USENIX Association

Proceedings of the 2002 USENIX Annual Technical Conference

Monterey, California, USA
June 10-15, 2002

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Ninja: A Framework for Network Services

J. Robert von Behren, Eric A. Brewer, Nikita Borisov, Michael Chen, Matt Welsh
Josh MacDonald, Jeremy Lau, Steve Gribble* and David Culler
University of California at Berkeley

Ninja is a new framework that makes it easy to create robust scalable Internet services. We introduce a new programming model based on the natural parallelism of large-scale services, and show how to implement the model. The first key aspect of the model is intelligent connection management, which enables high availability, load balancing, graceful degradation and online evolution. The second key aspect is support for shared persistent state that is automatically partitioned for scalability and replicated for fault tolerance. We discuss two versions of shared state, a cluster-based hash table with transparent replication and novel features that reduce lock contention, and a cluster-based file system that provides local transactions and cluster-wide namespaces and replication. Using several applications we show that the framework enables the creation of scalable, highly available services with persistent data, with very little application code — as little as one-tenth the code size of comparable stand-alone applications.

1 Introduction

The Ninja Project is focused on Internet infrastructure and the need for a better way to create, maintain and operate robust giant-scale distributed systems. Although the overall project [GWv+01] addresses wide-area systems, in this paper we study building robust large-scale centralized network services. Thus we focus on clusters within a single administrative domain that act as a centralized server for many users and potentially many services. The primary goal is to deal in full with the word "robust", which includes basic problems of scalability, availability, fault tolerance, and persistence.

Network services include almost all aspects of large web sites, including many non-HTTP services, such as instant-messaging, e-mail and the central-server aspects of peer-to-peer file sharing. These services have a form of *natural parallelism* that derives from supporting millions of independent users; we thus define scalability, concurrency and high availability in terms of users or requests. The basic unit of work is thus a query or connection (depending on the service) from a specific user.

We believe that the framework presented here is the right way to build these services: both that the programming model is the right way to think about the service, and that the mechanisms we use greatly simplify service authoring. In some sense, this framework is our fourth version over a period of five years (starting with [FGCB97]) and therefore represents considerable refinement of both the model and mechanisms. Unfortunately, it is nearly impossible to prove that a framework is "right" — instead we focus on describing the principles and invariants provided by the framework and why they simplify service authoring, and we examine the code size of several representative services and show that they are remarkably small given that they are scalable, highly available and persistent in the presence of faults.

We explicitly do not look at those parts of a site built on top of a database management system (DBMS) for several reasons. First, there is much work in industry on this topic and several products that work well. Second, our work is complementary to database research and would be easy to integrate with a DBMS by using Ninja as an "application server". Third, we tend to focus on high availability, rather than transactions, and support a wider range of semantics than ACID [GR97]. However, we do look at persistence, replication, atomicity, and consistency, and many things done with a database are perhaps better done directly in Ninja (see Section 6).

The requirements for network services are very demanding. By "robust" we mean all of the following:

Scalability: the ability to support 100M users.

High Availability: the ability to answer queries nearly all of the time. Ninja services should be able to reach 4 or 5 nines, that is, the probability of answering a query should be above 0.9999 (when desired). High availability means that most queries succeed and that if a query fails, retrying it has a high probability of success: ideally, retries should be independent trials. This differs from the harder goal of "fault tolerance" in which a query must complete correctly without a client-visible retry.

Persistent Data: Like high availability, this is a specific form of fault tolerance: that data survives faults. This requires replication, and much of the framework will deal with automating replication for availability and persistence. There is often, but not always, an implied sub-goal of *consistency* for the data. We support a range of performance and consistency tradeoffs, with the default being linearizability [HW87].

Graceful Degradation: We cannot assume that there will be sufficient resources to always handle the offered load. Instead, we aim for graceful

*: Now at the University of Washington
This work was supported in part by DARPA #DABT 63-98-C-0038.

degradation through admission control and prioritization of requests. We aim to achieve the maximum throughput even when overloaded.

Online Evolution: A variation of high availability, online evolution is the ability to upgrade the service in place without significant downtime. In most cases, we can upgrade a service without downtime.

One primary goal is to make achieving these properties *easy* for service authors. We have developed several example applications that exhibit robustness; we judge ease of authorship primarily by code size. To achieve ease of authorship, we follow employ three principles:

Exploiting Clusters: In a data-center environment, we can make many assumptions that are not true in general for distributed systems. These include a reliable source of power, temperature control, physical security, 24-hour monitoring, and a partition-free internal network.

Programming Model: We believe that a fully general programming model makes it impossible to provide robust services. Instead, we use namespaces and narrow interfaces to control the sharing, replication, and persistence of data, which means that we do not have to provide these properties for all data at all times. Second, we forsake general multi-threaded concurrency for a specific style that matches the natural parallelism. We thus focus on inter-task parallelism rather than intra-task parallelism, although we support asynchronous I/O. We show that this model is sufficiently expressive to write a wide variety of services.

Hide Complexity: We share with DBMS research the goal of hiding the complex details of replication, persistence, load balancing and fault tolerance from applications. However, we do so through the use of reusable data structures and libraries rather than via an abstract data model and a declarative language (SQL). Both approaches enable strong properties with relatively little application code, but our approach fits more naturally with applications written in imperative languages such as C or Java.

We define the programming model in Section 2, and our key mechanisms in Section 3. Section 4 describes the applications and Section 5 presents their evaluation. Section 6 discusses our principles and related and future work, and Section 7 provides a summary.

2 Programming Model

The goal of the programming model is to simplify the creation of complex network services; such services must map naturally onto the programming model. Second, the model must enable an underlying implementation that hides the details of fault and load management, scalability, high availability, and online evolution.

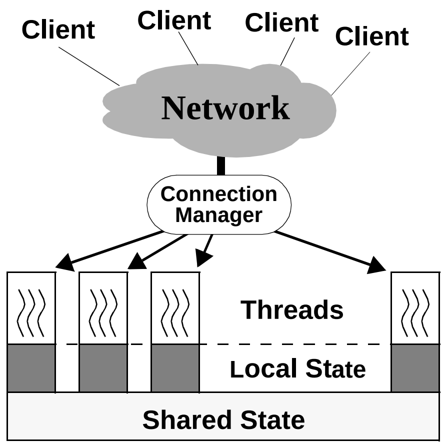

Figure 1: The Programming Model
A node consists of threads, local state and shared state. Nodes use the same “program” (code base), and receive connections from the Connection Manager in the style of data parallelism.

Given the natural parallelism described above, we choose a model based on request parallelism, in which we aim to partition users’ request streams across nodes. For ease of authoring, we would like to have a single program that is automatically spread across the cluster. Thus we choose to base our model on the *single-program-multiple-data (SPMD)* model commonly used in parallel computing [DGNP88]. We make two extensions to the SPMD model: support for shared state[1] and management of connections to the outside world. We refer to this new model as the *single-program-multiple-connection (SPMC)* model, as shown in Figure 1. We also assume many threads per node, which differs from SPMD in practice, but not in definition. There are expected to be many connections per node, and there may be more or fewer threads than connections. One big practical difference of course is that we seek to achieve high availability and tolerance for partial failures, whereas SPMD was developed in the context of the all-or-nothing fault models of parallel machines.

By “shared state” we mean that the threads and connections active on any subset of nodes may share global namespaces that support linearizable updates (i.e. strongly consistent, see [HW87]) to network-accessible storage in a uniform manner across the cluster. This notion does not require shared memory, as assumed in the original SPMD work; instead we provide multiple global namespaces accessed via method calls rather than load/store instructions. This narrow interface to shared

1: Although many SPMD systems had at least a shared namespace (e.g. CM-5 [HT93] and T3E [Sco96]), support was inconsistent and we thus treat this as an extension.

state simplifies consistency, replication and persistence.

Conversely, we define non-shared state as "local state", which includes local memory and files. Local state is simpler and faster to access than shared state, and is useful for session and thread state (e.g., stacks), caching shared state, and temporary files.

Invariant: Shared state is strongly consistent across all nodes used by a service.

Globally named shared state implies that any connection can be serviced from any node, which is a tremendous simplification. Shared state enables simple construction of groupware services, communication services (e.g. chat), and information dispersion (e.g. stock quotes); it also simplifies service management. For example, it is easy to count users, put them in (shared) queues, and track aggregate statistics about the service.

