoys are beginning to form in the server. The right-hand plateau in (d) signifies that the convoys in all stages of the pipeline have merged. Note that the x-axis of (a) and (b) are on a linear scale, while (c) and (d) are on logarithmic scales.

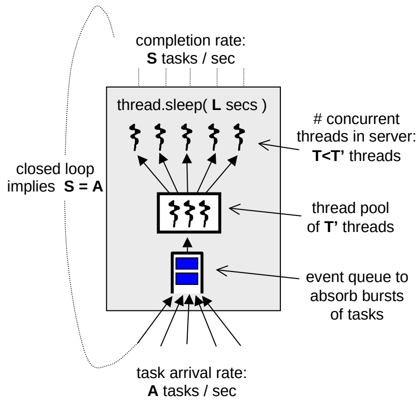
completion rate:
S tasks / sec
thread.sleep(L secs)
# concurrent threads in server:
T<T' threads
closed loop implies S = A
thread pool of T' threads
event queue to absorb bursts of tasks
task arrival rate:
A tasks / sec

Figure 6: *A hybrid thread and event system:* This server uses a constant-size thread pool of T threads to service tasks with an arrival rate of A from an incoming task queue; each task experiences a service latency of L seconds. If the number of tasks received by the hybrid server exceeds the size of the thread pool, the excess tasks are buffered by the incoming task queue.

2.4 The Case for a New Design Framework

There are many parameters to consider in the design space of highly concurrent systems. One must not only strike the right balance between event-driven and threads-based concurrency; issues such as fault isolation, data sharing, and ease of programming are important concerns as well. The number of choices in this design makes it difficult for application designers to construct new applications easily and reason about how they will perform.

This initial investigation into the spectrum of concurrency models indicates the need for systematic techniques for mapping an application design onto an implementation that provides efficiency and robustness despite the inherent limits of the underlying operating system. We feel that the design space that represents the full spectrum between threads and events has not yet been adequately explored; this paper attempts to understand its characteristics and the tradeoffs inherent when building applications in this space.

Rather than continue to build new concurrency models in an *ad hoc* manner for specific applications, we propose a general-purpose design framework that addresses the holistic requirements of highly concurrent systems. This framework provides a structure within which one can build and reason about large-scale server applications. By following a small set of specific design principles, application designers can make certain assertions about the performance and fault-isolation properties of their system.

3 Design Framework

3.1 Framework Components

There are four general-purpose building blocks for constructing applications under our framework: *tasks*, *thread pools*, and *queues*.

3.1.1 Tasks

A *task* is the fundamental unit of work in our framework. It is a typed message that contains a description of some work to be done, along with the data required to complete that task. An example of a task might be "retrieve the contents of a web page" where the data associated with that task is a URL.

Tasks are processed in a series of *stages* by individual components of the application. For example, for a web server to retrieve a static HTML page, the task's URL must be first parsed, the page looked up in a local cache, and finally (if necessary) read from disk. Stages of a task can either be executed in sequence or in parallel, or a combination of the two.

By decomposing the task into a series of stages, it is possible to distribute those stages over multiple physical resources, and reason about the flow of those tasks for load-balancing and fault-isolation purposes.

3.1.2 Thread pools

A *thread pool* is a collection of threads on the same machine that operate by continually processing tasks. Logically, a thread pool is associated with a set of task types, and each thread in the pool executes a piece of code that consumes a task, processes it, and dispatches one or more outgoing tasks to a thread pool. A thread pool must have at least one thread within it.

Thread pools are the only source of execution contexts within our framework. Modularity is accomplished by structuring applications as a set of thread pools each processing a particular set of task types. Parallelism can be exploited since each thread can run on a separate CPU in a multiprocessor system.

3.1.3 Queues

Queues are the means of communication between thread pools. A queue logically consists of a list of tasks; thread pools pull tasks from their incoming task queue, and dispatch tasks by pushing them onto the incoming queues of thread pools. The operation of two thread pools can be composed by inserting a queue between them, thereby allowing tasks to pass from one to the another. We call a thread pool coupled with its incoming task queue a *task handler*.

