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 presents a new persistent data management layer 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 a cluster. We have designed and implemented 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, consistency, and durability). The hash table uses two-phase commits to present a coherent view of its data across all cluster nodes, allowing any node to service any task. We show that the distributed hash table simplifies Internet service construction by decoupling service-specific logic from the complexities of persistent, consistent state management, and by 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, 1 terabyte of storage, and an in-core read throughput of 61,432 operations/s and write throughput of 13,582 operations/s.

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, and also to safeguard and provide access to their personal data (such as email and calendar entries). 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, strictly maintaining the 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 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.

Many projects and products propose using software platforms on clusters to address these challenges and to simplify Internet service construction [1, 2, 6, 15]. These platforms typically 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, the service properties, and 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 set 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 one such distributed data structure (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 [15], it is argued that clusters of workstations (commodity PC's with a high-performance network) are a natural platform for Internet services. Each cluster node is an independent failure boundary, which means that replicating computation and data can provide fault tolerance. A cluster permits incremental scalability: if a service runs out of capacity, a good software architecture allows nodes to be added to the cluster, linearly increasing the service's capacity. A cluster has natural parallelism: 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) that can achieve 1 Gb/s throughput with 10 to 100 μs latency.

1.2 Internet Service Workloads

Popular Internet services process hundreds of millions of tasks per day. A task is usually "small", causing a small amount of data to be transferred and computation to be performed. For example, according to 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's web proxy cache (`http://www.aol.com`) handles 5.2 billion web requests per day, with an average response size of 5.5 KB. 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 [19]. 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.

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 mechanisms used to access, partition, replicate, scale, and recover data. Because these complex mechanisms are hidden behind the simple DDS interface, authors only need to worry about service-specific logic when implementing a new service. The difficult issues of managing persistent state are handled by the DDS platform.

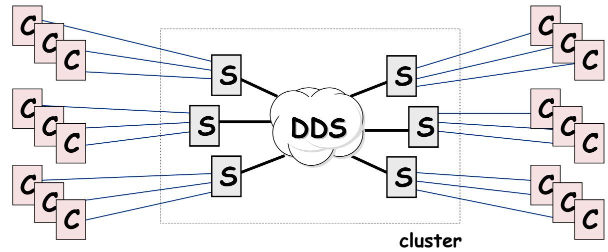

Figure 1: **High-level view of a DDS:** a DDS is a self-managing, cluster-based data repository. All service instances (S) in the cluster see the same consistent image of the DDS; as a result, any WAN client (C) can communicate with any service instance.

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.

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 implementation choices that are made based on its expected runtime environment and the types of failures that it should 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 [18], but these ACID properties can come at high cost in terms of complexity and overhead. As a result, 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 [15, 21, 32]. RDBMS's offer a high degree of data independence, which is a powerful abstraction that adds additional complexity and performance overhead. The many layers of most RDBMS's (such as SQL parsing, query optimization, access path selection, etc.) permit users to 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, but data independence can make parallelization (and therefore scaling) hard in the general case. 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.

Distributed file systems: file systems have less strictly defined consistency models. Some (e.g., NFS [31]) have weak consistency guarantees, while others (e.g., Frangipani [33] or AFS [12]) guarantee a coherent filesystem image across all 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 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 its elements 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 commits are used to keep replicas coherent, and thus all clients see the same image of a 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 single-element updates and coherence provided by our current DDS are strong enough to support interesting services.

A DDS's interface is more structured and at a higher level than that of 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 query parsing and optimization stages of an RDBMS are completely obviated in a DDS, but the DDS interface is less flexible and offers less data independence.

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 with all of the service properties. It has been our experience that the DDS interfaces, although not as general as SQL, are rich enough to successfully build sophisticated services.

3 Assumptions and Design Principles

In this section of the paper, we present the 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 the DDS can handle, and the workloads it will receive.

Separation of concerns: the clean separation of service code from storage management simplifies system architecture by decoupling the complexities of state management from those of service construction. Because persistent service state is kept in the DDS, service instances can crash (or be gracefully shut down) and restart without a complex recovery process. This greatly simplifies service construction, as authors need only worry about service-specific logic, and not the complexities of data partitioning, replication, and recovery.

