Scalable, Distributed Data Structures for Internet Service Construction

Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler
The University of California at Berkeley
{gribble,brewer,jmh,culler}@cs.berkeley.edu

Abstract

This paper identifies a new persistent data management layer specifically designed to simplify cluster-based Internet service construction. This self-managing layer, called a distributed data structure (DDS), presents a conventional single-site data structure interface to service authors, but partitions and replicates the data across the cluster. We have designed, implemented, and analyzed a distributed hash table DDS that has properties necessary for Internet services (incremental scaling of throughput and data capacity, fault tolerance and high availability, high concurrency, and consistency and durability of data). The hash table uses two-phase commits to present a coherent view of its data across all nodes of the cluster, allowing any node in the cluster to service any task. The distributed hash table is shown to simplify Internet service construction by decoupling service-specific logic from the complexities of persistent, consistent state management, and allowing services to inherit the necessary service properties from the DDS rather than having to implement the properties themselves. We have scaled the hash table to a 128 node cluster, achieving a read throughput of 61,432 operations per second, and a write throughput of 13,582 operations per second.

1 Introduction

Internet services are successfully bringing infrastructural computing to the masses. Millions of people depend on Internet services for applications like searching, instant messaging, directories, and maps, but also to safeguard and provide access to their personal data (such as email, calendar entries, and recently, arbitrary application code and files). As a direct consequence of this increasing user dependence, today's Internet services must possess many of the same properties as the telephony and power infrastructures. These *service properties* include the ability to scale to large, rapidly growing user populations, high availability in the face of partial failures, the maintenance of strict consistency of users' data, and operational manageability.

It is challenging for a service to achieve all of these properties, especially when it must manage large amounts of persistent state, as this state must remain available and consistent to the end-user even if individual disks, processes, or processors crash. Unfortunately, the consequences of failing to achieve the properties are harsh, including lost data, angry users, and perhaps financial liability. Even worse, there appear to be few reusable Internet service construction platforms (or data management platforms) that successfully provide all of the properties.

Using clusters of workstations as a platform for Internet services helps to address these challenges. Many projects and products propose using software platforms on clusters to simplify Internet service construction [1, 2, 12, 24]. These platforms either rely on commercial databases or distributed file systems for persistent data management, or they do not address data management at all, forcing service authors to implement their own service-specific data management layer. We argue that databases and file systems have not been designed with Internet service workloads or cluster environments specifically in mind, and as a result, they fail to provide the right scaling, consistency, or availability guarantees that services require.

In this paper, we bring scalable, available, and consistent data management capabilities to cluster platforms by designing and implementing a reusable, cluster-based storage layer, called a *distributed data structure (DDS)*, specifically designed for the needs of Internet services. A DDS presents a conventional single site in-memory data structure interface to applications, and durably manages the data behind this interface by distributing and replicating it across the cluster. Services inherit the aforementioned service properties by using a DDS to store and manage all persistent service state, shielding service authors from the complexities of scalable, available, persistent data storage, thus simplifying the process of implementing new Internet services. We believe that given a small class of DDS types (such as a hash table, a tree, and an administrative log), authors will be able to build a large class of interesting and sophisticated servers. This paper describes the design, architecture, and implementation of a distributed data structure (in particular, a distributed hash table built in Java). We evaluate its performance, scalability and availability, and its ability to simplify service construction.

1.1 Clusters of Workstations

In [12], it is argued that clusters of workstations (commodity PC's with a high-performance interconnect) are a natural platform for building Internet services. Each node in the cluster represents an independent failure boundary, which means that replication of computation and data can be used to provide fault tolerance. A cluster permits incremental scalability: if a service runs out of capacity, a good software architecture will allow administrators to add additional nodes to the cluster, linearly increasing the capacity of the service. There is also natural parallelism in a cluster: if appropriately balanced, all CPUs, disks, and network links can be used simultaneously, increasing the throughput of the service as the cluster grows. Clusters have high throughput, low latency redundant system area networks (SAN); modern SANs can achieve 1 Gb/s throughput with 10 to 100 μs latency.

1.2 Internet Service Workloads

Popular Internet services must process hundreds of millions of tasks per day. Each task is usually "small", in that it causes a small amount of data to be transferred and computation to be performed. For example, according to recent press releases, Yahoo (`http://www.yahoo.com`) serves 625 million page views per day. Randomly sampled pages from the Yahoo directory average 7KB of HTML data and 10KB of image data. Similarly, AOL (`http://www.aol.com`) handles 5.2 billion web requests per day from their web proxy cache, with an average size of 5.5 KB per response. Services often take hundreds of milliseconds to process a given task, and their responses can take many seconds to flow back to clients over what are predominantly low bandwidth last-hop network links [16]. Given this high task throughput and non-negligible latency, a service may handle thousands of tasks simultaneously. Human users are typically the ultimate source of tasks; because users usually generate a small number of concurrent tasks (e.g. 4 parallel HTTP GET requests are typically spawned when a user requests a web page), the large set of tasks being handled by a service are largely independent.

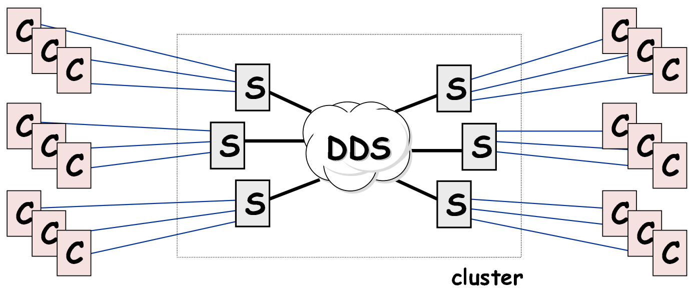


Figure 1: **High-level view of a DDS:** a DDS is a self-managing data repository running on a cluster of workstations. All service instances (S) in the cluster see the same consistent image of the DDS; as a result, any WAN client (C) can communicate with any service instance.

2 Distributed Data Structures

A distributed data structure (DDS) is a self-managing storage layer designed to run on a cluster of workstations [2] and to handle Internet service workloads. A DDS has all of the previously mentioned service properties: high throughput, high concurrency, availability, incrementally scalability, and strict consistency of its data. Service authors see the interface to a DDS as a conventional data structure, such as a hash table, a tree, or a log. Behind this interface, the DDS platform hides all of the the mechanisms used to access, partition, replicate, scale, and recover the data in the DDS. Because of this decoupling of complex mechanisms behind a simple interface, authors only need to worry about service-specific logic when implementing a new service. All of the difficult issues of managing persistent state are handled by the DDS platform.