Queues act as the separation mechanism between thread pools, by introducing an explicit control boundary. Because a thread cannot cross over this boundary (it can only pass data across the boundary by enqueuing a task), it is possible to constrain the execution of threads to a given task handler. This is desirable for two reasons. First, it makes applications easier to debug, as the thread pool's internal state is not visible to other thread pools. Second, it can eliminate cases where threads "escape" into a piece of code where they may never return — for example, into a library that performs blocking I/O operations.

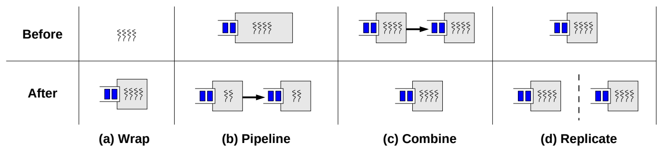

Figure 8: *The four design patterns.* This figure depicts each of the four design patterns used to construct applications within our framework.

In addition, queues provide a mechanism for overflow absorption, backpressure, and fairness. Queues act to buffer incoming tasks when their number outweighs the number of threads available to process them. 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. Fairness can be accomplished by scheduling thread pools based on their incoming queue length.

3.2 Design Patterns

We turn now to the question of how to map an application onto an implementation using the components presented above. The decision of where thread pools and queues are used has an important impact on the performance and fault-tolerance properties of the resulting application.

We propose four *design patterns* that can be applied to construct an application using our framework. These patterns encapsulate fundamental properties of our framework components and describe how to build up an application using them.

3.2.1 *Wrap*

The *Wrap* pattern involves "wrapping" a set of threads with a queue interface, as illustrated in Figure 8(a). Each thread processes a single task through some number of stages and may block one or more times. Applying *Wrap* places a single input queue in front of the set of threads, effectively creating a task handler out of them. This operation makes the processing within the task handler robust to load, as the number of threads inside of the task handler can now be fixed at a value that prevents thread overhead from degrading performance, and additional tasks that cannot be serviced by these threads will accumulate in the queue.

The *Wrap* pattern was used in our example thread/event hybrid system in Section 2.3. In that example, each thread in the pool processed a single task and made one blocking call (a sleep operation). By fixing the number of threads within the pool to some value $T < T'$ and placing a queue in front of them, peak throughput was be maintained despite the threads sleeping within the task handler.

3.2.2 *Pipeline*

The *Pipeline* pattern takes a single-threaded piece of code and splits it into multiple pipeline stages by introducing a queue and thread pool boundary at various points. This is illustrated in Figure 8(b). For example, if queues are introduced for each blocking I/O call, this operation makes each call appear to be non-blocking, as separate threads are responsible for processing the task on either side of the call.

The *Pipeline* pattern has two uses. The first is to limit the number of threads allocated for handling low-concurrency operations. If a pipeline stage has a task arrival rate of A tasks/second and a per-task latency of L seconds, the number of threads needed to maintain a completion rate of A for that stage is $A \times L$, as in Section 2.3. Now consider a pipeline stage which is limited in the number of concurrent tasks that it can handle. For example, UNIX filesystems can generally handle a fixed number (between 40 and 50) concurrent read/write requests before becoming saturated. We call this limit the *width* of the pipeline stage and denote it by the value W.

The width of a pipeline stage places an upper bound on the number of threads that yield concurrency benefit to that stage. That is, if $W < (A \times L)$ there is no need to supply the stage with more than W threads; additional threads within that stage would only remain idle. The completion rate S of the stage will be $S = W/L$; if $W < (A \times L)$ then $S < A$. Limited width therefore limits the completion rate of a task handler; width can only be increased through replication (discussed below).

Pipeline provides a way of limiting the threads needed for a particular pipeline stage to that stage's width, rather than "wasting" threads that would otherwise remain idle. Breaking the processing for a task into separate pipeline stages allows the size of the thread pool to be tailored for that stage, and allows that stage to be replicated across separate physical resources (as we will see below) to achieve greater parallelism.

The second use of the *Pipeline* pattern is to increase locality. Cache locality is an increasingly important concern for building high-throughput systems as the performance gap between cache and main memory increases. In addition, the use of multiple processors taxes the available memory bandwidth and can lead to lengthy stall times if a cache miss must be taken.

In a thread-per-task system, the instruction cache tends to take many misses as the thread's control passes through many unrelated code modules to process the task. In addition, whenever a context switch occurs (due to thread preemption or a blocking I/O call, say), other threads will invariably flush the waiting thread's state out of the cache. When the original thread resumes execution, it will need to take many cache misses in order to bring its code and state back into the cache. In this situation, all of the threads in the system are competing for limited cache space.