Appeal to properties of clusters: in addition to the properties listed in section 1.1, we require that our cluster is physically secure and well-administered. 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, its low latency SAN (10-100 μs instead of 10-100 ms for the wide-area Internet) means that two-phase commits are not prohibitively expensive. The SAN's high redundancy means that the probability of a network partition can be made arbitrarily small, and thus we need not consider partitions in our protocols. An uninterruptible power supply (UPS) and good system administration help to ensure that the probability of system-wide simultaneous hardware failure is extremely low; we can thus rely on data being available in more than one failure boundary (i.e., the physical memory or disk of more than one node) while designing our recovery protocols.[1]

Design for high throughput and high concurrency: given the workloads presented in section 1.2, the control structure used to effect concurrency is critical. Techniques often used by web servers, such as process-per-task or thread-per-task, do not scale to our needed degree of concurrency. Instead, we use asynchronous, event-driven style of control flow in our DDS, similar to that espoused by modern high performance servers [5, 20] such as the Harvest web cache [8] and Flash web server [28]. A convenient side-effect of this style is that layering is inexpensive and flexible, as layers can be constructed by chaining together event handlers. Such chaining also facilitates interposition: a "middleman" event handler can be easily and dynamically patched between two existing handlers. In addition, 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 are absorbed by additional threads.

3.1 Assumptions

If one DDS node cannot communicate with another, we assume it is because this other node has stopped executing (due to a planned shutdown or a crash); we assume that network partitions do not occur inside our cluster, and that DDS software components are fail-stop. The need for no network partitions is addressed by the high redundancy of our network, as previously mentioned. 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 replica is active (has not failed) at all times. We also assume some degree of synchrony, in that processes take a bounded amount of time to execute tasks, and that messages take a bounded amount of time to be delivered.

We make several assumptions about the workload presented to our distributed hash tables. A 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 a given key exists in the table is a function of the number of values in the table, but not of the particular key). We don't assume that all keys are accessed equiprobably, but rather that the "working set" of hot keys is 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. Our current DDS design does not gracefully handle a small number of extreme hotspots (i.e., if a handful of keys receive most of the workload). If there are many such hotspots, however, then our partitioning strategy will probabilistically balance them across the cluster. Failure of these workload assumptions can result in load imbalances across the cluster, leading to a reduction in throughput.

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.

[1] We do have a checkpoint mechanism (discussed later) that permits us to recover in the case that any of these cluster properties fail, however all state changes that happen after the last checkpoint will be lost should this occur.

4 Distributed Hash Tables: Architecture and Implementation

In this section, we present the architecture and implementation of a distributed hash table DDS. Figure 2 illustrates our hash table's architecture, which consists of the following components:

Client: a client consists of service-specific software running on a client machine that 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 [7], a service-specific protocol, or level 4 or level 7 load-balancing switches 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 of the DDS system runs on a client.

Service: a service is a set of cooperating software processes, each of which we call a service instance. Service instances communicate with wide-area clients and perform some application-level function. Services may 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 hash table API is the boundary between a service instance and its "DDS library". The API provides services with `put()`, `get()`, `remove()`, `create()`, and `destroy()` operations on hash tables. Each operation is atomic, and all services see the same coherent image of all exist-

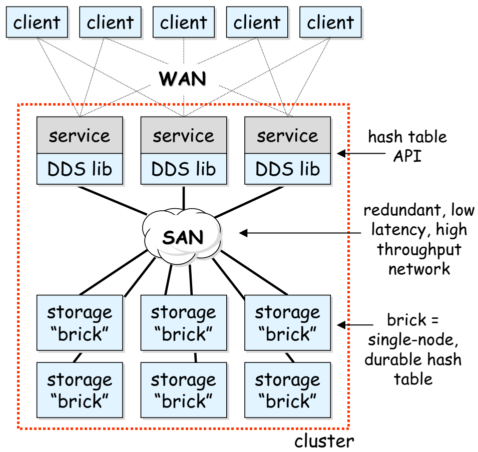

Figure 2: **Distributed hash table architecture:** each box in the diagram represents a software process. In the simplest case, each process runs on its own physical machine, however there is nothing preventing processes from sharing machines.

ing hash tables through this API. Hash table names are strings, hash table keys are 64 bit integers, and hash table values are opaque byte arrays; operations affect hash table values in their entirety.

