A Design Framework and a Scalable Storage Platform to Simplify Internet Service Construction

by

Steven D. Gribble

B.Sc. (The University of British Columbia) 1995
M.S. (The University of California at Berkeley) 1997

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Eric A. Brewer, Chair
Professor David Culler
Professor Marti Hearst

Fall 2000

The dissertation of Steven D. Gribble is approved:

Chair Date

Date

Date

University of California at Berkeley

Fall 2000

A Design Framework and a Scalable Storage Platform to Simplify Internet Service Construction

by

Steven D. Gribble

Abstract

A Design Framework and a Scalable Storage Platform to Simplify Internet Service Construction

by

Steven D. Gribble

Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Eric A. Brewer, Chair

The Internet infrastructure has evolved from a collection of loosely organized data repositories and web pages to a rich landscape populated with industrial strength applications and services. Although this evolution has been rapid, the task of building and maintaining these services nonetheless remains challenging, primarily because services must be exceptionally robust, remaining available and performing well in the face of voluminous and growing traffic demands. Coupled with a lack of suitable reusable building blocks and design methodologies for service construction, this challenge unfortunately implies that only organizations with very capable engineering and operations staff can currently successfully build and maintain new Internet services.

This dissertation represents a step towards ameliorating this situation; in it, we address two sets of challenges: the design and implementation of a programming model, concurrency model, and I/O substrate specifically geared towards Internet service construction, and the design and implementation of a storage platform that shields service authors

from the complexities of robust, scalable persistent data management.

The first half of this dissertation focuses on the development of a programming model and design framework that is well suited to the needs of scalable, highly concurrent services. The framework consists of a set of design patterns that can be applied to code in order to "condition" it against load, concurrency, failure, and performance bottlenecks. The framework also describes a way of structuring programs using queues to separate and decouple the program's stages.

The second half of this dissertation describes a scalable storage platform that we built using our design framework. This platform, called a distributed data structure (DDS), greatly simplifies the task of implementing a new Internet service by completely shielding authors from the complexities of scalable, available, consistent storage management. We describe the design, implementation, and performance of a distributed hash table, as well as a number of services implemented using it. The hash table design makes several assumptions and optimizations based on the properties of clusters of workstations.

We believe that this dissertation makes several contributions that greatly reduce the complexity of implementing new Internet services. As such, we hope that it will accelerate the evolution of the Internet by encouraging more people to implement and deploy new, innovative, creative services.

Professor Eric A. Brewer
Dissertation Committee Chair

Contents

List of Figures

List of Tables

Acknowledgements

Where to begin? This dissertation would not have happened if it were not for the help, feedback, support, and guidance of countless incredible people.

I owe a deep debt of gratitude to Armando Fox, whom I had the great fortune of working with and befriending for the first 3 years of my graduate student career. He served as an excellent role model for learning the art and science of research, in particular the skills of technical writing, speaking, and project selection.

Eric Brewer, my thesis advisor, has demonstrated an unwavering ability to see the deeper meaning of my own work, leading me towards a path of greater impact and relevance. He has also taught me the importance of being aware of the process beyond the research itself, such as being judicious about the selection of your peers, being deliberate about managing relationships and the perception of your own work, and fostering a sense of style with everything that you do.

I have had the privilege of working closely with some of the finest faculty and industrial researchers in the world, including David Culler, Marti Hearst, Joe Hellerstein, Jim Gray, Randy Katz, Larry Rowe, and Anthony Joseph. The content of this dissertation would not be as mature, relevant, or accurate without their collective wisdom. Without exception, they have all been exceptionally generous with their time and their friendship.

My office (445 Soda Hall) was an absolutely ideal work environment. The residents of this office have been phenomenal: I hope to retain my friendships with Armando, Ian, Dave, Paul, Yatin, Mike, and Nikita, and I would not hesitate for a second to work with any of these people again.

Thanks go to all of the members of the Ninja group for being the guinea pigs that first sampled the fruits of my research. In particular, I'd like to thank Matt Welsh and Rob von Behren for their insight and encouragement. Many other graduate students at large at Berkeley should also be singled out: Drew Roselli, Eric Anderson, and Remzi Arpaci-Dusseau all kept me honest by showing me the value of meticulousness and objectivity.

The staff in Soda Hall have done a tremendous job of supporting me as a student and researcher. Eric Fraser and Albert Goto have each gone out of their way to support and troubleshoot the clusters on which I ran my experiments, and Terry Lessard-Smith, Bob Miller, Glenda Smith, and Michelle Willard have shielded me from a legion of bureaucrats wielding red tape.

I would also like to personally thank Sunny, who has showered me with care, affection, and support throughout the sometimes arduous and stressful process of being a graduate student, and my parents, who have been unwaveringly supportive throughout my life.

Finally, acknowledgement and thanks must be given to the many funding agencies that have supported me personally, as well as supporting my research groups. Over the course of my graduate career, I have benefitted from the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), the Canadian Space Agency, the Defense Advanced Research Projects Agency (DARPA), the National Science Foundation (NSF), and donations and grants from many corporations, including Intel, Ericsson, Geoworks, Wink, and Hughes Aircraft.

Part I

Motivation

Chapter 1

Introduction

Over the past decade, the Internet has evolved from a collection of loosely organized data repositories and web pages to a rich landscape populated with industrial strength applications and services. Although this evolution has been rapid, the task of building and maintaining these services nonetheless remains challenging, primarily because services must be exceptionally robust, remaining available and performing well in the face of large and growing traffic demands. Coupled with a lack of suitable reusable building blocks and design methodologies for service construction, this challenge unfortunately implies that only organizations with very capable engineering and operations staff can currently successfully build and maintain new Internet services.

This thesis represents a step towards ameliorating this situation. We address two sets of challenges: the design and implementation of a programming model, concurrency model, and I/O substrate specifically geared towards Internet service construction, and the design and implementation of a storage platform that shields service authors from the complexities of robust, scalable persistent data management.

In order to best understand the Internet service construction problem, we first present a historical perspective on the evolution of Internet services, beginning with the advent of timeshared computing systems.

1.1 A Historical Perspective on Internet Services

A key aspect of Internet services is that they live in the infrastructure, and that they are shared across many users. The notion of timesharing can be traced back to the Compatible Time Sharing System (CTSS) operating system for the IBM 7094 mainframe computer [31]. A timesharing operating system has the ability to very quickly switch the control of the CPU between a number of concurrently executing programs, giving each the illusion that they are running on a dedicated, protected processor. As a result, many users and programs can share constrained and expensive computing resources, amortizing cost and also increasing the overall efficiency of the computing system.

CTSS lead to the development of the Multics operating system [32], which contributed or matured many of the fundamental mechanisms and abstractions necessary for robust timesharing operating systems, such as *protection and fault containment* through virtualization, and the development of appropriate I/O and communications interfaces for systems programmers. Even more remarkable, Multics had the foresight to consider the system as a computing utility, focusing on "remote access, continuous operation analogous to that of the power companies, [and] a wide range of capacity to allow growth or contraction without either system or user reorganization". In other words, the goal of Multics was to provide a community computing platform that completely shielded users and their programs from operational issues.

In the 1970's a new class of computers, called minicomputers, became popular. Unlike mainframes, minicomputers were designed to be affordable for small organizations, such as academic departments. Minicomputers also eschewed the complexity typical of mainframes, making it possible for individuals in the organizations that owned them to serve as administrators; comparatively, mainframe vendors typically sold administration contracts along with the computing hardware.

A new trend began to emerge in the late 1970's and early 1980's. Personal computers and desktop workstations became affordable enough so that people could use them as dedicated, personally owned computing platforms instead of relying on timeshared infrastructure computers. This trend had two major implications: the number of deployed computers in the world exploded, but each owner of such computers became saddled with the additional complexity of being a system administrator instead being just a user. This problem still plagues personal computers today; the administration and troubleshooting of a typical personal computer is complex enough to represent a significant portion of the total cost of ownership of a PC.[1]

The second important trend to shape this landscape was the emergence of the Internet, a global network with uniformly adopted transport protocols and a number of common infrastructure services. One such infrastructure service is the domain name system (DNS) [99], which provides translation from human-readable machine names to the more opaque dotted-quad numeric names (e.g. `128.32.130.48`) used by the network protocols and routers. An initial driving application for the Internet was academic document sharing, and in order to support this, a file transfer protocol (FTP) [17] and FTP servers were developed. DNS is an example of a "horizontal" service that is largely transparent to users and other services. FTP servers are "vertical" applications with which users explicitly interact using dedicated client software. Both services are infrastructural utilities (i.e., they are operated and managed by people other than users), and because of growing traffic

[1]According to the Gartner group, the total cost of ownership of a PC is $10,000.00 per year, 60-80% of which is attributed to technical support and user training.

demands, both began to suffer from robustness and availability challenges that Multics escaped.

By the mid-1990's, many of the hundreds of millions of deployed personal computers and workstations became connected to the Internet. Around the same time, a new driving application for the Internet emerged: the world-wide web, or WWW [16] provided a mechanism for delivering rich multimedia content to non-academic users. Soon, the web evolved from a content delivery mechanism to a service platform, with the emergence services such as online banking, travel reservations, news services, and email hosting. The most popular of these services began to attract traffic at an unprecedented scale; for example, according to a press release, the Yahoo! service [80] currently delivers over 625 million page views per day.

Infrastructural services have thus seemingly come full-circle to the days of time-shared Multics, but with the additional problems associated with immense scale. Modern Internet services, as with Multics, must provide "remote access, continuous operation analogous to that of the power companies, [and] a wide range of capacity to allow growth or contraction", with potentially very large capacity requirements.

However, there is an important additional distinction between Multics and Internet services besides scale. Vertical Internet services provide application-specific functionality to users, and horizontal Internet services provide functionality to applications. The Multics execution environment, in comparison, *was* the service. It provided I/O abstractions and programming models that were targeted to supporting relatively small numbers of interactive or batch programs run by humans. This thesis can be viewed as providing pieces of an execution environment for Internet services, specifically a programming model and a data storage platform. These pieces of the execution environment are specifically tailored towards ameliorating the significant challenges associated with providing an infrastructure

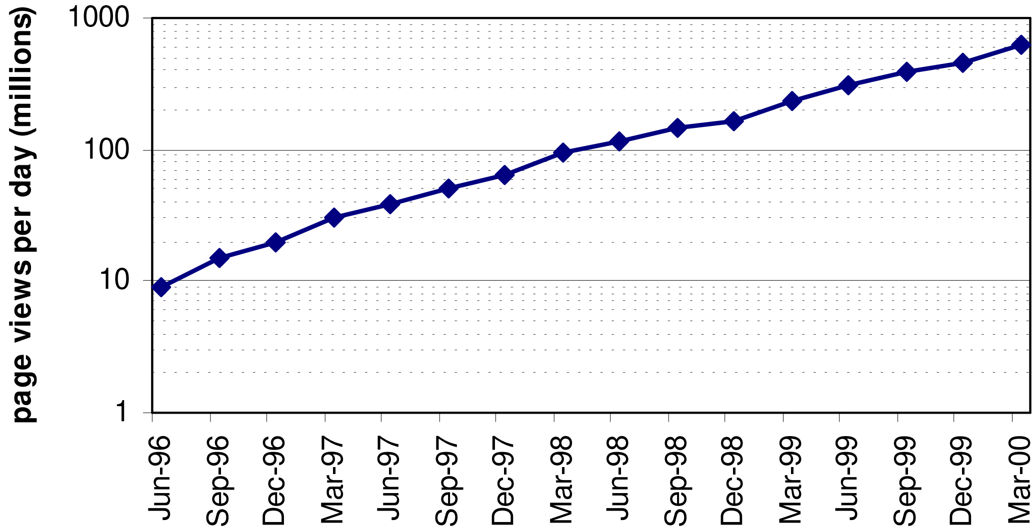


Figure 1: **Yahoo! traffic scaling:** The number of page views per day served by Yahoo! has been exponentially increasing since June 1996.

utility at scale. In the following section, we explore these challenges in depth.

1.2 Internet Service Challenges and the "Service Properties"

An Internet service faces a host of operational and performance challenges. The most obvious of these is the amount of traffic that a successful site must handle, and the rate at which this traffic scales up over time. Figure 1 shows the number of page views per day that the Yahoo! [80] service must deliver, according to their quarterly financial reports. The two remarkable features of this graph are the fact that traffic has been exponentially growing since 1996, and that the site currently must handle 625 million views per day, resulting in 30,000 web requests per second (given an average of 4 web requests per page view). A highly successful service must thus have both *high performance* and *scalability of throughput* in order to match its users' traffic demands.

Another property that services must have is the ability to handle *high concurrency*. In [68], a large-scale web client trace showed that the writeback phase of a web request has a median duration of approximately 3 seconds, due to the slow throughput of the typical 56Kb/s modem last-hop network link. For Yahoo, this means that there are more than 90,000 requests flowing through their service at any given moment. For middleware services (such as web proxy caches [20, 26]) or services in a composition chain, this concurrency demand can be even higher, depending on the aggregate latency of downstream services. This high concurrency has significant implications regarding the service's software architecture and programming model, as we will show in Part II of the thesis.

Recent network- and application-level studies [37, 59, 68, 87, 88, 108] have shown the prevalence of extremely high variance (or "bursty") traffic on the web, as exemplified in Figure 2b. Although there is still debate over the exact nature of this traffic, most traces and anecdotal experience supports the fact that large bursts of traffic can occur at many time scales, from milliseconds up to hours or even days. An extreme example of a burst can occur during denial of service attacks, which have recently become more frequent [114, 125]. Because of these bursts, services must be built to support *graceful degradation under overload*: the service must either absorb and buffer the burst without causing throughput to degrade, or excess traffic must be rejected early to avoid livelock [43].

A popular service must remain available as much as possible. Figure 2a shows the number of web requests per minute generated by a population of UC Berkeley web users, and Figure 2b shows the number of web requests per 180 seconds received by the UC Berkeley CS Division web server. From these illustrations, we see that there are clients generating requests 24 hours per day, and that servers receive requests 24 hours per day. The diurnal cycle seen in these two figures is typical of services with high geographical locality, but in more services with world-wide popularity, this cycle is much less pronounced. The fact

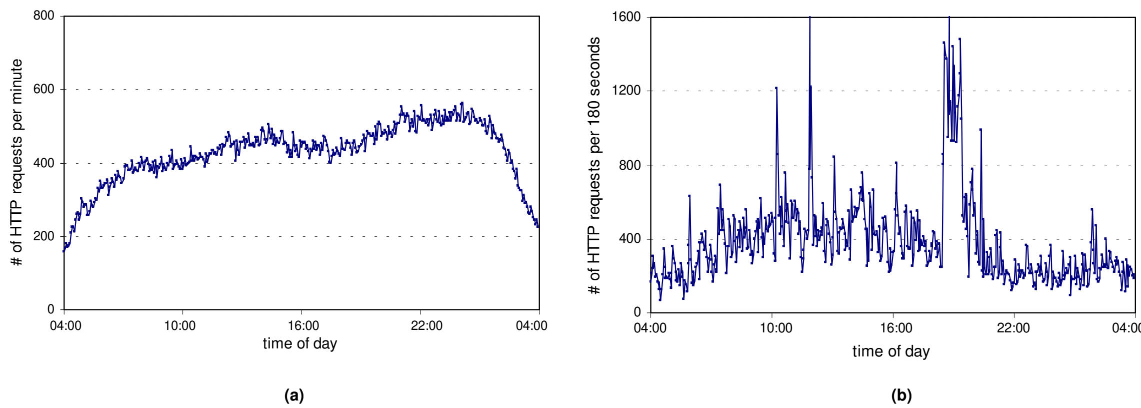


Figure 2: **HTTP traffic patterns:** (a) Illustrates the diurnal traffic pattern of web requests measured from a closed population of 8,000 modem pool users from UC Berkeley in the Fall of 1996, and (b) shows the number of web requests received every 180 seconds by the UC Berkeley CS division web server on February 2nd, 2000.

that services must deal with constant traffic means that there can be no planned "service downtime"; services must remain *highly available*, even in the face of hardware or software failures, and also during planned maintenance or upgrades. Being available in the face of failure implies that the service is fault tolerant.

If a service manages persistent data, e.g., a mail server [34] or an ecommerce service [1], then depending on the nature of the application, the service may need to keep that data consistent and durable. The degree of *data consistency and durability* depends on the application, but many applications require some degree of atomicity, coherence, and durability. Furthermore, as a service grows in popularity, the amount of data that the service must store and access may increase, forcing the service to have *scalability of storage capacity*. For example, according to press releases, the Hotmail mail service [34] has 80,000 new accounts registered per day, each of which may use up to 2MB of storage.[2]

To summarize, the workload presented to an Internet service coupled with its users'

[2]This is worst-case addition of 160GB of data per day; in practice, most mailboxes use considerably less than their allowed 2MB.

property	description
data consistency and durability	the service must maintain a level of data consistency and durability as appropriate to the given application
scalability of storage capacity	the amount of state that a service manages must be able to scale up to match the increasing demands of a growing user base
high availibility	the service must remain available even in the presence of partial failures
high concurrency	the service must be able to handle large numbers of simultaneous tasks from its user population
high performance	the service, if successful, must be able to handle a high volume of traffic with acceptable latency
graceful degradation under overload	if transient or long-lived bursts of arrivals exceed the current capacity of the service, the service should still maintain high throughput by shedding or queueing excess load
scalability of throughput	the service must be able to scale up to match increasing volumes of offered traffic

Table 1.1: **The "service properties":** This table outlines the set of properties that an Internet service must have in order to be successful.

expectations imply that it must have all of the properties outlined in Table 1.1. There are a number of additional challenges particular to Internet services that are beyond the scope of this dissertation, for example the problem of locating services in a wide-area environment ("service discovery"), or the problem of composing multiple horizontal services together to produce an interesting, dynamically assembled application ("service composition").

1.3 On the use of Clusters of Workstations

In this section, we describe how the use of a shared-nothing multicomputer architecture known as networks or clusters of workstations (NOW) [4] can help to achieve some of these properties. A NOW is a collection of possibly heterogeneous uniprocessor or multiprocessor workstations connected together by a high speed, low latency system area network (SAN). For example, the Millennium cluster at UC Berkeley [23, 29, 41] currently includes a mix of 28 2-way 500MB 500MHz Pentium-II and 39 4-way 1GB 550Mhz Pentium-II based machines, connected by both a myrinet [36, 102] (1.6 Gb/s throughput and 12 μs round-trip latency) and a gigabit Ethernet.

The NOW architecture has a number of interesting attributes that can be exploited to achieve the properties listed in Table 1.1. Because each node in the cluster is an independent failure boundary, it is possible to achieve high availability through the use of *replication* across nodes. Given a software architecture that includes failover abilities, the crash of a single node in the cluster need not interrupt the overall service, as the computation state and persistent data on that node are replicated elsewhere. The degree of replication and the rate at which failed components recover (mean time to recovery, or MTTF) affects how many concurrent failures a service can withstand without affecting availability.

Because a cluster is comprised of a number of independent resources (disks, CPU, memory, I/O buses, NICs, etc.), a cluster can support incremental scalability. If a service runs out of capacity, and if its software architecture is designed correctly, scaling can occur by adding additional nodes to the service. To achieve linear or superlinear scaling, the architecture must be able to rebalance data and computation across the cluster after such growth has occurred. The fact that there are so many independent resources in the cluster means that parallelism is naturally supported. Internet workloads typically have a large number of independent tasks, corresponding to requests from independent users; this task independence simplifies the process of exploiting this resource parallelism, leading to high service throughput.

A cluster also accomodates the partitioning of computation and data. This partitioning helps to mitigate the concurrency demands of services; for a given degree of service concurrency, adding additional nodes decreases the per-node concurrency requirements.

Clusters have a number of interesting operational attributes as well. A cluster can be physically situated in an environment that is most conducive to achieving the service properties. A cluster can be in a physically secure machine room, provided with redundant, uninterruptible power supplies to minimize failures, have controlled heterogeneity (in the sense that heterogeneity is predetermined and well understood), and have well-trained system administrators to reduce the chance of operator error.

1.3.1 Complexities of Clusters

Clusters have many sources of heterogeneity. Individual nodes may have different hardware or software characteristics, resulting in performance or capacity imbalances across the cluster. Imperfect run-time load balancing or data partitioning can also result in imbalance, leading to a loss of service performance or scalability. To ameliorate this, software and administrators must tune the static and dynamic balance of resources and tasks across the cluster.

When processes communicate with each other across a cluster, there are a number of different domains and associated domain crossing overheads that their messages may experience. These domains include methods or procedures within an address space, multiple address spaces on a single node, multiple processors within a single node, and multiple nodes across the SAN. In general, the coarser the domain, the more expensive it is to cross. The layout of software components and the placement and granularity of data partitions within the cluster must be designed to exploit locality within these domain wherever possible, otherwise the service may achieve poor performance due to excessive boundary crossings during communication.

The fact that clusters are comprised of many independent resources necessarily introduces complexities into the software architecture. In order to achieve availability,

Cluster Attribute	Service property	Challenge
low latency, high throughput SAN	high performance and scalability	overcoming network partitions, balancing traffic across the cluster avoid network bottlenecks
nodes as independent failure boundaries	high availability	detecting failures, restarting failed components, replicating data and/or computation for fault-tolerance and durability, maintaining consistency across replicas
independent resources in a shared-nothing architecture (IO backplanes, disks, CPUs, NICs, physical memories, etc.)	high concurrency, incremental scalability	balancing across resource types, preventing imbalance due to hardware and software heterogeneity, preventing imbalance due to dynamic differences in load or accidental coupling of resources, administration of the large collection of resources

Table 1.2: **Cluster attributes:** This table describes attributes that clusters have, how they help to address the service properties, and the challenges that must be overcome to do so.

data must be replicated across multiple nodes. But, to maintain the overall consistency of a service, these replicas must be kept consistent with each other. Node, process, and communication fabric failures make it difficult to achieve replica consistency. To make matters worse, as the promised consistency becomes stronger, delivering that promised consistency introduces a tight coupling between the replicas and the resources they use. This coupling detracts from the parallelism and the scalability of the system that arises from the independence of the cluster's resources.

Harmful coupling can occur from sources other than consistency protocols. For example, if a blocking RPC style of communication is used for inter-node communication, then that RPC couples two threads together, as the caller does not continue to execute until the callee returns. If RPC calls are chained across multiple levels, then the degree and duration of coupling grows with the call depth. To avoid this, careful thought must be given to the programming model and control structure of services, and appropriate I/O

API's and mechanisms must be used.

In Table 1.2, we summarize the cluster attributes, and describe how they help address the service properties and the challenges that must be overcome. All of the challenges relate to software infrastructure that must be in place to support the service and programming models that should be used to allow the service to be properly conditioned with respect to load and balance.

1.4 Contributions

The main contributions of this thesis are as follows:

1. We describe the specification, construction, and evaluation of a programming model and design framework specifically targeted to the needs of highly concurrent, scalable Internet services. Our description includes the exploration of the tradeoffs between threaded and event-driven programming styles, and our programming model presents a hybrid of these styles that retains the advantages of both, in terms of program performance and clean structure. Within our model, we identify several archetypical design patterns and composition operators that are common to Internet services; these patterns and operators serve to greatly simplify the conceptual design of new services. Finally, we present the implementation and evaluation of a class library based on this model and these patterns/operators.

2. We present the design, implementation, and evaluation of a storage platform for scalable Internet services that makes use of our programming model. This platform, called a distributed data structure (DDS), greatly simplifies service construction by shielding service authors from issues relating to the availability, consistency, recovery, and scalability of persistent data management. Our design picks a "sweet spot"

of consistency, providing one-copy equivalence and atomic updates, but avoiding the complexities of transactions. The evaluation of this storage platform demonstrates that both throughput and capacity linearly scale; we demonstrate in-core throughput of 60,000 read operations per second on a 128 CPU cluster, and a storage capacity of 1.28 TB over 128 disks.

3. We present a number of novel Internet services that were implemented using our programming model and scalable storage platform, including a scalable web server, an instant messaging proxy gateway, and a collaborative filtering engine for a community music Jukebox.

1.5 Thesis Map

In the remainder of Part I, we motivate the first two hypotheses by differentiating between WAN distributed systems and Internet services in a cluster. We also demonstrate that the service properties of graceful degradation and scalability imply that mechanisms such as task scheduling, resource management, and concurrency management must be exposed to applications rather than hidden through abstractions or through virtualization.

In Part II, we present a programming model and a design framework for highly concurrent Internet services. We present detailed performance measurements to highlight the distinction between event-driven and threaded programming models, and use this to motivate a hybrid programming model based on “thread boundaries”, in which queues separate layers of the system. We also describe a number of design patterns that can be applied to code in order to condition it against load bursts, resource bottlenecks, or failures.

In Part III, we describe a scalable storage subsystem that we built using the programming model described in Part II. This storage subsystem is called a distributed data structure (DDS), and it provides scalability (both in terms of throughput and capacity), availability, and strict consistency. A distributed data structure has a conventional single-site data structure interface, but partitions and replicates data across a cluster of workstations in a manner that is transparent to services that use it. We discuss the design and implementation of a hash table DDS, we quantify its performance, and we present several services implemented using it.

Finally, in Part IV of the dissertation, we present related work, and also discuss several open issues and avenues of future research that we uncovered while exploring our own work. We conclude with a summary of the contributions of this dissertation.

Chapter 2

Towards a Cluster-Based Internet Service OS

2.1 Dodging the WAN Bullet

Fundamentally, Internet services are distributed systems. Distributed systems face many significant challenges that must be overcome for them to operate correctly. These challenges all stem from the fact that a distributed system, by definition, includes multiple processors, independent failure boundaries, and unreliable networks. In the fully general case, a distributed system must be able deal with processor failures (including Byzantine failure [85]), differing execution speeds, multiple implementations of multiple programs executing within the system, network partitions,[1] and high variance in wide-area network bandwidths, latencies, and reliabilities.