Applying the *Pipeline* pattern can increase data and instruction cache locality to avoid this performance hit. Each pipeline stage can process a "convoy" of tasks all at once, keeping the instruction cache warm with its own code, and the data cache warm with any shared data used to process the convoy. In addition, each pipeline stage has the opportunity to service incoming tasks in an order that optimizes for data cache locality. For example, if queues are serviced in last in, first out order, then the tasks that arrived most recently may still be in the data cache.

Although *Pipeline* increases task processing latency, recall that the goal of our framework is to optimize for aggregate throughput, not the time to process an individual task.

3.2.3 *Combine*

The *Combine* pattern (Figure 8(c)) combines two separate task handlers into a single task handler with a shared thread pool. As we have seen, if the number of threads in the system is too high then performance will degrade. *Combine* is used to allow multiple task handlers to share a thread pool for the purposes of thread conservation.

Consider a set of three sequential pipeline stages resulting from the use of *Pipeline* to isolate a blocking operation into its own thread pool. If the first and third stages are both CPU bound, their width W is the number of CPUs in the system. Rather than have two thread pools of size W, a single thread pool can be shared between them by applying *Combine*. In this sense, *Combine* is the inverse of *Pipeline*.

3.2.4 *Replicate*

The *Replicate* pattern (Figure 8(d)) makes a "copy" of an existing task handler. Additionally, it either instantiates that task handler on a new set of physical resources, places a failure boundary between the two copies, or both. *Replicate* is used to achieve both parallelism and fault isolation.

By replicating a task handler across physical resources, the combined width W of the replicas is increased. This can be used to eliminate a bottleneck in a pipeline of task processing stages. A failure boundary can be introduced between two replicas by running them in separate address spaces or on separate machines. This is done to make the replicas highly available; if one fails, the other can continue processing tasks.

Replication raises concerns about distributed state management. The failure of a network link within the cluster can lead to partitioning, which is troublesome if task handlers residing on different cluster nodes need to maintain consistent state. There are several ways to avoid this problem. One is to employ one of various distributed consistency or group membership protocols [20, 27]. Another is to engineer the cluster interconnect to eliminate partitioning. This is the approach taken by DDS [10] (see Section 4.1) and the Inktomi search engine [6].

3.3 Applying the Design Patterns

In this section we provide a set of heuristics that can be used to determine how the above design patterns should be applied to a given application. Each of the patterns has an effect on the performance or fault-isolation properties of the application. In addition, there are a number of constraints that must be met to keep the application within its operational range. Obtaining the desired application properties, therefore, can be viewed as solving a multivariate constraint problem. The goal is to maximize the service task completion rate and minimize the probability of application failure. The various constraints include:

Physical resource limits: The number of cluster nodes available to the application is assumed to be fixed at some value. Likewise, the CPU speed, amount of memory, disk and network bandwidth, and other parameters are assumed fixed for a given configuration.

Thread limits: As we have seen, thread implementations generally impose a limit on the number of threads that the system can support before performance degradation occurs, which we call T'. To what extent this limit applies depends on the thread implementation. A user-level threads package may allow T' threads per process, while kernel-level threads may be limited to T' per machine.

Latency and width: The values for L and W for a given task processing stage drive much of the decision to apply the design patterns above.

Starting with a thread-per-task implementation, the following design rules can be used to transform that implementation into one that obtains the benefits of our framework.

Apply *Wrap* to introduce load conditioning: Placing the application threads within a task handler structure allows the number of threads to be limited to some optimum value (less than T'), with additional tasks absorbed by the queue. This was the transformation used to produce the hybrid system in Section 2.3.

Apply *Pipeline* to avoid wasting threads: After the above transformation, it is possible to increase throughput by isolating code with a low width W into its own task hander, and limit the number of threads within that task handler to W. For example, one can apply *Pipeline* at the point where filesystem calls are made, and limit the number of threads within the filesystem task handler to the number of concurrent accesses that the filesystem can handle. This effectively "frees" additional threads which can be placed into other task handlers that can make use of them.