DDS library: the DDS library is a Java class library that presents the hash table API to services. The library accepts hash table operations, and cooperates with the "bricks" to realize those operations. The library contains only soft state, including metadata about the cluster's current configuration and the partitioning of data in the distributed hash tables across the "bricks". The DDS 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 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. Bricks may run on dedicated nodes, or they may share nodes with other components.

4.1 Partitioning, Replication, and Replica Consistency

A distributed hash table provides incremental scalability of throughput and data capacity as more nodes are added to the cluster. To achieve this, we horizontally partition tables to spread operations and data across bricks. Each brick 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.

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 cluster node. The set of replicas for a partition form a *replica group*; all replicas in the group are kept strictly coherent with each other. Any replica can be used to service a `get()`, but all replicas must be updated during a `put()` or `remove()`. If a node fails, the data from its partitions is available on the surviving members of the partitions' replica groups. Replica group membership is thus dynamic; when a node fails, all of its replicas 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, described later).

To maintain consistency when state changing operations (`put()` and `remove()`) are issued against a partition, all replicas of that partition must be synchronously updated. We use an optimistic two-phase commit protocol to achieve consistency, with the DDS library serving as the commit coordinator and the replicas serving as the participants. If the DDS library crashes after *prepare* messages are sent, but before any *commit* messages are sent, the replicas will time out and abort the operation.

However, if the DDS library crashes after sending out any *commits*, then all replicas must commit. For the sake of availability, we do not rely on the DDS library to recover after a crash and issuing pending *commits*. Instead, replicas store short in-memory logs of recent state changing operations and their outcomes. If a replica times out while waiting for a *commit*, that replica communicates with all of its peers to find out if any have received a *commit* for that operation, and if so, the replica commits as well; if not, the replica aborts. Because all peers in the replica group that time out while waiting for a commit communicate with all other peers, if any receives a commit, then all will commit.

Any replica may abort during the first phase of the two-phase commit (e.g., if the replica cannot obtain a write lock on a key). 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.

If a replica crashes during a two-phase commit, the DDS library simply removes it from its replica group and continues onward. Thus, all replica groups shrink over time; we rely on a recovery mech-

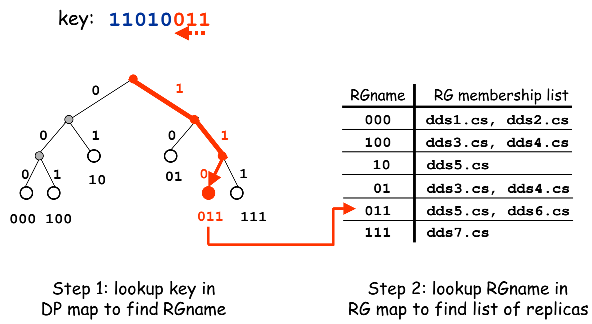

Figure 3: **Distributed hash table metadata maps:** this illustration highlights the steps taken to discover the set of replica groups which serve as the backing store for a specific hash table key. The key is used to traverse the DP map trie and retrieve the name of the key's replica group. The replica group name is then used looked up in the RG map to find the group's current membership.

anism (described later) for crashed replicas to rejoin the replica group. We made the significant optimization that the image of each replica must only be consistent 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 to ensure on-disk consistency. An implication of this is that if all members of a replica group crash, that partition is lost. We assume nodes are independent failure boundaries (section 3.1); there must be no systematic software failure across nodes, and the cluster's power supply must be uninterruptible.

Our two-phase commit mechanism gives *atomic updates* to the hash table. It does not, however, give transactional updates. If a service wishes to update more than one element atomically, our DDS does not provide any help. Adding transactional support to our DDS infrastructure is a topic of future work, but this would require significant additional complexity such as distributed deadlock detection and undo/redo logs for recovery.

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. We currently rely on system administrators to decide when to initiate checkpoints.