Figure 1 shows a high-level illustration of a DDS. All cluster nodes have access to the DDS and see the same consistent image of the DDS. As long as services keep all persistent state in the DDS, any service instance in the cluster can handle requests from any client, although we expect clients will have affinity to particular service instances to allow session state to accumulate.[1]

The idea of having a storage layer to manage durable state is not new, of course; databases and file systems have done this for many decades. The novel aspects of a DDS are the level of abstraction that it presents to service authors, the consistency model it supports, the access behavior (concurrency and throughput demands) that it presupposes, and its many design and

[1]This assumes that session state is "soft-state" and can be recomputed by or retransmitted to the new service instance if the old service instance handling client failures.

implementation choices that are made based on its expected runtime environment (namely a cluster) and the types of failures that it can withstand. A direct comparison between databases, distributed file systems, and DDS's helps to show this.

Relational database management systems (RDBMS): an RDBMS offers extremely strong durability and consistency guarantees, namely ACID properties derived from the use of transactions [15], but these ACID properties can come at relatively high cost in terms of complexity and overhead. RDBMS's offer a high degree of data independence. This independence is powerful, but it also comes with complexity and performance overhead. The many layers of an RDBMS (such as SQL parsing, query optimization, access path selection, etc.) permit users to conceptually decouple the logical structure of their data from its physical layout. This decoupling allows users to dynamically construct and issue queries over the data that are limited only by what can be expressed in the SQL language.

From the perspective of the service properties, an RDBMS always chooses consistency over availability; if there are media or processor failures, an RDBMS can become unavailable until the failure is resolved, which is unacceptable for Internet services. Furthermore, scaling a database is difficult; parallel and distributed databases have had success in this arena, but the data independence offered by an RDBMS makes parallelization (and therefore scaling) hard in the general case. Given the arbitrary queries that SQL permits, it is difficult to predict how a given partitioning of data will affect performance. Internet services that rely on RDBMS backends typically go to great lengths to reduce the workload presented to the RDBMS, using techniques such as query caching in front ends [12].

Distributed file systems: file systems have much less strictly defined consistency models. Some (such as NFS [23]) have poor consistency guarantees, while others (such as Frangipani [25] or AFS [9]) guarantee a coherent filesystem image across all file system clients, with locking typically done at the granularity of files. The scalability of distributed file systems similarly varies; some use centralized file servers, and thus do not scale. Others such as xFS [3] are completely serverless, and in theory can scale up to arbitrarily large capacities. File systems expose a relatively low level interface with little data independence; a file system is organized as a hierarchical directory of files, and files are variable-length arrays of bytes. These elements (directories and files) are directly exposed to file system clients; clients are responsible for logically structuring their application data in terms of directories, files, and bytes inside those files.

Distributed data structures (DDS): a DDS has a strictly defined consistency model: all operations on elements inside a DDS are atomic, in that any operation completes entirely, or not at all. DDS's have one-copy equivalence, so although data elements in a DDS are replicated, clients see a single, logical data item. Two-phase commit is used to keep all replicas coherent, i.e. all clients of a DDS see the same image of that DDS through its interface. Transactions across multiple elements or operations are not currently supported: as we will show later, many of our current protocol design decisions and implementation choices exploit the lack of transactional support for greater efficiency and simplicity. There are Internet services that require transactions (e.g. for e-commerce); we can imagine building a transactional DDS, but it is beyond the scope of this paper, and we believe that the atomic consistency provided by our current DDS is strong enough to support interesting services.

The interface to a DDS is more structured and at a higher level than a file system: the granularity of an operation is a complete data structure element rather than an arbitrary byte range. The set of operations over the data in a DDS is fixed by a small set of methods exposed by the DDS API, unlike an RDBMS in which operations are defined by the set of expressible declarations in SQL. The positive implication of this is that the query parsing and optimization stages of an RDBMS are completely obviated in a DDS. The negative implication of this is that the DDS interface is less flexible than that of an RDBMS, and that there is less data independence than in an RDBMS.

In summary, by choosing a level of abstraction somewhere in between that of an RDBMS and a file system, and by choosing a well-defined and simple consistency model, we have been able to design and implement a DDS that has all of the service properties (scalability, availability, consistency, and manageability). Furthermore, it has been our experience that the DDS interfaces, although not as general as SQL, are rich enough to successfully build sophisticated services.

3 DDS Design Principles and Assumptions

In this section of the paper, we present design principles that guided us while building our distributed hash table DDS. We also state a number of key assumptions we made regarding our cluster environment, failure modes that we can handle, and the workloads that the DDS will receive.

Separation of concerns: the clean separation of service code from storage management simplifies the overall architecture of the system by decoupling the complexities of state management from service construction. Because all persistent service state is kept inside the DDS, service instances can be shut down and restarted without a complex recovery process. Crashes become incidental: only session state needs to be regenerated after a crash. Authoring a service is greatly simplified, since service authors need only worry about service-specific logic (such as parsing HTTP in the case of a web server) rather than the complexities of data partitioning, replication, and recovery.

Appeal to properties of clusters: in addition to the properties listed in section 1.1 of this paper, we require that our cluster is physically secure and is well-administered to reduce the probability of component failures. Given all of these properties, a cluster represents a carefully controlled environment in which we have the greatest chance of being able to provide all of the service properties. For example, the low latency nature of the cluster (10-100 μs instead of 10-100 ms for the wide-area Internet) means that two-phase commits are not prohibitively expensive in terms of latency. Similarly, a highly redundant network means that the probability of a network partition can be made arbitrarily small, and thus we do not consider the case of partitions in our protocols. Having a reliable uninterruptible power supply (UPS) and good system administration implies helps ensure that the probability of a system-wide simultaneous hardware failure is negligible, and thus we can rely on data being available in more than one failure boundary in the cluster (i.e., in the physical memory or disk of more than one node) while designing our recovery protocols. We do have a checkpoint mechanism that permits us to recover in the case that any of these cluster properties fail, but all state changes that happen after the checkpoint will potentially be lost should this occur.

Design for high throughput and high concurrency: given the workloads presented in section 1.2, the control structure used within the DDS to effect concurrency is critical. Typical techniques found in web servers such as process-per-task or thread-per-task cannot scale to this magnitude of concurrency. Operating system overhead (such as thread scheduling) and thread-specific overheads (such as per-thread stack space and lock acquisition times) cause performance to degrade as concurrency increases, as seen in figure 2.

To achieve both high concurrency and high throughput, we use an asynchronous, event-driven style of control flow in our DDS implementation, similar to that espoused by the Harvest web cache project [6]. A convenient side-effect of this style of control flow is that layering is inexpensive and flexible, as layers can be constructed by chaining together event handlers. Such chaining also greatly facilitates interposition—a "middleman" event handler can be simply patched between two existing handlers. A second important characteristic of this control flow style is that if a server experiences a burst of traffic, the burst is absorbed in event queues, providing *graceful degradation* by preserving the throughput of the server but temporarily increasing latency. By contrast, thread-per-task systems degrade in both throughput and latency if bursts arrive and are absorbed by additional threads.