As an example of an Internet service that is a fully general distributed system, consider the case of the Domain Name Service (DNS) [99]. The DNS system consists of a

[1]A network partition occurs when a hardware or software failure splits a network into two or more disjoint subnetworks.

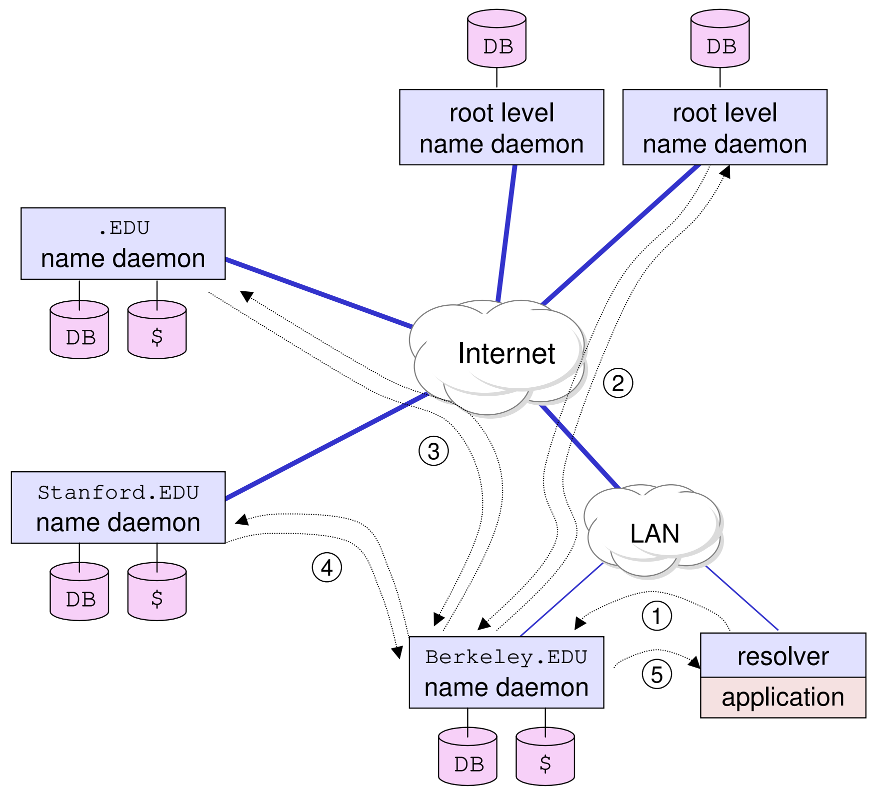


Figure 3: **DNS architecture:** This figure illustrates the software components, databases, caches, and network connections that are necessary in order for an application running on a Berkeley workstation to resolve a Stanford workstation's DNS name. The labeled arrows refer to messages exchanged between the software components; these messages are explained in detail below.

number of software processes, databases, and caches that are distributed across the wide-area, organized in a hierarchy that matches the name system hierarchy (Figure 3).

If an application running on a Berkeley workstation wishes to resolve a Stanford DNS name, e.g., `www.stanford.edu`, up to five message exchanges may need to occur between five different software entities. A library attached to the application first sends a message to a Berkeley DNS server (message 1). That server consults its cache; if it doesn't have `www.stanford.edu` in the cache, it attempts to contact the name server for `stanford.edu` in order to resolve it. However, to contact that name server, it must resolve another DNS record in order to determine the address of this name server. This resolution process continues up the name hierarchy, checking the local cache at each step, until the Berkeley DNS server resorts to contacting a root level name server (message 2). Messages 3 and 4 correspond to the Berkeley DNS server traversing down the name hierarchy, until it finally resolves `www.stanford.edu`. Once it has done so, it returns the resolved address to the application's library (message 5).

Although the application sees the DNS name space as a single, logical database, this database is actually partitioned among many independent fragments across the wide area. Furthermore, all name resolutions are done through a series of name servers' caches. There is thus a challenging consistency issue with DNS; updates to a local DNS database fragment are not immediately propagated to other databases or caches, resulting in potentially inconsistent name resolutions, depending on which name server an application consults. To address this issue, DNS records have expiration times associated with them, and distributed applications must tolerate inconsistencies within the window bounded by these expiration times.

DNS also has challenges related to service availability. If any of the DNS servers in the chain of requests in Figure 3 are unavailable, then the DNS resolution will fail. To overcome this, DNS stipulates that each DNS server must have at least one replica that can be used if the primary server is unavailable. Replicas are supposed to be geographically distributed, reducing the chance of correlated failures by ensuring that they are in independent failure boundaries. In reality, many organizations are configured with geographically proximate DNS server replicas, implying that the failure of a single network link could bring down DNS service for that entire organization. Also, if a network partition separates an application from its configured DNS server, then the application will not be able to resolve any DNS names, even for organizations that are not partitioned away.

2.1.1 The CAP Principle

The DNS example serves to demonstrate that there is a relationship between data consistency, service availability, and the ability to tolerate network partitions. This relationship is formalized by the CAP Principle [56]:

> **CAP Principle:** A system may have at most two of strong **C**onsistency, **A**vailability, and tolerance to network **P**artitions.

By strong consistency (C), the CAP principle means one-copy equivalence of a data store, even during update. Availability (A) means that the system continues to operate and be accessible to clients even though individual elements or replicas have failed. Tolerance to network partitions (P) implies that the system continues to operate and be available even though elements of the system cannot communicate with each other.

DNS is an example of a system that supports A and P, but not C. In fact, even without network partitions, clients may see two inconsistent name bindings, depending on the DNS server or cache through which they resolve a name. Such inconsistency is designed into the system as a normal mode of operation; a network partition merely increases the chance that a client will see such an inconsistency.

In reality, C, A, and P are not strictly boolean characteristics, but rather they exhibit degrees: a system can have weak consistency or partial availability. Accordingly, a more realistic view of the CAP principle is illustrated in Figure 4; a system can pick any operating point in the triangle. The vertexes of the triangle represent the tradeoff described above by the CAP principle. For example, the Bayou distributed database [40] allows transient inconsistencies, but as a result, it has high availability and can tolerate network partitions.

Systems are usually composed of multiple components. Each component in the system can select a different point within the CAP triangle. For example, many databases

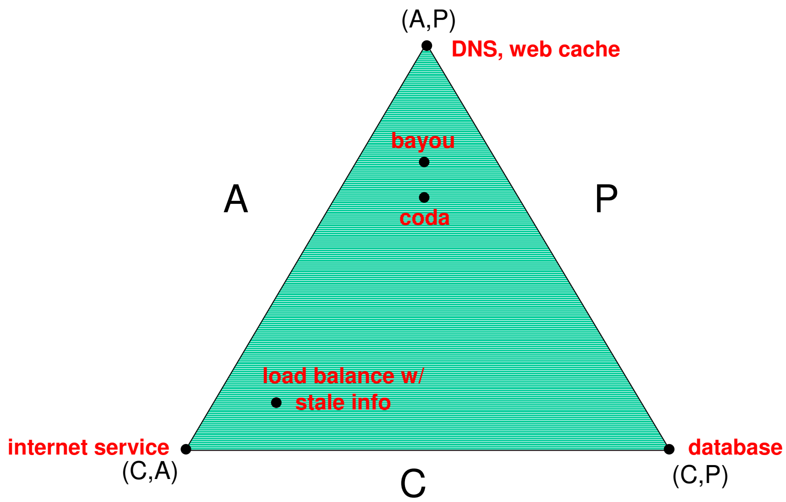


Figure 4: **The CAP tradeoff:** C,A, and P are not boolean characteristics; a system can be described in terms of weaker consistency or availability. Thus, a given system can choose any balance between C, A, and P that falls within the illustrated triangle. For example, a load balancer can operate with stale information during a network partition, but the efficacy of its load balancing decreases the longer that the network partition lasts.

are connected to the web using a web server front end. The front ends can select the AP vertex, since they have no data to keep consistent. The database will likely select the CP vertex, as a database typically becomes unavailable rather than tolerate inconsistencies during a network partition.

According to Table 1.1, two essential properties of Internet services are data consistency and high availability. In other words, services must have C and A. According to the CAP theorem, this implies that services will not be able to tolerate network partitions: if a network partition occurs between elements of an Internet service, that service will necessarily become unavailable, inconsistent, or both unavailable and inconsistent. The CAP theorem combined with our required service properties therefore imply the following corollary:

> **Service Operation corollary:** for an Internet service to operate with both high availability and data consistency under writes, it must be architected to operate in an environment in which network partitions do not occur.

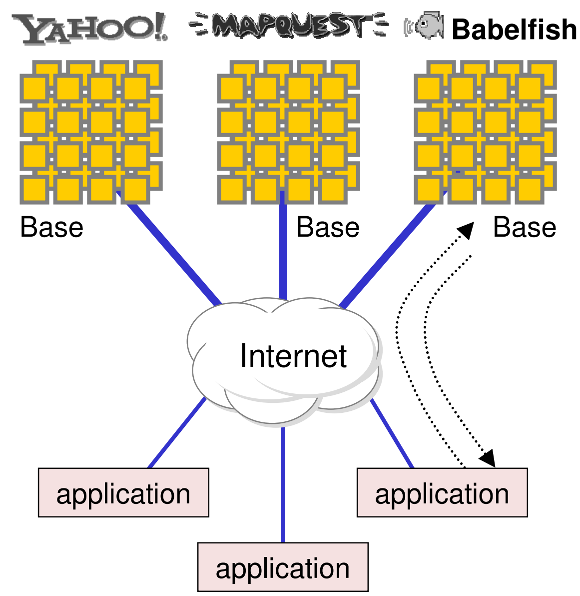


Figure 5: **Service architecture:** Services are confined to run inside bases, which are clusters of workstations that are engineered in such a way that the probability of a network partition is negligible.

2.1.2 Fighting our Battles in a Base

To satisfy the Service Operation corollary, we choose to require that services operate in a well-engineered environment that we call a *base*. A base consists of a cluster of workstations, located in a physically secured machine room, powered with redundant, uninterruptible power supplies, and connected with a redundant network. The network must have adequate redundancy so that the probability of a network partition is negligible.

Requiring that service run inside a base imposes a helpful structuring on our distributed system. The base is a boundary in which all of the challenges of distributed systems are contained, and an environment that is designed obviate (or at least lessen) these challenges as much as possible. Unlike the scattered architecture of DNS (previously illustrated in Figure 3), our service architecture has a clear structure (Figure 5). In particular, all service-specific data is encapsulated inside a single base; this implies that there are no data consistency issues outside of the base. There is also only one kind of network partition: one which separates clients from services. If such a partition occurs, there is no data consistency issue, as all replicas of the data are contained inside the base. There is an availability issue, but it is one that a service cannot eliminate, since it does not control the network leading from the client to the base. It can reduce the probability of such a partition by having multiple peering arrangements with different ISPs, however.

Putting all components and replicas of a service in a single base does add a new challenge: there is significantly more coupling between components in a single base than components across the wide area (such as a shared power supply, a single physical room, etc.), which introduces the possibility of correlated failures across the entire service. For example, if the building in which the base is located is destroyed, services running in the base will necessarily become unavailable, and data will likely be lost. Some coupling cannot be eliminated, however many sources of coupling can be ameliorated through careful engineering. We will return to this issue later in this dissertation.

2.2 The Trail of Cluster-Based Internet Service Platforms

This section describes the path of research that has been incrementally solving many of the challenges of building base platforms for supporting cluster-based Internet services. The previous chapter identified the properties that infrastructure services must have to be successful, and described why clusters can help to provide those properties, but also outlined significant challenges with cluster environments. The research described in this section represents a series of steps, each of which abstracts away an additional set of challenges into a platform or toolkit for building services, and each of which represents an expansion in the set of applications that can be supported.

2.2.1 The Inktomi Search Engine

One of the first driving applications of the Berkeley NOW project [4] was the Inktomi search engine [35]. Inktomi first demonstrated that the NOW architecture could be used to build a highly-available, scalable service. It makes heavy use of the myrinet SAN (achieving low overhead, 12 μs latency one-way communication through the active message [128] user-level network software) as a scalable interconnect. To achieve application scalability, Inktomi horizontally partitions its search index across the nodes of the cluster; a user's search thus results in parallel searches over all cluster nodes, followed by a sort-merge to aggregate the individual results. Because each Inktomi node is involved with each search, and since per-node searches are relatively inexpensive, the low overhead of the communication fabric is essential to achieve acceptable throughput from the system.

If a node fails, the portion of the overall document space on that node becomes unavailable but the overall service remains uninterrupted. Data consistency is not an issue for Inktomi, because their document search index is "read mostly" but also because the data in the index is not replicated. Failures reduce the quality of the service (as fewer hits are returned), but not its availability or consistency.

2.2.2 TACC

The TACC architecture [55] generalizes many of the ideas behind Inktomi by providing an extensible programming platform on which people could author services. TACC's programming model forces authors to decompose their services into stateless or soft-state workers; the application logic materializes a Unix pipe-like composition of these workers in order to service a task. Because workers are stateless or soft-state, if a worker dies, then tasks that previously would be dispatched to that worker can instead be rerouted to equivalent instances of that worker on other nodes. TACC fosters scalability by dynamically starting workers on additional cluster nodes if the workload warrants it, and by load balancing incoming tasks across all worker instances of a given type.

Task dispatches in TACC are blocking RPC-like operations; each worker node sequentially processes incoming tasks off a task queue. TACC thus relies on process-level parallelism in order to achieve concurrency across workers. As the composition depth grows in TACC, this concurrency model combined with the use of blocking RPC-style dispatch leads to an explosion of worker instances, since a worker at the head of a composition chain appears to be "busy" until a given task flows through the entire chain. However, the fact that TACC workers are generally very CPU intensive (performing operations such as image or audio transcoding) means that the overhead associated with this process explosion is small relative to worker computation.

2.2.3 MultiSpace

TACC has three deficiencies that the MultiSpace [70] service platform hoped to address. First, the names of TACC workers and their types are based on strings, and thus cannot be statically type-checked. Second, TACC workers can not be easily "pushed" into a running cluster platform by external clients, but rather have to be carefully introduced into the cluster by an administrator. Third, like Inktomi, TACC relies on having stateless or soft-state workers, thus greatly restricting the application space that it can support.

MultiSpace makes heavy use of the Java programming language and runtime environment [62] to attempt to address the first two of these deficiencies. MultiSpace uses Java's remote method invocation (RMI) facility as a communication primitive across workers. Workers are thus named according to their Java interface (a typed object in Java). Worker composition is accomplished by chaining together RMI calls across the composed workers; because of this, composition can be statically type checked. MultiSpace also makes use of Java's ability to perform dynamic class loading in order to ship bytecode into MultiSpace single-node execution environments. This mechanism makes it possible to dynamically "push" workers into a running MultiSpace installation, thereby reconfiguring existing services or deploying new services.

Experience with the MultiSpace platform revealed a number of serious design flaws stemming from these new mechanisms. The concurrency model implicit with RMI is thread-per-task, i.e. a thread is dispatched on the callee to perform the invocation. RMI attempts to be as transparent as possible, and as a result, service authors have no insight into or control over such important system characteristics as the number of concurrent RMI invocations that can run, the scheduling of RMI invocations, the number of threads or transport connections that would be created, or the lifetime of those threads and transport connections. Furthermore, the blocking-nature of RMI leads to coupling between resources across nodes, as explained in Section 1.3.1. The net effect of this "overzealous" transparency combined with the RMI-induced coupling is that MultiSpace's performance severely degrades as the number of users, services, service composition depth, or nodes in the MultiSpace scales.

The decision to name workers by their typed interface created several operational problems as well. Two workers with different semantic behavior cannot share the same interface, as there is no external way to distinguish them. This implies that different versions of the same worker cannot coexist, making service evolution difficult. Similarly, it is difficult to extend, specialize, or otherwise "subclass" a worker, since changing its interface by definition changes its name, and all people that depend on that worker must be made aware of the change. Finally, the use of typed interfaces implicitly requires an RMI-like composition mechanism, which, as we have stated above, introduces a coupling between the blocking caller and the callee.

To address TACC's lack of scalable, persistent state management, MultiSpace postulates the existence of a consistent, durable storage layer. This storage layer is based on the notion of exposing conventional data structure APIs such as hash tables or trees to service authors, but durably and coherently managing the state behind those APIs in a manner that is transparent to service authors. A prototype distributed hash table was built for the MultiSpace platform, but that prototype failed to achieve many of the service properties, in part due to naive design but also due to the use of an inappropriate programming model (the pervasive use of `mmap()` and thread-per-task concurrency) that failed to give the hash table implementation enough control over on-disk state management, in-memory buffer management, and scheduling over tasks. In Part III of this dissertation, we describe the design, implementation, and evaluation of a more sophisticated distributed hash table.

2.3 A Programming Model for Bases

Experience with MultiSpace demonstrated that a poor choice of programming model can cause an Internet service to scale poorly, and to exhibit poor performance under load. In this section, we focus on motivating a programming model that is well suited to Internet service construction; we will describe this model in detail in Part II of the dissertation.

Programming languages, models, and abstractions tend to reduce the complexity of building new systems by using virtualization. Virtualization gives programmers the illusion that their program's execution context has isolated access to a set of underlying system resources, even though there are multiple contexts competing for access to a shared resource. For example, a thread [5] is an abstraction that allows programmers to reason about the control flow of their systems in a simple, linear fashion; the thread subsystem virtualizes the processor by alternating the execution of multiple threads based either on timer events or on scheduling events (such as a thread entering a blocking state to wait for I/O). The use of the thread abstraction in highly concurrent systems such as Internet services typically leads to a thread-per-task programming model in which a thread is dispatched to handle an incoming client request. Using this model, each thread is completely independent and has no visibility into other threads. No component of the service has a complete view of the entire system.

As a second example, RPC [19] is a communications abstraction that naturally complements threaded programming. To the programmer, RPC allows the execution of a thread to be transparently transferred to a remote address space or machine across a procedure call; RPC thus can be thought of as another kind of virtualization by which address space or machine boundaries are hidden. The RPC subsystem automatically blocks the caller's thread, transfers marshalled arguments across the address spaces, and either creates or dispatches a thread to invoke the procedure on the callee. Because this mechanism is transparent to the programmer, the program itself typically has no visibility into or control over parameters such as the number of simultaneous contexts that can execute on the caller, or the order in which procedure calls execute.

Virtualization can greatly simplify programming, since it shields programmers from details such as scheduling decisions, marshalling complexities, and awareness of concurrency. However, we believe it is exactly these details that dictate whether or not a service is robust under scale and whether it can achieve all of the service properties. Services must be able to handle high throughput and high concurrency; in particular, a given process in a service may need to handle many thousands of simultaneous tasks. If the service author used a thread-per-task programming model, this would result in many thousands of threads per process. Most thread subsystems do not scale well to this number of threads (as we will explicitly demonstrate later). Furthermore, because RPC hides scheduling from the programmer, a system that relies on it to dispatch tasks across multiple components cannot easily introspect on the number of tasks flowing through a given component, making load balancing and admission control extremely difficult.

The simplification afforded by the programming abstractions such as threads and RPC (and their resulting programming models) can cause tremendous problems when the system must handle workloads that are typical for Internet services, in particular extremely high concurrency and high throughput. Under these workloads, resource management and visibility into resource consumption and scheduling is critical to maintaining the robustness of the service. The virtualization that is used by these abstractions takes away the system's ability to observe resource consumption, and often does not allow the system to perform any resource management. We therefore claim that the programming model presented to Internet service authors must explicitly expose mechanisms and resources to the programmer and to the program. Part II of this dissertation focuses entirely on our proposed programming model, which is based on asynchronous I/O abstractions, explicit queues, and message-passing oriented communication between components.

2.4 Hypotheses

This thesis represents the next step in the trail of Internet service platforms outlined in Section 2.2. The thesis attempts to explore issues related to the programming model for Internet services, and also attempts to introduce more sophisticated state management facilities to cluster-based service platforms. Specifically, the dissertation proposes the following three hypotheses:

1. There exists a programming model, a set of I/O abstractions, and a design framework

that are much better suited to high concurrency, I/O centric scalable Internet services than the traditional thread-per-task, synchronous I/O model. This better model (based on event-driven control flow, asynchronous I/O, message passing, and queues) behaves well as the service is scaled up and naturally imparts the service property of graceful degradation because of its explicit focus on exposing resources and enabling resource management and task scheduling.

2. Using this programming model, it is possible to build a scalable storage platform specifically designed for the needs of Internet services. This storage platform should contain all of the service properties, and if services relinquish all of their persistent state management to this platform, then they can easily inherit the service properties.

3. Given these two elements (the Internet service design framework and the scalable persistent state management platform), it is possible and easy to build a large and interesting class of Internet services that possess all of the service properties.

Part II of the dissertation addresses Hypothesis 1, by describing and evaluating a programming model that we have implemented. In Part III, we present the design, implementation, and evaluation of a storage management platform that has all of the features and properties stipulated in Hypothesis 2. Finally, in Chapter 8.3, we describe a number of interesting services that we have implemented using the programming model and storage management platform; these services validate Hypothesis 3.

Part II

A Programming Model for Highly Concurrent Servers

Chapter 3

The Thread and Event Spectrum

Internet services must expect to receive high throughput, high concurrency workloads. To deal with concurrency, programmers to date have primarily considered two programming models and structures for their systems: **thread-per-task** and **event-driven**. Thread-per-task programming allows programmers to write straight-line code and rely on the operating system to overlap computation and I/O by transparently switching across threads, each of which handles a single task as it flows through the system. When using the event-driven programming model, programmers manage concurrency explicitly by structuring code as a single-threaded handler that reacts to events (such as non-blocking I/O completions, application-specific messages, or timer events).

As shown in Figure 6, each task that a service handles can be structurally separated into a sequence of stages. A task stage consists purely of computation; stages are separated by high- or variable-latency operations, such as disk I/O, network I/O, and lock or mutex acquisition. The computation within a stage is typically small, consisting of tens of microseconds of protocol parsing or content generation; however, more complex computation (such as public key operations or image distillation [57]) may last tens of milliseconds.

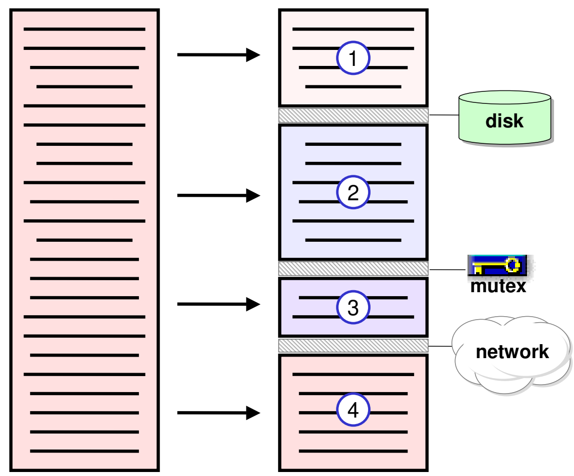


Figure 6: **Stages in a Task:** A task that enters an Internet service can be separated into a sequence of stages, each of which consists of pure computation. Stages are separated by high or variable latency operations such as disk I/O, network I/O, and lock or mutex acquisition.

The operations that separate stages, however, can be virtually instantaneous, such as for disk reads that are serviced from a buffer cache, or they may last hundreds of milliseconds, such as for network reads over the wide area. A system can achieve high concurrency and high throughput by executing other stages while a particular task is blocked.

In this chapter, we present a detailed performance and structural evaluation of these two programming models. We will use this evaluation to guide the design of our own programming framework, which is presented in the following chapter. As a reference model for our evaluation, we frame our discussion around a hypothetical server that is illustrated in Figure 7a. This server, which receives A tasks per second from a number of clients, handles each task with a service latency of L seconds (the result of invoking an external resource such as a disk or network), but overlaps as many tasks as possible. We denote the task completion rate of the server as S. A concrete example of such a server would be a web proxy cache; if a request to the cache misses, there is a large latency while the page is

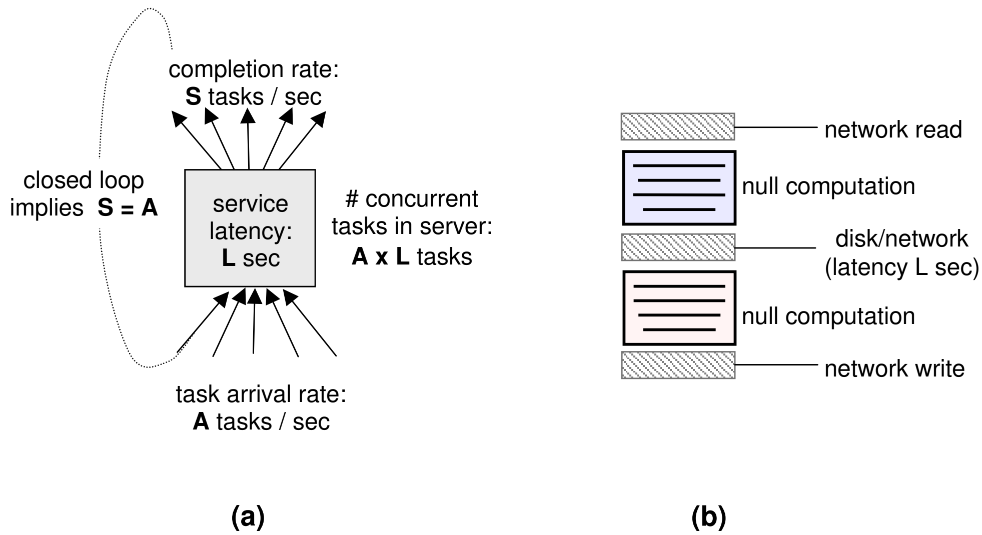


Figure 7: **Concurrent server model:** (a) The server receives A tasks per second, handles each task with a service latency of L seconds (the result of invoking an external resource such as a disk or network), and has a service response rate of S tasks per second. The system is closed loop: each service response causes another tasks to be injected into the server; thus, $S = A$ in steady state. (b) A task from the programmer's point of view: a task consists of two (null) stages separated by three long-latency operations.

fetched over the wide-area network from the authoritative server, but during that time the task doesn't consume CPU cycles. For each response that a client receives, it immediately issues another task to the server; this is therefore a closed-loop system.[1]

Figure 7b shows the view of the system from the point of view of the programmer; each task that enters the system has two stages, and three stage boundaries. The task first must read data from a network connection. A computation stage (which is null in our implementations) separates the read completion from an L second operation such as a disk access or a network access. Then, a second computation stage (which again is null in our implementations) separates the completion of the L second operation from a final network write operation, which sends the task response to the client.