4.2 Metadata maps

To find the partition that manages a particular hash table key, and to determine the list of replicas in partitions' replica groups, 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 key's partition. The DP map thus controls the horizontal partitioning of data across the bricks. As shown in figure 3, the DP map is a trie over 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 3 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 3 could be merged into a single partition `00`.

The second map is called the *replica group (RG) membership map*. Given a partition name, the RG map returns a list of bricks that are currently serving as replicas in the partition's replica group. The RG maps are dynamic: if a brick fails, it is removed from all RG maps that contain it. A brick joins a 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.

The maps are replicated on each cluster node, 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 libraries' maps to drift out of date, but lazily update them when they are used to perform operations. The DDS library piggybacks hashes of the maps[2] 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. Because of this mechanism, 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 consults its DP map to determine the correct partition for the key. It then looks up that partition name in its RG map to find the current set of bricks serving as replicas, and finally performs a two-phase commit across these replicas. To do a `get()` of a key, a similar process is used, except that the DDS library can

[2]It is important to use large enough of a hash to make the probability of collision negligible; we currently use 32 bits.

select any of the replicas listed in the RG map to service the read. We use the locality-aware request distribution (LARD) technique [14] to select a read replica—LARD further partitions keys across replicas, in effect aggregating their physical caches.

4.3 Recovery

If a brick fails, all replicas on it become unavailable. Rather than making these partitions unavailable, we remove the failed brick from all replica groups and allow operations to continue on the surviving replicas. When the failed brick recovers (or an alternative brick is selected to replace it), it 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 for vast simplifications.

Firstly, 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 (e.g., 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 system logic, since we don't worry about correctly handling operations in these rare situations. Instead, we rely on the DDS library (or, ultimately, the service and perhaps even the WAN client) to retry the operation. Secondly, 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, operations fail until the maps are reconciled.

We make our partitions relatively small (~100MB), which 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 recovery. When a node initiates recovery, it grabs a write lease on one replica group member from the partition that it is joining; this write lease means that all state-changing operations on that partition will start to fail. 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 will increase. 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.

4.3.1 Convergence of Recovery

A challenge for fault-tolerant systems is to remain consistent in the face of repeated failures; our recovery scheme described above has this property. In steady state operation, all replicas in a group are kept perfectly consistent. During recovery, state changing operations fail (but only on the recovering partition), implying that surviving replicas remain consistent and recovering nodes have a stable image from which to recover. We also ensure that a recovering node only joins the replica group after it has successfully copied over the entire partition's data but before it release its write lease. A remaining window of vulnerability in the system is if recovery takes longer than the write lease; if this seems imminent, the recovering node could aggressively renew its write lease, but we have not currently implemented this behavior.

If a recovering node crashes during recovery, its write lease will expire and the system will continue as normal. If the replica on which the lease was grabbed crashes, the recovering node must reinitiate recovery with another surviving member of the replica group. If all members of a replica group crash, data will be lost, as mentioned in Section 3.1.

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 implementation by eliminating the need for data locks, and race conditions due to threads. Hash table layers 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 that exceed the system's throughput capacity.

All interfaces in the system (including the DDS library APIs) are split-phase and asynchronous. This means that a hash table `get()` doesn't block, but rather immediately returns with an identifier

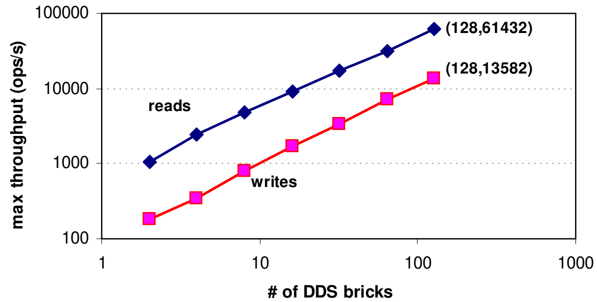

Figure 4: **Throughput scalability:** this benchmark shows the linear scaling of throughput as a function of the number of bricks serving in a distributed hash table; note that both axis have logarithmic scales. As we added more bricks to the DDS, we increased the number of clients using the DDS until throughput saturated.