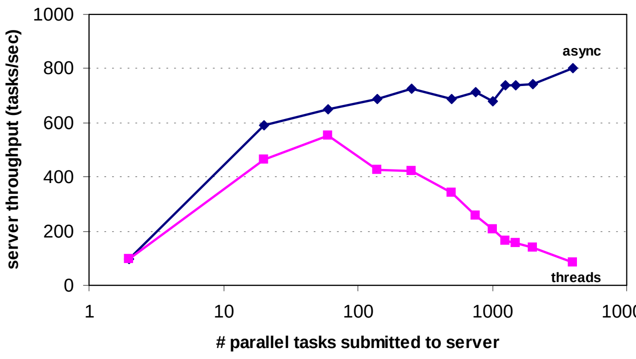


Figure 2: **Thread-per-task vs. asynchronous state machines:** this microbenchmark has a simple network server that receives a pipeline of many concurrent 150 byte request packets over a single TCP connection, and responds with 150 byte packets over a second TCP connection. In the "threads" case, a thread is dispatched to service each request, and in the "async" case, an event-driven state machine is created to handle the request. Both cases are run on a 167 Mhz UltraSparc running Solaris 5.6, and in both cases the per-task service latency is set to 10 ms.

3.1 Assumptions

If one node in the hash table cannot communicate with another, we assume it is because this other node has stopped executing (due to a planned shutdown or a crash). We thus assume that network partitions do not occur inside our cluster, and that software components in the cluster are fail-stop. The need for no network partitions is addressed by the high redundancy of our network, as mentioned in the previous section. We have attempted to induce fail-stop behavior in our software by having it terminate its own execution if it encounters an unexpected condition, rather than attempting to gracefully recover from such a condition.

These strong assumptions have been valid in practice; we have never experienced an unplanned network partition in our cluster, and our software has always behaved in a fail-stop manner. We further assume that software failures in the cluster are independent. We replicate all durable data at more than one place in the cluster, but we assume that at least one of these replicas is active (i.e. has not failed) at all times.

Our next assumption concerns the workload presented to our distributed hash tables. Each table's key space is the set of 64-bit integers; we assume that the population density over this space is even (i.e. the probability that any key has a value associated with it is a function of the number of values in the table, and not of the particular key). We don't assume that all keys are accessed equiprobably, but rather that the "working set" size of accessed keys is much larger than the number of nodes in our cluster. We then assume that a partitioning strategy that maps fractions of the keyspace to cluster nodes based on the nodes' relative processing speed will induce a balanced workload. It is up to service authors to adhere to these assumptions; failing to do so can result in imbalances across the cluster, leading to a reduction in throughput. Automatically tuning the hash table to smooth imbalances or hotspots is a topic of future work.

Finally, we assume that tables are large and long lived. Hash table creations and destructions are relatively rare events: the common case is for hash tables to serve read, write, and remove operations.

4 Distributed Hash Tables: Architecture and Implementation

In this section, we present the architecture and implementation of a fully functional distributed hash table DDS. Figure 3 illustrates the architecture of the distributed hash table. The complete system is comprised of the following components:

Client: the client consists of service-specific software running on a client machine. It communicates across the wide area with one of many service instances running in the cluster. The mechanism by which the client selects a service instance is beyond the scope of this work, but it typically involves DNS round robin [5], a service-specific protocol, or a transport or application-level multiplexing router on the edge of the cluster. An example of a client is a web browser, in which case the service would be a web server. Note that clients are completely unaware of DDS's—no part

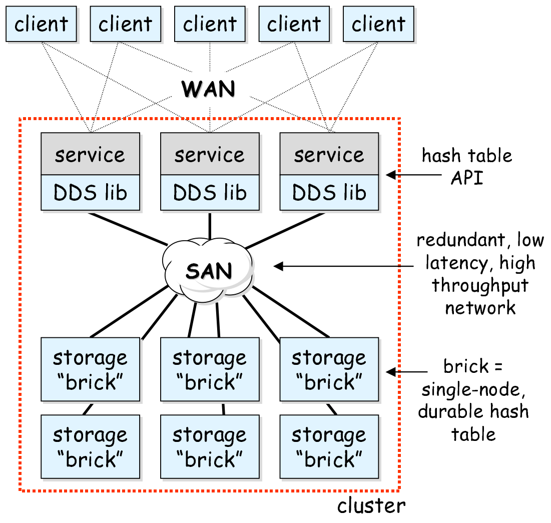


Figure 3: **Distributed hash table architecture:** each box in the diagram represents a software process. In the simplest case, each process runs on its own physical machine, however there is nothing preventing processes from sharing machines.

of the DDS system runs on a client.

Service: a service is composed of a set of cooperating software processes, each of which we call a service instance. These service instances perform some application-level function and communicate with wide-area clients. Service instances perform computation and may optionally have soft state (state which may be lost and recomputed if necessary), but they rely on the hash table to manage all persistent state.

Hash table API: the boundary between the service and the "DDS library" component of the architecture is the hash table interface. This interface provides services with `put()`, `get()`, `remove()`, `create()`, and `destroy()` operations on hash tables. Each of these operations is atomic, and all services see the same coherent image of all existing hash tables through this interface. Hash tables are named by strings, and hash table keys are 64 bit integers. Hash table values are opaque byte arrays, and each hash table operation atomically affects a hash table value in its entirety.

DDS library: the DDS library is a Java class library that presents the hash table interface to services. This class library accepts hash table operations, and cooperates with the "brick" components of the hash table to realize those operations. The DDS library contains only soft state; this soft state includes metadata about the current configuration of the cluster, and the current partitioning of data in the distributed hash tables across the "bricks". This library acts as the two-phase commit coordinator for state-changing operations on the distributed hash tables.

Brick: bricks are the only system components that manage durable data. Each brick manages a set of network-accessible single node hash tables. A brick consists of a buffer cache, a single-site lock manager, a persistent chained hash table implementation, and network stubs and skeletons for remote communication. Typically, we run one brick per CPU in the cluster, and thus a 4-way SMP will house 4 bricks.

4.1 Partitioning, Replication, and Replica Consistency

A distributed hash table provides incremental scalability as more nodes are added to the cluster; this scalability is both in terms of the throughput of operations that the distributed hash table can sustain, and also the amount of data that it can contain. In order to achieve this scalability, operations and data must be spread across the nodes in the cluster. This spreading is done by horizontally partitioning each table across the cluster. Each brick in the cluster thus stores some number of *partitions* of each table in the system, and when new nodes are added to the cluster, this partitioning is altered so that data is spread onto the new node. Because of our workload assumptions (section 3.1), this horizontal partitioning evenly spreads both load and data across the cluster.