We have implemented this simple hypothetical server using the three programming

[1]A closed-loop system is one in which the output signal of a system is fed back into the system's input.

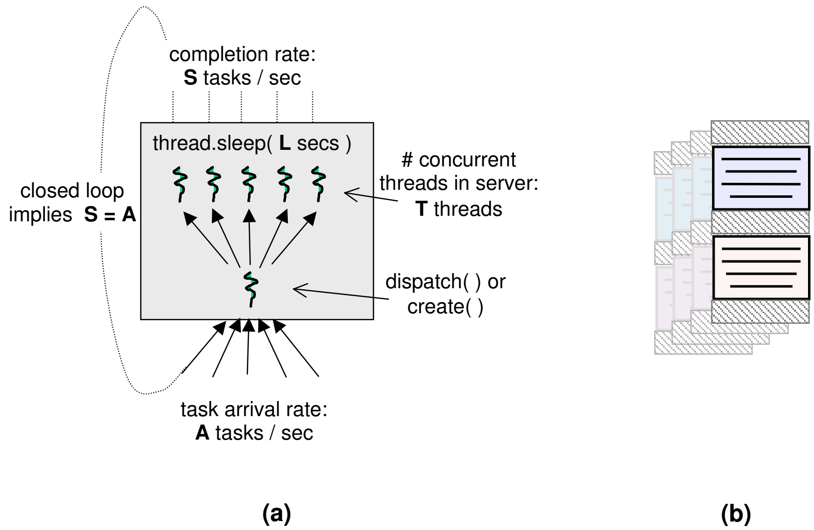


Figure 8: **Threaded server:** (a) For each task that arrives at the server, a thread is either dispatched from a statically created pool, or a new thread is created to handle the task. At any given time, there are a total of T threads executing concurrently, where $T = A \times L$. (b) From the programmer's point of view, the program consists of the logic for a single, linear task; programmers must worry about synchronization, but not scheduling or state management.

models that we evaluate in this chapter (thread-per-task, event-driven, and a hybrid model that we introduce later); we used these implementations to measure the performance characteristics of each model. In our implementations of this server, we simulate the L second operation either with a thread sleep operation in the case of the thread-per-task or hybrid server, or with a sleep queue in the case of the event-driven server.

3.1 Threaded Servers

A simple threaded implementation of this server (Figure 8a) uses a single, dedicated thread to service the network, and hands off incoming tasks to individual task-handling threads, which step through all of the stages of processing that task. One handler thread is created per task. An optimization of this simple scheme creates a pool of several threads in advance and dispatches tasks to threads from this pool, thereby amortizing the high cost of thread creation and destruction. In steady state, the number of threads T that execute concurrently in the server is $S \times L$. As the per-task latency increases, there is a corresponding increase in the number of concurrent threads needed to absorb this latency while maintaining a fixed throughput, and likewise the number of threads scales linearly with throughput for fixed latency.

From the programmer's point of view, the program consists of implementing the logic for a single, linear task. The programmer must use synchronization primitives to protect shared state from concurrent access, but otherwise, the programmer is not at all aware of the multiple, concurrent threads in the system. In particular, the programmer doesn't worry about scheduling, as the thread scheduler handles this, or per-task state management, as the compiler transparently places task state on thread stacks in the form of automatic variables. If tasks are invoked by RPC, then the programmer also has no control over task admission (i.e., whether or not tasks are admitted into the system or are dropped), as the RPC subsystem will transparently handle this.

Threads have become the dominant form of expressing concurrency. Thread support is standardized across most operating systems, and it has become so well established that it is directly incorporated into modern languages such as Java [62]. Programmers are comfortable coding in the sequential programming style of threads, and programmer tools (such as debuggers and thread-safe class libraries) are relatively mature. In addition, threads allow applications to scale with the number of processors in an SMP system, as the operating system can schedule threads to execute concurrently on separate processors. However, thread programming does present a number of correctness and tuning challenges. Synchronization primitives (such as locks, mutexes, or condition variables) are a common source of bugs. Lock contention can cause serious performance degradation as the number

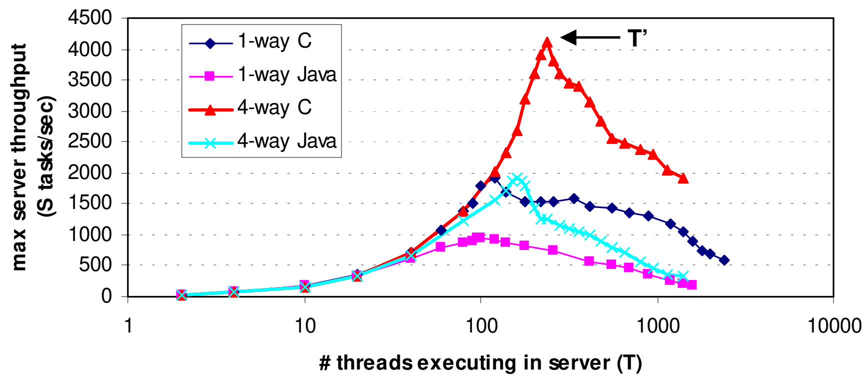


Figure 9: **Threaded server throughput degradation:** This benchmark has a very fast client issuing many concurrent 150-byte tasks over a single TCP connection to a threaded server as in Figure 8 with $L = 50$ms. We implemented the server in two languages (C and Java), and gathered measurements on two machines (a 167MHz UltraSPARC running Solaris 5.6, and a 4-way 296MHz UltraSPARC SMP also running Solaris 5.6). The arrival rate determined the number of concurrent threads; sufficient threads are preallocated for the load. As the number of concurrent threads T increases, throughput increases until $T \geq T'$, after which the throughput of the system degrades substantially.

of threads competing for a lock increases.

Regardless of how well the threaded server is crafted, as the number of threads in a system grows, operating system overhead (such scheduling and the aggregate memory footprint of the program) increases, leading to a decrease in the overall performance of the system. There is typically a maximum number of threads T' that a given system can support, beyond which performance degradation occurs. This phenomenon is demonstrated clearly in Figure 9. In this figure, we show measurements of our implementation of the threaded server; we implemented the server in two languages (Java and C), and gathered measurements on two machines (a 4-way 296MHz UltraSPARC SMP running Solaris 5.6, and a 1-way 167MHz UltraSPARC also running Solaris 5.6).

For all configurations that we tested, the thread limit T' was no more than 300 threads. While this degree of multiplexing is large for many applications and time-sharing

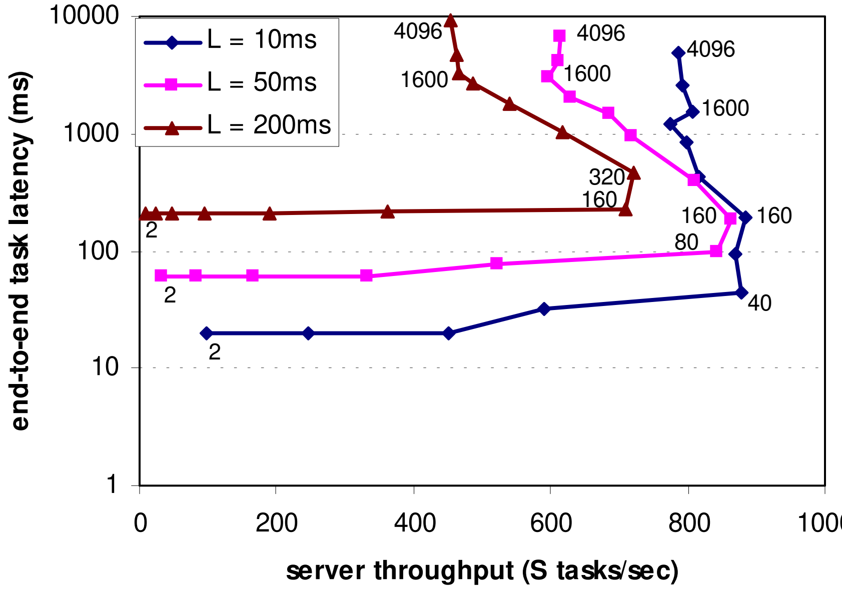


Figure 10: **Threaded server performance curve:** This parametric curve demonstrates the 1-way SMP Java server implementation's throughput and end-to-end task latency as the load on the server is increased. "L" in this graph refers to the per-task latency introduced by the server, as defined in Figure 7. The numbers next to points on the curves represent the # of parallel tasks handled by the server for those points. Note that after the system saturates, additional load drives the server into a regime where both throughput degrades and latency continues to increase.

systems, it isn't adequate for the concurrency requirements of an Internet service. There is a particularly serious implication of this: because load increases at times when the service is most valued, if load ever exceeds the thread limit, the attainable system throughput will degrade when the service needs to perform its best. The thread-per-task systems thus do not exhibit graceful degradation, one of our required service properties.

This effect can be most clearly seen in Figure 10. Each line in this graph was generated parametrically by varying the number of tasks in the closed-loop pipeline, and for each value, measuring the average end-to-end latency (i.e. the latency as measured by the client) and throughput of the 1-way SMP Java implementation of the server. End-to-end latency is plotted on the X-axis, and throughput is plotted on the Y-axis. The number of tasks in the pipeline determines the load on the server, since more tasks in the pipeline mean that the server must handle more tasks concurrently. As expected, as the system saturates under load, throughput reaches a maximum and latency begins to increase. The lack of graceful degradation can be seen as these parametric curves bend backwards, reaching a spot where system throughput degrades **and** end-to-end latency increases. In other words, once the system has saturated, additional load drives the system into a state of overload in which throughput degrades. Unfortunately, the programmer is not aware of this state of overload, as the single-task perspective of threads (and particularly of RPC) doesn't expose system-level characteristics such as the number of simultaneously executing tasks, or the number of tasks that should be admitted into the system.

3.2 Event-Driven Servers

An event-driven implementation of this server uses a single thread and non-blocking interfaces to I/O subsystems or timer utilities to "juggle" between concurrent tasks, as shown in Figure 11a. For example, if a task requires a disk read, in a threaded system a thread will block on the disk read, while in an event-driven system, a non-blocking read will be issued and a completion event will later be queued for that thread.

From the perspective of the programmer (Figure 11b), an event-driven system is structured as a single thread that loops continuously, processing events of different types from an event queue. Each long-latency, operation that would have resulted in a thread blocking in the threaded server is instead cast as an asynchronous request whose completion will trigger an event being placed on this queue. There is thus a direct correspondence between blocking operations in the threaded server and event types in the event-driven server. The program itself consists of the same stages as in the threaded case, however

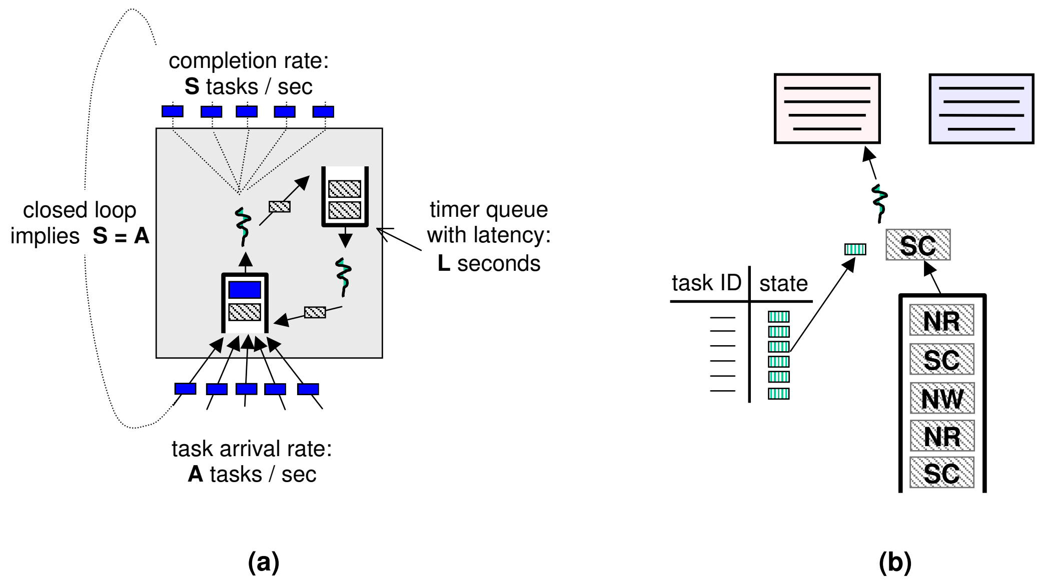


Figure 11: **Event-driven server:** (a) Each task that arrives at the server is placed in a main event queue. The dedicated thread serving this queue sets an L second timer per task; the timer is implemented as a queue which is processed by another thread. When a timer fires, a timer event is placed in the main event queue, causing the main server thread to generate a response. (b) From the programmer's perspective, the program has a single thread of execution that services completion events (NR=network read, NW=network write, SC=sleep completion) from an event queue. The program must manage all task state in a table, and dispatches events plus task state to the program's two stages.

the programmer must dispatch events to stages based on the event type, and based on knowledge of the previous history of the task associated with this event.

Unlike a threaded server, with an event-driven system, the programmer must explicitly manage all tasks' state. Partial state associated with a task must be bundled into a self-contained object, and stored in a table indexed by a unique identifier associated with the task. When the program thread pulls an event off of the queue, it associates the event with its task, retrieves this task's state, and invokes the correct task stage with the event and task state as arguments.

Event-driven programming has its own set of inherent challenges. The sequential

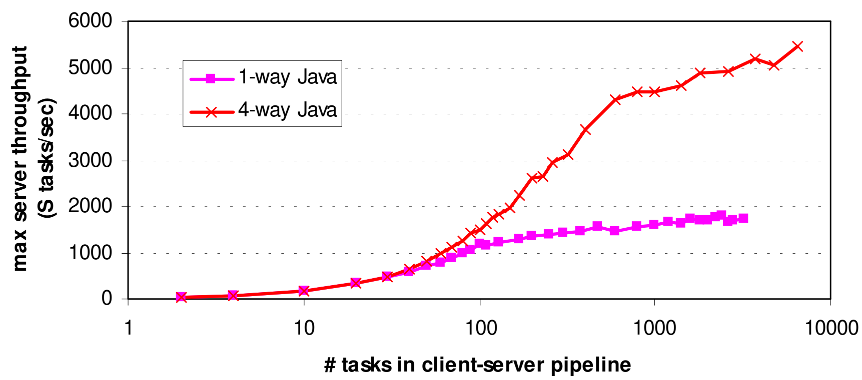


Figure 12: **Event-driven server throughput:** Using the same benchmark setup as in Figure 9, this figure shows the event-driven server's throughput as a function of the number of tasks in the pipeline. The event-driven server has one thread receiving all tasks, and another thread handling timer events. The throughput flattens in excess of that of the threaded server as the system saturates, and the throughput does not degrade with increased concurrent load.

flow of each task is no longer handled by a single thread; rather, one thread processes all tasks in disjoint stages. This can make debugging difficult, as stack traces no longer represent the control flow for the processing of a particular task. Events generally cannot take advantage of SMP systems for performance, unless multiple event-processing threads are used. Also, event processing threads can block regardless of the I/O mechanisms used. Page faults and garbage collection are common sources of thread suspension that are generally unavoidable.

However, because there is only a single thread of execution in the system, event-driven programming avoids many of the bugs associated with the synchronization of multiple threads, such as race conditions and deadlocks. In addition, concurrency is explicit in the event-driven approach, as all tasks are visible to the system programmer (in the form of task state bundles). The programmer is also able to influence the order in which tasks are processed by the system; because queues expose all events, programmers can make use of

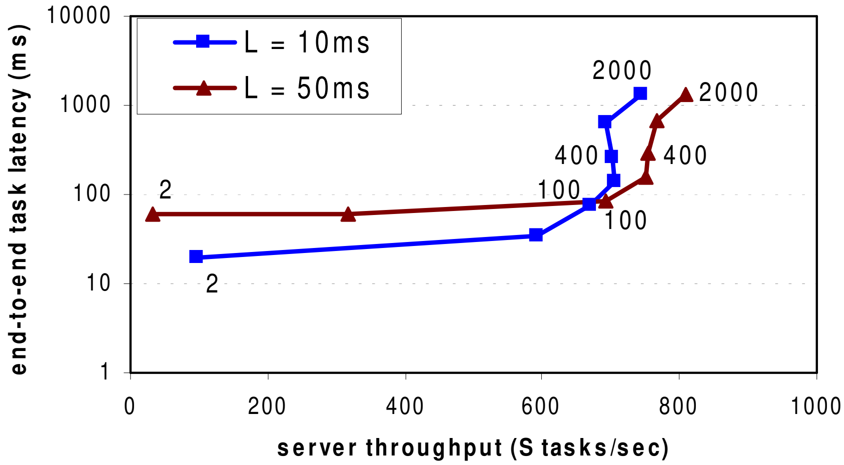


Figure 13: **Event-driven server performance curve:** This parametric curve demonstrates the 1-way SMP Java server implementation's throughput and end-to-end task latency as the load on the server is increased. The numbers next to points on the curves represent the # of parallel tasks handled by the server for those points. Note that after the system saturates, additional load increases the latency of the system, but doesn't degrade its throughput.

application-specific knowledge to reorder event processing.

Event-driven systems tend to be robust to load, with little degradation in throughput as bursts of tasks drive the offered load beyond what the system can sustain. Excess load is absorbed by the event queue, whereas in a threaded system, excess load causes additional threads to execute. Figure 12 shows the throughput achieved on an event-driven implementation of the network service from Figure 11 as a function of the load. The throughput exceeds that of the threaded server shown in Figure 9, but more importantly, throughput does not degrade with increased concurrency.

As the number of tasks increases, the server throughput increases until the pipeline fills and the bottleneck (the CPU in this case) becomes saturated. If the number of tasks in the pipeline is increased further, the excess tasks are absorbed in the queues of the system, either in the main event queue of the server, or in the network stack queues associated with the client/server transport connection. The throughput of the server remains constant in such a situation, although the latency of each task increases. This effect can be more clearly seen in Figure 13, which shows the parametric curve relating the throughput and latency of the 1-way SMP Java implementation as load is increased. Note in particular that the throughput of the system increases until the server saturates. From this point onwards, the throughput remains essentially constant, while latency increases.

An opportunity that the explicit event queue enables is the ability for the system to perform admission control. If the system is saturated, the event queue will begin to grow, increasing task latency. The system can shed load and reduce latency by simply dropping tasks from this queue, optionally sending error messages back to the sender. Queues in an event-driven system are thus a useful mechanism which exposes resources and the ability to impose resource management policies to the programmer and program.

3.3 The Thread and Event Spectrum: a Hybrid Server

We believe that the design space for concurrent servers is not limited to just threaded and event-driven systems, but rather that there is a spectrum between these extremes, and it is possible to build hybrid systems that can exploit the advantages of both programming models. For example, a hybrid system can expose a threaded programming style to programmers, while simultaneously exposing an event queue, and limiting the number of concurrently executing threads so as to prevent performance degradation. The simplest example of a hybrid thread and event server is shown in Figure 14. It limits the number of concurrent threads running in the system to no more than T threads in a preallocated thread pool, and buffers incoming tasks in an event queue from which the thread pool feeds.

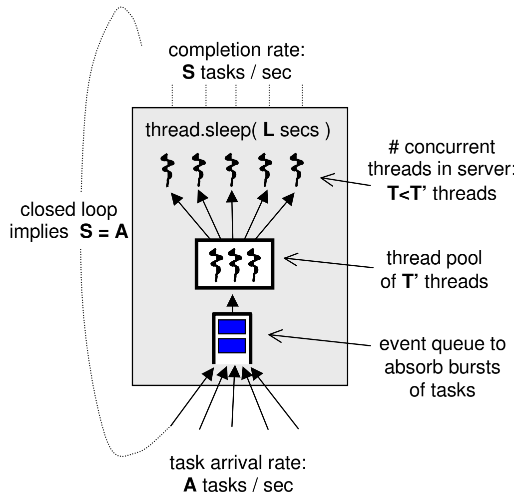


Figure 14: **A hybrid thread and event system:** This server uses a constant-size thread pool of T threads to service tasks with an arrival rate of A from an incoming task queue; each task experiences a service latency of L seconds. If the number of tasks received by the hybrid server exceeds the size of the thread pool, the excess tasks are buffered by the incoming task queue.

Returning to our example service, the concurrency demand on the system is $A \times L$, which is serviced by the T threads in the pool. Within the operating regime where $A \times L \leq T \leq T'$, the hybrid server performs as well as an event-driven server, as shown in Figure 15(a). However, if $A \times L > T'$, then it will be impossible to service the concurrency demand without creating more than T' threads, as shown in Figure 15(b). If the size of the thread pool exceeds T', throughput degrades regardless of whether the thread pool is as large as the concurrency load. Therefore, T should be set to never exceed T', and if $A > T/L$, then the excess tasks will accumulate in the task queue, which absorbs bursts but increases the latency to process each task. In this case, users experiencing this higher latency will presumably back off until the task arrival rate slows to $A = S = T/L$.

Figure 15(c) shows the performance of our Java implementation of this hybrid server, running on the 1-way Solaris machine, for various values of L. As as L increases, the

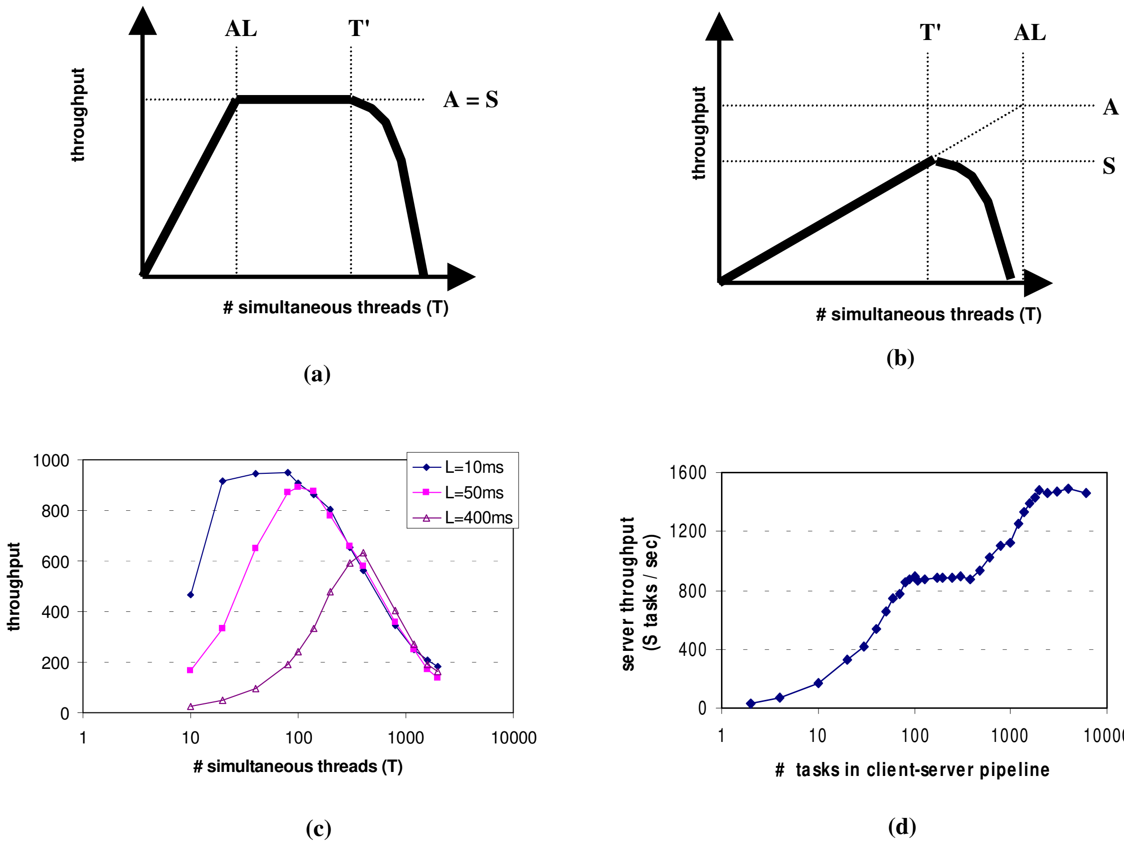


Figure 15: **Throughput of the hybrid event and thread system:** (a) and (b) illustrate the theoretical performance of the hybrid server, where T' is larger or smaller than the concurrency demand $A \times L$. (c) shows measurements of the benchmark presented in Figure 9, augmented by placing a queue in front of the thread pool, for different values of L and T. (d) shows the throughput of the hybrid server when $T = T'$, which is the optimal operating point of the server. Here, $L = 50$ms. The middle plateau in (d) corresponds to the point where the pipeline has filled and convoys are beginning to form in the server. The right-hand plateau in (d) signifies that the convoys in all stages of the pipeline have merged. Note that the x-axis of (a) and (b) are on a linear scale, while (c) and (d) are on logarithmic scales.