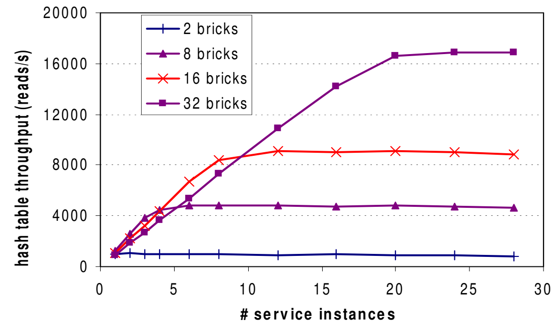

Figure 5: **Graceful degradation of reads:** this graph demonstrates that the read throughput from a distributed hash table remains constant even if the offered load exceeds the capacity of the hash table.

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 Performance

In this section, we present performance benchmarks of the distributed hash table implementation that were gathered on a cluster of 28 2-way SMPs and 38 4-way SMPs (a total of 208 500 MHz Pentium CPUs). Each 2-way SMP has 500 MB of RAM, and each 4-way SMP has 1 GB. 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 Sun's JDK 1.1.7v3, using the OpenJIT 1.1.7 JIT compiler and "green" (user-level) threads on top of Linux v2.2.5.

When running our benchmarks, we evenly spread hash table bricks amongst 4-way and 2-way SMPs, running at most one brick node per CPU in the cluster. Thus, 4-way SMPs would have at most 4 brick processes running on them, while 2-way SMPs would have at most 2. We also made use of these cluster nodes as load generators; because of this, we were only able to gather performance numbers to a maximum of a 128 brick distributed hash table, as we needed the remaining 80 CPUs to generate enough load to saturate such a large table.

5.1 In-Core Benchmarks

Our first set of benchmarks tested the in-core performance of the distributed hash table. By limiting the working set of keys that we requested to a size that fits in the aggregate physical memory of the bricks, this set of benchmarks investigates the overhead and throughput of the distributed hash table code independently of disk performance.

5.1.1 Throughput Scalability

This benchmark demonstrates that hash table throughput scales linearly with the number of bricks. The benchmark consists of several services that each maintain a pipeline of 100 operations (either `gets()` or `puts()`) to a single distributed hash table. We varied the number of bricks in the hash table; for each configuration, we slowly increased the number of services and measured the completion throughput flowing from the bricks. All configurations had 2 replicas per replica group, and each benchmark iteration consisted of reads or writes of 150-byte values. The benchmark was closed-loop: a new operation was immediately issued with a random key for each completed operation.

Figure 4 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.

5.1.2 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 5 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

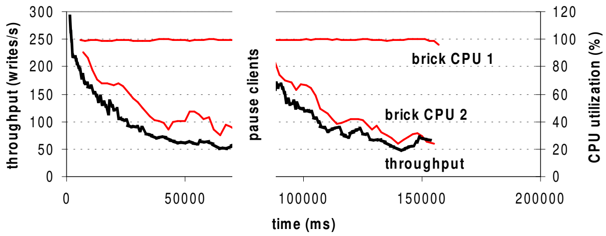

Figure 6: **Write imbalance leading to ungraceful degradation:** the bottom curve shows the throughput of a two-brick partition under overload, and the top two curves show the CPU utilization of those bricks. One brick is saturated, the other becomes only 30% busy.

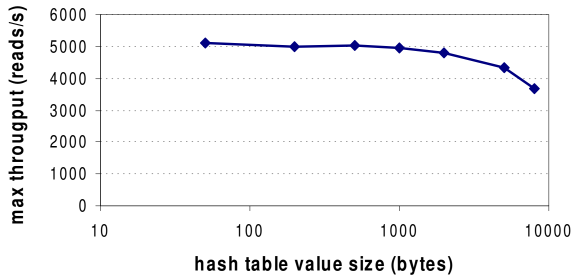

Figure 7: **Throughput vs. read size** the X axis shows the size of values read from the hash table, and the Y axis shows the maximum throughput sustained by an 8 brick hash table serving these values.

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.

5.1.3 Ungraceful Degradation for Writes

An unfortunate performance anomaly emerged when benchmarking `put()` throughput. As the offered load 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 unloaded, but one brick in the hash table was completely saturated and had become the bottleneck in the closed-loop benchmark.