Apply *Pipeline* for cache performance: As discussed above, *Pipeline* can also be used to increase cache locality by isolating the code for related task processing stages within its own task handler. The cache locality benefit of structuring applications in this way depends on two major factors. The first is the amount of code and shared data used internally by a task handler when processing a convoy of tasks. This determines the potential cache locality benefit that can be achieved internally to a task handler. The second is the amount of data carried between task handlers when a task is pushed onto a queue. This determines the potential data cache locality benefit of processing tasks out of order. The performance effect of pipelining task processing can be measured directly, and used to determine whether a particular application should use this structure.

Apply *Replicate* for fault tolerance: Replication of task handlers across multiple resources increases the reliability of the application as a whole, by introducing redundancy for task processing stages. If we replicate a task handler n times, and the probability of a failure for each task handler during a given time interval is F_0, then the probability that all replicas will fail during this time is $F = F_0^n$. As we can see, replication has an exponential effect on the reliability of the set of task handlers.

The benefit of replication depends on several factors. One is whether enough physical resources are available to support replication. If we wish to replicate across cluster nodes n times to achieve a particular failure probability F, then clearly n cluster nodes must be available. Another is whether the task handlers to be replicated will rely on shared state, which raises several design concerns, as discussed above.

Apply *Replicate* to scale concurrency: *Replicate* can also be used to effectively increase the concurrency width W of a particular task handler by running multiple instances on separate resources. In general replicating a task handler n times increases the width of the collective replicas to $W \times n$. For a task handler with latency L, this increases the task completion rate of the combined replicas from $S = W/L$ to $S = (W \times n)/L$, which can eliminate the throughput bottleneck.

Apply *Combine* to limit the number of threads per node: After pipelining and replicating the thread-per-task application, we may end up with a large number of task handlers each with their own private thread pool. If the latency of some of these task handlers is low, then it makes sense to "share" thread pools between them, thereby reducing the overall number of threads. As discussed above it is important to limit the number of threads per node to some value T', which can be measured for a given system. *Combine* merges the thread pools for disparate task handlers on the same node, allowing threads to be conserved.

These heuristics assumes that the application designer is also keeping in mind the inherent resource limits of the underlying platform. For example, a task handler should not be placed onto a node if that would increase the total number of threads on that node beyond T'. While these heuristics can help to structure an application using our framework, they are by no means exhaustive.

The structure of an application constructed using our framework is illustrated in Figure 9. Here we see 6 thread pools, each with their own incoming task queue, running on 5 separate machines. The communication arrows are only intended as an example of how tasks might flow between application components; a real application would involve many more components and communication paths between them.

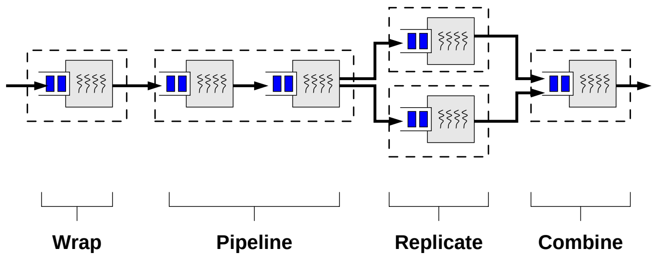

Figure 9: *An application resulting from the use of the design patterns.* Data flows from left to right, with tasks entering the application at the leftmost queue. Dashed boxes represent machine boundaries. Each design pattern is labelled below its use.

3.4 Principles

Apart from the design patterns presented above, our framework reveals several other principles should be followed when engineering a system.

First, task handlers should be stateless whenever possible. This allows a task handler to be lock-free as no state is shared between its threads. In addition this allows task handlers to be easily created or restarted on demand, for example, in response to a load spike or failure.

Second, data associated with tasks should be passed by value, rather than by reference, whenever possible. Data sharing between two task handlers raises a number of concerns. Consistency of shared data must be maintained using locks or a similar mechanism; locks can lead to race conditions and long acquisition wait-times when contended for, which in turn reduces concurrency. Also, passing data by reference is problematic when two task handlers are located in different addresses spaces or machines. While Distributed Shared Memory (DSM) [18] can be used to make cross-address-space sharing transparent, DSM mechanisms are complex and raise concurrency concerns of their own. Data sharing requires task handlers to agree upon who is responsible for deallocating data once it is no longer used. In a garbage-collected environment (within a single address space) this is straightforward; without garbage collection, more explicit coordination is required. Perhaps most importantly, data sharing reduces fault isolation. If a task handler fails and leaves shared data in an inconsistent state, any other task handlers sharing that data must be able to recover from this situation or risk failure themselves.