Availability is a second goal of the distributed hash table. Given that the data in the hash table is spread across multiple nodes, if any of those nodes fail, then a portion of the hash table will become unavailable. For this reason, each partition in the hash table is replicated on more than one physical node in the cluster. The set of replicas for a partition form a *replica group*; all partitions in the replica group are kept strictly coherent with each other. Any partition in the replica group can be used to service a `get()`, and all partitions in the replica group must be synchronously updated for `put()` or `remove()` operations. If a cluster node fails, the data from partitions on that node is available on the surviving members of its replica groups. Replica group membership is thus dynamic; when a node fails, all of its partitions are removed from their replica groups. When a node joins the cluster, it may be added to the replica groups of some partitions (such as in the case of recovery, as described later).

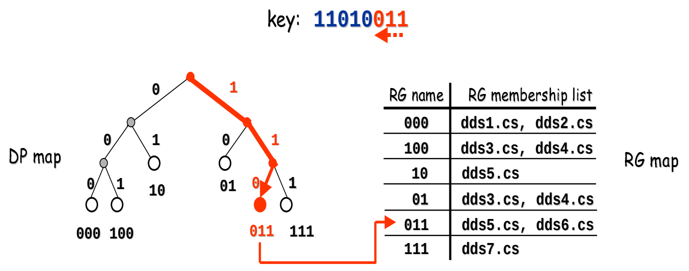
A third goal of the distributed hash table is consistency. When state changing operations (`put()` and `remove()`) are issued against a partition in the hash table, all replicas of that partition must be synchronously updated. We use an optimistic form of the two-phase commit protocol to achieve consistency, with the DDS library serving as the two-phase commit coordinator, and the replicas serving as the participants. In our system, if the DDS library crashes after *prepare* messages are sent, but before any *commit* messages are sent, the replicas will time out and abort.

If the DDS library crashes after sending out any *commit* messages, then all replicas must commit; because we want the hash table to be highly available, we cannot rely on the DDS library recovering after a crash and issuing pending *commits*. Thus, our replicas store short in-memory logs of recent state changing operations and their outcomes. If a replica times out while waiting for a *commit* from the DDS library, that replica communicates with its peers to find out if any of them have received a *commit*, and if so, the replica commits as well. If not, the replica aborts.

Any replica may abort an operation in the first phase of the two-phase commit (for example, if the replica cannot obtain a write lock on the key it needs to change). If the DDS library receives any *abort* messages at the end of the first phase, it sends *aborts* to all replicas in the second phase. Replicas do not commit side effects unless they receive a *commit* message in the second phase of the operation.

If a replica crashes during a two-phase commit, the DDS library simply removes that replica from its replica group, and continues the two-phase commit. Thus, in the face of crashes, all replica groups will shrink over time. We rely on a recovery mechanism (described later) for crashed replicas to rejoin the replica group. Because we have assumed that brick failures are independent (section 3.1), we made a significant optimization by requiring the image of each replica to be consistent only through its brick's cache, rather than having a consistent on-disk image. This allows us to have a purely conflict-driven cache eviction policy, rather than having to force cache elements out in a manner that ensures on-disk consistency. An important implication of this optimization is that if all members of a partition's replica group crash, that partition is lost. We rely on nodes in a cluster to be independent failure boundaries; for this to be true, there must be no systematic software failure in the cluster, and the power supply to the cluster must be uninterruptible.

Our two-phase commit mechanism gives us *atomic updates* to the hash table. It does not, however, give us transactional updates. If a service wishes to update more than one element atomically, our DDS does not



RG name	RG membership list
000	dds1.cs, dds2.cs
100	dds3.cs, dds4.cs
10	dds5.cs
01	dds3.cs, dds4.cs
011	dds5.cs, dds6.cs
111	dds7.cs

Figure 4: **Distributed hash table metadata maps:** this illustration highlights the steps taken to discover the set of replica groups which serve as the backing store for a specific hash table key.

provide any help. Adding transactional support to our DDS infrastructure is a topic of future work.

We do have a checkpoint mechanism in our distributed hash table that allows us to force the on-disk image of all partitions to be consistent; the disk images can then be backed up for disaster recovery. This checkpoint mechanism is extremely heavyweight, however; during the checkpointing of a hash table, no state-changing operations are allowed.

4.2 Metadata maps

To select a partition that is responsible for a particular key of a hash table, and to determine the replicas that are currently in the replica group of that partition, the DDS libraries consult two metadata maps that are replicated on each node of the cluster. Each hash table in the cluster has its own pair of metadata maps.

The first map is called the *data partitioning (DP) map*. Given a hash table key, the DP map returns the name of the partition that this key belongs to. The DP map thus controls the horizontal partitioning of data across the bricks. As shown in figure 4, the DP map is a trie on the least-significant bits of hash table keys; to find a key's partition, key bits are used to walk down the trie, starting from the least significant key bit until a leaf node is found. As the cluster grows, the DP trie subdivides in a "split" operation. For example, partition `10` in the DP trie of figure 4 could split into partitions `010` and `110`; when this happens, the keys in the old partition are shuffled across the two new partitions. The opposite of a split is a "merge"; if the cluster is shrunk, two partitions with a common parent in the trie can be merged into their parent. For example, partitions `000` and `100` in figure 4 could be merged into a single partition `00`.

The second map in the system is called the *replica group (RG) membership map*. Given a partition name returned from the DP map, the RG map returns a list of bricks that are currently serving as replicas in the replica group associated with that partition. The RG maps are dynamic: if a brick fails, it is removed from all RG maps that contained it. During recovery, if a brick joins a replica group for a table, it is added to that replica group after finishing recovery. An invariant that must be preserved is that the replica group membership maps for all partitions in the hash table must have at least one member; for the sake of fault tolerance, they should always have multiple members.

The maps are replicated on each node of the cluster, in both the DDS libraries and the bricks. The maps must be kept consistent, otherwise operations may be applied to the wrong bricks. Instead of enforcing consistency synchronously, we allow the maps on the libraries to drift out of date, but they are lazily updated when a DDS library uses them to perform operations. The DDS library piggybacks hashes of both maps on operations sent to bricks; if a brick detects that either map used is out of date, the brick fails the operation and returns a "repair" to the library. Thus, all maps become eventually consistent as they are used. An implication of this repair mechanism is that the libraries can be restarted with out of date maps, and as the library gets used its maps become consistent.

To `put()` a key and value into a hash table, the DDS library servicing the operation first consults its DP map in order to determine the appropriate partition for the key. Then, it looks up that partition name in its RG map to determine the current set of bricks serving as replicas for the key. The DDS library then performs a two-phase commit across these replicas.

To do a `read()` of a key, a similar process is used, except that the DDS library can select any of the replicas listed in the RG map to service the read. We use the locality-aware request distribution (LARD) technique [11] to select a read replica—LARD further partitions keys across replicas, in effect aggregating the physical cache size of all the replicas. If hotspots become an issue for a particular workload, we could instead randomly select a replica and increase the size of the replica group to sufficiently absorb the hotspot, but we have not needed to do this as of yet.