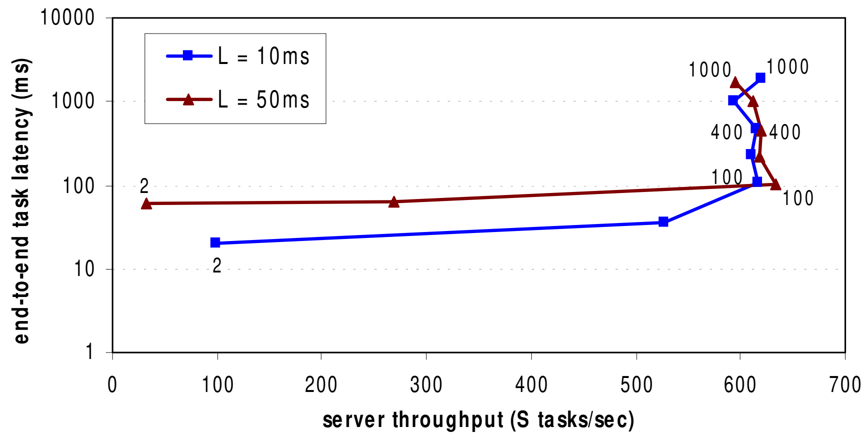


Figure 16: **Hybrid server performance curve:** This parametric curve demonstrates the 1-way SMP Java server implementation's throughput and end-to-end task latency as the load on the server is increased. The numbers next to points on the curves represent the # of parallel tasks handled by the server for those points. this hybrid server exhibits the same graceful degradation under overload as the event-driven server, since the fixed-size thread pool limits the maximum number of concurrently executing threads. After the system's throughput has saturated, additional load is absorbed on the queue, increasing latency but not degrading throughput.

hybrid system eventually becomes unable to meet the concurrency demand $A \times L$ without exceeding T'. However, because tasks are buffered by the incoming queue, throughput will not degrade as long as the size of the thread pool is chosen to be less than T', as shown in Figure 15(d).

In Figure 16, we illustrate that the hybrid server has the same graceful degradation property as the event-driven server. As the load on the hybrid server is increased, the server eventually saturates; at this point, either the CPU has saturated, or all threads in the thread pool have been dispatched. As load increases past saturation, additional load becomes absorbed on the queue. Latency increases at this point, but because the maximum number of concurrent threads is limited by the thread pool, throughput doesn't degrade.

3.4 Summary

In this chapter, we explored the distinction between threaded and event-driven programming models for high-concurrency servers. We quantitatively demonstrated that event-driven servers have as good or better throughput than threaded servers. Furthermore, as the degree of concurrency scales, event-driven servers exhibit graceful degradation; the server's event queues absorb bursts of load, resulting in an increase in latency without a degradation of throughput. Next, we introduced a hybrid model that uses admission control and an event queue to induce graceful degradation on a threaded server. In the following chapter, we generalize this notion of inducing good behavior by describing how to "condition" a service against load, concurrent, failure, and performance bottlenecks by applying several design transformations.

Chapter 4

A Design Framework for Well-Conditioned Systems

The logic of an Internet service is only a piece of its overall design complexity. In addition to functioning properly, a service must be designed to be operationally robust. A service is typically composed of a number of components, each of which may be deployed and replicated across multiple nodes in a cluster. The service's design must ensure that enough resources are provisioned to each individual component to satisfy its concurrency requirements and resource demands. Components must be adequately replicated to ensure that the service can provide enough throughput, but also to ensure that the service can withstand failures without becoming unavailable. The service also must be able to gracefully handle temporary bursts during which the offered load exceeds the capacity of the service.

We call the process of achieving this operational robustness **conditioning the service**. A necessary (but not sufficient) step in conditioning is selecting an appropriate programming model and concurrency strategy that allows the service author and the service's execution environment to observe and manage constrained resources such as threads

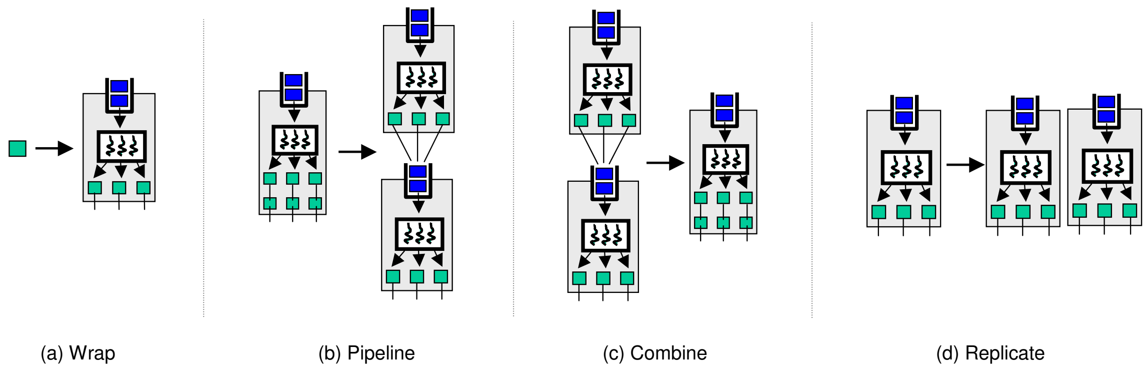


Figure 17: **The Four Design Patterns:** The four design patterns, **wrap**, **pipeline**, **combine**, and **replicate**, can be applied to stages of a service to condition it against load, failures, and limited or bottleneck resources.

and tasks being processed by the system. Additionally, an overall design framework must be considered that allows the service author to reason about the performance of individual components, but also to reason about the composition of the components and the performance and availability of the resulting service.

4.1 The Four Design Patterns

In this section, we present a design framework that was devised to provide a method for reasoning about service design, but also a set of steps for functionally decomposing a service into components that can be conditioned individually. According to this design framework, services have the task and stage abstraction previously illustrated in Figure 6. Given a request from a wide-area client, the service processes that request through a sequence of logically distinct stages, each of which is separated by a high- or variable-latency operation. For example, a web server might have three stages: reading and parsing an HTTP request from a browser, retrieving the requested file from the file system, and returning a formatted response to the browser. We impose the constraint that all data sharing between these stages is done using pass-by-value semantics, for example through the exchange of messages containing the data to be shared. This constraint acts to decouple the stages, allowing them to be isolated from each other and perhaps be physically separated across address spaces or physical machine boundaries.

Our framework offers four **design patterns** that authors and the service infrastructure can apply to compose and condition a service's stages (Figure 17):

Wrap: The wrap pattern places a queue in front of a stage, and assigns a bounded number of threads to the stage in order to process tasks that arrive on its queue; the hybrid server previously discussed in Section 3.3 is an example of a wrapped stage. The queue serves to condition the stage to load; excess work that cannot be absorbed by the stage's threads is buffered in the queue, increasing latency through the stage. This queue also serves to expose scheduling and admission control mechanisms to the stage: because the queue is explicitly exposed to the stage, it can decide the order in which to process tasks, and it can also choose to drop tasks in the case of excessive or long-lasting overload. Because threads are dedicated to the stage, the wrap pattern allows the stage to execute independently of other stages. Wrapping also introduces a **thread boundary** between stages (Figure 18): because chained stages communicate with message passing, a thread from one stage cannot directly execute the code of another stage. Thread boundaries between stages thus serve to introduce a rigid layering on the control flow across stages of a program; in Section 5.2.3 we will illustrate how thread boundaries can improve program structure, and as a result, program efficiency.

Pipeline: The pipeline pattern takes a wrapped stage, and splits it into two pipelined, wrapped stages. Pipelining further decouples a stage, and allows for increased throughput through functional parallelism across processors or cluster nodes. Pipelining permits optimizations such as having a single thread repeatedly execute the same code

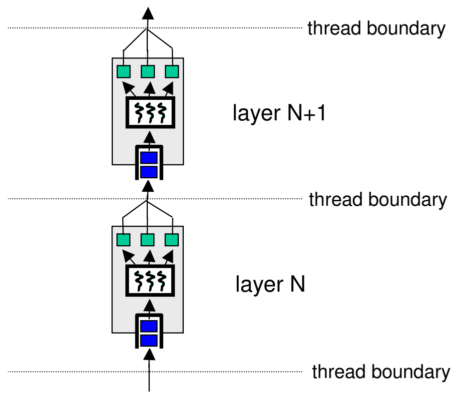


Figure 18: **Using Wrap for Thread Boundaries:** The Wrap pattern introduces a "thread boundary" between stages. Because composition across wrapped stages is done through message passing on queues, threads from one stage cannot directly call code in a wrapped stage. This thread boundary imposes a strict layering on the control flow of the program.

while processing many tasks from a queue, thereby increasing instruction cache locality. Many stages have a natural **width** to them; for example, because disks can generally only handle about 40-50 concurrent requests before saturating, there is no need to supply a stage that interacts with a disk with more than 40-50 threads. Pipelining allows stages with small widths to be extracted from larger stages, resulting in an overall conservation of threads.

Combine: The combine pattern is the logical inverse of the pipeline pattern. Given two previously independent, wrapped stages, the combine operator fuses the code of the two stages into a single, wrapped stage. Combine permits resource sharing and fate sharing between these previously independent stages.

Replicate: Given a wrapped stage, the replicate pattern duplicates that stage on a number of independent processors or cluster nodes. Replication is used to eliminate bottlenecks; by replicating a stage, the resources that can be applied to its bottleneck and therefore its width are augmented, hopefully increasing the throughput of the stage. Replication also duplicates the stage's functionality across multiple failure boundaries, making it possible to retain availability across failures. However, if a stage manages state, then replication introduces distributed state consistency issues.

4.2 Composition Operators

To design a service given the four patterns of the previous section, an author must first decompose the service into stages, apply the appropriate design patterns to the stages, and then compose the resulting conditioned stages together into a graph. Composition entails "connecting" the output messages from one source stage into the input queue of one or more destination stages. Because the source and destination could reside in different address spaces or one physically distinct machines, this connection may entail network communication and the marshalling of the message content. Whereas the application of design patterns dictates the layout of the code into stages of a service, the placement of composition operators determines the data flow across these stages.

There are many different mechanisms by which this composition could be effected. For example, two stages could be directly composed together using a best-effort unicast message delivery (e.g. UDP [21]). Alternatively, one source and many destinations could be composed together using a reliable broadcast or multicast of the message. There are five primary composition operators that we identify (Figure 19):

Direct composition: this composition operator connects the output messages of the source stage into the input queue of a single destination stage. If the two stages span an address space or machine boundary, the operator must marshall the messages and dispatch them using IPC or network connections. There are a number of variants of this operator, such as a best-effort operator which may drop messages, a reliable operator which transparently retransmits messages to achieve at-most-once delivery semantics, and a flow-

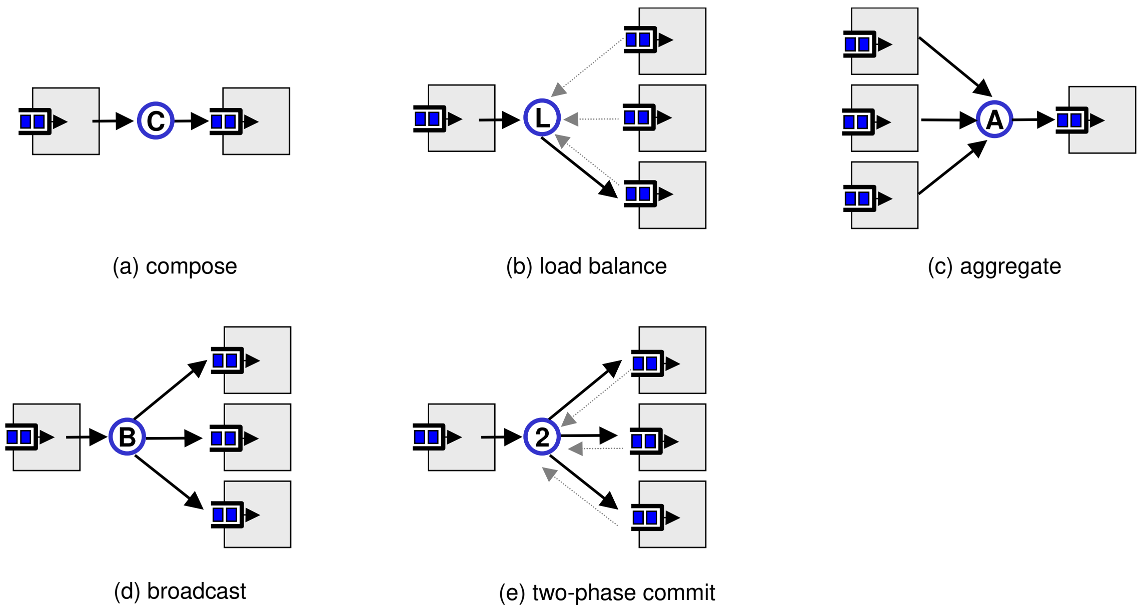


Figure 19: **Composition Operators:** Services are constructed by composing stages together into a directed graph. Composition can be done using a number of semantically different operators, including the four operators shown in this figure. Solid arrows represent data flow, and dashed arrows represent the flow of metadata or control information.

controlled operator which exerts backpressure on the source to prevent the destination stage from being swamped.

Load balanced composition: this composition operator load balances the output messages of a single source stage across the input queues of a set of destination stages. In order to perform this load balancing, load information must flow back from the destination stages into the source stage. It is possible to perform load balancing by only looking at the lengths of queues of the destination stages, but this assumes that queue length is an appropriate metric of load.

Aggregation composition: an aggregation operator merges the output messages of a set of source stages into the input queue of a single destination stage. The operator may impose one of a number of merging disciplines, including FIFO, round-robin, or a priority-based scheme.

Broadcast composition: a broadcast operator accepts messages from a single source stage and delivers them to all input queues of a set of destination stages. This broadcast may be reliable or unreliable, ordered or unordered, and it may make use of an efficient multicast-based transport, or it may rely on a broadcast mechanism built into the underlying network.

Two-phase commit composition: a two-phase commit operator functions similarly to a broadcast operator, but it performs a two-phase commit in order to atomically deliver the input message to either all output stages or none. This operator is useful in order to keep replicated stages perfectly consistent.

It is possible to create additional, more complex operators through the composition of these basic operators. For example, if an aggregation operator is composed with a load balancing operator, the result is equivalent to a distributed queue as described in [10]. However, the naive composition of these two operators would result in a bottleneck in the aggregation stage; a real distributed queue would only perform a virtual aggregation, instead directly sending data from source stages to destination stages, bypassing the aggregation stage altogether.

4.3 Constructing a Conditioned Service

Given the design patterns of Section 4.1 and the composition operators presented in the previous section, the process of assembling a well-conditioned service can now be described in terms of identifying the service's stages, judiciously applying the design patterns, and composing the resulting stages together in a graph using the composition operators. In this section, we walk through this process, using a service constructed with vSpace [27]

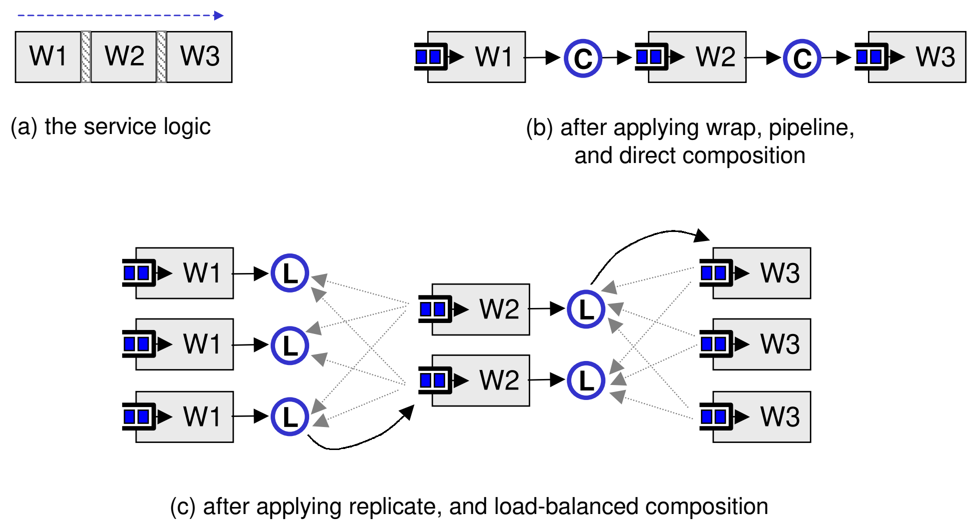


Figure 20: **A vSpace Service:** A vSpace service is a composition of workers. In (a), the service logic is broken into 3 stages. In (b), the wrap operator is used to convert each stage into a conditioned vSpace "worker"; direct composition is used to form the workers into a pipeline. In (c), each worker is replicated, and the pipeline composition is load-balanced across replicas.

service platform and programming model as a reference example. vSpace makes extensive use of the previously presented design patterns: vSpace services are constructed as a composition of "workers", each of which is equivalent to a wrapped stage as described by the design patterns. vSpace itself provides load-balancing, automatic failover, restart, and versioning of services and workers, shielding service authors from these complexities.

Logically, a vSpace service can be described as a graph of computational operations on an input task. For example, a natural language translation service could be described as a sequence of three computations (Figure 20a): speech-to-text conversion, text translation, and then text-to-speech conversion. A naive first step towards designing a service would be to create a single piece of code with these three operators directly fused together; this would result in a single-staged service. To condition this single stage to load, the wrap pattern could be applied to add a fixed-size, dedicated thread pool and queue to the stage. The presence of this queue means that the service can now smoothly handle bursts of tasks that exceed its throughput capacity. This also implies that the current load being experienced by the service can be inferred from the length of its queue.

This service can exploit parallelism across tasks (through concurrently executing threads), but it cannot easily exploit parallelism within a task. To do this, the pipeline pattern should be applied, resulting in three wrapped stages directly composed together (Figure 20b). The three stages can reside on different processors, exploiting functional parallelism within the service. The use of pipelining also means that the thread pools for each stage can now be independently sized, minimizing the overall number of threads allocated to the service. Pipelining shortens the code path executed by any given thread by constraining threads to execute only within their assigned stage. This has the opportunity for increasing the instruction cache locality of processors in the system, but only if a scheduling policy is used that executes multiple tasks within a stage before context switching to a thread from another stage.

The pipelined, wrapped service is susceptible to failure, because if any of the three stages fails, the service will become unavailable. It also cannot scale: if any stage becomes saturated, the overall service will saturate with it. To add fault-tolerance and the ability to scale, the replicate pattern can be applied to each stage. Because the stages are separated, each stage can be independently replicated; the degree to which each stage is replicated can depend on that stage's probability of failure, and that stage's resource demands. In order to route tasks to replicas, the direct composition that was used to produce the pipelined service needs to be replaced with a load-balancing composition operator. Note that the load balancing operator can use the queue lengths of the destination replicas as good indicators of their relative load. The resulting service is shown in Figure 20c.

The programming model offered by the vSpace platform requires authors to explicitly decompose their services into stages (dubbed workers). Programmatically, vSpace authors subclass a `vSpaceWorker` class in order to create a new worker. Upon instantiation, vSpace itself transparently applies the wrap pattern to a worker by associating a dedicated thread pool and task queue with it. The programming model also forces authors to establish the pipeline for their workers; the pipeline is implicitly defined by the closure of task dispatches emitted by the initial worker.

Each node in the cluster has a vSpace execution environment executing on it. Nodes cooperate with each other in order to determine automatically how many replicas of a worker should execute (based on the currently experienced load by replicas, and the available resources within the cluster). Each vSpace environment load balances locally dispatched tasks across all available replicas. Thus, the vSpace execution environment transparently applies the wrap and replicate patterns on a service, and also transparently introduces the load-balanced composition operator between workers. Service authors must decompose their services into stages and implicitly define how the pipeline pattern is applied to these stages; the remaining design patterns and composition operators are embedded inside the vSpace platform itself.

4.4 Putting It All Together

In this section, we present a detailed examination of the effects of applying the design patterns to an example service. We start with a threaded implementation of a simple service, and then successively apply the wrap, pipeline, and replicate patterns, showing the effect on both the architecture and the performance of the resulting services. All performance benchmarks were gathered on a cluster of SMPs, each of which has two 500 MHz CPUs, 250 MB of memory, and a disk with 15 MB/s sequential write capability. All nodes are connected with a 100 Mb/s shared Ethernet. We used IBM's JDK 118 with kernel threads for the purposes of this benchmark.

Our simple test service is called the "SignServer"; it was designed to represent a service that performs both CPU and I/O intensive operations. SignServer is a networked service that receives 5KB tasks from clients, and performs the following operations on each task:

1. **MD5 hash #1:** an MD5 hash is computed over the arriving packet. This operation is computationally intensive: on our benchmark machines, each MD5 hash computation takes 250 microseconds;

2. **Disk write:** after hash #1 is computed, 128 bytes are written to the disk. Disk writes are structured in such a way as to result in sequential writes. On our unloaded benchmark machines, we measured the latency of such a single disk write as 140 microseconds.

3. **MD5 hash #2:** after the disk write has completed, we then compute a second MD5 hash over the input packet.

4. **Writeback:** finally, after the second MD5 hash has been computed, the results of the MD5 hashes are written over the network to the originating client.

Our experimental setup is based on the concurrent server model depicted in Figure 7. To gather our measurements, we set up a closed-loop pipeline of requests between client nodes and servers, and varied the number of tasks in the closed-loop pipeline.

The source code for all server implementations is included in Appendix A.

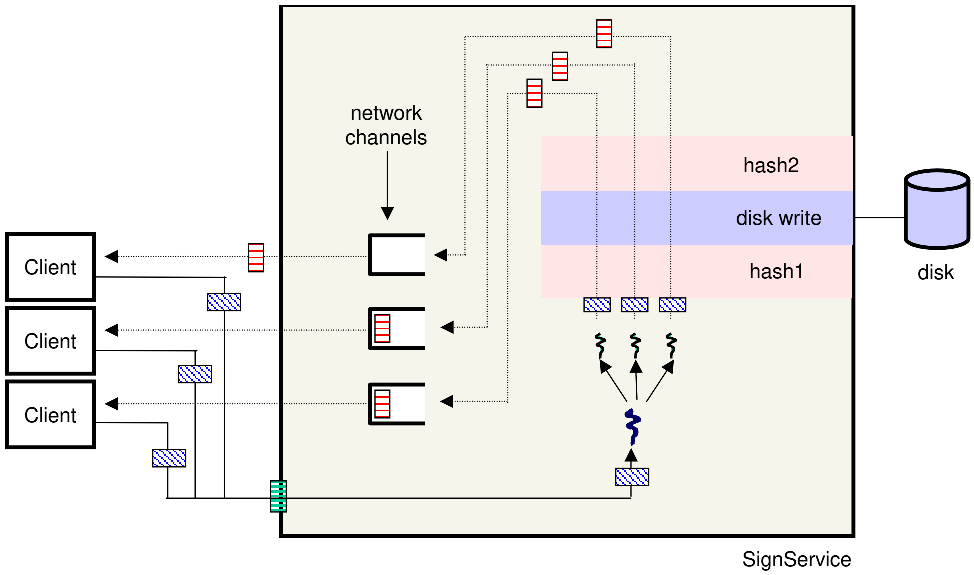


Figure 21: **Threaded SignServer Architecture:** This diagram illustrates the architecture of the thread-per-task implementation of the SignServer.

4.4.1 The Threaded SignServer

In Figure 21, we show the architecture of our initial thread-per-task implementation of this service. Each arriving task is dispatched to a thread from a thread pool. The dispatched thread sequentially shepherds its task through the first three operations (hash #1, disk write, and hash #2). The thread therefore is actively performing computation during hash #1 and hash #2, but is blocked during the disk write. After completing these three operations, the thread drops a response packet into the network channel to the client, and then rejoins its thread pool to await a new task.

In Figure 22, we show the performance of the threaded server. Figure 22(a) shows that the throughput of the server increases with the number of simultaneous tasks, until it reaches a maximum throughput of 750 requests/s for 270 tasks. For greater numbers of simultaneous tasks, throughput degrades. Similarly, latency increases linearly until 270 simultaneous tasks, after which it increases superlinearly. In Figure 22(b), we show a

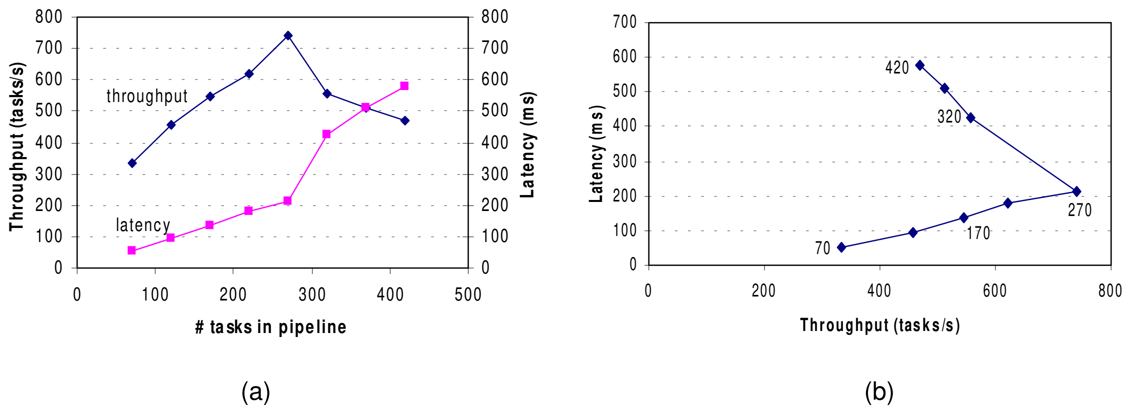


Figure 22: **Threaded SignServer Performance:** In (a), we show the throughput and latency of the thread-per-task SignServer as a function of the number of simultaneous tasks in the pipeline. In (b), we show a parametric curve showing the relationship between throughput and latency as the number of tasks in the pipeline is varied. The value of the parameter (# of simultaneous tasks) is displayed next to a number of points on the curve.

parametric curve of throughput versus latency as a function of the number of simultaneous tasks. We see that throughput increases until the server saturates, but then throughput degrades while latency continues to increase.