An alternative to passing by value is to pass by reference with the originator relinquishing access. Another means of reducing data sharing is to space-partition application state, in which multiple pipeline stages or replicas of a task handler process their own private partition of the application state, rather than sharing state and using locks to maintain consistency.

The third principle is that fate sharing should be avoided. If two task handlers share physical resources, they also share their fate: that is, if the physical resources fail (e.g., the node crashes), both task handlers will also fail. Clearly, replicas of the same task handler should be kept on separate physical nodes, or at least in separate address spaces, to avoid this fate sharing.

Note that task handlers can be linked in terms of load, not just in terms of failures. For example, if one replica of a task handler fails, other replicas will be required to take on its load in order to maintain throughput.

The fourth principle is that admission control should be done at the "front door" of the application, that is, at its interface to the outside world. By denying or delaying task entry at the application's topmost layer, task handlers are able to stay within their operational range in terms of resource demand. This approach requires that all task handlers implement backpressure by rejecting new tasks when queue lengths become too long. Without an external admission control mechanism, a slow task handler can become overloaded and many tasks can queue up within the application. This increases latency as well as resource demand within the system.

4 Analysis of the Framework

To demonstrate its utility, we analyze a number of existing systems in terms of our design framework. These systems predate this paper, and so were not built with our framework explicitly spelled out, although they were designed with the framework's key features in mind.

4.1 Distributed Data Structures

In [10], we present a scalable, available cluster-based storage layer for Internet services. The system exposes conventional data structure APIs (such as hash tables and trees) to service authors, but replicates the state of the data structures throughout the cluster for the sake of scalability and availability. All persistent state is managed by software components called *bricks*; applications run in the same cluster as the bricks, but communicate with them using distributed data structure (DDS) libraries. These libraries handle the distribution, replication, and recovery of the data, shielding services from this complexity. The DDS bricks were designed using a hybrid thread/event framework tailored for the high concurrency and throughput required by Internet services. Figure 10(a) illustrates the bricks' design. The DDS layer is implemented in Java using Linux as the cluster operating system.

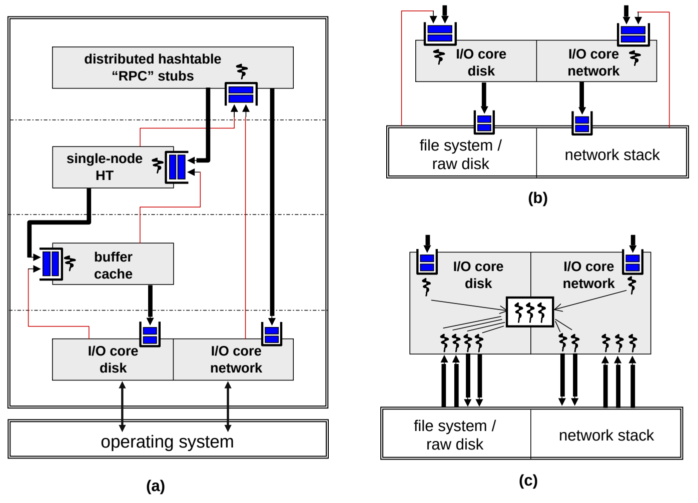

Figure 10: *Distributed hash tables:* (a) illustrates the structure of the distributed hash table "brick" process; the thick arrows represent request tasks, and the thin arrows represent completion tasks. (b) shows the ideal operating system and I/O core implementation, in which the OS exports a non-blocking interface. (c) shows our current implementation, in which blocking the I/O interfaces exposed by the Java class libraries force us to dispatch threads to handle I/O requests from a fixed-sized thread pool. There are also dedicated threads that block listening on incoming sockets.

A brick is a single process composed of many layers, each of which is separated by a thread boundary. References to tasks are placed on queues that bridge these boundaries. The bottom layer of a brick is an abstraction library called the *I/O core*. The library provides non-blocking interfaces to both disk and network I/O, as well as implementations of queues, events, and thread pools. A buffer cache layer sits on top of the disk interface, which is used to implement a single-node hash table layer. We expose the single-node hash table to the rest of the cluster with an asynchronous RPC layer using the networking interfaces of the I/O core.