4.3 Recovery

If a brick fails, all of the partition replicas on that brick become unavailable. Rather than making the entire partition available, we remove the failed brick from all replica groups and allow operations to continue. When the failed brick recovers (or an alternative brick is selected to replace it), the recovering brick must "catch up" to all of the operations it missed. In many RDBMS's and file systems, recovery is a complex process that involves replaying logs, but in our system we use properties of clusters and our DDS design to vastly simplify recovery.

First, we allow our hash table to "say no"—bricks may return a failure for an operation, such as when a two-phase commit cannot obtain locks on all bricks (as would happen if two `puts()` to the same key are simultaneously issued), or when replica group memberships change during an operation. The freedom to say no greatly simplifies the hash table logic, since we don't worry about correctly handling operations in these rare operational situations. Instead, we rely on the DDS library (or, ultimately, the service perhaps even the WAN client) to retry the operation.

Second, we don't allow any operation to finish unless all participating components agree on the metadata maps. If any component has an out-of-date map, the operation will fail until the maps are reconciled.

Finally, we make our partitions relatively small (˜100MB), but we have many of them for large tables. This relatively small partition size means that we can transfer an entire partition over a fast system-area network (typically 100 Mb/s to 1 Gb/s) within 1 to 10 seconds. Thus, during recovery, we can incrementally copy entire partitions to the recovering node, obviating the need for the undo and redo logs that are typically maintained by databases for the sake of recovery.

When a node initiates recovery, it grabs a write lock on one replica group member from the partition that it is joining; this write lock means that all state-changing operations on that partition will start to fail. Although these locks are very coarse grained, we don't expect failures to occur often. Next, the recovering node copies the entire replica over the network. Then, it sends updates to the RG map to all other replicas in the group, which means that DDS libraries will start to lazily receive this update. Finally, it releases the write lock, which means that the previously failed operations will succeed on retry. The recovery of the partition is now complete, and the recovering node can begin recovery of other partitions as necessary.

There is an interesting choice of the rate at which partitions are transferred over the network during recovery. If this rate is fast, then the involved bricks will suffer a loss in read throughput during the recovery. If this rate is slow, then the bricks won't lose throughput, but the partition's mean time to recovery is increased. We chose to recover as quickly as possible, since in a large cluster only a small fraction of the total throughput of the cluster will be affected by the recovery.

A similar technique is used for DP map split and merge operations, except that all replicas must be modified and both the RG and DP maps are updated at the end of the operation.

4.4 Asynchrony

All components of the distributed hash table are built using an asynchronous, event-driven programming style. Each hash table layer is designed so that only a single thread ever executes in it at a time. This greatly simplified the implementation by eliminating the need for data locks and race conditions due to threads. The layers of the hash table are separated by FIFO queues, into which I/O completion events and I/O requests are placed. The FIFO discipline of these queues ensures fairness across requests, and the queues act as natural buffers that absorb bursts of requests that exceed the throughput capacity of the system.

All interfaces in the system, including the DDS library interfaces, are split-phase asynchronous interfaces. This means that a hash table `get()` operation doesn't block, but rather immediately returns with an identifier that can be matched up with a completion event that is delivered to a caller-specified upcall handler. This upcall handler can be application code, or it can be a queue that is polled or blocked upon.

5 Distributed Hash Table Performance

In this section of the paper, we present some performance metrics of the distributed hash table implementation. All of the performance benchmarks were gathered on a cluster of 28 2-way SMPs and 38 4-way SMPs (a total of 208 CPUs, each of which is a 500 MHz Pentium). The 2-way SMPs have 500 MB each of physical memory, and the 4-way SMPs have 1 GB each of physical memory. All are connected with either 100 Mb/s switched Ethernet (2-way SMPs) or 1 Gb/s switched Ethernet (4-way SMPs). The benchmarks are run using the Blackdown port of Sun's JDK 1.1.7 v3, using the OpenJIT 1.1.7 JIT compiler and "green" (user-level) threads on top of Linux v2.2.5. We ran at most one brick per CPU.

5.1 Throughput Scalability

This benchmark demonstrates that our hash table throughput scales linearly with the number of bricks.

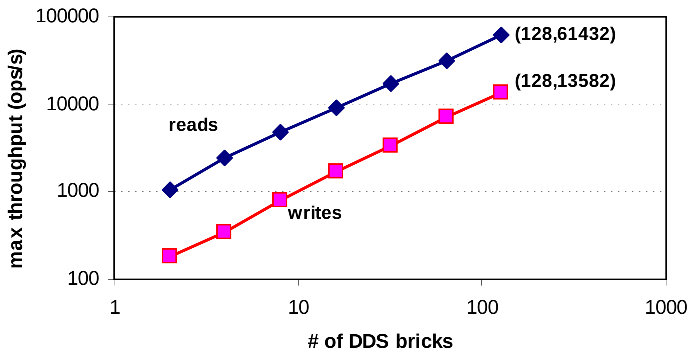


Figure 5: **Throughput scalability:** this benchmark shows the linear scaling of throughput as a function of the number of bricks serving in a distributed hash table; note that both axis have logarithmic scales. As we added more bricks to the DDS, we increased the number of clients using the DDS until throughput saturated.

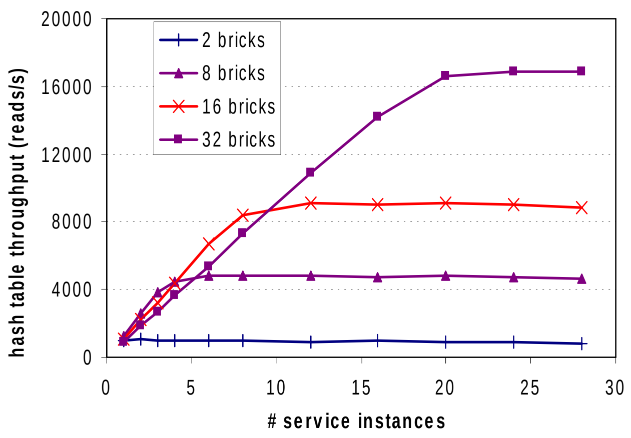


Figure 6: **Graceful degradation of reads:** this graph demonstrates that the read throughput from a distributed hash table remains constant even if the offered load exceeds the capacity of the hash table.

The benchmark consists of a number of services that each maintain a pipeline of 100 operations (either `gets()` or `puts()`) to a single distributed hash table. We vary the number of bricks in the hash table; for each configuration, we slowly increase the number of services interacting with the hash table, measuring the completion throughput flowing from the bricks. All configurations have 2 replicas per replica group, and each benchmark iteration consists of reads or writes of 150 byte values. We restrict the keys accessed by the benchmarks so that the working set of hash table values fits in the aggregate physical memory of all bricks; this is thus an in-core benchmark. The benchmark is closed-loop: a new operation is immediately issued with a random key for each completed operation.