4.4.2 The Wrapped SignServer

The throughput degradation exhibited by the thread-per-task implementation is identical in nature to that exhibited by the example threaded server in Section 3.1. Increasing load on the server causes additional threads to executed; once too many threads are active b, the overhead associated with the threads (such as context switching, scheduling, and memory footprint) causes the throughput of the server to degrade.

The "wrap" design pattern is meant to condition a service against load. In Figure 23, we illustrate the architecture of an event-driven version of the server. Instead of dispatching threads to handle incoming tasks, tasks are placed on a queue; a single thread picks up tasks from this queue and either performs one of the hash functions on the task,

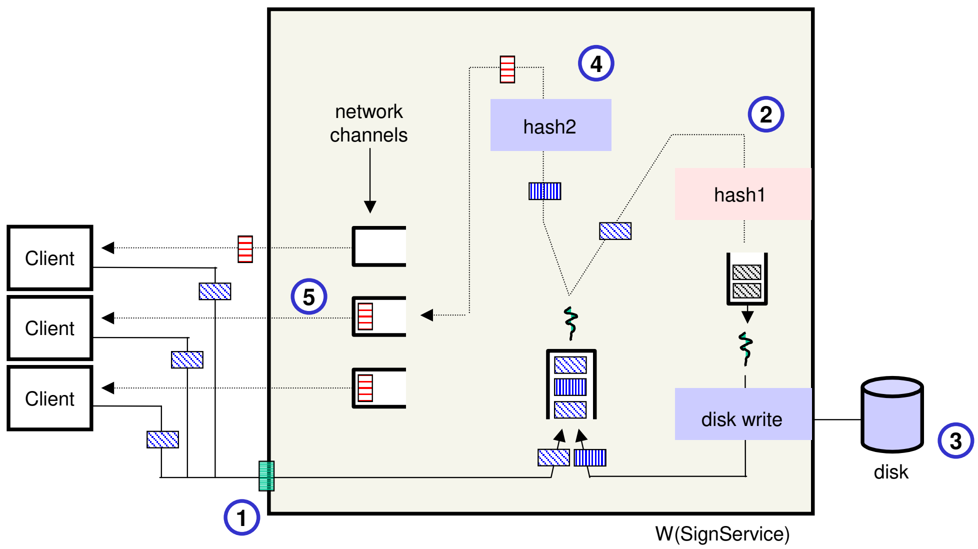


Figure 23: **Wrapped SignServer Architecture:** This diagram illustrates the architecture of a single-threaded, wrapped implementation of the SignServer. Incoming tasks (1) are placed on the single server thread's queue. The thread dequeues tasks, applying hash function #1 (2) and then issuing an asynchronous disk write (3). The disk write completions flow back onto the main queue. The single server thread dequeues completions, and applies hash function #2 (4). After this hash has completed, response packets are enqueued on network channels destined for the originating client (5).

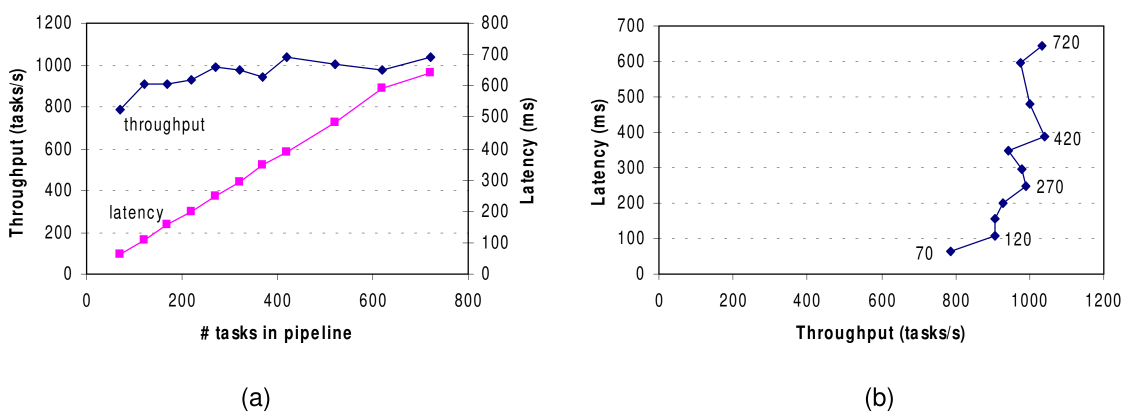


Figure 24: **Wrapped SignServer Performance:** In (a), we show the throughput and latency of the wrapped SignServer as a function of the number of simultaneous tasks in the pipeline. In (b), we show a parametric curve showing the relationship between throughput and latency as the number of tasks in the pipeline is varied. The value of the parameter (# of simultaneous tasks) is displayed next to a number of points on the curve.

or initiates the asynchronous disk write for the task. This queue/thread combination is the result of applying the "wrap" pattern with a minimally sized thread pool (i.e. a pool of size one).

In Figure 24, we show the performance of the wrapped server implementation. Figure 24(a) shows that the throughput of the wrapped server increases with load, until it saturates at approximately 1000 requests/s, which is faster than the saturated thread-per-task server. Unlike the threaded server, additional load does not cause the performance of the server to degrade; the wrap pattern has successfully conditioned the server against load. The latency of the wrapped server increases linearly with additional load. In Figure 24(b), we show the parametric performance curve of throughput versus latency as a function of load; this curve clearly illustrates that after the server has saturated, additional load increases latency but does not degrade throughput.

4.4.3 The Wrapped, Pipelined SignServer

The wrapped server has only a single thread that performs both hash functions, even though the node on which the server runs has two processors. Although the second processor is partially utilized by handling network and disk traffic, it could be further utilized by having a second service thread performing the hash computation.

In Figure 25, we show the result of applying the pipeline design pattern to the wrapped SignServer implementation. Instead of having one service queue that feeds a single thread, the pipelined implementation has two queues and two threads. The first of these threads pulls tasks received from the network off of its queue, and applies hash #1 to them. This thread then issues an asynchronous disk write, and then returns to its own queue to service the next task. The completion from the disk write flows into the second threads' queue; this second thread dequeues completions, applies hash #2, and then places

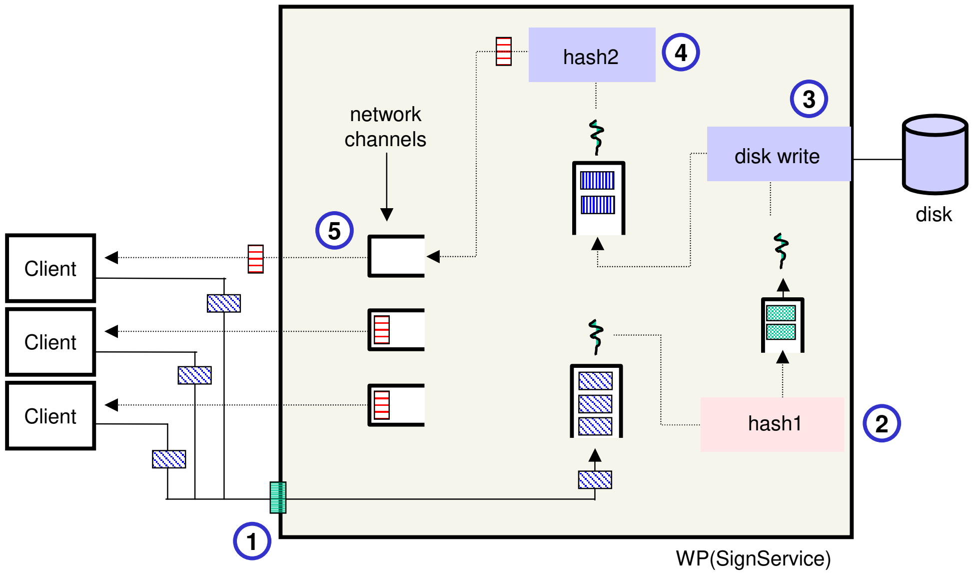
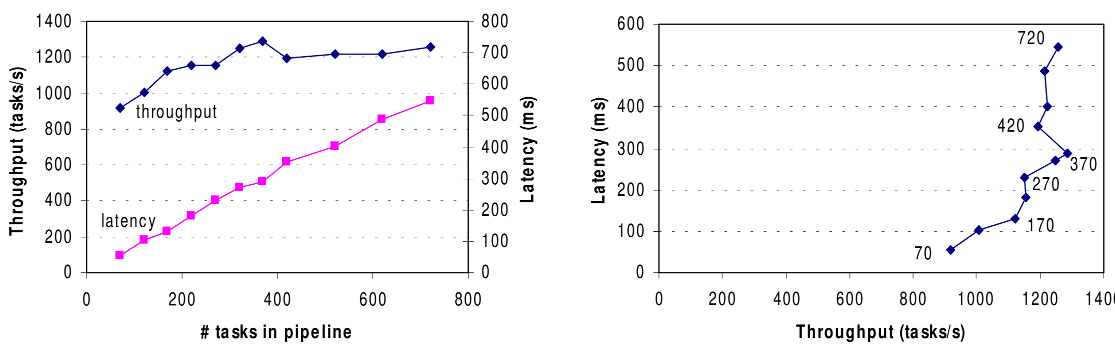


Figure 25: **Wrapped, Pipelined SignServer Architecture:** This diagram illustrates the architecture of the wrapped, pipelined SignServer. Incoming tasks are placed on a queue (1). A thread dequeues tasks, applies hash #1 to them (2), and then issues an asynchronous disk write. The write completions (3) flow onto a second queue. A second thread dequeues completions, applies hash #2 to them (4), and then enqueues a response into a network channel destined for the originating client (5).



(a) (b)

Figure 26: **Wrapped, Pipelined SignServer Performance:** In (a), we show the throughput and latency of the wrapped, pipelined SignServer as a function of the number of simultaneous tasks in the pipeline. In (b), we show a parametric curve showing the relationship between throughput and latency as the number of tasks in the pipeline is varied. The value of the parameter (# of simultaneous tasks) is displayed next to a number of points on the curve.

a response packet in an outgoing network channel destined for the originating client. Thus, in this pipelined implementation, the server threads can execute the two hash stages in parallel on the server node's two CPUs.

The performance of the wrapped, pipelined server is shown in Figure 26. In (a), we show that the throughput of the server increases with load until the server saturates. The saturated throughput is 1200 requests/s; by pipelining the hash function computations, we successfully increased the maximum throughput of the server. In (b), we show the parametric performance curve; like the wrapped server, this curve clearly demonstrates that applying additional load to the server after it has saturated increases latency but does not decrease throughput.

4.4.4 The Wrapped, Pipelined, Replicated SignServer

The wrapped, pipelined server successfully saturated both CPUs of the server's 2-way SMP. To increase the throughput of the server beyond that which can be achieved on this 2-way SMP, we must apply the replicate pattern to duplicate the server on more than one node. In Figure 27, we show the results of such replicate. Instead of sending tasks to a single server, clients now spread tasks across multiple servers. Our implementation of this balances tasks across two servers. We achieve load balancing across the servers by maintaining separate closed-loop pipelines from each client to each server; because the pipelines are all disjoint, in steady state each client will generate load at the same rate that completions arrive from the server.

The performance of the wrapped, pipelined, replicated server is shown in Figure 26. As before, in (a) we show that the throughput of the server increases with load until the server saturates. The saturated throughput is 1800 requests/s; by replicating the server, computations, we successfully increased the maximum throughput of the server. Note that

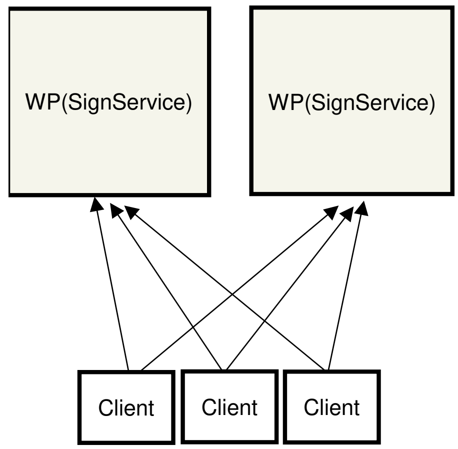


Figure 27: **Wrapped, Pipelined, Replicated SignServer Architecture:** This diagram illustrates the architecture of the wrapped, pipelined, replicated SignServer. Two instances of a wrapped, pipelined server are run on different nodes; each client spreads its tasks across the two server instances.

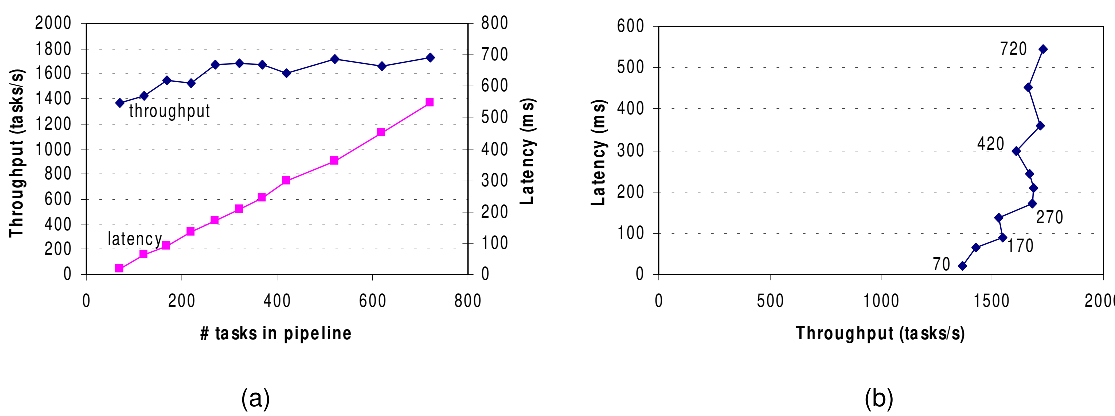


Figure 28: **Wrapped, Pipelined, Replicated SignServer Performance:** In (a), we show the throughput and latency of the wrapped, pipelined, replicated SignServer as a function of the number of simultaneous tasks in the pipeline. In (b), we show a parametric curve showing the relationship between throughput and latency as the number of tasks in the pipeline is varied. The value of the parameter (# of simultaneous tasks) is displayed next to a number of points on the curve.

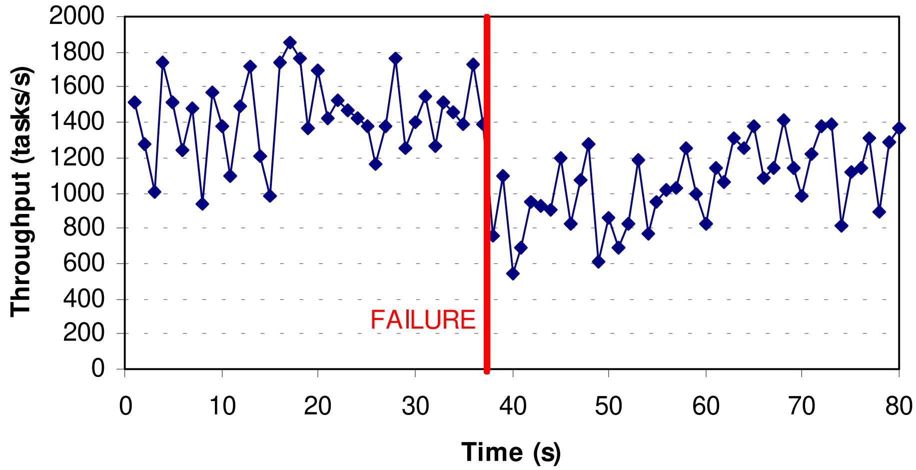


Figure 29: **Fault Tolerance in the Wrapped, Pipelined, Replicated SignServer:** This chart shows the throughput of the replicated server over time. After 37 seconds, we deliberately crashed one of the two servers. The clients detected this crash, and began routing all tasks to the surviving server; because of this, the system continued to operate, although at a diminished capacity.

even though we added a second server, the maximum throughput did not double. At 1800 requests/s, the total load on the network was 72 Mb/s, causing the network to saturate and become the bottleneck of the system. In (b), we show the parametric performance curve; once again, this curve clearly demonstrates that applying additional load to the server after it has saturated increases latency but does not decrease throughput.

In addition to increasing the capacity of the system, the replicate pattern also introduces redundancy to the system. Because there are two SignServer instances running on two different nodes, if one node fails, the system can continue to execute (but at lower capacity). Figure 29 demonstrates this fault-tolerance ability.

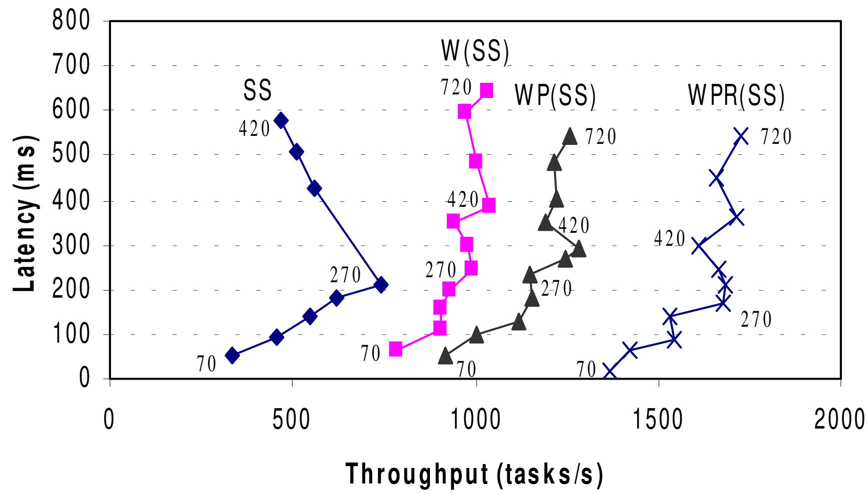


Figure 30: **Performance Comparison between Servers:** This graph shows the parametric curves of throughput version latency as a function of load, for all four server implementations. SS = thread-per-task SignServer. W(SS) = wrapped SignServer. WP(SS) = wrapped, pipelined SignServer. WPR(SS) = wrapped, pipelined, replicated SignServer.

4.4.5 Summary

In Figure 30, we compare the parametric performance curves for all four server implementations. The most obvious feature of this graph is that the application of each additional design pattern increased the throughput capacity of the system. **Wrap** eliminated the overhead of large numbers of threads, **pipeline** added functional parallelism of hash computations across two CPUs, and **replicate** added data parallelism of tasks across cluster nodes.

In addition, **wrap** successfully conditioned the server against load, resulting in graceful degradation when excess load is generated. Finally, **replicate** caused the system to be fault-tolerant.

Chapter 5

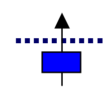
The I/O Core

We have implemented a Java-based programming library (called the **I/O core**) that makes it simple for both service authors and the service's execution environment to apply the design framework of the previous chapter to pieces of code. The I/O core provides a set of uniform abstractions for interacting with disks, network peers, and queues, as well as for generating events and composing together software modules that are built with the design framework in mind. All network communication and disk I/O provided by the library exports a non-blocking, asynchronous event-driven style of programming. This event-driven style nicely matches the task-driven composition of stages, and as seen in Section 3.2, facilitates scaling to many thousands of concurrent tasks, including network connections and disk interactions. In this chapter, we present the design, implementation, and performance of the I/O core.

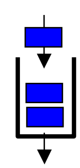
5.1 Interface Design

The I/O core is largely defined by its programmatic interfaces. The design of these interfaces was extremely subtle, as they influenced the programming model and structure of both the I/O core but also the structure and performance of systems built on top of it, as we will later show. We heavily exploited the object-orientedness and strong typing of Java [62] in our API design. For example, all of the interfaces discussed below are Java `interface` types, and are declared independently of their implementation, which allowed us to experiment with a number of semantically equivalent but structurally different implementations of the I/O core. There are several major abstractions defined in the I/O core interface; we discuss each in turn.

Memory regions: A `MemRegionIF` interface abstracts away the behavior of a memory region or data element. Through this interface, regions of memory can be copied, created, or updated. All events and messages used in the I/O core are wrapped behind this interface. Having this abstraction allows several different memory management styles to be used, including the garbage collected byte arrays that are naturally supported by Java, but also user-managed pinned physical memory regions such as those required by the Via user-level network [24].



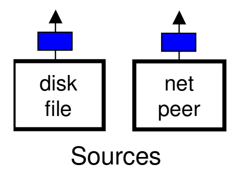
Event handlers: Events and messages can be directed to one of a number of types of components, as we will describe below. The ability to receive an event is abstracted behind an event handler interface called `UpcallHandlerIF`.[1] This event handler interface allows both single events and batches or arrays of events to be passed in to the handler.



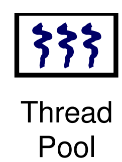
Sinks: A `SinkIF` interface is an abstraction for performing writes to an I/O destination. Callers can synchronously or asynchronously drop data elements into a sink. Elements are serialized in the sink and drain to an I/O device in the background. When an element has drained, the sink generates a completion event (directed to a caller-specified

[1]The name `UpcallHandlerIF` is a misnomer, as more than just upcalls are represented by this interface. All event handlers extend `UpcallHandlerIF`.

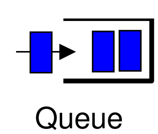
`UpcallHandlerIF`) in order to notify the caller that the element has reached the device. Currently, we have implemented both disk and network extensions of the basic sink interface. A disk sink sequentially writes data into a raw disk segment or file, and a network sink provides best-effort packet delivery to a peer.



Sources: A source is the logical opposite of a sink; it is an I/O device that "spontaneously" emits data that is wrapped in a `MemRegionIF` and directed to an application-specified `UpcallHandlerIF`. Currently, network peers and disk files are examples of sources. An event is emitted when a packet arrives from a network peer, and an event is emitted when a disk read completes (although a caller had to request this data originally, so a disk completion is not really spontaneously emitted).



Thread pools: A thread pool is an abstraction that provides concurrency to otherwise blocking operations. The thread pool interface permits the caller to dispatch a thread from the pool to a caller-specified method. If there are no more threads left in the pool, this dispatch request is queued up for later dispatch. Many thread pool implementations are possible, including fixed-sized thread pools and adaptive thread pools that grow and shrink according to currently experienced demand). A thread pool is one of the two abstractions that are needed to apply the wrap pattern to a task stage.



Queues: A queue provides a generic mechanism for buffering events. The queue interface extends `UpcallHandlerIF`, and thus is a composable element that can be used to receive completions from sinks and events emitted from sources. The interface also supports a number of styles of notification; a caller can use polling, timed waiting (similar to the UNIX `select()` system call), or blocking in order to receive elements from a queue. The queueing discipline used is hidden inside the queue implementation, although a default FIFO queue is provided to service authors. Queues can also be composed in a chain with other event handling interfaces; when composed, events that end up in the queue will be automatically forwarded to the next event handler in the chain.

5.1.1 Pitfalls

There are many subtleties in the specific design details of the I/O core's interfaces: to get them correct, we needed to go through six iterations of the interface design process. In this section, we present a few of the more important design decisions that we uncovered.

Because all of the interfaces in the I/O core are non-blocking interfaces, there is a disjunction between the context in which an I/O request was issued from that in which it completes. To overcome this, some demultiplexing information must be kept by the system to associate I/O completions with the tasks for which the I/O was generated. In the first few versions of this interface, the I/O core didn't provide any information to the receiver of a completion event beyond the event type and data content itself. This forced callers to peek inside the data payload in order to determine appropriate demultiplexing of completions, which is of course awkward. The next version of the interface returned a unique ID to the caller on every I/O dispatch, and passed the same ID to the receiver of completion events; by matching up these ID's, the receiver could thus pair requests with completions, thereby performing this demultiplexing.

However, even this proved to be limiting; if multiple applications wanted to share an I/O channel, then they needed to have a shared demultiplexing table in which to store these ID's, otherwise completions could not be routed to their appropriate application. While it was possible to build such a shared table, it forced applications to be aware of each other. To overcome this, the final version of the interface allowed callers to specify a completion handler for every I/O request that was issued, instead of specifying a completion handler for an I/O channel. By doing so, completions are naturally routed to their correct application.

A related demultiplexing issue arose for network I/O. Early versions of the I/O core had a limited namespace for network peers; multiple applications running on a single host would need to share the same network peer name, and thus demultiplexing messages to multiple receivers on the same host required introspection in the data payload. To overcome this, the final version of the interface allowed applications to register port numbers to completion handler interfaces; senders specify a port number as well as a peer name when dispatching a message, allowing message reception events to be immediately demultiplexed to the appropriate application event handler.

Deciding upon a name space for network peers was challenging as well. One goal of the I/O core was to allow for the possibility of multiple, independent transport layers underneath a common network sink and source API, but each transport would likely have a different name space. To overcome this, the abstract network source and sink API has a flat, string-based name space. The underlying transport implementation is required to parse whatever structure is necessary out of these flat names; for example, our TCP-based transport layer expects names of the format `hostname:portnum`. We further assert that no code should statically declare any peer names, but rather should read names out of a configuration file. This file can be modified in order to switch transports, and thus can be constructed to contain appropriately structured network names.