Shared state can also be highly available and persistent (up to some configurable number of faults):

Invariant: Shared state is highly available and persistent (when desired).

High availability requires automatic replication, while durability requires management of replicas and disk writes. The power and simplicity of the SPMC model come from the automatic management of consistent, highly available, persistent shared state. Finally, we allow services to relax these invariants for better performance (Sections 3.4.1 and 5.5).

We support multiple independent namespaces for shared data. This provides both encapsulation and logical isolation of different services and system components, thus providing a simple form of security: services cannot read or write the shared state of other services or of shared system components. Additionally, support for many namespaces enables fine-grain control over consistency, availability (replication), and persistence, all of which are attributes of a namespace.

We have found two kinds of shared state to be particularly useful: shared data structures and shared files:

Shared Data Structures: These have the same interface as normal data structures and are therefore very easy to use. We provide hash tables and B-trees. Hash tables are sufficient to support other models including tuple spaces and shared arrays. We also provide extensible atomic operations that enable programmers to create high-concurrency sharing primitives, such as compare-and-swap.

Shared Files: Although we can store large items persistently in the hash table, we found that file usage is sufficiently different and common to support directly. Some of the key differences include larger objects, larger working set sizes, lower expectation of being in memory, and the need for data streaming over the network.

The second extension in SPMC is explicit support for connection management. The SPMD model does not define how outside I/O interacts with the nodes, except for possibly spreading files across the nodes. For network services the problem is much more dynamic: some state is long lived, and we must isolate down nodes from the clients to provide high availability.

Invariant: New or retried connections arrive at "up" nodes.

Note that we do not promise that connections do not go down: existing connections are lost when a node goes down. Although possible in theory, moving active connections when the server side dies is not practical. For example, every potentially client visible state change must be durable, which requires tracking those changes to either a replica or a persistent store, as they occur. Instead, we promise that retried connections are not affected by the failure. Using shared state, it is possible to keep session state across this transition as needed, which is simpler and much more tractable than tracking all client-visible state automatically.

To enable more intelligent connection management, we add one key idea to the programming model: connections are partitioned into application-defined classes, which we call *partitions*. By default a service has only one class, in which case all connections are treated the same, but in practice explicit partitioning gives the author more control over the service. In particular, a partition is the:

Unit of Affinity: Connections in the same partition go to the same node(s), which enables cache affinity (similar to LARD [PAB+98]), and reduces communication for users within the partition. For example, if all users in a chat room are in the same partition, then the group state resides on that node.[2]

Unit of Priority: Partitions allow the author to control graceful degradation and quality of service. In particular, we can support application-defined admission control, by dropping connections in low-priority partitions first. The same idea enables differentiated quality of service by partition: we can support different densities of users/node for high- and low-priority partitions. For example, high-paying stock traders might have less congestion and thus faster trades, especially during overload.

2: Some communication is still required if the chat state is replicated, but typically chat rooms are neither persistent nor highly available; the application code would be almost identical regardless.

Unit of Migration: Under a load imbalance or a fault, it may be necessary to migrate users' state to a new node. Partitions are the unit of migration for fault recovery and load balancing. This ability also simplifies online evolution, as we can do a rolling upgrade by partition.

In general, explicit partitions are powerful because we get simple application-level guidance on how to group connections. We can then use these groups to provide fine-grain control over replication, cache affinity, quality of service, graceful degradation and online evolution. Note that we partition connections and not users; we can use them to partition users or we can have the same user in different partitions simultaneously depending on the task. Finally, service authors can ignore partitions if they need only even load balancing.

3 Mechanisms

In this section we examine the four key building blocks that we use to achieve the SPMC model.

3.1 Clone Groups

The first mechanism is to virtualize the SPMC model: instead of each service running on a whole cluster, we instead run services on *clone groups*, which are a set of *clones* with common code and state. A clone is a virtual node that we map onto a real node dynamically; we refer to them as clones because they share the same code base (the "single program") and shared state. Thus when we discuss shared state or namespaces, it is always for a specific clone group. Similarly, connections are managed across clone groups, not the cluster.

Principle: Clone groups provide each service with a virtual cluster.

Clone groups typically map onto a subset of the real nodes, and may vary in size depending on load. More than one clone group may map onto a node, in which case they are isolated in terms of state and namespaces, but not in performance. However, the connection manager described below can maintain even load balancing even if clones have uneven throughput due to differences in hardware, work per connection, or interference from other groups.

Clone groups provide several useful mechanisms to the programmer, including membership, broadcast and barrier synchronization. Changes in membership lead to notification of all clones via birth and death events. Membership is approximate and eventually consistent, which has proven sufficient in practice. For example, we use death notification to instigate recovery within the shared data structures.

Broadcast is mostly useful for notification, since there is a better mechanism for sharing state. Barriers could be implemented on top of shared state, but are actually done via message passing (i.e. events) because of the need to integrate dynamic membership information. A barrier is considered done when all live nodes reach the barrier, so death events may complete a barrier. As with SPMD, barriers are used to ensure that all clones are in the same stage; the biggest use seems to be to denote the completion of an initialization phase.

An overall manager, called the "shogun", dynamically modifies the size of each service's virtual cluster based on utilization. Remarkably, most services don't care about the size of the cluster, since the shared state is managed across the transition automatically, and no connections are lost during the transition (see Section 5.4). A service can track changes using the birth/death events when needed.

Typically, replication uses subsets of a clone group of storage nodes. A *replica group* is thus a subset of a clone group that handles replication for part of the shared data, so that we can decouple the degree of replication from the number of clones. Replicas use the clone-group mechanisms to handle replica membership and synchronization. We use many small replica groups in one storage clone group, with overlapping membership. For example, with 2-way replication, a replica group is a two-node subset of a larger storage clone group. The use of lots of small groups reduces the recovery latency per group, and enables incremental recovery, where each small group is one step. By design, the groups are small enough that we can just copy the whole contents of another replica atomically, without too much concern for the fact that we prevent updates to that group (only) during the copy.

3.2 Single-node Run-Time System: SEDA

An important aspect of building scalable services is to support very high concurrency and to avoid overcommitment of server resources. Building highly concurrent systems is inherently difficult: structuring code to achieve high throughput is not well-supported by existing programming models, and traditional concurrency mechanisms, particularly threads, make it difficult for applications to exercise control over their resource usage.

Ninja makes use of a concurrency design called SEDA, or *staged event-driven architecture*. Services are structured as a set of *stages* connected by explicit *event queues*. This design permits each stage to be individually conditioned to load (e.g., by performing thresholding on its incoming event queue), and facilitates modular application construction. SEDA, covered in detail in [WCB01], enables not only very high concurrency, but also graceful degradation through resource management and adaptive load shedding.

For the purposes of this paper, SEDA provides two key capabilities: support for more connections/node and thus better overall performance, and detection of overload at the node level, which we need to provide graceful degradation for the overall service.

The scalability limits of threads are well-known and have been studied in several contexts, including Internet services [PDZ99] and parallel computing [RV89]. Generally, as the number of threads grows, OS overhead (scheduling and aggregate memory footprint) increases, which leads to a decrease in overall performance. Direct use of threads presents several other correctness and tuning challenges, including race conditions and lock contention.

Principle: *Concurrency is implicit in the programming model; threads are managed by the runtime system.*

Since we must avoid excess threads to achieve graceful degradation, we simply prevent services from creating threads directly. Instead, services only define *what could be* concurrent, via (explicitly) concurrent stages. Conceptually, each stage has a dedicated but bounded thread pool, but the allocation and scheduling of threads is handled by SEDA. Thus the system as a whole is event-driven, but 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 stage's thread pool must be balanced between obtaining sufficient concurrency and limiting the total number of threads; SEDA uses a feedback loop to manage thread pools automatically. The particular policies are beyond the scope of this paper, as they only effect the node performance. Roughly, allocation is based on effective use of threads (non idle) and priorities, while scheduling is based on queue size and tries to batch tasks for better locality and amortization (as in [Lar00]).

Internal framework modules, such as the shared-state mechanisms in Section 3.4, also use stages and avoid explicit thread creation. The internal modules are often written in the event-driven style, common for high-performance servers [PDZ99], which we enable by providing non-blocking interfaces for all network and disk activity, and for the shared data structures.