Figure 6 illustrates this imbalance. To generate it, we issued `puts()` 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 computation (but perhaps in slightly different orders). We 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 a significant amount of time in garbage collection. 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, as more objects would be examined before a free object was discovered. Random fluctuations in arrival rates and garbage collection caused one replica to spend more time garbage collecting than the other. This replica became the system bottleneck, and more operations piled up in its queues, further amplifying this imbalance. Write traffic particularly exacerbated 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 consistently saw this effect. Some JDKs (such as JDK 1.2.2 on Linux 2.2.5) developed this imbalance for read traffic as well as write traffic. This sort of performance imbalance is fundamental to any system that doesn't perform admission control; if the task arrival rate temporarily exceeds the system's ability to handle them, then tasks will begin to pile up in the system. Because systems have finite resources, this inevitably causes performance degradation (thrashing). In our system, this degradation first materialized due to garbage collection. In other systems, this might happen due to virtual memory thrashing, to pick an example. We are currently exploring using admission control (at either the bricks or the hash table libraries) or early discard from bricks' queues to keep the bricks within their operational range, ameliorating this imbalance.

5.1.4 Throughput Bottlenecks

In figure 7, we varied the size of elements that we read out of an 8 brick hash table. Throughput was 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 Out-of-core Benchmarks

Our next set of benchmarks tested performance for workloads that do not fit in the aggregate physical memory of the bricks. These benchmarks stress the single-node hash table's disk interaction, as well as the performance of the distributed hash table.

5.2.1 A Terabyte DDS

To test how well the distributed hash table scales in terms of data capacity, we populated a hash table with 1.28 terabytes of 8KB data elements. To do this, we created a table with 512 partitions in its DP map, but with only 1 replica per replica group (i.e., the table would not withstand node failures). We spread the 512 partitions across 128 brick nodes, and ran 2 bricks per node in the cluster. Each brick stored its data on a dedicated 12GB disk (all cluster nodes have 2 of these disks). The bricks each used 10GB worth of disk capacity, resulting in 1.28TB of data stored in the table.

To populate the 1.28TB hash table, we designed bulk loaders that generated writes to keys in an order that was carefully chosen to result in sequential disk writes. These bulk loaders understood the partitioning in the DP map and implementation details about the single-node tables' hash functions (which map keys to disk blocks). Using these loaders, it took 130 minutes to fill the table with 1.28 terabytes of data, achieving a total write throughput of 22,015 operations/s, or 1.4 MB/s per disk.

Comparatively, the in-core throughput benchmark presented in Section 5.1.1 obtained 13,582 operations/s for a 128 brick table, but that benchmark was configured with 2 replicas per replica group. Eliminating this replication would double the throughput of the in-core benchmark, resulting in a 27,164 operations/s. The bulk loading of the 1.28TB hash table was therefore only marginally slower in terms of the throughput sustained by each replica than the in-core benchmarks, which means that disk throughput was not the bottleneck.

5.2.2 Random Write and Read Throughput

However, we believe it is unrealistic and undesirable for hash table clients to have knowledge of the DP map and single-node tables' hash functions. We ran a second set of throughput benchmarks on another 1.28TB hash table, but populated it with random keys. With this workload, the table took 319 minutes to populate, resulting in a total write throughput of 8,985 operations/s, or 0.57 MB/s per disk. We similarly sustained a read throughput of 14,459 operations/s, or 0.93 MB/s per disk.[3]

This throughput is substantially lower than the throughput obtained during the in-core benchmarks because the random workload generated results in random read and write traffic to each disk. In fact, for this random workload, every `read()` issued to the distributed hash table results in a request for a random disk block from a disk. All disk traffic is seek dominated, and disk seeks become the overall bottleneck of the system.

We expect that there will be significant locality in DDS requests generated by Internet services, and given workloads with high locality, the DDS should perform nearly as well as the in-core benchmark results. However, it might be possible to significantly improve the write performance of traffic with little locality by using disk layout techniques similar to those of log-structured file systems [29]; we have not explored this possibility as of yet.