An asynchronous interface, by nature, allows callers to inject multiple requests into the system before any completions occur. This introduces the design decision of the order in which requests are issued to the underlying I/O device, and the order in which request completions are returned to the application. For both network and disk writes,

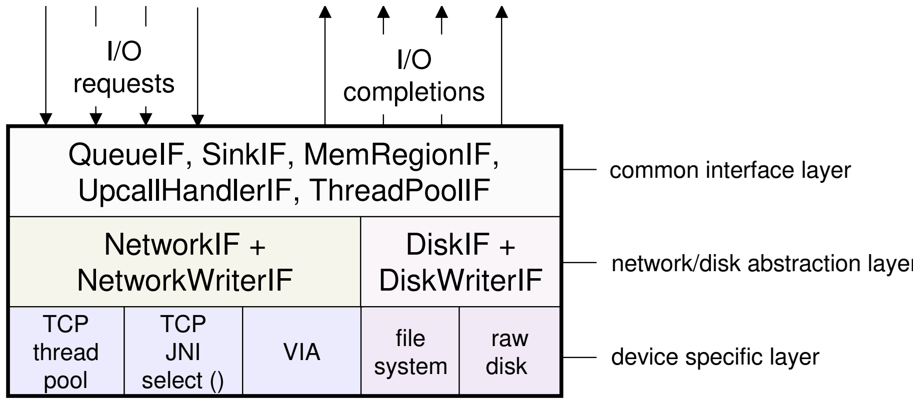


Figure 31: **I/O core structure:** The I/O core has three layers to it: the common interface layer defines abstractions such as sinks, queues, and event handlers. The network/disk abstraction layer consists of source and sink interface extensions that are specific to networks and disks (e.g., defining the ability to open a connection to a network peer). The device specific layer consists of implementations of the network/disk abstraction layer for particular devices, such as a Via user-level network stack or a raw disk.

we decided to maintain a strict sequential order in the requests, ensuring that requests are delivered to the underlying device in the order that they were received in. This strict ordering is important for applications that must control order to maintain consistency; for example, file system metadata writes must be ordered to ensure the consistency. However, we decided to allow reads to be satisfied out-of-order (this is obviously only relevant for disk I/O, since network reads can only finish when packets arrive). By allowing this, multiple disk reads could be issued to the disk device driver simultaneously, allowing it to more optimally schedule disk seeks.

5.1.2 Code Structure

The interfaces and implementation of the I/O core is structured as three layers, as presented in Figure 31. We discuss each in turn.

Common interface layer: This layer presents all of the interfaces previously discussed in Section 5.1. These interface definitions include sinks, queues, events, event handlers, and thread pools. An application that uses the I/O core will predominantly make use of this layer, since code that uses it is independent of particular I/O devices and device classes (such as storage subsystems or network peers). Abstractly, such code processes events, but doesn't necessarily care where those events came from. This implies that the source of these events could be established dynamically; this would make it very simple to seamlessly add mechanisms like network-backed paging or storage access, for example.

Network/disk abstraction layer: This layer extends the `SinkIF` interface in the common interface layer to add semantics that are particular to a specific device class, such as a network or a storage subsystem. In addition, it defines source interfaces for these device classes, allowing callers to request disk blocks, or establish connections to a named network peer. These extensions and source definitions are still abstract, in that they have no implementation, and do not define any semantics particular to any particular network transport or storage device. Only a small amount of application code is expected to interact with this layer; this code will typically be used to establish event bindings between the rest of the application and particular I/O device classes, for instance setting up a network packet reception channel between a particular named peer and an event handler.

Device specific layer: This layer consists of device-specific implementations of the network/disk abstraction layer interfaces. Currently, we have implemented 4 device-specific implementations. The first three are extensions of the basic network interfaces; one uses the underlying blocking TCP stack APIs presented by the OS, a second uses the Java Native Interface (JNI) [122] in order to access the non-blocking `select()` system call but also uses the TCP stack, and the third uses the Jaguar [132] mechanism to access a Via network implementation running on top of Myrinet [102]. The fourth extension implements the storage interfaces on top of the standard file-system interface presented by the OS. A hypothetical (but currently unimplemented) fifth extension is shown in Figure 31, namely a raw disk extension to the storage interfaces. Application code should be completely unaware of these device-specific interfaces, with the possible exception of needing to pass correct arguments to class constructors at application initialization time.

A great success of the I/O core is that we were able to implement all of these different device-specific extensions without requiring changes to the network/disk abstraction layer, the common interface layer, or applications that were using the I/O core. In particular, we were able to seamlessly introduce the JNI `select()` version of the network interfaces, resulting in a significant performance and concurrency scaling increase with no change to applications.

5.2 Disentangling Control Flow

The I/O core interfaces define event sources and sinks that can be composed together into an event flow graph to define an application. Queues, sinks, and sources all support an interface that allows callers to direct events or completions generated by these components to event handler interfaces of their choice. Sinks, queues, and application code can implement this event handler interface; it is thus possible to establish an event graph (either statically or dynamically), and then have a long sequence of events automatically flow through this graph, as exemplified by Figure 32.

A significant design decision related to these event flow graphs is that of picking a control flow model for deciding when and under which execution context events will flow from component to component. For example, it is possible to associate threads only with application code, and have the application poll queues to determine whether events

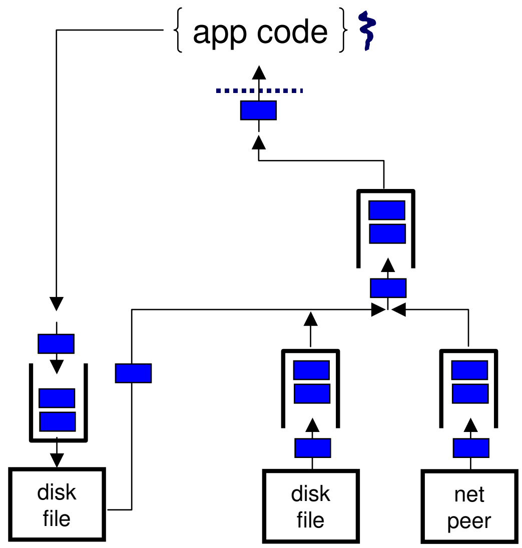


Figure 32: **Example I/O core event flow graph:** In this example I/O core application, data flows from a disk file source into a queue, and from a network peer source into another queue. Data from these two queues is aggregated into a third queue, which also receives completion events from a disk file sink. Events from this aggregation queue flow into an application-defined `UpcallHandlerIF` event handler, which processes them, and generates data to be sent to the disk file sink.

are available; in such a scheme, polling a high-level queue would automatically trigger polls to any other queues further down the flow graph. Alternatively, it is possible to associate threads only with queues and sources, resulting in events percolating upwards through the flow graph, eventually causing an upcall into an application-specific event handler. We experimented with three different control flow designs with the I/O core, each of which is presented in the following sections. As we will discuss, we discovered that these design decisions had very significant implications on application throughput, latency, code structure, code simplicity, and reliability.

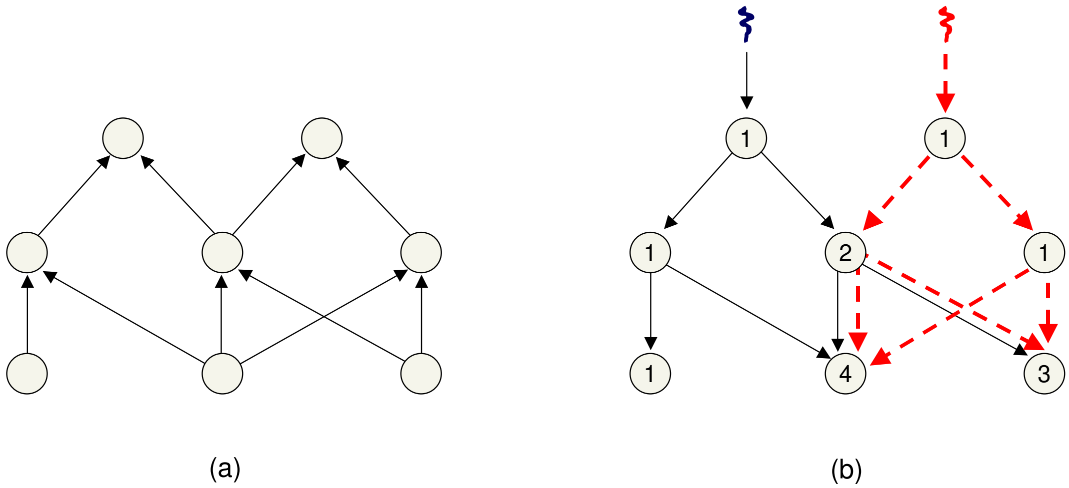


Figure 33: **Polling-based control flow:** (a) Illustrates a hypothetical composition of elements in an I/O core graph. Only upwards flowing composition is shown, i.e. the flow of completions or data upwards through a layered system. (b) Shows two threads polling for completions on the top-most layer, and the cascade of downward polls that this triggers. The in-degree of each node is labelled, and represents the number of times that node is polled by the two threads.

5.2.1 Polling

Our first attempt at solving this control-flow problem was to base all event flow on polling. In this model, an event is queued inside the component that it is initially delivered to, whether the component is a source, a queue, or the completion path from a sink. If a downstream component is composed with an upstream component, then the downstream component must poll the upstream component in order to pull enqueued events from it. Thus, if a long composition path is built, polling the component that is furthest downstream will cause a "ripple" of polls up the composition path. This composition strategy was chosen because our first implementation of several sources, including a Via network source, required callers to poll them for completions and data delivery.

In Figure 33a, we show a hypothetical composition graph of elements. This graph is meant to represent a layered system, in which completions and arrivals from I/O elements flow upwards through the layers of the system. In Figure 33b, the graph of poll operations that result when two independent threads poll the two top-level elements is shown. There are a number of very important characteristics of this poll graph that had significant performance and software engineering implications.

Firstly, note that the two thread contexts ended up duplicating effort. The in-degree of each node in the poll graph (labelled on the node) represents the number of times that node was polled. Some elements are polled more than once, even up to four times for one of the bottom-layer elements. This duplication of effort happens for two separate reasons. The first is that nodes in Figure 33a may have more than one parent (i.e. the graph is not a tree). Because of this, a traversal of the graph in Figure 33b may visit a given node more than once. The second reason for the duplication of effort is that the two threads are unaware of each other's presence, and thus even though the first thread may have recently polled a node, the second thread may poll it again if that node is also in its poll graph. This duplicated effort has performance implications; polls on empty queues are wasted CPU cycles, and the duplication of effort results in an amplification of these wasted CPU cycles.

A second implication of the poll graph is that multiple threads may be simultaneously polling elements of the graph, and that multiple threads may be simultaneously executing in application-defined event handlers. Because of this, all elements of the composition graph and all application-defined event handlers must employ mutexes or locks to prevent race conditions. This introduces significant complexity; building reentrant code is known to be error prone [115].

Another implication relates to the fact that a given node in Figure 33a may have more than one parent. Logically, this means that the node is being multiplexed by more than one caller. This means that events and completions flowing from this multiplexed node must be demultiplexed across the downstream nodes. Unfortunately, because downstream nodes do the polling, the demultiplexing must be done at the time that the poll happens, and in the thread context of the downstream node. This is adds even more complexity, as it implies that the poller must examine events on the polled queue, and decide which of those events are appropriate to pull.

Experience with this poll-based model showed that its performance was extremely poor, and worse, degraded as the number of elements or polling threads increased. Furthermore, the design of an application that uses this model is complicated by the need to decide upon an appropriate polling rate. The best possible polling rate is a function of the hardware speed, operating system overhead, graph structure, and number of concurrent application threads. As such, this polling rate cannot be easily determined, introducing either "voodoo constants" or a very difficult adaptation problem.

5.2.2 Unstructured Upcalls

Our second attempt at building a control-flow model inverts the control-flow graph of Figure 33b by replacing upstream-flowing polls with downstream-flowing "upcalls". In this model (illustrated in Figure 34), thread contexts live in I/O sources. When an I/O completion event or a data arrival happens, a thread context is emitted from the source (the lowest-level elements in Figure 34b) and pushes the data upwards (i.e. downstream) through the composition graph. Multiple thread contexts may emerge from a single I/O source, and multiple I/O sources may emit threads simultaneously.

An additional wrinkle introduced in Figure 34b is that the thread context used to deliver an upcall can be diverted by the callee in order to do other work, such as performing a downcall to dispatch additional work. For example, consider the case of a disk buffer cache. A read miss on a buffer cache will cause a disk read to be issued to a disk source.

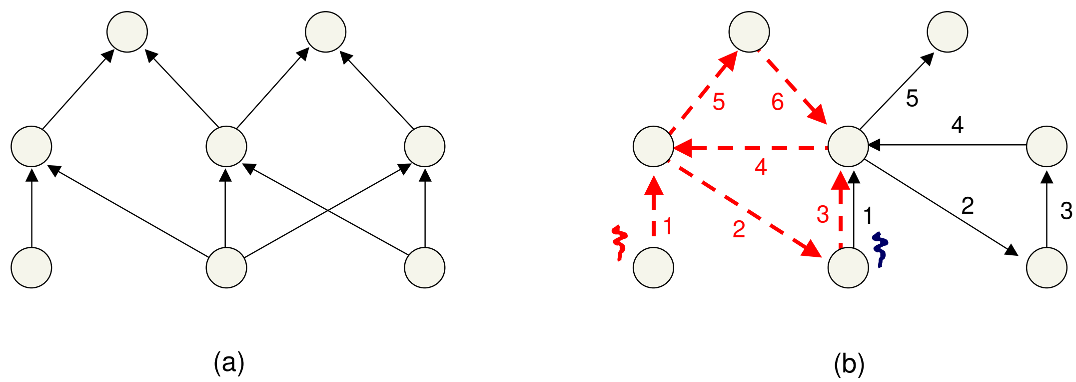


Figure 34: **Unstructured upcall-based control flow:** (a) The same composition graph as in the previous figure. (b) Shows two thread contexts pushing events upwards through the composition graph, but also shows I/O request downcalls, and the resulting "spaghetti" control flow. The labels next to edges show the order of control transfer across the elements in the graph.

When the read completes, an upcall will be issued into the buffer cache; if the buffer cache performs prefetching, it may want to immediately issue another read request. In this case, the thread context used to deliver the read completion upcall to the buffer cache is diverted downwards to dispatch a prefetch read. After this diversion, the thread context returns to the buffer cache, which then likely dispatches an upcall to the application to deliver a buffer cache read completion.

This upcall-based model eliminates the terrible inefficiency of the poll-based model, because work is only performed when events arrive. There is no speculative polling: threads are emitted by sources only when completions or data arrivals occur. However, like the upcall-based model, multiple threads may simultaneously enter a given node in the graph, and thus queues and application-defined event handlers need to use mutexes or locks in order to be reentrant. A subtle implication of this is that unless nodes are treated as monitors (in that the mutex or lock is released if a thread temporarily diverts outside of the monitor), it is possible for the system to deadlock by having the paths of two threads overlap.

The problem of demultiplexing events across downstream elements is simplified by this upcall-based model as well. Because this model is "push" oriented, the thread context that delivers events can examine an event before delivering it, and at that point can decide upon the appropriate downstream branch to take in the composition graph. The "push" threads route events to downstream branches, whereas in the poll-based model, "pull" threads need to perform a selection-search on events residing in upstream nodes. Furthermore, the ability for callers to specify an `UpcallHandlerIF` completion handler for each event (as discussed in Section 5.1.1) eliminates the routing decision; demultiplexing information is provided by downstream nodes when they originally dispatch I/O requests.

Experience with this model uncovered a major complication. Because of the ability of upcall handlers to divert control flow, it became extremely difficult to predict the sequence of nodes in the composition graph that a given upcall thread would visit. Furthermore, it was possible for threads to enter into cycles in the composition graph by having a downwards diversion trigger an upcall. Even with relatively simple composition graphs, a thread-aware debugger revealed that many threads in the system would have call-depths of over 30 or 40 nodes, and that they had followed many cycles.

In addition to increasing the chance of deadlock for those nodes that weren't built as monitors, this had the effect of depleting the thread pools that were driving upcalls out of I/O sources. In fact, in one situation, the thread pools were emptied, and a deadlock occurred because a poorly written upcall handler was blocking one thread while waiting for a completion from another. Another side effect of this spaghetti-like control flow graph was an increase in the probability that an individual thread's stack would overflow.

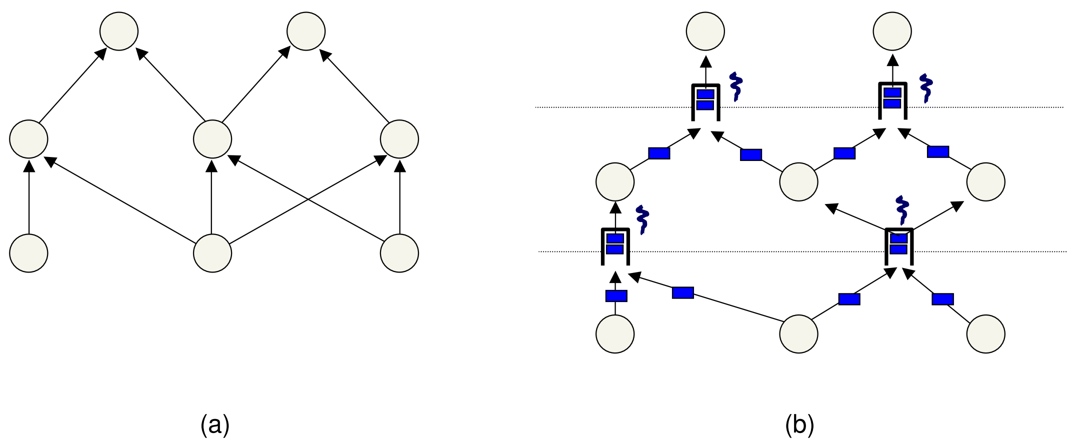


Figure 35: **Structured upcall-based control flow:** (a) The same composition graph as in the previous two figures. (b) Shows a structured upcall-based graph, in which queues are used to impose thread boundaries between layers, thereby disentangling control flow, but also eliminating the need for mutexes.

5.2.3 Structured Upcalls

Our final control-flow model for the I/O core retains the upcall-based control flow of the previous model, but also imposes additional structure on the system that eliminates spaghetti-like control flows. As shown in Figure 35, this model uses queues to introduce thread boundaries between layers of the system. Upcalls between elements in the graph are placed in downstream queues, meaning that the thread context issuing the upcall returns immediately. Each layer in the system (more specifically, each thread boundary's queue) has its own thread context associated with it that permanently blocks on its queue, draining events and pushing upcalls downstream as needed.

Because the thread boundaries prevent thread contexts from propagating across layers, cycles in threads' call graphs are completely eliminated. This structuring also prevents threads' call chains from growing long; instead, a thread receives an event from its queue, and processes it by passing it into event handling code. (Note that multiple event handlers may share a queue, as shown in Figure 35b.) If that event handling code wants to generate an additional upcall, it is placed in a downstream queue, and the thread immediately returns to the upstream caller, instead of being diverted to the downstream upcall handler.

There is an additional benefit to this structured upcall model; because a single thread pulls events from a thread boundary queue, as long as each event handler is fed by a single thread boundary then there can only be a single thread context entering a given event handler at a time. This completely eliminates the need for mutexes or locks, also eliminating the possibility of deadlock or race conditions due to thread synchronization.

The structured upcall model can also experience better processor instruction cache performance than the other models. If the thread pulling events from the thread boundary handles many events before a context switch occurs, then the instructions executed by that thread are likely to be loaded into the instruction cache when the first event is handled. The thread then will execute with a warm instruction cache for subsequent events.

A disadvantage of the structured upcall model is that an event flowing through the graph experiences several thread context switches throughout its lifetime. These context switches can add to the latency of event delivery through the system, as can the time the event spends behind other events in a thread boundary queue. This model thus trades latency for both throughput and simplicity.

Overall, the structured upcall model solves nearly all of the problems encountered by the poll-based model and the unstructured upcall model. Race conditions are eliminated, the upcall-based nature of the model simplifies demultiplexing, the introduction of thread boundary eliminates spaghetti-like thread call chains, and the overall system is efficient, performing work only when there is useful work to be done (rather than wasting cycles on speculative polling).

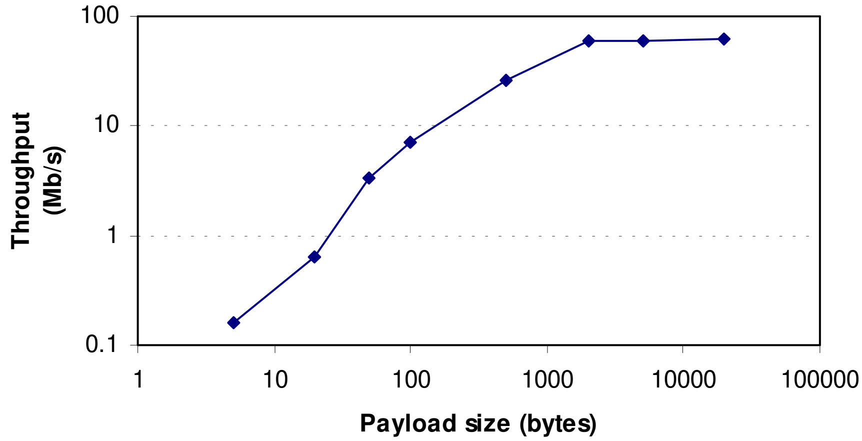


Figure 36: **Network throughput:** This benchmark shows the measured throughput of the client-server pipeline as a function of packet size. Both request and reply packets were taken into account to calculate the total throughput. The Ethernet saturated (reaching 85 Mb/s) at a 2000 byte packet size.

5.3 I/O Core Performance

In this section, we present the results of several microbenchmarks that explore the performance of both the network and disk channels in the I/O core. All benchmarks are performed on 500MHz dual Pentium SMPs with 256 MB of RAM and 100 Mb/s Ethernet cards, running Linux 2.2.13 and the IBM JDK v1.1.8 (which uses kernel threads). The network tests use the threadpool-based TCP network extension in the I/O core device-specific layer.

5.3.1 Network Performance

Our network performance benchmark consists of a client and server process, each running on a different node on the 100 Mb/s Ethernet. The client process opens a network sink to the server, and drops packets of a configured size into the sink. For each packet that the server receives, it sends a reply packet of the same size back to the client. The

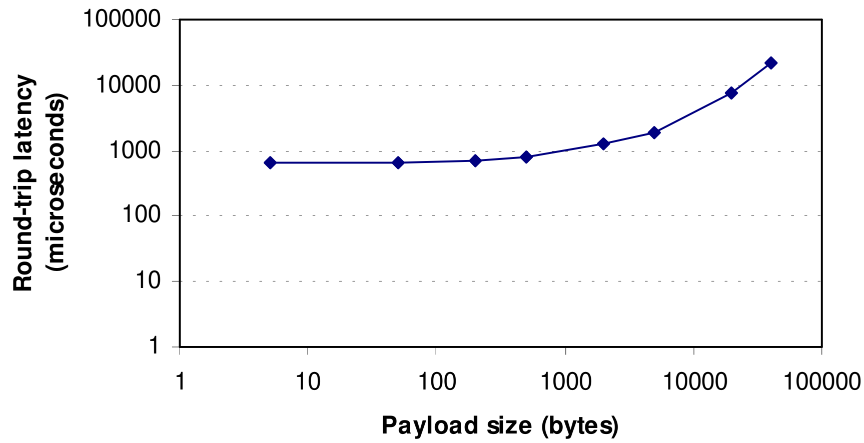


Figure 37: **Network latency:** This benchmark shows the roundtrip latency of a message as a function of its size. Latency increases with message size, from a minimum of 650 μs.

client sends a new packet for every reply that it receives, thereby establishing a closed-loop pipeline between the itself and the server. We gathered two separate benchmarks: the first measured network latency, and had only a single packet in the client-server pipeline. The second measured network throughput, and filled the pipeline with enough packets to reach maximum measured throughput. Both benchmarks were repeated several times with a number of different packet sizes.

In Figure 36, the network throughput benchmark results are shown. This graph shows that network throughput increases linearly with payload size until the 100 Mb/s Ethernet saturates, reaching a maximum 85 Mb/s for packets that are 2000 bytes or larger. For packet sizes less than 2000 bytes, the overhead of interaction with the TCP stack and the JDK's TCP class libraries is the throughput bottleneck of the system, while for packet sizes greater than 2000 bytes, the Ethernet itself saturates and becomes the bottleneck.

In Figure 37, we show the roundtrip latency of the system as a function of message size. As the graph shows, the minimum latency of the system is 650 microseconds for messages that have less than 50 bytes of payload. Latency gradually increases with message

Figure 38: **Disk latency, random reads, cache miss:** This graph shows the latency of reading a random disk block through the I/O core during a file system cache miss, thus incurring a disk seek. This latency number is graphed as a function of the number of concurrent read requests issued to the disk source.

size; for messages less than 2000 bytes, the overhead of interacting with the TCP stack and copying bytes from the network card dominates the end-to-end latency. Beyond messages of 2000 bytes, latency increases linearly with message size; at this point, the throughput of the network has become the bottleneck in the system.

5.3.2 Disk Performance

To measure disk performance, we wrote a benchmark client that opens up a disk source from a multi-gigabyte file, and issues random read requests to that source. We measured three different characteristics: the latency to perform a read from a disk block that is not in the file system cache (thus incurring a disk seek and block transfer), the latency to perform a read from a disk block that is in the file system cache (thus measuring the latency overhead of the I/O core), and the throughput of the I/O core for both sequential and random reads and writes.