The buffer cache, the single-node hash table, and the asynchronous RPC layer each have a single, dedicated thread that waits for tasks on that layer's task queue. When a task arrives in a given layer, the task's state is retrieved and the task is passed to an event handler within that layer. No data locks are necessary due to the single-threaded design of these layers, although one implication is that event handlers can never block.

The state for each distributed data structure is horizontally partitioned. For the sake of availability, partitions are replicated across at least two cluster nodes. Replicas are kept consistent with two-phase commits, yielding one-copy equivalence.

An ideal implementation of the I/O core layer is shown in Figure 10(b). Here, the operating system exposes non-blocking interfaces to the disk and network, allowing the I/O core to be implemented using a single-threaded state machine. Unfortunately, Java's class libraries do not expose non-blocking I/O interfaces, and the version of Linux used in our cluster does not provide non-blocking file system interfaces. Therefore, the I/O core uses a thread pool (as shown in Figure 10(c)) to provide non-blocking interfaces to the high layers. For each task placed in the I/O core's queue, a thread is dispatched from the pool to issue the blocking I/O request. The thread pool is made large enough to handle the width of the disk and of the inter-cluster network I/O. In this case the disk width is approximately 30 threads, and the network width is 2 times the number of cluster nodes (two threads are dedicated per network connection: one for reading and one for writing). Note that as the number of cluster nodes grows, the number of threads required by the network layer increases.

The DDS is an example of a complex, hybrid thread and event system that fits into our framework. Each layer of the DDS brick is an event-driven task handler with a single thread. The lowest layer of the DDS uses a thread pool to gain concurrency from blocking I/O interfaces; this is an example of the *Wrap* pattern. The *Pipeline* pattern is used to separate the higher layers of the DDS, and the *Replicate* pattern is used to replicate bricks across multiple cluster nodes. The fact that there is a single thread pool serving both disk and network I/O exemplifies the *Combine* pattern.

4.2 vSpace

vSpace is a platform for scalable, flexible Internet services.[1] vSpace uses workstation clusters and the Java programming language to provide a high-level programming model which hides most of the details of Internet service construction from application developers. vSpace borrows many of the ideas embodied in MultiSpace [11] and TACC [7], but goes a step further by providing data persistence (through a Distributed Data Structure layer) and an event-driven execution model for high concurrency.

In vSpace, applications are constructed as a set of *workers* which run across the nodes of a workstation cluster. Workers operate by processing *tasks*, each of which generally consists of a small amount of data. Each task has an associated *task type* which specifies its contents; workers register with the system their ability to handle one or more task types. For example, an HTTP protocol worker might handle tasks of type *IncomingHttpRequest* while a disk-storage worker might handle types *DiskWriteRequest* and *DiskReadRequest*. In addition, workers can dispatch outgoing tasks for other workers to handle.

The vSpace model maps very neatly onto our design framework. vSpace workers correspond directly to task handlers, and queues are implemented through the task dispatch operation in vSpace, which can cause tasks to flow either to workers on the same node or across the cluster network to other nodes. Each vSpace worker has a thread pool of size one; running multiple workers on a single node is used to achieve high concurrency.

In vSpace, the *Pipeline* pattern is used to decompose applications into workers, and *Replicate* to achieve scalability and fault tolerance across nodes. *Wrap* is used to introduce a single logical task queue for the workers each node. Because all vSpace workers use soft state, except that managed by the Distributed Data Structure layer, the failure of a cluster node does not affect other nodes, and the workers on that node can be restarted elsewhere. Load balancing is accomplished by interposing on the task dispatch operation: vSpace determines the most suitable cluster node, and worker on that node, to handle each new task. This determination is based on load information periodically gathered from cluster nodes.

4.3 Flash and Harvest

Our framework can be used to analyze other successful concurrent system designs. The Flash web server [23] and the Harvest web cache [4] are based on an asynchronous, event-driven model very much like that proposed here. Although each was conceived for a specific application, they can be mapped onto our design framework. In Flash, each component of the web server responds to particular types of events, such

[1]vSpace is still in the early stages of development, and as of yet no concrete performance results are available.