Invariant: *Overload detection is automatic; services are notified when they are overloaded.*

A key property of queues is that it becomes possible to implement backpressure by thresholding the event queue for a stage. We use this to detect overloaded stages and thus to initiate *overload mode* and graceful degradation. With only the implicit queues of blocked threads, it is difficult to detect overload until too late.

Thus our single-node runtime system provides two key capabilities. First, it provides control over thread allocation and scheduling, which enables either thread-based or event-driven programming and ensures thread limits consistent with the operating range of the node. Second, it provides backpressure via explicit queues that enables the detection of overload and thus graceful degradation, which is shown in Section 5.3.

3.3 Connection Manager (CM)

The Connection Manager (CM) is responsible for all external names. It dynamically maps external names to clone groups and connections to an external name to a specific clone. It must hide failed nodes and balance load across the clone group. Although it appears in the figure as a single point of failure, it is actually a pair of "layer 7" switches [Fou01] that provide automatic failover for each other. Based on ethernet switch reliability, we estimate the uptime of these switches at about $1\text{-}10^{-7}$ each (seven 9's), so the pair is extremely reliable.

As an optimization, services can define partitions that are subgroups of names (and thus connections), to provide fine-grain control over resource allocation and graceful degradation. The CM can map partitions to subsets of clones in a clone group, or in the case of admission control deny partitions altogether.

3.3.1 External Names

The connection manager provides a level of indirection for external service names. The CM maps external names to clone groups, which may change dynamically, and load balances connections among the clones. In general, the CM maps external (IP, port) pairs to the set of internal pairs corresponding to the clone group. When there are multiple clones, connections are balanced across the target set based on open connections.

The CM tracks clone birth and death events in order to maintain high availability. Starting a clone is a two-step process. During initialization, Ninja allocates server sockets for a clone and starts it. It then registers the clone with the CM, which starts to forward connections to the clone. Stopping a clone is the reverse process: the CM stops forwarding connections to the clone and then removes it from the clone group. To provide higher availability, the clone may finish processing outstanding requests before it exits.

Invariant: *Ninja can remap or resize clone groups without dropping connections.*

The ability to shutdown clones gracefully makes it possible for Ninja to remap clones to nodes dynamically or to reduce the size of an underutilized clone group. The same ability enables online evolution to a new version with no downtime (shown in Section 5.4).

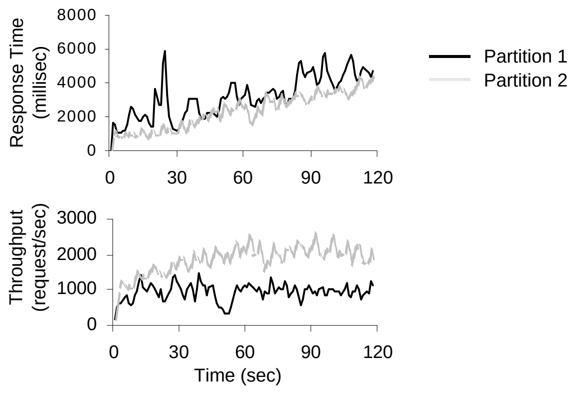

Figure 2: Delivering different Qualities of Service
A three-node web server is divided into two partitions: Partition 1 maps onto one node, Partition 2 onto two. The CM achieves twice the asymptotic throughput and better response time for the larger partition, and both tiers show graceful degradation.

3.3.2 Partitions

In the current implementation, services can define partitions in two ways, either via ports or URL string matching. With ports, services map external port numbers to partitions, which are then dynamically mapped to clones. Typically, services would use one external port number per partition, although more are allowed. For HTTP requests, the service can define partitions based on URL hashing and string matching; we currently support prefix, suffix, and substring matching. This can be done at wire speed using current "layer 7" switches [Fou01]. Given these partitions, the CM will dynamically map partitions to clones.

Principle: Partitions enable division of the working set for higher throughput.

As in LARD [PAB+98], partitions provide better locality and cache performance as the working set is partitioned across the clone group. Without partitions, the CM spreads load evenly, which effectively replicates the working set at each clone.

Partitions are also the unit of priority which helps with tiered quality of service and graceful degradation:

Principle: Partitions enable tiered quality of service.

First, the CM enables differential qualify of service by allocating varying resources to different partitions: partitions need not map evenly onto clones. Figure 2 shows this proactive form of uneven load balancing. Partition 1 maps to one clone, while Partition 2 maps to two: the latter has twice the asymptotic throughput and better latency, particularly as Partition 1 reaches its overload point (about time 20). The variance of the one-node partition is higher as well, due to averaging effects. Note that both graphs show graceful degradation, with smooth asymptotes and linearly increasing latency.

Second, priorities enable more graceful degradation during overload. The CM implements an admission-control policy based on partition priorities: requests to low-priority partitions are dropped first. Individual nodes can detect overload directly (Section 3.2) and notify the CM. In overload mode, the CM drops requests by routing them to a generic "drop" clone, analogous to the use of /dev/null in UNIX. This is complementary with the first strategy and they may be used together. In addition to admission control, nodes may take action themselves in overload mode to reduce the average work per request. We evaluate these mechanisms in Section 5.3.

There is also a relationship between failures and overload: when a clone fails, the remaining clones typically receive increased load, which may put them into overload mode. The movement of load due to failures and the reaction to overload are independent mechanisms, but both are automated and overload mode will kick in only if needed.

Finally, it is important to realize that partitions to do not effect shared state or replication. In the graph above for example, all three clones have the same program and shared state, but the CM allocates traffic unevenly by partition. This means that any clone *can* handle any partition, although they do not in normal operation. If a node fails, the remaining two nodes are given both partitions automatically, which affects the difference in quality but maintains high availability.

To summarize, the connection manager provides management of all external names, including dynamic mapping of names to physical nodes for load balancing and fault tolerance. It also implements policies based on partitions that allow a service to define relative quality of service and prioritized admission control.

3.4 Shared State

The fourth mechanism provides services with shared state: currently shared hash tables, B-trees, and file systems. The shared state mechanisms need to support robust applications, and therefore must be scalable, highly available, durable and consistent. By implementing these properties in the shared state mechanisms, we essentially eliminate the burden of achieving them. In particular, we hide all of the issues of atomicity, replication, consistency and recovery from service authors.

Because high availability and consistency are incompatible in the presence of partitions [FB99], we opt to use a redundant system-area network for communication within our cluster (currently gigabit ethernet with redundant switches). A partition-free network allows us to use a two-phase commit protocol (2PC) [GR97] to ensure consistency and atomicity of updates to shared state across several nodes. The version of 2PC we use is optimized for high availability in two ways. First, if a member of the protocol dies in the second phase, the 2PC completes without it, because the replica will be able to recover a consistent image of its state from its peers later. Second, if the coordinator fails, we cannot afford to wait until it recovers to complete the protocol; instead, the replicas contact each other proactively after a timeout and commit the action if any member received a commit; otherwise, they all abort. The protocol is also available to application writers to extend the framework with additional shared state mechanisms.

In the next two sections, we examine the cluster hash table and file system in more detail. The cluster B-tree is ongoing work.

3.4.1 Cluster Hash Table (CHT)

Our prototypical shared data structure is the cluster hash table, which uses the traditional interface of three operations: get, put, and remove. Each operation is atomic with strong consistency (equivalent to having a single copy). The underlying data is partitioned across the cluster for scalability, and replicated for high availability (see [GBH+00] for more details). The degree of replication can be varied based on the requirements of the application, and different tables in the same service may use different replication strategies. This control enables tradeoffs among performance, fault tolerance and storage requirements, and also enables the composition of modules without name or policy collisions.

3.4.1.1 Non-Blocking Synchronization

The atomic put operation on the CHT returns the old value prior to the update, in essence implementing an atomic swap. Atomic swap can be used to implement various synchronization primitives, such as locks (using test-and-set) or read-modify-write (swapping in a "locked" value first and then the updated value). Such implementations, however, can be classified as *blocking* [Her91], in that a process holding a lock may take an arbitrary time to complete. This reduces both scalability, due to lock contention, and availability, since a process may die while holding the lock.