Figure 5 shows the maximum throughput sustained by the distributed hash table as a function of the number of bricks. Throughput scales linearly up to 128 bricks; we didn't have enough processors to scale the benchmark further. The read throughput achieved with 128 bricks is 61,432 reads per second (5.3 billion per day), and the write throughput with 128 bricks is 13,582 writes per second (1.2 billion per day); this performance is adequate to serve the hit rates of most popular web sites on the Internet.

5.1.1 Graceful Degradation for Reads

Bursts of traffic are a common phenomenon for all Internet services. If a traffic burst exceeds the service's capacity, the service should have the property of "graceful degradation": the throughput of the service should remain constant, with the excess traffic either being rejected or absorbed in buffers and served with higher latency. Figure 6 shows the throughput of a distributed hash table as a function of the number of simultaneous read requests issued to it; each service instance has a closed-loop pipeline of 100 operations. Each line on the graph represents a different number of bricks serving the hash table. Each configuration is seen to eventually reach a maximum throughput as its bricks saturate. This maximum throughput is successfully sustained even as additional traffic is offered. The overload traffic is absorbed in the FIFO event queues of the bricks; all tasks are processed, but they experience higher latency as the queues drain from the burst.

5.1.2 Ungraceful Degradation for Writes

An unfortunate performance anomaly emerged when benchmarking the `put()` throughput of the hash table. As the write throughput approached the maximum capacity of the hash table bricks, the total write throughput suddenly began to drop. On closer examination, we discovered that most of the bricks in the hash table were nearly idle, but one brick in the hash table was completely saturated and had become the bottleneck in the closed-loop benchmark.

Figure 7 illustrates this imbalance. To generate figure 7, we issued write traffic to a hash table with a single partition and two replicas in its replica group. Each `put()` operation caused a two-phase commit across both replicas, and thus each replica saw the same set of network messages and performed the same computa-

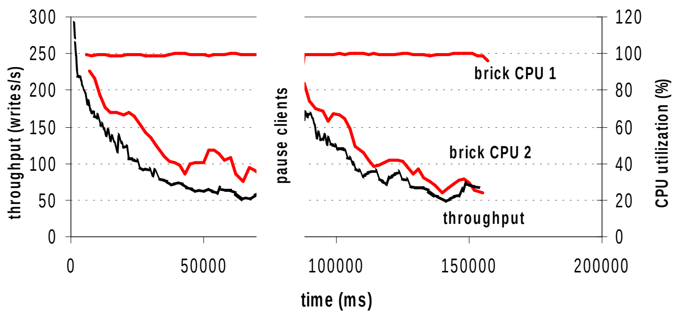


Figure 7: **Write imbalance leading to ungraceful degradation:** the bottom curve shows the throughput of a two-brick partition under overload, and the top two curves show the CPU utilization of those bricks. One brick is saturated, the other becomes only 30% busy.

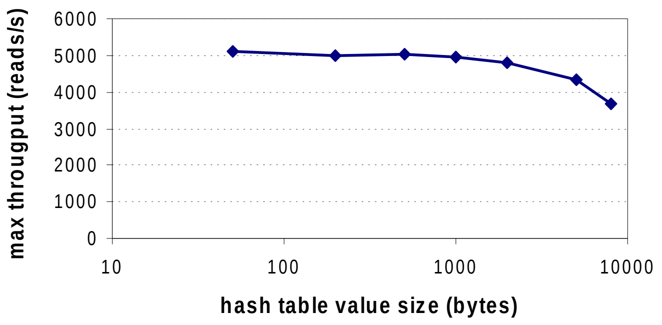


Figure 8: **Throughput vs. read size** the X axis of this graph represents the size of values read from the hash table, and the Y axis shows the maximum throughput sustained by an 8 brick hash table serving these values.

tion, but perhaps slightly out of order from each other. We thus expected both replicas to perform identically, but instead one replica became more and more idle, and the throughput of the hash table dropped to match the CPU utilization of this idle replica.

Investigation showed that the busy replica was spending all of its time garbage collecting. As more live objects populated that replica's heap, more time needed to be spent garbage collecting to reclaim a fixed amount of heap space since more objects would be examined before a free object was discovered. Random fluctuations in arrival rates and garbage collection behavior would cause one of the two replicas to spend more time garbage collecting than the other. This replica became the performance bottleneck of the system, and more operations would "pile up" in its queues, further amplifying this imbalance.

Write traffic particularly exacerbates the situation, as objects created by the "prepare" phase must wait for at least one network round-trip time before a commit or abort command in the second phase is received. The number of live objects in each bricks' heap is thus proportional to the bandwidth-delay product of hash table `put()` operations. For read traffic, there is only one phase, and thus objects can be garbage collected immediately after read requests are satisfied.

We experimented with many JDKs, but saw the same issue with all of them. Some JDKs (such as Sun's JDK 1.2.2 on Linux 2.2.5) developed this imbalance for read traffic as well as write traffic. We are exploring using admission control or early discard from bricks' queues to keep the bricks within their operational range, ameliorating this imbalance.

5.1.3 Throughput Bottlenecks

In figure 8, we fixed the number of bricks in the system at 8, but varied the size of elements that we read out of the hash table. The throughput of the hash table remained fairly flat from 50 bytes through 1000 bytes, but then began to degrade. From this we deduced that per-operation overhead (such as object creation, garbage collection, and system call overhead) saturated the bricks' CPUs for elements smaller than 1000 bytes, and per-byte overhead (byte array copies, either in the TCP stack or in the JVM) saturated the bricks' CPUs for elements greater than 1000 bytes. At 8000 bytes, the throughput in and out of each 2-way SMP (running 2 bricks) was 60 Mb/s. For larger sized hash table values, the 100 Mb/s switched network became the throughput bottleneck.

5.2 Availability and Recovery

To demonstrate the availability of the hash table in the face of node failures, and the ability for the bricks to recover after a failure, we repeated the read benchmark for a hash table full of 150 byte elements. The table was configured with a single 100 MB partition and three replica bricks in that partition's replica group. Figure 9 shows the throughput of the hash table over time as we induced a fault in one of the bricks, and also as we initiated recovery of that brick. During recovery, the rate at which the recovered partition is copied was limited to 12 MB/s; this number was chosen to match the maximum sequential bandwidth we could obtain from the disk on which the partition resided.