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.[2]

Flash and Harvest's main thread and helper processes can be considered task handlers in our framework as would be produced by applying the *Pipeline* pattern. Queues are implemented using UNIX IPC mechanisms (e.g., a pipe). Each of the task handlers has a thread pool of size one, and multiple helper processes are created (using the *Replicate* pattern) to increase throughput. Our framework suggests improvements on the Flash and Harvest designs: *Pipeline* could be used to further decompose event processing stages to increase bandwidth, and *Replicate* could be applied to produce a clustered version of each application.

5 Related Work

Several other programming models have been proposed for building scalable, highly concurrent systems. None of these systems, however, attempt to cover the wide range of requirements addressed by our framework.

Application servers, including BEA WebLogic [3], ObjectSpace Voyager [22], and IBM WebSphere [15] aim to support scalable, concurrent applications using industry-standard programming interfaces, such as Enterprise Java Beans [25] and Java Servlets [16]. Generally, application servers are used as a "middleware" component between a presentation system (such as a web server) and a back-end database or mainframe. While these systems provide a variety of application programming interfaces, their internal structure is generally thread-based. Threads, network connections, and database connections are usually pooled to limit resource consumption on the server; replication is used across several servers to provide scalability and fault isolation. Because these systems tend to be implemented on standard Java Virtual Machines and operating systems, they have little control over low-level I/O and concurrency mechanisms. As such, these systems require overprovisioning of resources to handle load bursts.

The structure of our framework was influenced greatly by TACC [7] and SNS [5], which propose scalable service models based on collections of *workers* replicated across a cluster. Although the structural components of task handlers and queues are evident in these systems, they did not adopt an event-driven approach to achieving robust high concurrency. Rather, they rely upon threads, and obtain scalability by growing the number of cluster nodes.

The JAWS web server [13] combines an event-driven concurrency mechanism with a high-level programming construct, the *Proactor pattern* [14], which is intended to simplify the development of highly concurrent, event-driven applications. While JAWS does not directly address fault isolation or clustering, the use of the Proactor pattern to describe the event-driven programming model is a useful tool which could be applied to the task handlers in our framework.

Kaashoek *et al.* [17] propose specializing operating system architectures for server applications, using Internet services as a specific example. While this work focuses on low-level aspects of O/S performance for servers (such as disk and network access overhead), it also realizes the benefit of an event-driven concurrency model. Application-Specific Handlers [29] are used to install application-level event handlers in the kernel for added performance. This approach complements our design framework by providing novel kernel-level functionality to improve I/O performance.

Other work has looked at improving threaded and event-driven programming models. Banga *et al.* [2] suggest changes to the UNIX system call interface to better support event-driven applications, and scheduler activations [1] can be used to reduce the cost of kernel-based thread implementations.

The Click modular packet router [19] 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, so threads can cross module boundaries. Click is a domain-specific system for obtaining high concurrency in a packet router, and as such is less general and lower-

[2]Interestingly, Harvest's asynchronous disk read mechanism relied on the use of the *select()* call in UNIX, which under most implementations (including Solaris, Linux, and FreeBSD) returns immediately regardless of whether or not the requested data is in fact in memory. Therefore, the application thread could block waiting for data to be read from disk even when non-blocking file descriptors are used. Squid [21] corrected this bug by using a separate thread pool to issue disk I/O requests.

level than the framework presented here.

Click uses both *push* and *pull* semantics for flow control; that is, packet processing modules can send data downstream or request that data be pushed upstream to it. The rationale for pull processing is that push semantics require packets to be queued up before stages which are not ready to process a packet (for example, when a network port is busy). The push/pull distinction is important when threads can cross module boundaries, as is the case in Click. Our framework always imposes a queue between modules, so threads push data downstream (to other queues) and pull data upstream (from their incoming queue). Click could be implemented using our framework, by creating a thread pool boundary (using the *Wrap* pattern) where a queue exists between two Click modules. Click modules that communicate through function calls would operate within a single task handler using our framework.

6 Future Work and Conclusions

We believe that our design framework will enable the construction of a new class of highly concurrent applications. We are in the process of building several systems based upon this design. These include novel Internet services [30, 8], a new segment-based database storage manager [12], and a secure, consistent, highly available global filesystem [26]. All of these applications share the concurrency and availability requirements targeted by our framework.