To overcome these obstacles, we extended the hash table interface with an *apply* operation, which implements an atomic read-modify-write:

```
apply (key, update_function) {
    temp = get(key)
    put(key, update_function(temp))
    return temp
}
```

The apply operation is implemented by shipping the name of the update function to the nodes that store the data and executing it there, analogous to function shipping in databases. We can use the name rather than the code, because of the "single program" facet of the SPMC model. Atomicity is ensured, as before, by the 2PC protocol. Read-modify-write is sufficiently general to implement a wide range of atomic primitives, such as compare-and-swap, fetch-and-add, etc. Several of these primitives, such as compare-and-swap, are *universal* [Her91], and thus can be used to build non-blocking and wait-free implementations of a data structure from a sequential one.

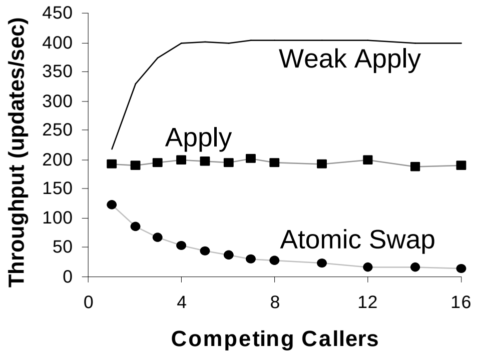

Figure 3: Optimizations for Update Contention
This graph plots single-node throughput under heavy contention in the CHT (without replication). Atomic Swap drops off due to long lock-holding times, while Weak Apply performs twice as well as apply, by not holding locks across the 2PC round trip.

However, we can build non-blocking data structures directly: unlike conventional shared-memory systems, each location in a hash table stores an entire object, as opposed to just a pointer or a primitive value. This allows us to provide the update function from a sequential implementation as the argument to the apply function. The update is atomic and non-blocking.[3] Therefore, the operation is naturally wait-free, without the complexity or overhead usually associated with wait-free protocols.

The improved scalability of apply-based updates can be seen in Figure 3. We compare an update implemented using two atomic swaps to one implemented by an apply operation; the graph shows the aggregate throughput of several clients continuously updating the same data value. The atomic swap implementation performance quickly degrades as concurrency increases, since more time is spent trying to obtain the lock. The apply-based implementation performs better at the outset, since it requires half as many operations to complete

3: This is not strictly true (nor could it be) for an arbitrary update function; however, we assume simple non-blocking update functions, which holds in actual use. Our most complex update function appends to a list represented as an array and sometimes has to resize the array.

an update, and the aggregate throughput remains virtually flat as we scale up to 16 nodes.

The graph also shows performance of a "weak" version of the apply operation; this version has exactly the same interface, but weaker consistency semantics. Namely, it commits the update in the first phase of the 2PC; the second phase is only to let replicas know that everyone made the update (in case the coordinator fails). The update is eventually executed atomically on each node; however, cluster-wide atomicity is not achieved. In particular, updates may be executed in different orders at different nodes. These relaxed semantics allow for a significant performance improvement, since locks are held only for the duration of the local update and not across the round-trip interaction with a coordinator.

An example data structure that takes advantage of these weaker semantics is an unordered list. Insert and remove operations are commutative in unordered lists, so "weak apply" semantics are sufficient. Such lists are used by several of our applications.

3.4.1.2 Replication and Performance

As mentioned above, the CHT replicates data for high availability. The implementation distinguishes storage clones from the libraries that clones include to use the CHT, which decouples clone group size from which nodes actually store data. The set of storage nodes remains stable except for faults and explicit operator-controlled repartitioning. Thus, the replica groups and recovery are managed entirely within the CHT implementation, and storage clones are shared by many tables and clone groups (with namespace isolation). Replication and durability polices are table-specific, but the storage clones are not.

The degree of replication can have an impact on performance: more replicas will deliver higher read throughput, but lower throughput for updates. An extreme case of the latter effect can be observed when many updates to a single location in the hash table are attempted simultaneously: different nodes may prepare successfully for different instances of the 2PC protocol, causing all instances to fail. The chances of livelock increase with the number of conflicting updates and the degree of replication.

Most data updates are largely independent, so such conflicts do not happen frequently, but when they do occur, their impact is significant. We were forced to add an algorithm that detects livelock and serializes prepare interactions with each replica. As Figure 4 illustrates, such detection improves performance of atomic apply to be tolerable, but there is still significant degradation. If this is unsatisfactory, the application designer has the choice of reducing the amount of replication or relaxing consistency requirements by using "weak" apply, which

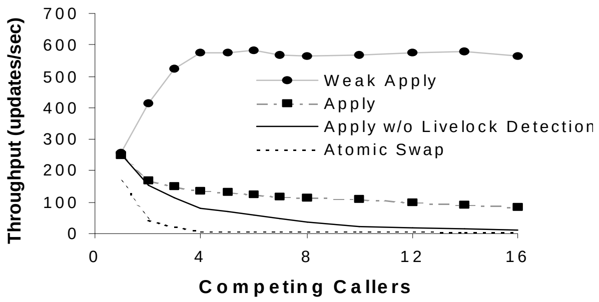

Figure 4: Update Contention with 2PC (2 replicas) This graph reveals the impact of livelock on updates using 2PC. Atomic Swap encounters livelock with even two competing updates. Proactive livelock detection is significant for Apply.

does not experience livelock. Another possibility, not yet implemented, would be to use exponential backoff. For comparison, we also measured the performance of atomic-swap-based updates; we found that it becomes unusable under a moderate amount of contention, despite livelock detection.

3.4.2 Cluster File System (CFS)

The second form of shared state is the cluster file system (CFS). In contrast to the CHT, the cluster file system manages large blobs of persistent storage that are normally on disk, and supports the streaming of data directly from disks to clients. Although it is closely related to a traditional file system, we chose not to implement the normal UNIX file system interface for several reasons:

- The traditional API limits atomicity. First, the only atomic operation is "rename" which requires copying whole files even for small updates. Second, file metadata operations are path based, which mixes path and file updates, and presents problems if the path changes during a file update. We provide first-class i-nodes, which eliminate redundant path resolutions, and provide natural support for atomic file updates, since you can name them directly.
- File consistency across multiple nodes is very limited. We desire a range of consistency and durability options, including both strong consistency across the cluster with replication, and local temporary storage.
- There is only one kind of index on files, the directory. We would like files to belong to multiple indices of different types simultaneously, including hash tables, B-trees, and version trees.
- We would like extensible metadata to simplify service-specific file operations, such as version numbers, TCP or MD5 checksums, and caching/ expiration directives.

Thus, the basic strategy is to provide a "toolbox" local file system that can we reuse in multiple ways to build service-specific cluster-based file systems. The toolbox deals with entirely local instances of storage, called *volumes*. We provide a simple physical global namespace using (node, volume, i-node) triplets.

A "file" consists of an i-node that has several values: typically a *segment,* which holds the data, and some metadata attributes. The metadata is extensible, which allows services to store their own metadata. A "directory" is just one kind of index on top of the i-nodes.

We provide atomic transactions, which in turn enables files to have multiple indices (or multiple parents), and simplifies renaming, deletion, and path operations. Direct exposure of i-nodes also allows database-style iteration through sets of files.

The real power of the CFS comes from the ease with which an author can create file-system like things. To build a shared file system across a clone group, the author need only define a global namespace. For example, storage for web pages need not have a directory at all, and can just use the CHT to store name→i-node mappings, thus enabling single seek access to the data segment. Or even simpler, a CFS with a fixed number of nodes can be built just by using a static hash function to map file names to nodes. Thus with little service-level code, we can achieve a variety of file systems.

Replication is completely orthogonal to the partitioning of a cluster-wide namespace and is handled quite differently than in the CHT. For a replicated CFS, the author must define the replica groups and use 2PC to update them. Since operations in the file system are atomic, the general 2PC manager can be used to build replication easily. This is intentionally quite a different policy from the CHT, in which replication was managed transparently. We take a different tack in CFS for two reasons: 1) there are a wider array of strategies for a replicated file system, making it harder to have any single one, and 2) the existence of the CHT makes it really easy to manage replica groups within a service, since it handles atomicity and recovery of this metadata automatically. This approach enables powerful service-specific CFSs (a service can have more than one) with very little application-level code. We have built three different service-specific file systems so far.

Finally, as a performance optimization, we support streaming of data segments across the cluster (versus store-and-forward copying). Any stage in the cluster may issue a stream task (acting as stream client) to another stage (the stream server), thus establishing a virtual channel within the cluster for reading or writing data. This is particularly useful for streaming data directly from the CFS out to the wide-area network, and is used for both our web and e-mail servers.