In Figure 38, we show the latency of reading a random 4 KB disk block through

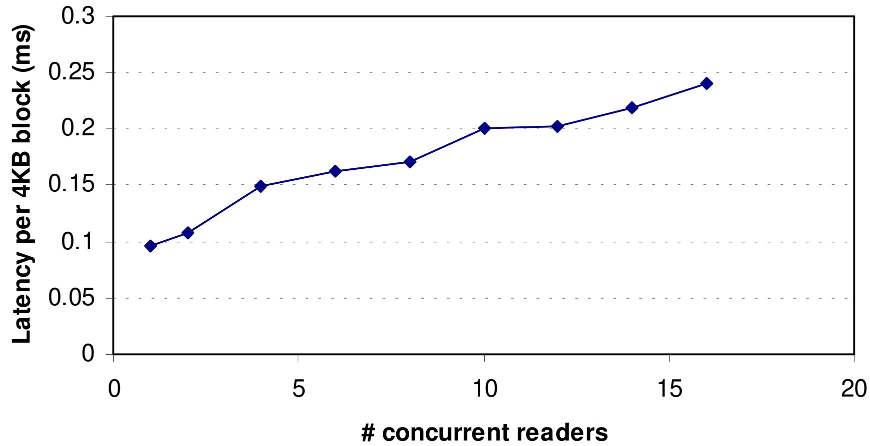


Figure 39: **Disk latency, random reads, cache hit:** This graph shows the latency of reading a random disk block from the file system cache as a function of the number of concurrent readers.

the I/O core when incurring a file system cache miss and disk seek; this latency is plotted as a function of the number of concurrent read requests issued to the underlying I/O core disk source. For a single read request, the latency is 10ms, which roughly corresponds to the seek time of the disk. As the number of concurrent read requests grows, this latency grows sublinearly; this sublinear growth is due to the greater efficiency obtained by the elevator scheduling algorithm in the disk device driver as the number of concurrent requests grows.

Figure 39 shows the latency of random 4 KB reads under file system cache hits, as a function of the number of concurrent readers. For a single reader, the measured latency is 96 microseconds, resulting in a bandwidth of 41 MB/s. Latency increases linearly with the number of concurrent readers; for 15 readers, the measured latency is 237 microseconds, corresponding to a read bandwidth of 262 MB/s. At this point, the bottleneck of the system is the copying of data from the file system buffer cache into the JVM, and garbage collection within the JVM.

We also measured the throughput of the I/O core disk sources and sinks. To measure throughput, we opened a closed-loop pipeline of 4 KB read or write requests, and increased the number of requests in the pipeline until throughput saturated. All throughput benchmarks were done against a large enough file so that all requests missed in the file system cache. For sequential disk reads, we achieved 11 MB/s throughput; sequential disk writes saturated at 8 MB/s. We measured the maximum throughput of the disk at 12 MB/s using a tuned C program. Disk writes obtained less throughput than disk reads because of the design decision (described in Section 5.1.1) to issue only a single write at a time from a disk sink, but to allow multiple simultaneous read requests to be issued from a disk source.

For random disk read and writes, the throughput of the I/O core saturated at 2.1 MB/s and 1.8 MB/s, respectively. The high latency of performing disk seeks was the bottleneck in this benchmark, resulting in much lower throughput than in the sequential read and write case.

5.4 Experience

We have gathered over a year of experience using the I/O core as a foundation for many different cluster-based and wide-area infrastructure services (some of which we will mention in the following section). This experience has lent us considerable insight into the efficacy of its design and its interfaces.

An interesting surprise that emerged from the I/O core is that its message-passing oriented, asynchronous interfaces have the convenient property that local, cross-JVM, and cross-machine event delivery all have exactly the same interface (`UpcallHandlerIF` or `SinkIF`) and the same semantics. The asynchronous interface imparts the correct expectations on programmers, namely that the operation behind the interface may have some finite latency associated with it, and that operation may fail or time out, returning an error condition instead of a successful completion. Because the interface and semantics are the same in all three cases, it is possible to change the routing of events between these cases at run-time. Such a change would have obvious performance implications, but also subtle availability implications, since in local event delivery, both the source and destination are within a single failure boundary, while for cross-JVM or cross-machine delivery, they span failure boundaries.

The layering of the I/O core was extremely successful, in that we were able to implement three network device layers that use very different mechanisms, while retaining the higher-level interface abstractions. Similarly, the structured upcall control flow model was very successful for building robust, layered systems on top of the I/O core. One aspect of the Java programming language that made this layering very successful was its garbage collection. Because of the garbage collector, low level layers in the system could allocate memory regions, embed references to them in events, and pass the events upwards through the application stack. After passing an event upwards, a layer can immediately forget about it, relying on the garbage collector to eventually deallocate the embedded memory region. If the system didn't have a garbage collector, then either reference counting on events would need to be used to determine when to free the embedded memory region, or the layers would need to coordinate with each other to send the event back down to the allocator for it to free it, which would greatly complicate layering and compromise modularity.

Another benefit of the Java language was its type system, particularly the existence of strongly typed interfaces and inheritance. The strongly typed interfaces made it very easy to declare the high-level abstract interfaces to the I/O core (such as `SinkIF`), and to maintain those interfaces in lower-level device-specific layers. The strong typing and inheritance also imparted elegance and an economy of mechanism to the event hierarchy defined by the I/O core.

The decision to wrap all memory regions with an abstract memory interface `MemRegionIF` ended up being poor. The original motivation for this was to permit low-level device-specific implementations (such as the Via network layer) to perform their own memory management, such as to allow pinned physical memory regions. However, the decision to use pinned memory regions effectively prevents the system from using garbage collection, since the device-specific memory manager is responsible for unpinning memory regions and recycling them when they are no longer needed. Losing the ability to do garbage collection complicates the layer of the system, as previously discussed. Because of this, we ended up copying data from the Via network layer's memory regions into a generic Java memory region to allow garbage collection to occur. In the end, the `MemRegionIF` thus became a "latency engine", adding extra access costs without providing truly useful functionality.

Another design decision was to use a single interface (`UpcallHandlerIF`) to abstract away all event handlers in the system. This design decision was exceptionally successful at allowing interposition and dynamic extension of the system, since interposed or extended layers have the same interface as the original layers of the system. However, the cost of this decision was that much type information is lost through this interface; `UpcallHandlerIF` declares a single method, which is the delivery of an abstract event class. To recover type information, introspection needed to be done to determine the specific class of the event delivered through this interface. Fortunately, this kind of introspection is extremely cheap in Java. The resulting code structure is familiar to event-driven system programmers, specifically an event loop with a "switch" statement that determines the appropriate action for a given event type.

Another implication of the narrow `UpcallHandlerIF` event delivery interface is that it is impossible for the caller of this interface to determine whether the event handler will enqueue the event and immediately return, or "borrow" the caller's thread context to do other work. Thus, if an untrusted piece of code is called through this interface, it is possible for that code to steal threads from the callee, effectively performing a denial of service attack. Fortunately, the design patterns show us the way to solving this problem. If a programmer wishes to use an untrusted piece of code through such an interface, the programmer can simply apply the "wrap" design pattern to dedicate a queue and a thread pool to the untrusted code. Because queues implement the `UpcallHandlerIF` interface, the wrap operation preserves the interface of the wrapped code, but conditions it by dedicating a pool of thread contexts to the code. Thus, the caller's code doesn't need to change at all; wrapping is a completely transparent operation from the perspective of program code, specifically because of the decision to use the narrow `UpcallHandlerIF` for all event handlers.

5.4.1 Impact

The I/O core has had significant impact both within Berkeley and at other institutions. Within Berkeley, the I/O core is the foundation for the current version of the Ninja project [71]. More specifically, the I/O core is being used as the programming model and I/O substrate for the vSpace [27] service platform and the distributed data structure (DDS) [69] storage platform. In addition, other projects within Berkeley have adopted the I/O core as their foundation, including the Telegraph segment-based storage manager, [74], some components of the OceanStore [18] wide-area data storage utility, and a highly-concurrent single-node Btree implementation. Outside of Berkeley, the I/O core API's have influenced the design process of Sun's upcoming Java high-performance I/O extensions, and they have influenced the design of the `one.world` pervasive computing architecture project [72] at the University of Washington.

Part III

Distributed Data Structures

Chapter 6

A Storage Management Layer for Internet Services

It is challenging for a service to achieve all of the service properties outlined in Section 1.2, 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.

As previously mentioned, many projects and products propose using software platforms on clusters to address these challenges and to simplify Internet service construction [2, 4, 15, 55]. For example, the TACC platform [55] masks node failures by automatically restarting failed software components on new nodes, and it routes requests to the least-loaded of equivalent software components in order to perform load-balancing. However, TACC assumes that all shared state in the system can be reconstructed if lost and that it does not need to be kept persistent or consistent. Similarly, the Rivers I/O processing platform [10] implicitly load balances across I/O multiplexing operators by allowing them to "pull" data towards themselves at their own speed, but Rivers does not deal with data persistence or fault tolerance. Fundamentally, none of these cluster toolkits provide support or abstractions for fault-tolerant, distributed, scalable data storage, and as a result, none of them support the family of applications that require explicit support for persistent data.

Platforms that do support persistent state management 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 part of the thesis, 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.

In this chapter, we present an overview of distributed data structures, describing their design relative to that of relational databases and file systems, and describing the assumptions that we made about their operating environment and the types of failures that

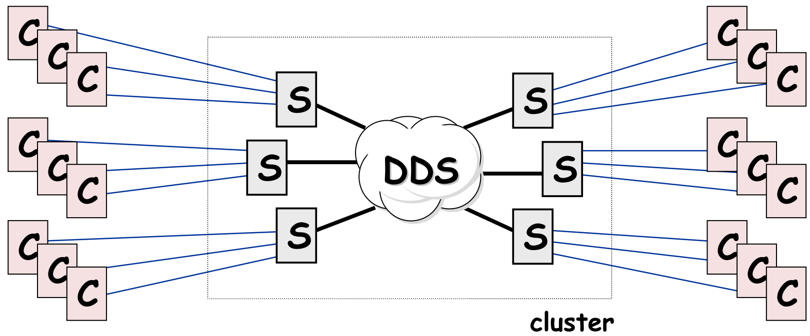


Figure 40: **High-level view of a DDS:** A DDS is a self-managing, cluster-based data repository. All service instances (S) in the cluster see the same consistent image of the DDS; as a result, any WAN client (C) can communicate with any service instance.

we expect and can handle. We also describe an early prototype of a distributed hash table; although this prototype failed to provide most of the service properties, through its failure we gained valuable insight into the design space of DDS's.

In the following chapter, we describe the design, architecture, and implementation of a much more successful distributed hash table, built in Java. We used this second implementation of a distributed hash table as a mechanism for exploring the programming framework and I/O core described in Part II. From this perspective, we evaluate its performance, scalability and availability, its ability to simplify service construction, and the effectiveness of the programming framework for this large system. We also present operational experience that we gained from running the distributed hash table over several months.

6.1 DDS Overview

A distributed data structure (DDS) is a self-managing storage layer designed to run on a cluster of workstations [4] and to handle Internet service workloads. A DDS has all of the previously mentioned service properties: high throughput, high concurrency, availability, incremental 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. All of the difficult issues of managing persistent state are transparently handled by the DDS platform.

Figure 40 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. This assumes that session state is “soft-state” and can be recovered by a new service instance if the old service instance handling a particular client fails.

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 services. We believe that an adequate set of structures consists of a distributed, append-only log (that provides guarantees about the ordering of writes), a distributed hash table, and some form of distributed tree. Logging is a common operation in many services, and includes both informational logging, and the style of logging done to provide ACID semantics in transactional systems. A hash table is a generic, fast storage primitive that has proven itself to be invaluable in many existing services. Finally, a tree provides an ordered index into data, and facilitates range queries. For this dissertation, we have focused solely on the implementation of a distributed hash table.

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 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.

6.1.1 Relational database management systems (RDBMS)

An RDBMS offers extremely strong durability and consistency guarantees, namely ACID properties derived from the use of transactions [66], 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 [55, 81, 119]. 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 construct and dynamically 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.

6.1.2 Distributed file systems

Most Internet web caches [26, 53] and web servers [109] use file systems in order to store their persistent data. They either use local file systems, in which case scaling of storage capacity must be handled within the service logic, or they rely on distributed file systems.

Distributed file systems have less strictly defined consistency models than an RDBMS. Some file systems (e.g., NFS [113]) have weak consistency guarantees, while others (such as Frangipani [126] or AFS [48]) 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 [6] 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. As a result of this, the intent of an operation issued by a client is hidden from the file system; it is impossible to tell if a write to bytes 100-250 of file `/X/Y` corresponds to a modification of part of a logical application record, an entire logical record, multiple records, or something else altogether. This limits the ability of a file system to optimize based upon such knowledge, resulting in performance anomalies from effects such as false sharing.

6.1.3 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.

Rather than attempting to provide a general abstraction that is useful to all conceivable services, we have deliberately focused on providing a narrow interface with carefully chosen properties. 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.

6.2 DDS Design Principles

In this section, we present a number of general design principles that guided us while designing and building a hash table distributed data structure.

6.2.1 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. Crashes become incidental: only session state needs to be regenerated in the case of a crash. This greatly simplifies service construction, as authors need only worry about service-specific logic, and not the complexities of data partitioning, replication, and recovery.

6.2.2 Appeal to properties of clusters

In addition to the properties listed in section 1.3, 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. 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.

6.2.3 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 the asynchronous, event-driven style of control flow in our DDS that is outlined in Part II of this thesis. This style is similar to that espoused by modern high performance servers [13, 76] such as the Harvest web cache [26] and Flash web server [105]. As we will show, 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 the DDS experiences a burst of traffic, the burst is absorbed in event queues, providing *graceful degradation* of the DDS by preserving its throughput but temporarily increasing latency.

6.3 An Early, Failed Prototype: the `mmap()`-based Distributed Hash Table

As an early exploration into distributed data structures, we implemented a rudimentary prototype of a distributed hash table DDS that supports multiple node failures, but does not provide any atomicity, consistency, or on-line recovery guarantees, nor is it extensible (in terms of being able to add more nodes to the SDDS while it is running). This prototype was built using the "C" programming language. Architecturally, wide area clients communicate with service-specific front ends, such as web servers. These front ends export a service-specific interface such as HTTP, and in the process of handling a client request, make use of the distributed hash table for persistent, available storage. Interactions with the distributed hash table are done through an abstraction library that is linked into each front end (or other such hash table clients); it is assumed that hash table clients are on the same SAN as the hash table storage nodes.

The abstraction library exports a "C" language interface (as shown in Figure 41), and also provides Java glue to a subclass of the standard `java.util.hashtable` class. The library converts each hash table operation into an RPC-like message that is sent to appropriate nodes in the hash table; currently, tables are partitioned across nodes according to a simple modulus-based hash. Each storage node in the cluster is an independent "brick" that contains an isolated, single-node persistent hash table implementation. Storage nodes are not aware of their peers; it is the abstraction libraries that contain the logic for partitioning requests across the bricks, and for forming replication groups for partitions. This partitioning logic is configurable at service launch time. Services can manipulate a table that binds replica group numbers to one more physical storage bricks; because of this, each service has control over the degree of replication for all of that service's hash tables.

In addition to the standard hash table insert, delete, and lookup routines, we included the ability to create and destroy hash tables, as well as to enumerate through all of the entries in a particular hash table asynchronously. These additional operations were necessary in order to support features required by the `java.util.hashtable` Java interface for which we provided glue, but they significantly complicated the implementation of both

```
int MR_createTable(char *table_name, UINT32 num_buckets);
int MR_destroyTable(char *table_name);
int MR_openTable(char *table_name)
int MR_closeTable(char *table_name);
int MR_lookup_value(char *table_name, UINT64 lookup_hashval,
                    mr_hash_rec *returned_record);
int MR_insert_value(char *table_name, mr_hash_rec record);
int MR_delete_value(char *table_name, UINT64 delete_hashval);
int MR_num_elements(char *table_name, UINT64 *num_elements);
int MR_num_lookup_next(char *table_name,
                       int  *logical_node_num,
                       int  *element_num, int  *chain_num,
                       mr_hash_rec *returned_record);
```

Figure 41: **Prototype hash table "C" language API:** All functions return zero on success, and non-zero values in case of error.

the abstraction library and the storage bricks. Conspicuously absent from the interface is the ability to lock elements in the hash table or perform atomic transactions: indeed, the prototype provided no atomicity or consistency guarantees to callers. Two or more simultaneous state-changing operations on the same element would produce undefined and perhaps disastrous results.

6.3.1 Storage "Bricks"

As previously mentioned, each storage node in the cluster maintains an independent, self-contained storage "brick". These bricks contain single-node hash tables that can be accessed via an RPC-like interface, similar in nature to networked file systems such as Sun's NFS [113]. Significant design and engineering effort went into the implementation of these bricks, since we wanted to be able to run bricks under different operating systems, thread models, and network primitives and communications media. Figure 42 illustrates the architecture of one of these bricks.

Persistent storage is managed by using the Unix `mmap` system call to map a large

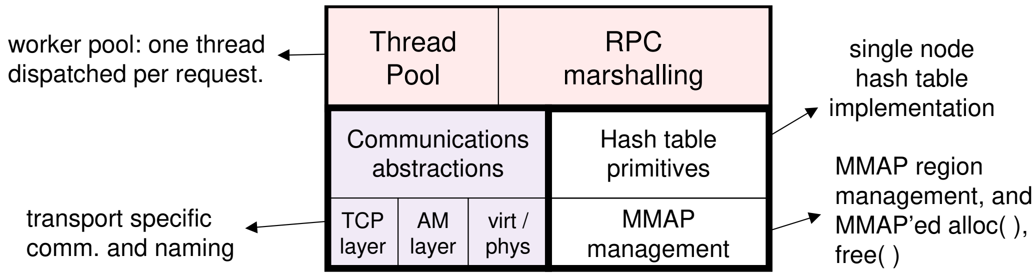


Figure 42: **Distributed hash table prototype "storage brick"**: A brick contains a single-node hash table and RPC-like stubs so that it can be remotely accessed.

file into the virtual address space of the brick. All modifications to persistent state are thus done by manipulating virtual memory structures; this significantly simplified the construction of the brick, but had the implication that write ordering is now controlled by the virtual memory subsystem instead of the brick. Also, page faults or dirty page writes cause the entire brick to seize, since the brick process is swapped out until the virtual memory operation completes. Finally, because memory mapped regions may not always be loaded at the same absolute virtual address, the mapped memory could not contain any pointers. This implied that we had to implement our own offset based `malloc` and `free` memory allocation routines as well an offset based in-memory chained hash table access method.

To support network communications, we implemented a simple communications abstraction layer that facilitated multiple underlying network transports. We currently only have a TCP/IP based brick, but this abstraction layer will allow us to easily port the brick to more sophisticated cluster-oriented transports such as active messages [128] or U-Net [127].

Using the underlying memory-based chained hash table and the abstract communications layer, we implemented an RPC skeleton layer that accepts incoming network connections, unmarshalls hash table operations, executes those operations, and returns marshalled results. This RPC skeleton layer works in conjuction with a worker thread pool;

as an incoming network connection or message is detected, a thread is dispatched to the skeleton layer to process the encapsulated request. After handling the request, the thread rejoins the pool and sleeps until it is redispatched for some subsequent request.

6.3.2 Lessons from the Prototype

We learned many lessons during the implementation and operation of this distributed hash table prototype:

Service simplicity: our hypothesis about SDDS's simplifying service construction was backed; we were able to implement an interesting "related site" service[1] that inherited the properties of the prototype hash table. The service code is devoid of any code related to data persistence and availability—that complexity is successfully hidden inside the hash table implementation. Unfortunately, because the hash table didn't maintain consistency or have a recovery path, neither did the service.

Client-side abstraction libraries: embedding the hash table partitioning, replication, and fail-over logic in the client-side abstraction libraries simplified the implementation of the hash table, since we could make each storage brick completely independent. However, this had negative implications for administration and monitoring: there was no place in the system that had a complete view of the activity of the hash table. Furthermore, each client must make isolated decisions: they cannot share load balancing information or knowledge about discovered failures in the cluster. From this, we believe that the correct design point is to have partitioning and replication information durably stored by the bricks, and to share this information with the client-side abstraction libraries on demand. The libraries would thus learn the current state of the hash table topology, and they would be able to independently route requests to bricks based on this.

[1] Described at `http://ninja.cs.berkeley.edu/demos/parallelisms/what_is_it.html`.

Incremental scaling: a second effect of our poor decision to embed partition and replication metadata in the client-side abstraction library was that in order to change the configuration of the hash table (e.g., by adding more nodes to increase capacity), the metadata kept by all client-side libraries needed to be updated synchronously and atomically. Again, we believe the correct solution to this is to rely on the bricks to retain authoritative knowledge of the cluster topology, but to have the client-side libraries lazily discover topology changes and adapt based on that.

Recovery: there was no on-line recovery strategy built into the prototype hash table. If a node failed, then the number of replicas in the hash table would dwindle. The only recourse available was to shut down the hash table temporarily, copy data from a surviving brick to the failed brick, and then turn the hash table back on. This resulted in unacceptable periods of unavailability.

Consistency: the fact that the prototype hash table made no guarantees about the consistency of data in the face of simultaneous state changing operations was disastrous. It effectively limited the usefulness of the hash table to read-only services that were bulk-loaded with a single write-intensive process. This lead us to believe that the next implementation of the hash table must promise atomicity of reads and writes, allowing the possibility of simultaneous writes to the same key.

`mmap`: the `mmap` system call proved to have severe limitations. Writes from virtual memory could not be ordered by the hash table implementation, preventing it from maintaining any consistency. Even worse, writes are done on page boundaries rather than hash table element boundaries. Furthermore, if a brick crashes, the operating system would spend tens of seconds flushing dirty data back to disk, increasing the amount of time it takes to recover a brick, and potentially overwriting the on-disk representation with corrupted data. Finally, page faults would cause the entire process to hang.

Language issues: when we went to implement Java glue that exposed the "C" distributed hash table implementation to Java, we discovered that the fact that we included a user-level thread package in the "C" client-side abstraction library caused great difficulties. The Java run-time environment has its own notion of threading, and correspondingly, it's own implementation of a user-level thread package. These two thread packages, as it turned out, could not coexist: we had to reauthor significant portions of the client-side abstraction library so that it did not depend on a thread subsystem at all, which unavoidably introduced complexities and inefficiencies. Because of this, we decided that the next implementation of the distributed hash table should be implemented purely in Java, even though we anticipated the performance implications of this decision.

Chapter 7

A Robust Distributed Hash Table Implementation

In this chapter, we present the design, architecture, and implementation of our second version of a distributed hash table DDS. This second implementation was designed to be much more robust than our original prototype, and it was also designed to enforce consistency, to provide an online recovery path, and to allow incremental scaling and reconfiguration of the hash table across the cluster.

7.1 Assumptions

To simplify the hash table design and to specify precisely the operational attributes that we expect from it, we made number of key assumptions we made regarding our cluster environment. In particular, we examine the failure modes that the DDS can handle and the workloads that we assume that it will receive.

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 justification 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 recover gracefully 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 current working set of popularly accessed 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).[1] If there are many such hotspots, however, then our partitioning strategy will balance them across the cluster. Failure of these workload assumptions can result in load imbalances across the cluster, leading to a reduction in

[1] We believe that the use of caching plus invalidation of write locks (similar to that used in [48]) could help to solve this; we have not yet explored this possibility.

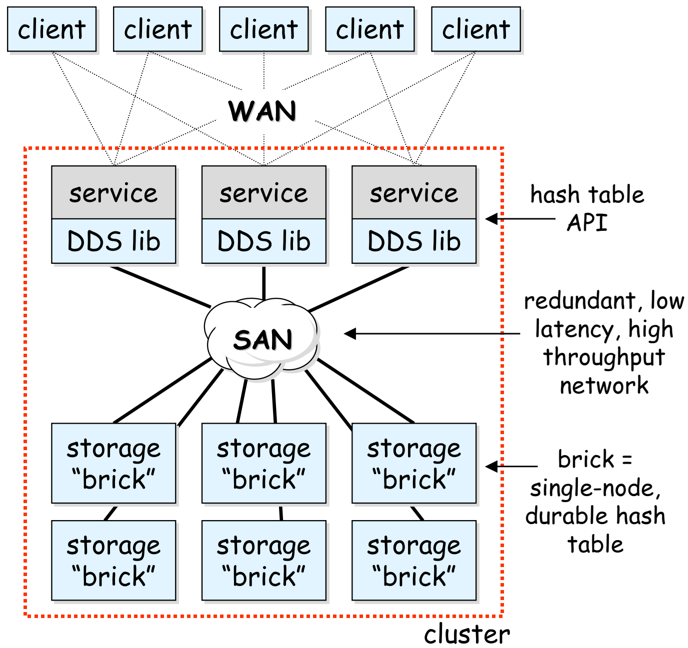


Figure 43: **Distributed hash table architecture:** Each box in the diagram represents a software process. In the simplest case, each process runs on its own physical machine, however there is nothing preventing processes from sharing machines.

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.

7.2 Architecture

Figure 43 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 [22], a service-specific protocol, or

```
public int create_dis_hashtable(int num_bucks, UpcallHandlerIF compQ);
public int destroy_dis_hashtable(UpcallHandlerIF compQ);
public int get(long key, UpcallHandlerIF compQ);
public int put(long key, byte[] bytes, UpcallHandlerIF compQ);
public int remove(long key, UpcallHandlerIF compQ);
```

Figure 44: **Hash table Java language API:** All methods return an integer, which is a unique ID that will be passed in as a field of the completion event. This completion event is delivered to the compQ `UpcallHandlerIF` specified as the final argument to all methods. The `put()` and `remove()` methods return the old value in addition to updating the current value in the table.

level 4 or level 7 load-balancing switches (such as Cisco's Local Director [124]) 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 (Figure 44) 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 existing 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.

7.2.1 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, 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 mechanism (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; 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.