In addition, we continue to explore the design tradeoffs within this framework. Many approaches to load-balancing and resource management within a cluster have yet to be investigated in depth. In particular, we are interested in using economic models to manage arbitration for "competing" applications sharing the same physical resources [24, 28].

Building event-driven systems could be supported by better high-level language features for managing state, consistency, and scheduling. While much of our current work is based in Java [9], new language abstractions supporting our framework reveal several avenues for future research.

Our goal has been to map out the design space of highly concurrent systems, and to present a framework which provides a way to reason about their performance, fault isolation, and software engineering characteristics. Our framework is based on three simple components — *tasks*, *queues*, and *thread pools* — which capture the benefits of both threaded and event-driven concurrency models. We have presented a set of implementation guidelines, expressed as a small number of design patterns, which allow application designers to directly obtain the benefit of these framework components.

The demands of highly concurrent systems, such as Internet services, present new challenges for operating system and language designers. High concurrency, high throughput, burstiness, and fault-tolerance are all required for these applications, yet existing systems generally fail to provide all of these features in a form which is "ready to use" by application developers. Our design framework presents a straightforward programming model which allows these applications to be constructed using existing techniques (such as threads, non-blocking I/O, and clustering); we suggest that future O/S and language designs should support our framework as a first-class programming schema.

References

[1] 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.

[2] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999.

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

[4] 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.

[5] Yatin Chawathe and Eric Brewer. System support for scalable and fault tolerant internet services. In *Proceedings of Middleware '98*, September 1998.

[6] Inktomi Corporation. Inktomi search engine. `http://www.inktomi.com/products/portal/search/`.

[7] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-Based Scalable Network Services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, October 1997.

[8] Ian Goldberg, Steven D. Gribble, David Wagner, and Eric A. Brewer. The Ninja Jukebox. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, CO, October 1999.

[9] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, Reading, MA, 1996.

[10] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, Distributed Data Structures for Internet Service Construction. Submitted to the Fourth Symposium on Operating System Design and Implementation (OSDI 2000), October 2000.

[11] Steven D. Gribble, Matt Welsh, David Culler, and Eric Brewer. MultiSpace: An evolutionary platform for infrastructural services. In *Proceedings of the 16th USENIX Annual Technical Conference*, Monterey, California, 1999.

[12] Joe Hellerstein, Eric Brewer, and Mike Franklin. Telegraph: A Universal System for Information. `http://db.cs.berkeley.edu/telegraph/`.

[13] James C. Hu, Irfan Pyarali, and Douglas C. Schmidt. High Performance Web Servers on Windows NT: Design and Performance. In *Proceedings of the USENIX Windows NT Workshop 1997*, August 1997.

[14] James C. Hu, Irfan Pyarali, and Douglas 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.

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

[16] Sun Microsystems Inc. Java Servlet API. `http://java.sun.com/products/servlet/index.html`.

[17] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, and Deborah A. Wallach. Server Operating Systems. In *Proceedings of the 1996 SIGOPS European Workshop*, September 1996.

[18] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 94 Usenix Conference*, pages 115–131, January 1994.

[19] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans 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.

[20] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Proceedings of the Fourteenth International Conference on Distributed COmputing Systems*, pages 56–65, Poznan, Poland, June 1994.

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

[22] ObjectSpace Inc. ObjectSpace Voyager. `http://www.objectspace.com/Products/voyager1.htm`.

[23] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the 1999 Annual Usenix Technical Conference*, June 1999.

[24] M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, A. Sah, and C. Staelin. An economic paradigm for query processing and data migration in Mariposa. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems*, September 1994.

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

[26] UC Berkeley OceanStore Project. `http://oceanstore.cs.berkeley.edu`.

[27] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

[28] C. A. Waldspruger, T. Hogg, B. A. Huberman, J. O. Kephart, and S. Stornetta. Spawn: A Distributed Computational Economy. *IEEE Transactions on Software Engineering*, 18(2):103–177, February 1992.

[29] Deborah A. Wallach, Dawson R. Engler, and M. Frans 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.

[30] Matt Welsh, Nikita Borisov, Jason Hill, Rob von Behren, and Alec Woo. Querying large collections of music for similarity. Technical Report UCB/CSD-00-1096, U.C. Berkeley Computer Science Division, November 1999.