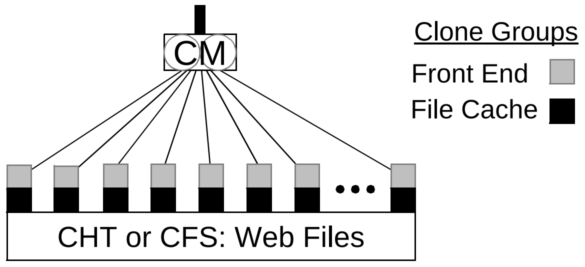

Figure 5: Ninja Web Service
Only Front Ends receive external connections, Cache nodes serve files locally or retrieve them from either the hash table or CFS (two different versions).

We have not built a generic cluster file system yet, although the one in our e-mail server is relatively general and could be packaged up for reuse by other services. We are still learning about the revised CFS API and expect to generalize support for replication and partitioning in the future, which will eliminate the small amount of service-level code required now.

4 Applications

In this section, we review three applications built using the Ninja framework. We then use these applications in the next section to evaluate the framework and our goals of robustness and ease of authoring. We have also built several other applications, including other web and mail servers, and a Napster-like file-sharing service.

4.1 Ninja Web Server

The prototypical service for Ninja is the web server, and we thus use it to evaluate all of our goals. The web server is relatively simple but achieves scalability, high availability, graceful degradation, and online evolution.

We have implemented several web server prototypes, serving both static and dynamic pages using either the hash table or the CFS for page storage. Our latest prototype builds upon the Haboob web server [WCB01] and modifies it to retrieve pages from the CHT. Haboob uses SEDA to handle a large number of simultaneous connections, making it an ideal front-end for the Ninja cluster web server. Haboob maintains an in-memory cache; we performed minor modifications to the cache miss component to fetch page data from the hash table instead of from local disk. Adding a thin wrapper to make an instance of Haboob behave as a Ninja clone allows us to create a clustered web service, with the CM directing external HTTP requests to one of the clones. Figure 5 shows the structure of the web server.

The shared persistent state maintained by the CHT allows any front-end node to answer any request; the CM masks front end failures. The replication policy used for the tables storing page data can be tuned to

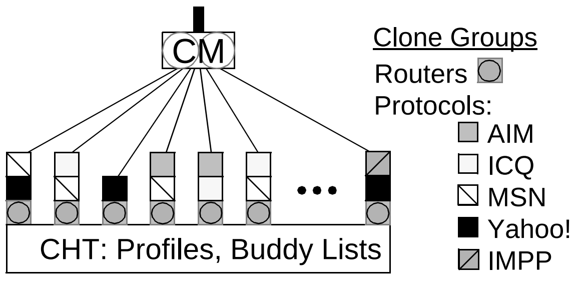

Figure 6: NinjaIM Architecture
Each protocol has a clone group, whose size depends on the traffic in that protocol. All protocols use Message Routers to communicate with each other, and all use the CHT for profiles and buddy lists.

achieve desired tradeoffs between availability and performance, and different classes of pages may be split among tables with different replication strategies. Similarly, front ends may be partitioned to provide differing quality of service or achieve better cache performance, as described in Section 3.3.2.

4.2 Universal Instant Messaging Proxy

NinjaIM is an instant-messaging (IM) proxy that performs protocol translation among popular instant messaging protocols and e-mail. It currently supports AIM, ICQ, MSN, Yahoo!, and IMPP protocols. Users can use the unmodified MSN client software or our Java applet-based client to communicate with users on all five IM systems. NinjaIM forwards messages bidirectionally among the five systems, which allows all users to reach each other.

There are several challenges in implementing an IM service. First, it must scale to a huge number of connections that are mostly idle; AOL's AIM has over 90M registered users [Hu00]. Most IM systems use long-lived TCP connections for every active user. Second, it requires scalable persistent storage for user profiles and buddy lists. Third, it must be able to route messages efficiently and process buddy status updates.

Figure 6 shows the NinjaIM architecture. The Connection Manager enables NinjaIM to easily scale up the number of connections linearly with nodes, and provide high availability. The CHT is used to store user profiles and buddy lists, which allows users to connect to any node in the cluster. To provide efficient buddy status notification, both a forward buddy list and a reverse buddy list are stored. In addition, we store the node to which a user is connected, which is used to route messages between nodes. All of the shared state may be cached locally (local soft state) to improve performance. Finally, partitions are used to provide better affinity for chat sessions. For example, when a user initiates a chat session, all the parties are given the same partition number (externalized as a port number) to which to connect. The CM maps the port number to a single node in the normal case, but need not in the presence of unusual load or faults.

4.3 NinjaMail

E-mail is one of the most widely used Internet applications, with hundreds of millions of users world-wide. Moreover, many of these users are concentrated in large e-mail services. AOL currently has over 23 million e-mail accounts [Lei00], while Hotmail has over 110 million [WB01].

NinjaMail is a scalable, highly available and extensible e-mail service, built on the Ninja architecture. At NinjaMail's core is the MailStore module, a message access library that uses the CHT and CFS to store user profiles, e-mail messages, and message indices. Built on top of this are various access modules, which support interaction between users and the message store. We have fully functional modules for sending and receiving messages via SMTP, and reading messages via POP and HTML. Additionally, we have nearly completed an implementation of the IMAP protocol.

Figure 7 shows the architecture for NinjaMail. The NinjaMail modules keep all long-lived state in the CHT or the CFS. This allows the infrastructure to create and destroy clones in response to load changes or faults. NinjaMail's use of the underlying mechanisms is illuminated by examining a typical message cycle:

Message arrival (`SMTP`): 1) Accept a new SMTP connection from the CM, 2) check the CHT, to verify that the recipient is a valid user, 3) stream the message to the replicated file system (MailStore), and 4) use the CHT apply function to add the message to the user's message index.

Message retrieval (`POP`): 1) Accept a new POP connection from the CM, 2) check the user's login name and password in the CHT, 3) retrieve the user's message index, 4) stream messages from the MailStore to the user, as requested, and 5) use the CHT apply function to update the persistent copy of the message index when the user deletes messages or updates the status flags.

The cluster-based file system of the MailStore is built using the CFS to provide atomic local storage volumes, and the CHT maintains the mappings from partitions to replica groups, and from replica groups to MailStore clones. It uses the 2PC library to update the replicas. This gives us a replicated cluster file system with very little application code. There are no "directories" in the file system; the only index on files is the global hash table that maps replica groups to storage nodes.

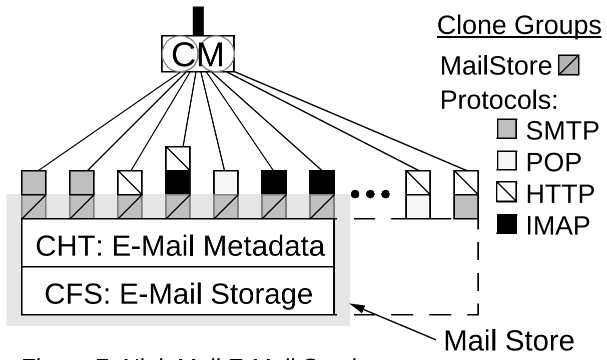

Figure 7: NinjaMail E-Mail Service
The service is divided into protocol handlers, each of which has its own clone group, and the Mail Store, which uses both the CHT and CFS to manage e-mail, user and folder information.