At point (1), all three bricks were operational and the throughput sustained by the hash table was 450 operations per second. At point (2), one of the three

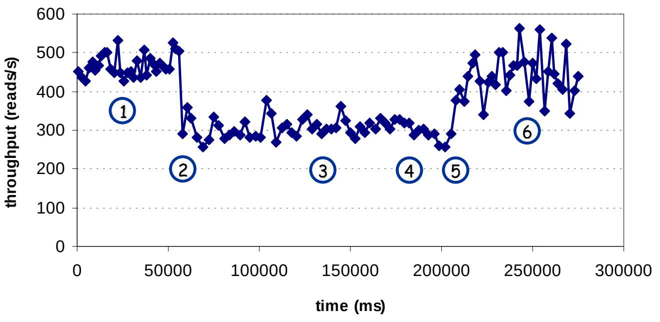


Figure 9: **Availability and Recovery:** this benchmark shows the read throughput of a 3-brick hash table as a deliberate single-node fault is induced, and afterwards as recovery is performed.

bricks was killed. Performance immediately dropped to 300 operations per second (two-thirds of the original capacity); the fault detection was immediate, and the performance overhead of the replica group map updates could not be observed. At point (3), recovery was initiated, and recovery completed at point (4). Between points (3) and (4), there was no noticeable performance overhead of recovery; this is because there was ample excess bandwidth on the network, and the CPU overhead of transferring the partition during recovery was negligible.

After recovery completed, performance briefly dropped at point (5). This performance degradation corresponds to the warming of the buffer cache on the recovered node. Once the cache became warm, performance resumed to its previous throughput of 450 operations per second at point (6). An interesting anomaly at point (6) is the presence of noticeable oscillations in throughput; these oscillations were traced to garbage collection triggered by the "extra" activity of recovery. When we repeated our measurements, we would occasionally see this oscillation at other times besides immediately post-recovery. Performance unpredictability due to garbage collection seems to be a pervasive problem; probabilistic admission control at the DDS libraries is a potential fix to this problem, but we have yet to implement this.

6 Example Services

We have implemented a number of interesting services that make use of our distributed hash table. In all cases, the implementation of the service was greatly simplified by using the DDS; the service logic was simplified, and the service trivially scaled by adding more service instances. One aspect of scalability not covered by using the hash table was in routing and load balancing of WAN client requests across service instances; this issue is beyond the scope of this work.

Sanctio: Sanctio is an instant messaging gateway that provides protocol translation between popular instant messaging protocols (such as Mirabilis' ICQ and AOL's AIM), conventional email, and voice messaging over cellular telephones. Sanctio acts as a middleman between all of these messaging protocols, routing and translating messages between the networks. In addition to protocol translation, Sanctio also can transform the content of messages. We have built a "web scraper" that allows us to compose AltaVista's BabelFish natural language translation service with Sanctio. We can thus perform language translation (such as English to French) as well as protocol translation; a Spanish speaking ICQ user can send a message to an English speaking AIM user, with Sanctio providing both language and protocol translation.

A user of the service may be reached on a number of different addresses, one for each of the networks that Sanctio can communicate with. The Sanctio service must therefore keep a large table of bindings between users and their current transport addresses on these networks. We used the distributed hash table for this purpose. The expected workload on the DDS includes significant write traffic as users change networks or log in and out of a network, and the data in the table must be kept consistent (otherwise messages will be routed to the wrong address).

Sanctio took approximately 1 person-month to develop, and most of that time was spent authoring the protocol translation code. The code that interacts with the distributed hash table took less than a day to write.

Scalable web server: we have implemented a scalable web server using the distributed hash table. The server speaks HTTP to web clients, hashes requested URLs into 64 bit keys, and requests those keys from the hash table. The server takes advantage of the event-driven, queue-centric programming style to interpose on the URL resolution path in order to introduce CGI-like behavior. This web server was written in 900 lines of Java, 750 of which deals with HTTP parsing and URL resolution, and only 50 of which deals with interacting with the hash table DDS.

Others: We built many other interesting services as part of the overall Ninja project[2]. The "Parallelisms" service recommends related web sites to user-specified URLs by looking up ontological entries in an inversion

[2]`http://ninja.cs.berkeley.edu/`

of the Yahoo web directory. We built a collaborative filtering engine for a digital music jukebox service [13]; this engine stores all of its users' music preferences in a distributed hash table. Similarly, we implemented a private key store and a composable user preference service, both of which use the distributed hash table for persistent state management.

7 Discussion

Our experience with the distributed hash table implementation has taught us many lessons about using it as a storage platform for scalable services. The hash table was a resounding success in simplifying the construction of interesting services, and these services inherited the scalability, availability, and data consistency of the hash table. Exploiting properties of clusters also proved to be remarkably useful. In our experience, most of the assumptions that we made regarding properties of a clusters and component failures (specifically the fail-stop behavior of our software and the probabilistic lack of network partitions in the cluster) were valid in practice.

One of our assumptions was initially problematic: we observed a case where there was a systematic failure of all replica group members inside a single replica group. This failure was caused by a software bug that enabled service instances to deterministically crash remote bricks by inducing a null pointer exception in the JVM. After fixing the associated bug in the brick, this situation never again arose. However, it serves as a reminder that systematic software bugs can in practice bring down the entire cluster at once. Careful software engineering and a good quality assurance cycle can help to ameliorate this failure mode, but we believe that this issue is fundamental to all systems that promise both availability and consistency.

As we scaled our distributed hash table, we noticed scaling bottlenecks that weren't associated with our own software. As we scaled to 128 bricks, we approached the point at which the 100 Mb/s Ethernet switches would saturate; upgrading to 1 Gb/s switches throughout the cluster would delay the onset of saturation. We also noticed that the combination of our JVM's user-level threads and the Linux kernel began to induced poor scaling behavior as each node in the cluster opened up a reliable TCP connection to all other nodes in the cluster. The brick processes began to saturate due to a flood of interrupts associated with TCP connections that had data waiting to be read.

7.1 Java as a Service Platform

We found that Java was an adequate platform from which to build a scalable, high performance subsystem. However, we ran into a number of serious issues with the Java language and runtime. The garbage collector of all JVMs that we experimented with inevitably became the ultimate performance bottleneck of the bricks and also a source of high throughput and latency variation. Whenever the garbage collector became active on a brick, it had a serious impact on all other system activity; unfortunately, current JVMs do not provide adequate interfaces to allow systems to control garbage collection behavior.

The type safety and array bounds checking features of Java vastly accelerated our software engineering process, and helped us to write stable, clean code. However, these features got in the way of writing efficient code, especially when dealing with multiple layers of a system each of which wraps some array of data with layer-specific metadata. We found ourselves needing to perform copies of regions of byte arrays, whereas in a C implementation we would have been able to exploit pointers into `malloc'ed` memory regions to the same effect without needing copies.

Java also lacks asynchronous I/O primitives, which necessitated the use of a thread pool at the lowest-layer of the system. This thread pool is much more efficient than a thread-per-task system, since the number of threads in the system is equal to the number of outstanding I/O requests rather than the number of tasks. Nonetheless, it introduced performance overhead and scaling problems, since the number of TCP connections required per brick node increased with the cluster size. We are working on introducing high-throughput asynchronous I/O completion mechanisms into the JVM using the JNI native interface feature.