5.3 Availability and Recovery

To demonstrate availability in the face of node failures and the ability for the bricks to recover after a failure, we repeated the read benchmark with a hash table of 150 byte elements. The table was configured with a single 100MB partition and three replicas in that partition's replica group. Figure 8 shows the throughput of the hash table over time as we induced a fault in one of the replica bricks and later initiated its recovery. During recovery, the rate at which the recovered partition is copied was 12 MB/s, which is maximum sequential write bandwidth we could obtain from the bricks' disks.

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 bricks was killed. Performance immediately dropped to 300 operations per second, two-thirds of the original capacity. Fault detection was immediate: client libraries experienced broken transport connections that could not be reestablished. 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. It should be noted that between points (3) and (4), the recov-

[3] Write throughput is less than read throughput because a hash bucket must be read before it can written, in case there is already data stored in that bucket that must be preserved. There is therefore an additional read for every write, nearly halving the effective throughput for DDS writes.

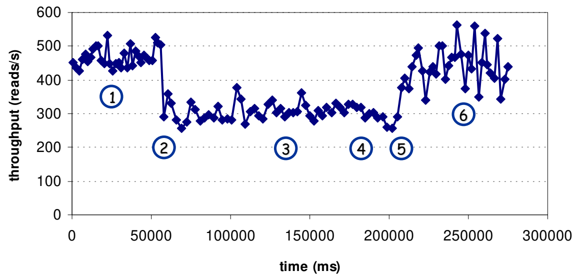

Figure 8: **Availability and Recovery:** this benchmark shows the read throughput of a 3-brick hash table as a deliberate single-node fault is induced, and afterwards as recovery is performed.

ering partition is not available for writes, because of the write lease grabbed during recovery. This partition is available for reads, however.

After recovery completed, performance briefly dropped at point (5). This degradation is due to the buffer cache warming on the recovered node. Once the cache became warm, performance resumed to the original 450 operations/s at point (6). An interesting anomaly at point (6) is the presence of noticeable oscillations in throughput; these 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. This sort of performance unpredictability due to garbage collection seems to be a pervasive problem; a better garbage collector or admission control might ameliorate this, but we haven't yet explored this.

6 Example Services

We have implemented a number of interesting services using our distributed hash table. The services' implementation was greatly simplified by using the DDS, and they trivially scaled by adding more service instances. An aspect of scalability not covered by using the hash table was the routing and load balancing of WAN client requests across service instances, but this 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 is a middleman between these protocols, routing and translating messages between the networks. In addition to protocol translation, Sanctio also can transform the message content. 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 (e.g., 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 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 generated when users change networks or log in and out of a network. The data in the table must be kept consistent, otherwise messages will be routed to the wrong address.

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

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 introduce CGI-like behavior by interposing on the URL resolution path. 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 have built many other services as part of the Ninja project[4]. The "Parallelisms" service recommends related sites to user-specified URLs by looking up ontological entries in an inversion of the Yahoo web directory. We built a collaborative filtering engine for a digital music jukebox service [16]; this engine stores users' music preferences in a distributed hash table. We have also 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 behav-

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

ior 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. At 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 this 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 signals from the kernel to the user-level thread scheduler associated with TCP connections with 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 performance bottleneck of the bricks and also a source of throughput and latency variation. Whenever the garbage collector became active, it had a serious impact on all other system activity, and 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 code efficiency, especially when dealing with multiple layers of a system each of which wraps some array of data with layer-specific metadata. We often found ourselves performing copies of regions of byte arrays in order to maintain clean interfaces to data regions, whereas in a C implementation it would be more natural to exploit pointers into `malloc'ed` memory regions to the same effect without needing copies.

Java lacks asynchronous I/O primitives, which necessitated the use of a thread pool at the lowest-layer of the system. This is much more efficient than a thread-per-task system, as the number of threads in our 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 per brick increases with the cluster size. We are working on introducing high-throughput asynchronous I/O completion mechanisms into the JVM using the JNI native interface.