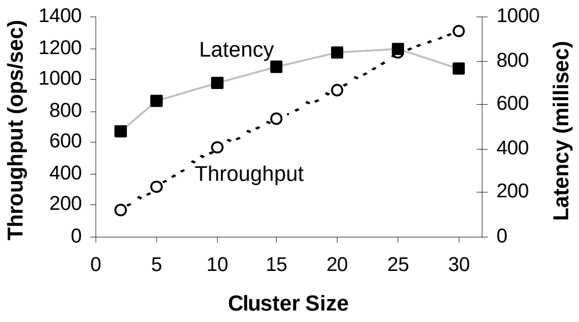

Figure 8: NinjaMail Scalability

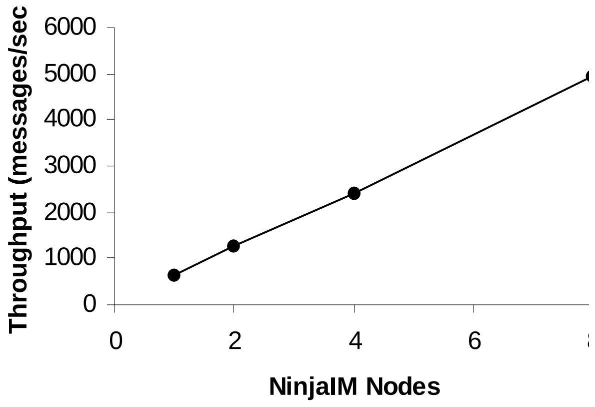

Figure 9: NinjaIM Scalability
Scalability is measured in terms of total throughput of IM messages per second. In addition to the n front-end nodes there are 2 additional nodes that store the replicated CHTs for NinjaIM.

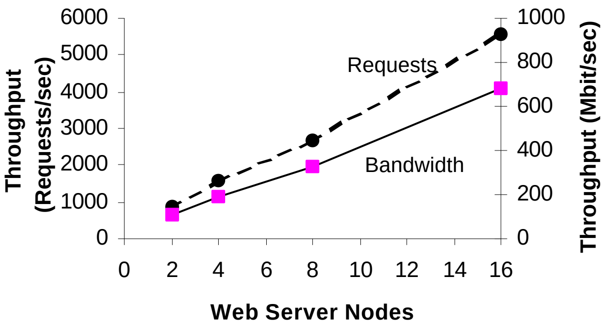

Figure 10: Web Server Scalability

5 Evaluation

In this section, we use the above applications to evaluate each of our goals: scalability, high availability, graceful degradation, online evolution, range of semantics, and ease of service authoring.

5.1 Scalability

For scalability, the overall goal is to support a very large number of users. For most services this corresponds directly to the number of simultaneous connections, including the web server, IM server and music server. For NinjaMail, scalability is tied more directly to messages per second. Ninja also supports linear scaling of database size, which comes directly from simple partitioning; we have built services using the CHT with more than 1TB of storage and over 100 nodes [GBH+00].

Figure 8 shows the scalability of NinjaMail in a message receipt, storage, and retrieval test. Each cluster node functions both as a front-end for SMTP and POP, and as a member of the CHT. The cluster nodes used for this experiment were 2-way SMPs with 500-Mhz processors and 512 MB of RAM, running Linux 2.4.7. Each test was performed with a user base of 1 million times the cluster size. Our test harness executes a simple loop. It first selects a random user and node, and submits a 4K message. Next, it selects another random user and node, and uses POP to read and delete all messages for that user. The per-node performance is excellent, at about 14 times the performance of a typical sendmail setup on the same hardware (for the message receipt portion), perhaps due the efficiency of the CHT for metadata. Extrapolating, we expect NinjaMail (as is) should be able to handle the Yahoo! mail workload, about 12 billion messages per month [Yah01], with a cluster of around 100 nodes.

Figure 9 shows the scalability for NinjaIM, measured in total throughput of IM messages. Simulated clients saturate the server using the full MSN IM protocol by sending messages every 5 seconds. Each front end node ran one message router and one MSN IM server. At the peak of 4941 messages/sec (with 8 front ends), this corresponds to almost 25,000 simultaneous extremely active clients.

Figure 10 shows the scalability of the web server under the SPECweb99 benchmark with 600MB data/node; single node performance is consistent with a solid single-node web server such as Apache [Apa01]. Note that the Ninja web server is not just a web farm, but actually reflects strongly consistent data across the clus-

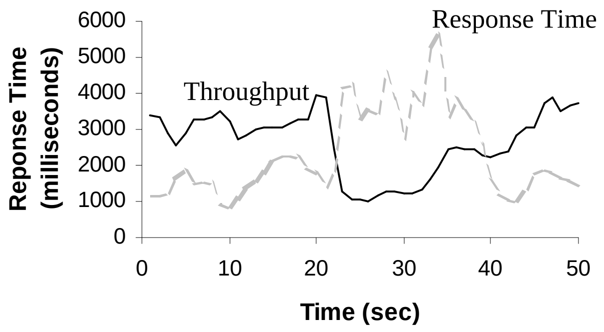

Figure 11: Recovery time for an Unexpected Death
This graph shows the recovery process for a two-node web server with one failure at time 20. A new clone takes over the traffic in six seconds, and recovery completes in about 20 seconds.

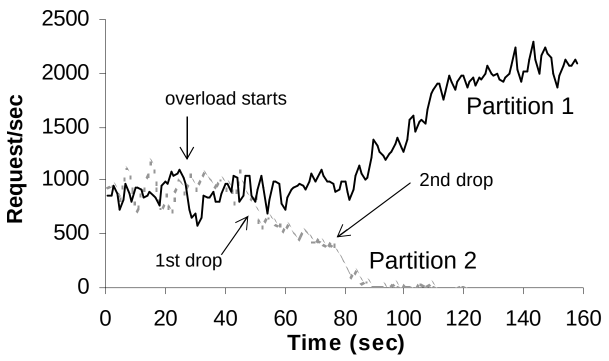

Figure 12: Prioritized Admission Control (Overload)
We start with a 2-node clone group with two partitions, 1 and 2; Partition 1 has higher priority. Initially both clones handle both partitions. Overload is detected by one of the nodes, which initiates overload mode. At "1st drop" the CM drops half the traffic to the lower priority partition; after a second drop, Partition 2 is not admitted at all and Partition 1 throughput doubles.

ter, and integrated connection management for high availability and online evolution.

5.2 High Availability

Our goal for high availability is to show that users have a high probability of success, and that we can provide independent retry for a given query or connection. It is not our goal to provide fault-tolerant *connections*. Rather, we forfeit active connections on lost nodes, but retries should automatically locate another node and work correctly (by using shared replicated state).

Figure 11 shows the recovery after an unexpected death in a two-node web server. A third clone took over the affected traffic within 6 seconds, and the overall server was fully recovered in about 20 seconds. Response time increased by several seconds during the recovery process, and active connections on the dead node were lost.

In the case of graceful shutdown, Ninja normally does not drop any connections. This case is covered shortly under the discussion of online evolution, which uses controlled shut downs to upgrade a running service without drops.

5.3 Graceful Degradation

At the service level, we support several strategies for graceful degradation. The goal is to react gracefully to offered loads that exceed capacity. In practice, peak loads can be 5x the average load, making it impractical to provision for peak load [Mov99]. Even with overprovisioning, 10x load spikes still occur [WS00].

The default strategy is simply to reject new connections when the service is saturated. This preserves the maximum throughput, but is not all that graceful. We provide three strategies that exploit service-level knowledge, via partitions, to degrade more gracefully.

The first strategy is simply to prioritize partitions and assign separate resources for each partition. This enables low-priority partitions to be overloaded without affecting high-priority connections, and enables independent throughput asymptotes and overprovisioning ratios. This was shown in Figure 2, in the CM section.

The second strategy, shown in Figure 12, is to prioritize partitions and drop low-priority connections during overload. We refer to this as prioritized admission control. This ensures that under overload the most important connections (or users) are handled first, potentially to the exclusion of lower priority connections. An improvement would be to drop connections probabilistically based on the priority of the partition, but this can essentially be done by first using different resources for each partition, and then dropping connections independently as each partition becomes overloaded.

As discussed in Section 3.3.2, Ninja sheds excess connections by sending them to a "drop" clone. We have not fully explored the power of this mechanism, which could be service specific to enable very fine-grain admission control, since the drop clone could decide on a case-by-case basis to handle some connections.

The third strategy we employ for graceful degradation is to try to serve more requests, but in a degraded form, which is possible because clones *know* that they are in overload mode. For example, a web server might serve generic versions of pages rather than personalized versions. The degraded service moves out the absolute scale limit, at which point further degradation using one of the first two strategies would have to take place.