7.2 Future Work

In the future, we plan on investigating more interesting data-parallel operations on a DDS (such as a hash table iterator, or the `maplist()` operator from Lisp). We also plan on building other distributed data structures, including a B-tree and an administrative log; in doing so, we hope to reuse many of the components of the distributed hash table (such as the brick storage layer, the RG map infrastructure, and the two-phase commit code). We also have yet to explore automatic caching in the DDS libraries, but instead rely on services to build their own application level caches. Finally, we are currently exploring adding other interesting single-element operations to the hash table such as testandset() in order to provide global lock leases to services that may have many service instances competing to write to the same hash table element.

8 Related Work

Litwin et al.'s families of scalable, distributed data structures (SDDS) such as RP^* [17, 21] helped to motivate our own work. The RP^* work focuses on algorithmic properties, while we focused on the systems issues of implementing a persistent distributed hash table that satisfies the concurrency, availability, and incremental scalability needs of Internet services.

Our work has a great deal in common with database research. The problems of partitioning and replicating data across shared-nothing multicomputers has been studied extensively in both the distributed and parallel database communities [7, 14, 20]. We make use of mechanisms such as horizontal partitioning and two-phase commits, but we do not need an SQL parser or a query optimization layer since we have no general-purpose queries in our system.

We also have much in common with distributed and parallel file systems [3, 18, 23, 25]. Our DDS's present a higher level interface than a typical file system, and DDS operations are data-structure specific and atomically affect entire elements. Our research has focused on achieving scalability, availability, and consistency under high throughput and highly concurrent traffic, which is a different focus than these systems. Our work is perhaps most similar to the Petal [19] distributed virtual disk project, in that a Petal virtual disk can be thought of as a simple hash table with fixed sized elements; our hash tables have variable sized elements, have an additional name space (the set of hash tables), and focus on Internet service workloads and properties as opposed to file system workloads and properties.

The CMU etwork attached secure disk (NASD) storage architectures [8] explore variable-sized object interfaces as an abstraction to allow storage subsystems to optimize on-disk layout. This philosophy is similar to our own data structure interface, which is deliberately higher level than the block or file interfaces of Petal and parallel or distributed file systems.

Distributed object stores [10] attempt to solve a related problem, namely transparently adding persistence to distributed object systems. The persistence of (typed) objects in such systems is typically determined by reachability through the transitive closure of object references, and the removal of objects is handled by garbage collection. A DDS has no notion of pointers or object typing, and applications must explicitly use API operations to store and retrieve elements from a DDS store. Distributed object stores are also usually built with the wide-area in mind, and thus do not focus on the scalability, availability, and high throughput requirements of cluster-based Internet services.

Many projects have explored the use of clusters of workstations as a general-purpose platform for building Internet services [1, 4, 12]. To date, these platforms rely on file systems or databases for persistent state management; our DDS's are meant to augment such platforms with a state management platform that is better suited to the needs of Internet services. The Porcupine project [22] includes a storage platform built specifically for the needs of a cluster-based scalable mail server, but they are attempting to generalize their storage platform for arbitrary service construction.

9 Conclusions

This paper presents a new persistent data management layer that enhances the ability of clusters to support Internet services. This self-managing layer, called a distributed data structure (DDS), fills in an important gap in current cluster platforms by providing a data storage platform specifically tuned for services' workloads and for the cluster environment.

This paper focused on the design and implementation of a distributed hash table DDS, empirically demonstrating that it has many properties necessary for Internet services (incremental scaling of throughput and data capacity, fault tolerance and high availability, high concurrency, and consistency and durability of data). These properties were achieved by carefully designing the partitioning, replication, and recovery techniques in the hash table implementation to exploit features of cluster environments (such as a low-latency network with a lack of network partitions). By doing so, we have "right-sized" the DDS to the problem of persistent data management for Internet services.

The hash table DDS simplifies Internet service construction by decoupling service-specific logic from the complexities of persistent state management, and by allowing services to inherit the necessary service properties from the DDS rather than having to implement the properties themselves.

References

[1] Elan Amir, Steven McCanne, and Randy Katz. An Active Service Framework and its Application to Real-Time Multimedia Transcoding. In *Proceedings of ACM SIGCOMM '98*, pages 178–189, October 1998.

[2] T. E. Anderson, D. E. Culler, and D. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 12(1):54–64, February 1995.

[3] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[4] D. Andresen, T. Yang, O. Egecioglu, O. H. Ibarra, and T. R. Smith. Scalability Issues for High Performance Digital Libraries on the World Wide Web. In *Proceedings of IEEE ADL '96*, Washington D.C., May 1996.

[5] T. Brisco. RFC 1764: DNS Support for Load Balancing, April 1995.

[6] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 Usenix Annual Technical Conference*, January 1996.

[7] D. DeWitt et al. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[8] G. A. Gibson et al. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, California, 1998.

[9] J. H. Howard et al. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[10] P. Ferreira et al. PerDiS: Design, Implementation, and Use of a PERsistent DIstributed Store. In *Recent Advances in Distributed Systems*, volume 1752 of *Lecture Notes in Computer Science*, chapter 18, pages 427–452. Springer-Verlag, February 2000.

[11] V. S. Pai et al. Locality-Aware Request Distribution in Cluster-Based Network Servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, CA, Oct 1998.

[12] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, October 1997.

[13] I. Goldberg, S. D. Gribble, D. Wagner, and E. A. Brewer. The Ninja Jukebox. In *Submitted to the 2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, Colorado, USA, October 1999.

[14] Goetz Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *ACM SIGMOD Conference on the Management of Data*, Atlantic City, NJ, USA, May 1990.

[15] Jim Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings of VLDB*, Cannes, France, September 1981.

[16] S. D. Gribble and E. A. Brewer. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems (USITS 97)*, Monterey, California, USA, December 1997.

[17] J. S. Karlsson, W. Litwin, and T. Risch. LH*LH: A Scalable High Performance Data Structure for Switched Multicomputers. In *Proceedings of the 5th International Conference on Extending Database Technology*, pages 573–591, Avignon, France, March 1996.

[18] O. Krieger and M. Stumm. HFS: A Flexible File System for Large-Scale Multiprocessors. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 6–14, Hanover, NH, June 1993.

[19] E. K. Lee and C. A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Cambridge, MA, 1996.

[20] B. G. Lindsay. A Retrospective of R*: A Distributed Database Management System. *Proceedings of the IEEE*, 75(5):668–673, May 1987.

[21] W. Litwin, M. Neimat, and D. A. Schneider. RP*: A Family of Order Preserving Scalable Distributed Data Structures. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 342–353, Santiago, Chile, 1994.

[22] Y. Saito, B. Bershad, and H. Levy. Manageability, Availability and Performance in Porcupine: a Highly Scalable, Cluster-based Mail Service. In *Proceedings of the 17th Symposium on Operating System Principles*, Kiawah Island, SC, December 1999.

[23] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the USENIX 1985 Summer Conference*, El Cerrito, CA, USA, June 1985.

[24] BEA Systems. BEA WebLogic Application Servers. http://www.bea.com/products/weblogic/.

[25] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, October 1997.