7.2 Future Work

We plan on investigating more interesting data-parallel operations on a DDS (such as an iterator, or the Lisp `maplist()` operator). 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 hash table, such as the brick storage layer, the RG map infrastructure, and the two-phase commit code. We would like to explore caching in the DDS libraries (we currently rely on services to build their own application-level caches). We are also exploring adding other single-element operations to the hash table, such as `testandset()`, in order to provide locks and 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 scalable, distributed data structures (SDDS) such as RP^* [22, 26] helped to motivate our own work. RP^* focuses on algorithmic properties, while we focused on the systems issues of implementing an SDDS 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 the distributed and parallel database communities [10, 17, 25]. We use 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, 23, 31, 33]. A DDS presents 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 scalability, availability, and consistency under high throughput, highly concurrent traffic, which is a different focus than file systems. Our work is most similar to Petal [24], in that a Petal distributed virtual disk can be thought of as a simple hash table with fixed sized elements. Our hash tables have variable sized elements, 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 network attached secure disk (NASD) architecture [11] explores variable-sized object interfaces as an abstraction to allow storage subsystems to optimize disk layout. This 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 [13] attempt to transparently adding persistence to distributed object systems. The persistence of (typed) objects 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. Distributed object stores are often 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, 15]. 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 [30] 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.

There have been many projects that expolored wide-area replicated, distributed services [9, 27]. Unlike clusters, wide-area systems must deal with heterogeneity, network partitions, untrusted peers, high latency and low throughput networks, and multiple administrative domains. Because of these differences, wide-area distributed systems tend to have relaxed consistency semantics and low update rates. However, if designed correctly, they can scale up enormously.

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.

Acknowledgements

We are very grateful to Eric Anderson, Rob von Behren, Nikita Borisov, Mike Chen, Armando Fox, Jim Gray, Ramki Gummadi, Drew Roselli, Geoff Voelker, the anonymous referees, and our shepherd Bill Weihl for their very helpful suggestions that greatly improved the quality of this paper. We would also like to thank Eric Fraser, Phil Buonadonna, and Brent Chun for their help in giving us access to the Berkeley Millennium cluster for our performance benchmarks.

References

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

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

[3] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Dec 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] G. Banga, J. C. Mogul, and P. Druschel. A Scalable and Explicit Event Delivery Mechanism for UNIX. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, Jun 1999.

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

[7] T. Brisco. RFC 1764: DNS Support for Load Balancing, Apr 1995.

[8] 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*, Jan 1996.

[9] A.D. Birrell et al. Grapevine: An Exercise in Distributed Computing. *Communications of the ACM*, 25(4):3–23, Feb 1984.

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

[11] G. A. Gibson et al. A Cost-Effective, High-Bandwidth Storage Architecture. In *ASPLOS-VIII*, San Jose, California, 1998.

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

[13] 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, Feb 2000.

[14] V. S. Pai et al. Locality-Aware Request Distribution in Cluster-Based Network Servers. In *ASPLOS-VIII*, San Jose, CA, Oct 1998.

[15] 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, Oct 1997.

[16] I. Goldberg, S. D. Gribble, D. Wagner, and E. A. Brewer. The Ninja Jukebox. In *The 2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, CO, Oct 1999.

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

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

[19] 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, CA, Dec 1997.

[20] J. C. Hu, I. Pyarali, and D. C. Schmidt. Applying the Proactor Pattern to High-Performance Web Servers. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems*, Oct 1998.

[21] A. Iyengar, J. Challenger, D. Dias, and P. Dantzig. High-Performance Web Site Design Techniques. *IEEE Internet Computing*, 4(2), Mar 2000.

[22] 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, Mar 1996.

[23] 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, Jun 1993.

[24] E. K. Lee and C. A. Thekkath. Petal: Distributed Virtual Disks. In *ASPLOS-VII*, Cambridge, MA, 1996.

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

[26] 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.

[27] P. V. Mockapetris and K. J. Dunlap. Development of the Domain Name System. In *ACM SIGCOMM Computer Communication Review*, 1988.

[28] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the 1999 Annual Usenix Technical Conference*, Jun 1999.

[29] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 1991.

[30] 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, Dec 1999.

[31] 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, Jun 1985.

[32] J. Song, E. Levy, A. Iyengar, and D. Dias. Design Alternatives for Scalable Web Server Accelerators. In *Proceedings of the 2000 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2000)*, Austin, TX, Apr 2000.

[33] 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, Oct 1997.