5.4 Online Evolution

Online evolution is enabled by our ability to shut down clones gracefully, without dropped connections. Figure 13 shows online evolution between two versions of a three-node web server. To upgrade a node, the

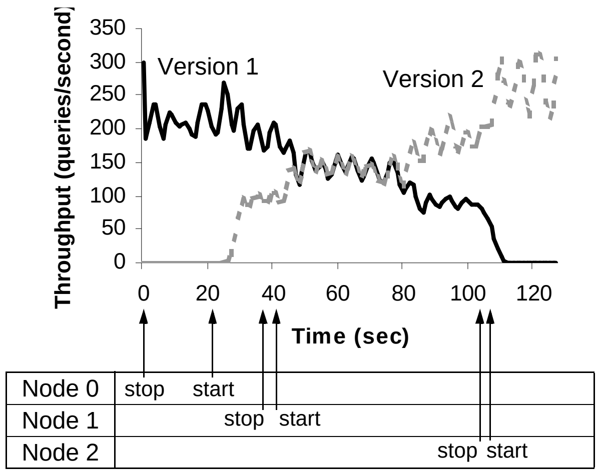

Figure 13: Online Evolution for a 3-node Web Server Each node starts with Version 1 and is then gracefully shut down and restarted with Version 2. No connections are dropped during the transition. The first restart takes longer due to creation of the clone group for Version 2.

infrastructure first updates the CM to stop sending in new connections to the Version 1 clone. Next, Ninja starts a Version 2 clone on the node, which begins receiving new connections. The Version 1 clone exits once all established connections have been serviced. By repeating this process on all nodes in sequence, we can upgrade the entire cluster with no downtime and no dropped connections.

Note that the two versions typically coexist on the same node for some time while the Version 1 clone finishes servicing existing connections. This is possible because Ninja's virtualization of resources prevents the two versions from interfering with each other (other than performance).

5.5 Range of Semantics

Our support for a range of data consistency semantics comes primarily from the CHT and from the ability to build service-specific file systems. The simplest form of this is choosing non-replicated storage, which is possible with both the CHT and the file system toolkit. We have found this useful for local file caches in some of our web server implementations and in the Ninja version of Napster (not discussed).

Figures 3 and 4 show that we can reduce lock contention if we accept updates that have an inconsistent ordering across replicas (using "weak apply"). This approach can achieve five times the throughput with two replicas under heavy contention. Our primary use for this so far has been to maintain unordered lists, which are useful in NinjaMail (since internal message order is not critical), and in various membership lists, such as members of a chat room.

Although not discussed, we have also exploited

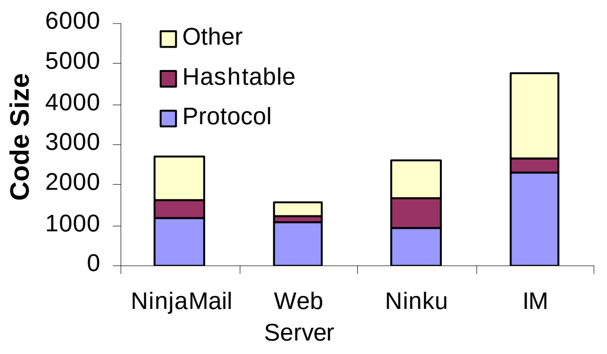

Figure 14: Code Size for Services in Lines (Java)

delayed disk writes in the CFS to improve latency and throughput at the expense of a small window for lost data (much like NFS updates). Similarly, the CHT allows two independent memory copies to be considered "durable", rather that than more strict definition of two copies on disk. In the former case, the disks are updated shortly thereafter in the style of group commit.

5.6 Service Authoring

Overall, we found it hard to write the underlying mechanisms, but easy to write the services, which fulfills our primary goal. Figure 14 shows the code size for four applications. The Ninku application, which was not discussed, is the Ninja version of the Napster service. In comparison, the Ninja infrastructure code is about 20,000 lines of code, not counting various third-party libraries used by the CHT and CFS. These services are remarkably small given that they are full-fledged robust services. For example, the Porcupine scalable e-mail server [SBL99] is about 30,000 lines by itself. Both the e-mail and web servers seem to be about one-tenth the size of comparable stand-alone applications.

The primary burden that we did not lighten is the difficulty of authoring protocol code, which presents an obvious place for future work. We also found event-based programming, used for most internal modules and some services to be harder than using threads.

One other important point is that none of these application require *any* code for high availability, online evolution or graceful degradation, although some may have a few configuration lines to define partitions (if used).

6 Discussion

In this section, we review the principles behind Ninja and discuss related and future work.

The programming model has three important principles. First, we want to exploit the natural parallelism of Internet services. The two advantages of this approach are that applications fit naturally and that we can ban more general types of concurrency, which are historically hard to get right and require unknown resources for correctness. In particular, services do not create or manage threads.

Second, like databases, we want to hide the complexity of fault tolerance, persistence and scalability.

Third, the explicit use of shared state allows us to simplify recovery greatly. The invariant is that anything that must survive faults must be kept in the shared state. Local state thus requires no recovery, and the shared state is recovered transparently. We apply the same principle recursively to build the shared state mechanisms: we differentiate storage clones, which require recovery, from all other clones, which do not. It is the clean recovery story that allows us to provide high availability, online evolution, and dynamic resizing and remapping of clone groups.

We have also pursued a bottom-up approach that provides a range of semantics. For example, the 2PC library and CFS are really tools that are used to simplify building complex systems. The primary difference between the CFS and a traditional file systems is exactly that the CFS is a toolbox with a more appropriate interface for authoring services that need replicated persistent storage. Even the CHT is used as a tool to build the cluster-based file system of NinjaMail. Similarly, we try to make these tools configurable to enable tradeoffs among performance, consistency and replication. The "weak apply" function is the best example of this.

Connection management seems fundamental to robust Internet services. In general, there must at least be a dynamic mapping between external names and currently working internal nodes; otherwise failures are visible to clients. Partitions enable application input into how the CM should prioritize connections. This is essentially a use of static type information (partitions are usually defined statically) to enable run-time optimization during overload. They also provide better cache affinity for all kinds of "front end" clone groups.

The CHT exploits the use of a narrow interface to simplify the maintenance of consistency. Unlike a shared address space, the CHT can only be accessed via method calls and thus only needs to ensure consistency at these points. This is most noticeable in the ability to support atomic updates, as a hash table value is simply not visible during updates.

6.1 Related Work

Lightweight recoverable virtual memory [MMK+94] provides an integrated approach to in-memory data structures with durability. It could be used to implement the non-replicated versions of our shared data structures, but does not support replication or 2PC.

The traditional way to simplify persistent applications is to store all data in a DBMS and use a declarative query language for all access and updates. We intentionally desire a "navigational" rather than relational interface for better integration with the rest of the service, which is navigational. Databases also focus on consistency under faults at the expense in practice of availability, where we explicitly provide a range of semantics and tradeoffs. We find that availability is often more important for Internet services than strict consistency. DBMSs also provide a large whole solution with little ability to customize semantics or make tradeoffs. In contrast, we may decide to consider something committed if it is in memory on two independent nodes, and only later move objects to disk, which increases throughput for updates. We believe services should use a combination of our techniques and DBMS solutions.

Object-oriented databases, such as Thor [LAC+96] or Persistent Java [ADJ+96], share our use of controlled interfaces, and can implement all of our shared data structures, although they are more heavyweight and generally don't offer a range of semantics. They are strictly more powerful, with support for transactions and nested objects. We also find power in our "toolbox" approach that has allowed us to build a range of persistent data structures out of logging, 2PC, and the apply function. We also depend on and exploit our partition-free, high-performance network (typical for a cluster).

Application servers, such as BEA's WebLogic [BEA01], provide persistent shared state by wrapping navigational structures around a relational database. These servers also target large-scale highly available services and were developed concurrently. Application servers typically also provide integration with legacy systems. Ninja provides better support for in-memory data structures, variable semantics, graceful degradation and online evolution. Use of Ninja's techniques would complement these servers' use of RDBMS systems, and one vendor is incorporating some of our techniques.

"Layer 7" switches, such as the Foundry switch that we use, provide some aspects of connection management, as does HACC [ZBCS99]. In particular, they can provide load balancing and basic partitioning by URL. The primary advantage of Ninja is the integration of the control of the manager into the framework. We dynamically reconfigure the switches as clone groups change, and we provide integrated support for online evolution and graceful degradation. In fact, our dynamic use of these switches was clearly novel, as we uncovered many new bugs in production hardware that we had to work around.

The Porcupine mail server [SBL99] shares the goals of scalability and availability, and even some of the techniques for replication and scalability. However, Porcupine is a single application rather than a framework. The existence of Ninja makes it easy to write Porcupine: NinjaMail has about one-tenth the code size of Porcupine for similar functionality and robustness. In addition, Ninja is more efficient and allows a wider range of performance tradeoffs than were present in Porcupine.

The TACC framework [FGB97] is a predecessor to this work and shares most or our goals, but does not address persistent shared state, which is the hardest part. It also uses application-specific front ends to do connection management, which we avoid.

The Ninja project [GWv+01] that led to this work has a much broader scope, and includes support for distributed systems built on top of clusters, which we refer to as "bases" in the overall architecture. Some of the additional pieces include support for proxies and end devices, such as laptops, phones, or PDAs; support for paths that connect these elements; security; and OS and proxy support for small devices. There are also papers that cover subsets of the work here in greater detail, including the CHT [GBH+00,Grb00] and SEDA [WCB01].

6.2 Future Work

There are at least three key areas of future work: ease of authoring, ease of use, and support for shared state. Section 5.6 covered some enhancements to ease authoring.

To simplify ease of use, there is much we could do to automate online evolution and graceful degradation. Evolution should have an explicit publishing process and a way to revert to the previous version easily. Graceful degradation is mostly automated, but is still very service specific. We don't help much with how a service should define partitions or trade off quality and performance. We could also use a unified way to test overload conditions and in general administer running services.

Our support for shared state should evolve to include true transactions rather than the atomic actions that we support now. This is quite a bit harder and the current set has proven very useful as is. There is also more we can do with the interaction between lock contention and 2PC, as discussed in Section 3.4.1. Finally, our recovery code remains immature due to the difficulty of thorough testing. However, it is exactly the complexity of recovery code that makes it so valuable to build once for the framework, rather than separately for each service. The automation of recovery is the most valuable aspect of the Ninja framework to service authors.

7 Summary

Ninja defines a new programming model and then uses the model to simplify the implementation of complex network services. The model exploits the natural parallelism of large-scale services and hides the complexity of threads, locks, shared state, recovery, load balancing and graceful degradation. It provides several invariants that greatly simplify service authoring:

- Each service has its own virtual cluster, which may vary in size transparently over time. We have shown linear scalability up to 100 nodes for toy applications and to 30 nodes for the e-mail server.
- Services can have many shared namespaces. Each namespace provides strongly consistent shared state across the nodes of the service.
- Shared state can be persistent and highly available with automatic recovery from faults.
- Concurrency is implicit in the programming model, which avoids the creation and management of threads in applications. Atomicity is provided by the shared state primitives and by isolation of namespaces and local state.
- Connections and external names are managed automatically for load balancing and fault tolerance.
- Overload is detected automatically, which initiates graceful degradation as needed.
- The CM enables online evolution and graceful degradation without help from service authors. They may use partitions for fine-grain control of both quality of service and graceful degradation.
- The CM and highly available shared state together enable highly available services.

Because of these powerful invariants, Ninja services remain remarkably simple despite being scalable, highly available and persistent. We have been able to write several real services using Ninja, including instant messaging, a Napster-like file sharing system, and scalable web and e-mail servers. In all cases, the code for the service was small and relatively simple (e.g. no recovery or logging code). In the case of e-mail, we achieved a ten times reduction in code size for a comparable scalable server by using Ninja.

Acknowledgments: We would like to thank Amin Vahdat and the referees for their helpful feedback.

[ADJ+96] M. Atkinson, L. Daynes, M. Jordan, T. Printezis and S. Spence. "An Orthogonally Persistent Java." *ACM SIGMOD Record*, 25(4), December 1996.

[Apa01] The Apache Web Server. http://www.apache.org.

[BEA01] *The BEA WebLogic Server Datasheet*. http://www.bea.com

[DGNP88] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister. "A single-program-multiple-data computational mode1 for epex/fortran." *Parallel Computing*, 5(7), 1988.

[FB99] A. Fox and E. Brewer. "Harvest, Yield, and Scalable Tolerant Systems." *Proc. of HotOS-VII*. March 1999.

[FGCB97] A. Fox, S. D. Gribble, Y. Chawathe and E. Brewer.

"Scalable Network Services" *Proc. of the 16th SOSP*, St. Malo, France, October 1997.

[Fou01] Foundry Networks ServerIron Switch. http://www.foundrynet.com/

[GBH+00] S. Gribble, E. Brewer, J. M. Hellerstein, and D. Culler. "Scalable, Distributed Data Structures for Internet Service Construction." *Proc. of OSDI 2000*, October 2000.

[GR97] J. Gray and A. Reuter. *Transaction Processing*. Morgan-Kaufman, 1997

[Gri00] S. Gribble. *A Design Framework and a Scalable Storage Platform to Simplify Internet Service Construction*. Ph.D. Dissertation, UC Berkeley, September 2000.

[GWv+01] S. Gribble, M. Welsh, R. von Behren, E. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, and B. Zhao. "The Ninja Architecture for Robust Internet-Scale Systems and Services." *Journal of Computer Networks*, March 2001.

[Her91] Maurice Herlihy. *A Methodology for Implementing Highly Concurrent Data Objects*. Technical Report CRL 91/10. Digital Equipment Corporation, October 1991.

[Hu00] J. Hu. "AOL instant messaging rivals file complaint with FCC." CNET News.com. http://news.cnet.com/news/0-1005-200-1755834.html, April 25, 2000.

[HT93] D. Hillis and L. W. Tucker. "The CM-5 Connection Machine: a scalable supercomputer." *Communications of the ACM*, 36(11), pp. 31–40, November 1993.

[HW87] M. P. Herlihy and J. M. Wing. "Axioms for concurrent objects." *Proc. of the 14th SOSP*, pp. 13–26, January 1987.

[LAC+96] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day and L. Shrira. "Safe and efficient sharing of persistent objects in Thor." *Proc. of ACM SIGMOD*, pp. 318–329, 1996.

[Lar00] J. Larus. *Enhancing server performance with Staged-Server*. http://www.research.microsoft.com/~larus/Talks/StagedServer.ppt, October 2000.

[MMK+94] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, J. J. Kistler. "Lightweight Recoverable Virtual Memory." *ACM Transactions on Computer Systems*, 12(1), pp. 33–57, February 1994.

[Mov99] MovieFone Corporation. "MovieFone Announces Preliminary Results from First Day or Star Wars Advance Ticket Sales." Press Release, May 12, 1999.

[PAB+98] V. S. Pai, M. Aron, G. Banga, M, Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. "Locality-Aware Request Distribution in Cluster-based Network Servers." *Proc. of ASPLOS 1998*. October 1998.

[PDZ99] V. S. Pai, P. Druschel, and W. Zwaenepoel. "Flash: An efficient and portable Web server." *Proc. of the 1999 Annual USENIX Technical Conference*, June 1999.

[RV89] E. Roberts and M. Vandevoorde. *Work crews: An abstraction for controlling parallelism*. DEC SRC Technical Report 42, Palo Alto, California, 1989.

[SBL99] Y. Saito, B. Bershad and H. Levy. "Manageability, Availability and Performance in Porcupine: A Highly Scalable, Cluster-based Mail Service." *Proc. of the 17th SOSP*. October 1999.

[Sco96] S. L. Scott. "Synchronization and Communication in the T3E Multiprocessor." *Proc. of ASPLOS 1996*, October 1996.

[SPEC99] Standard Performance Evaluation Corporation. *The SPECweb99 Benchmark*. http://www.spec.org/osg/web99.

[WM01] M. Williams and M. Berger. "Two days late, Hotmail gets an upgrade." *InfoWorld*, July 19, 2001.

[WCB01] M. Welsh, D. Culler and E. Brewer. "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services." *Proc. of the 18th SOSP*. October, 2001.

[WS00] L. A. Wald and S. Schwarz. "The 1999 Southern California Seismic Network Bulletin." *Seismological Research Letters*, 71(4), July/August 2000.

[Yah01] Yahoo! Inc. "Yahoo! Introduces Yahoo! Mail - Business Edition." Press Release, October 15, 2001.

[ZBCS99] X. Zhang, M. Barrientos, J. B. Chen, and M. Seltzer. "HACC: An Architecture for Cluster-Based Web Servers." *Proc. of the 3rd USENIX Windows NT Symposium*, Seattle, WA, July 1999.