7.2.2 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 45, 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

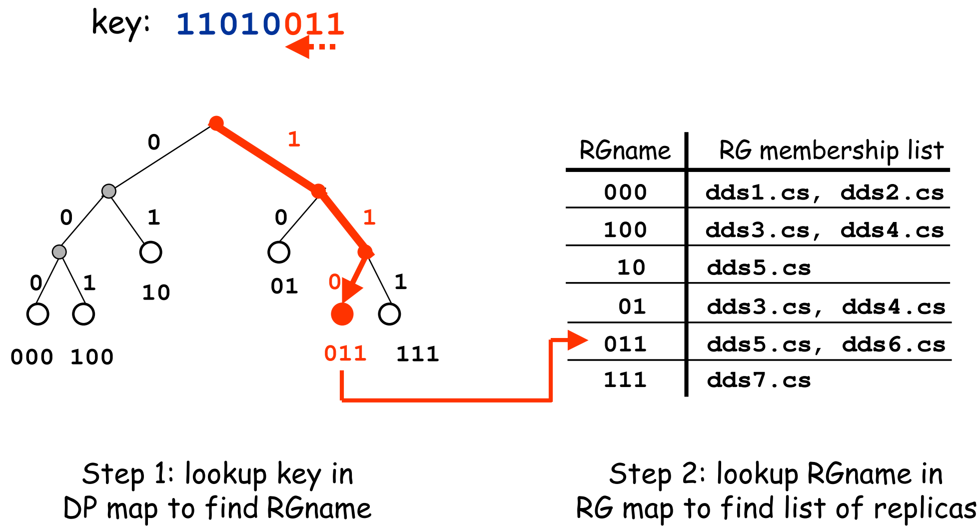


Figure 45: **Distributed hash table metadata maps:** This illustration highlights the steps taken to discover the set of replica groups which serve as the backing store for a specific hash table key. The key is used to traverse the DP map trie and retrieve the name of the key's replica group. The replica group name is then used looked up in the RG map to find the group's current membership.

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 45 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 45 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 select any of the replicas listed in the RG map to service the read. We use the locality-aware request distribution (LARD) technique [50] to select a read replica—LARD further partitions keys across replicas, in effect aggregating their physical caches.

7.2.3 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

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

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.

7.2.4 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. The recovering node could detect this, and either proactively renew its write lease or abort recovery, 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.

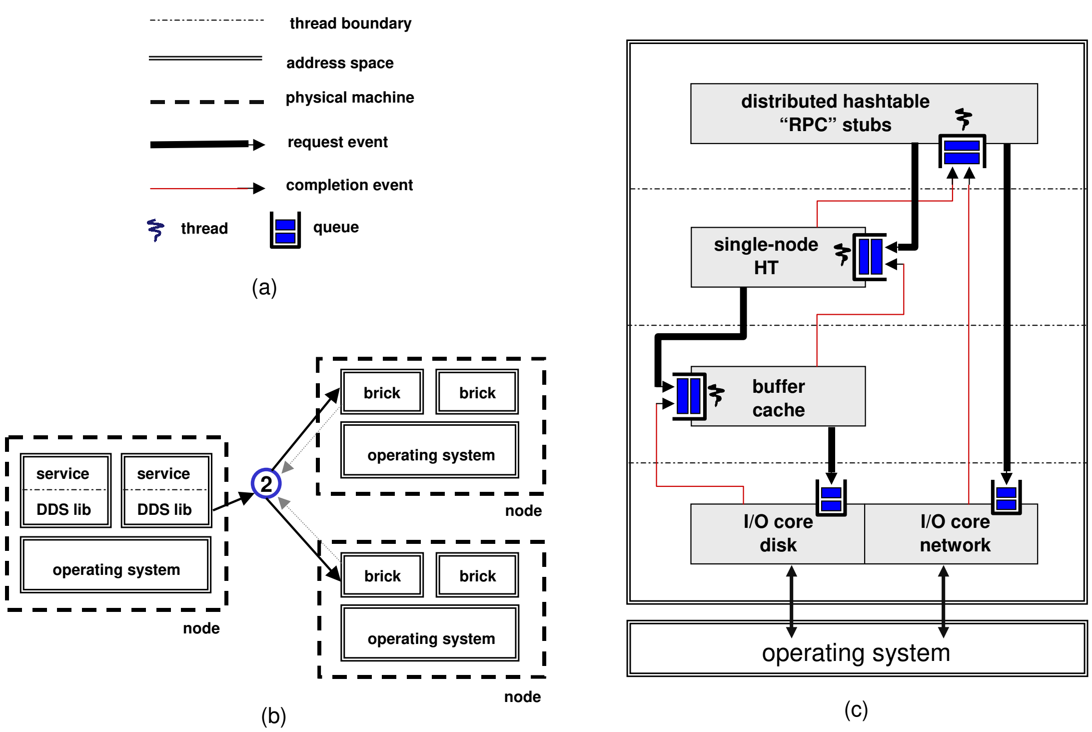


Figure 46: **Hash table structure:** This figure shows the architecture of the distributed hash table in terms of the programming model described in Part II of this thesis. (a) shows a key of icons, (b) illustrates the mapping of processes to machines across the cluster and a two-phase commit composition operator joining a library to two bricks, and (c) shows the structure of a "brick" process in terms of queues, thread pools, thread boundaries, and events.

7.2.5 Programming Model

All components of the distributed hash table are built using the asynchronous, event-driven programming model described in Part II of this dissertation. Each hash table layer has its own thread boundary (as shown in Figure 46), and thus it 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. The thread boundaries between the 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 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.

7.3 Hash Table Performance

In the following sections, 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 across 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.

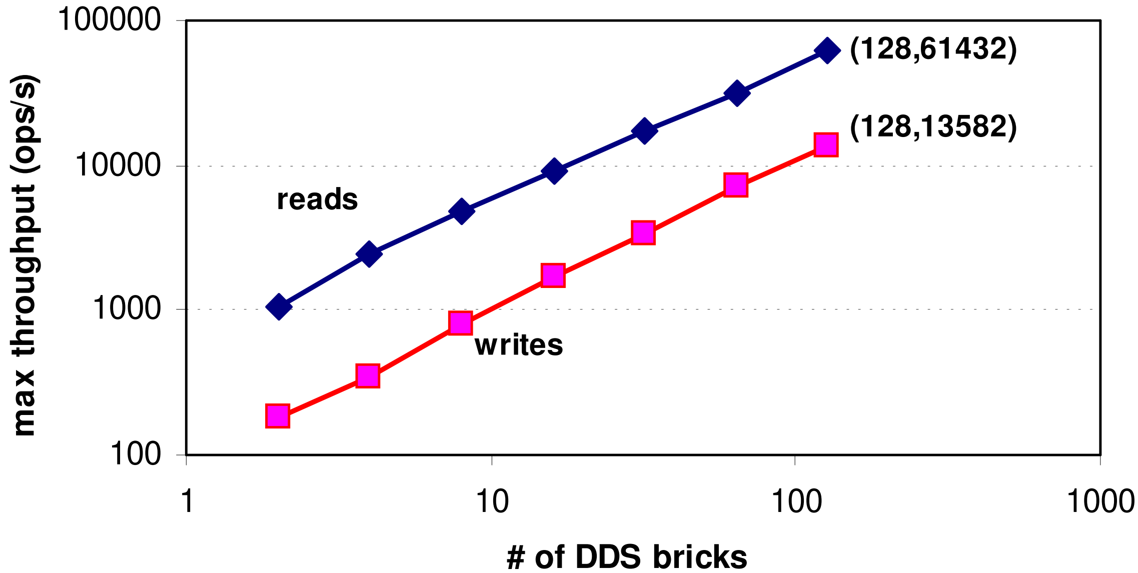


Figure 47: **Throughput scalability:** This benchmark shows the linear scaling of throughput as a function of the number of bricks serving in a distributed hash table; note that both axis have logarithmic scales. As we added more bricks to the DDS, we increased the number of clients using the DDS until throughput saturated.

7.3.1 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.

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 47 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.

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 48 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 reach a maximum throughput as its bricks eventually 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 because of the longer event queue lengths.

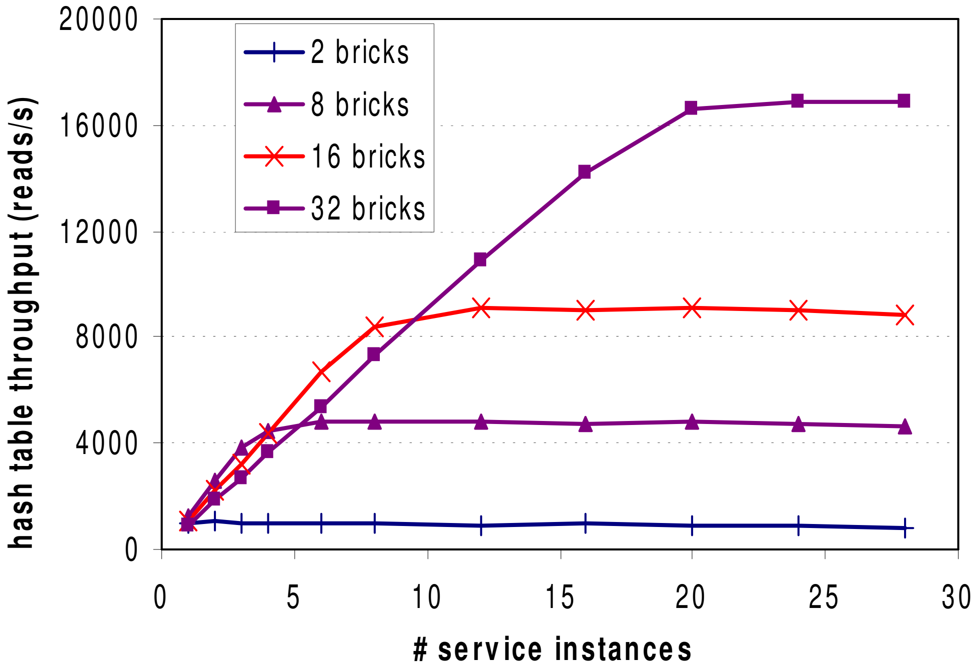


Figure 48: **Graceful degradation of reads:** This graph demonstrates that the read throughput from a distributed hash table remains constant even if the offered load exceeds the capacity of the hash table.

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 49 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

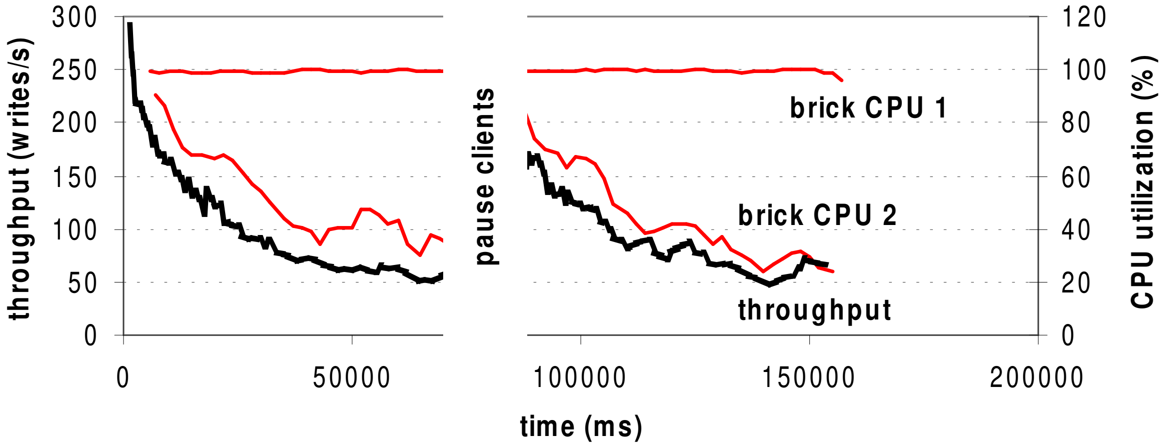


Figure 49: **Write imbalance leading to ungraceful degradation:** The bottom curve shows the throughput of a two-brick partition under overload, and the top two curves show the CPU utilization of those bricks. One brick is saturated, the other becomes only 30% busy.

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.

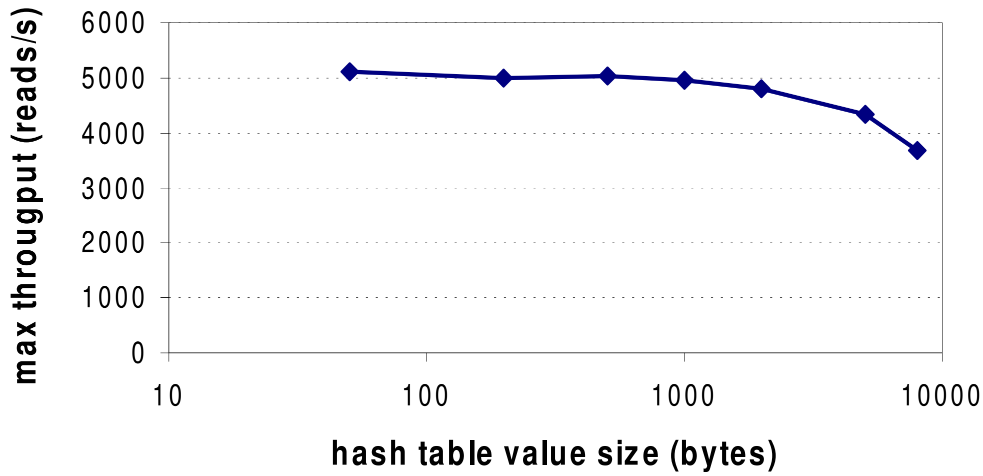


Figure 50: **Throughput vs. read size** The X axis shows the size of values read from the hash table, and the Y axis shows the maximum throughput sustained by an 8 brick hash table serving these values.

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.

Throughput Bottlenecks

In Figure 50, we show the results of varying 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.

7.3.2 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.

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 7.3.1 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.

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

[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.

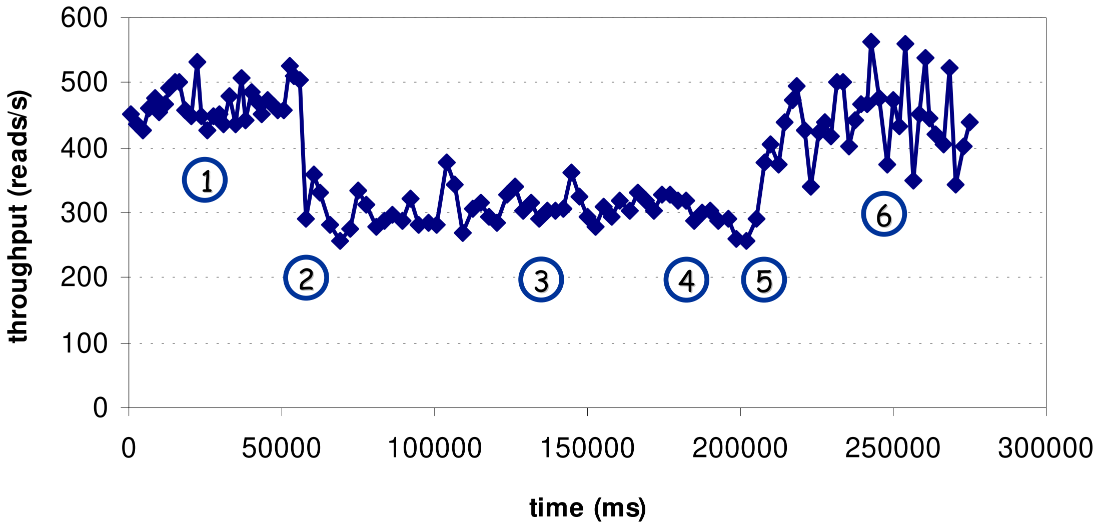


Figure 51: **Availability and Recovery:** This benchmark shows the read throughput of a 3-brick hash table as a deliberate single-node fault is induced, and afterwards as recovery is performed.

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 improve the write performance of traffic with little locality by using disk layout techniques similar to those of log-structured file systems [111]; we have not explored this possibility as of yet.

7.4 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 51 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 recovering 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.

Chapter 8

Experience and Applications

In this chapter, we discuss experiences that we gained by administrating a distributed hash table over a long period of time, and by building several interesting services using the hash table.

8.1 Operational Experience: Violations of Assumptions

The performance metrics presented in Section 7.3 validate that, for the workloads and operational environment of our benchmarks, our hash table implementation has all of the service properties enumerated in Table 1.1. However, benchmarks are almost always a poor reflection of the operational environment that a deployed system will experience; by their very nature, benchmarks are carefully controlled so that they may be reproduced and independently verified by multiple parties. This careful control is contrary to the requirement that our DDS must be robust and available in the face of unexpected circumstances.

In this Section, we present more realistic operational experiences that we gathered from managing a distributed hash table DDS over a period of six months. This DDS was run on a dedicated 8-CPU cluster in the UC Berkeley CS department. In addition to our own services, which we will describe in Section 8.3, many external research projects built systems that depended on this hash table.

Some of the external projects that relied upon our DDS during this six-month experiment include:

vSpace: The vSpace Internet service platform provides fault-tolerance and load balancing abstractions for computational elements of Internet services. (Comparatively, the DDS provides similar abstractions for persistent state management aspects of services.) One component of vSpace makes use of the DDS to store descriptions of services, their operational requirements, and service instantiation metadata. The DDS provides a cluster-wide mechanism for vSpace nodes to access all of this data, and as such it greatly simplified the consistent management of service state. The workload generated by vSpace is read-mostly; writes occur only when services are instantiated or halted.

The ICEBERG Preference Registry: The ICEBERG project [131] seeks to integrate telephony and data services by designing and implementing an IP-centric signalling and service architecture. The ICEBERG user preference registry exploits the hash table DDS to store user profiles that dictate the routing of incoming telephone calls to users' devices. Accesses to this preference registry are read-mostly; an access occurs for every incoming telephone call. Updates to the registry occur less frequently, on the order of once every week for each user.

Proxy framework: As part of an undergraduate class (Berkeley CS199) on scalable services, the DDS was used as the storage manager for a proxy framework that allows untrusted, heterogeneous devices to access secure services. The proxy framework transforms content on-the-fly to reduce the value of the information in the content, so that it can be safely displayed on untrusted devices. The proxy relied on the DDS to store filters, device profiles, and user preferences.

A number of interesting incidents occurred during this experiment that caused the DDS to fail in some manner. These incidents all represented violations of the assumptions we made while designing the hash table (Section 7.1), and they resulted in a loss of one or more of the service properties. In all cases, they could have been avoided with more careful engineering, administration, or documentation. The violations occurred because of latent bugs in the system, or because the "customers" using the hash table were not aware of an assumption, causing them to build something outside of the intended operating regime of the DDS. Nonetheless, these incidents happened, and it is useful to examine closely why they happened to reflect upon the reasonableness of our assumptions.

8.1.1 NFS Considered Harmful

A running distributed hash table consists of many bricks executing on multiple cluster nodes; each brick is a set of classes executing in a JVM that manages a slice of the distributed hash table on its node's local disk. To run the hash table, Java `.class` files that comprise the brick program must be shipped to each cluster node. Rather than implement our own `.class` file distribution mechanism, we initially chose to rely on an NFS networked file system to house both these `.class` files and the JVM executable itself. This has the nice property that changes to the `.class` files during development are instantly accessible throughout the cluster.

Unfortunately, this decision introduced a coupling between the nodes in the cluster that led to multiple violations of our assumptions. Our NFS file server was not scalable; as we scaled the size of the hash table, the number of bricks that relied on the NFS server to read the JVM executable and the brick `.class` files increased, and thus the workload on the NFS server increased. By the time we scaled up to a 128 node hash table, the NFS server was completely saturated, and it took over a minute for JVMs to launch on all nodes, and another minute for the minimal set of `.class` files necessary to execute the bricks to be classloaded by the JVM. Strictly speaking, this isn't a violation of any assumption; we expected that bricks will be restarted independently, and thus during steady-state operation, the demand on the NFS server would be extremely low.

However, modern JVM's perform lazy classloading: they only read, validate, and JIT compile `.class` files when program execution happens to first touch that class. Thus, if there is an execution path in a program that happens rarely, it is possible that `.class` files touched by that execution path won't be classloaded for many minutes, hours, or even days. Two such rare execution paths correspond to a timeout, or to a DP map update from a brick failure. Since timeouts and failures occur infrequently, this execution path is rarely exercised. The unfortunate consequence of this is that the first time a brick fails in a running hash table, all DDS libraries and bricks will quickly detect this, and simultaneously issue reads against the NFS server for the necessary `.class` files. This results in a barrage of read traffic that saturates the NFS server, causing all of the bricks to "seize up" for many seconds or tens of seconds.

This behaviour represents a failure of our assumption of **synchrony**, namely that processes take a bounded amount of time to execute tasks and that messages take a bounded amount of time to be delivered. Because the NFS server doesn't scale, but was relied upon by all nodes in the system, we inadvertently introduced a coupling between system elements that resulted in poor scaling behavior and a violation of our synchrony assumption. The degree of this violation grows with the size of the cluster, since the NFS server doesn't scale.

A second violation occurred because of this coupling. As previously mentioned, changes to `.class` files served by the NFS server are instantaneously visible across the cluster. Because `.class` files are lazily loaded, it is possible that a given class file has not been loaded, but the JVM has already made assumptions about its interface signature based on a dependent class that has already been loaded. Changing a `.class` file on NFS can thus lead to the situation where a JVM loads a class with an interface signature that it doesn't expect. When faced with this situation (which arises due to a race condition and is extremely non-deterministic), the JVM will crash. Thus, by recompiling `.class` files on the NFS server, it becomes possible to cause many nodes in the cluster to simultaneously crash. Even worse, this simultaneous crash may happen at a much later time than the `.class` files were changed, because of the lazy class loading of the JVM. This represents a violation of our assumption of **independent failures**.

The obvious solution to these violations was to eliminate the NFS server, and instead rely on a more carefully controlled `.class` file distribution mechanism. We also believe that it would be extremely useful to introduce formal versioning mechanisms to `.class` files, to prevent the situation where dependent class files' signatures inadvertently change. Even though there was an obvious solution in this case, the lesson learned from these violations is that even a seemingly innocuous coupling between nodes can lead to correlated failures and a violation of our assumption of synchrony.

8.1.2 DDS as a Lock Manager

The design of the vSpace execution platform required a globally accessible lock in order to serialize nodes' accesses to a group membership map. To build this global lock, the vSpace authors chose to exploit the fact that the `put()` method in the distributed hash table API returned the old value of the element in the hash table. Using this, they built a `test-and-set` lock primitive by populating a hash table with a boolean element behind a globally known key. If the key contains the value "0", the lock associated with the key is available, and if it contains the value "1", the lock has been grabbed. To grab the lock, a vSpace node would `put()` the value 1 into the lock; if the old value returned by the `put()`

method is 0, then that node successfully grabbed the lock. If not, somebody else has the lock, and the node repeatedly spins until it eventually obtains the lock. To release a lock, the vSpace node would `put()` the value "0" into lock.

This lock implementation was functionally successful, and performed well if there was no contention for the lock. However, if there was any contention at all (i.e., more than one node simultaneously attempting to access the lock), performance became intolerably poor. Investigation showed that there were two compounding effects leading to this poor performance.

The first effect that lead to poor performance was the impact of having the competing nodes spinning in order to acquire the lock. If one node successfully grabs a lock, then all other nodes competing for the lock will repeatedly issue `put()` calls to the hash table, generating vast amounts of load on the table. This load causes the latency of operations issued against the table to increase, which in turn increases the time it takes for a node to release and acquire a lock. The competition for locks has effectively dilated time, increasing the latency to successfully acquire and release a lock.

The second effect that lead to poor performance was an interesting implementation artifact arising from the two-phase commit protocol used to effect state changing operations. In the first phase of the two-phase commit, the issuing DDS library sends "prepare" messages to all replicas serving the specific key. These prepare messages are sent in parallel, and thus the order in which they arrive at the replicas depends on factors such as network load and the timing of message arrivals in the face of competing traffic. For the two-phase commit to succeed, all replicas that receive the prepare message must successfully grab a local lock; if any cannot, that replica will send back an abort.

When multiple nodes compete for the vSpace test-and-set lock, those multiple nodes will each issue parallel "prepare" messages to the same replicas that control the lock element. As the number of nodes competing for the test-and-set lock increases, the chance that any given node will successfully grab all brick local locks needed for the two-phase commit rapidly diminishes, since it is likely that at least one of the locks has been grabbed by another node. Thus, as the number of competing nodes increases, it becomes much less likely that any forward progress would be made.

These performance implications arose because of a violation of two of our workload assumptions. We assumed that keys accessed by services would be largely **independent**, and that the **working set** of hot keys would be greater than the number of bricks in the cluster. For the vSpace test-and-set workload, the vSpace lock element is an extremely hot key that is not independently accessed by the underlying vSpace, leading to these two performance degradation effects.

To fix the first effect (performance degradation due to competing nodes spinning to grab the lock), we experimented with adding randomized exponential backoff to the lock acquisition algorithm. Each time a node unsuccessfully bids for the lock, it doubles the amount of time ($\pm$ a random offset) that it waits before bidding for the lock again. To address the second effect (starvation due to parallel "prepare" messages in the first phase of the two-phase commit), we experimented with modifying the two-phase commit algorithm so that the prepare messages sent by nodes were serialized and ordered across bricks. Because of this, if a node successfully grabbed a local lock on the first brick, it is guaranteed to grab all of the locks on the other brick, eliminating the potential for starvation. These experiments successfully improved the performance of the vSpace lock manager, but nonetheless it still had unacceptably high latency (100-500ms to grab a lock under contention from 5-6 nodes) and high overhead (an average of 5 attempts were necessary to grab the lock when under contention from 5-6 nodes).

We believe that the use of the DDS as a global lock manager falls out of the expected operating regime of our current system design. Although it may be possible to further modify the design of the hash table to better accomodate this workload, we believe it would be better either to add additional operations to the hash table that would simplify the construction of a lock manager, or to design a distributed lock manager specifically for this purpose that is completely independent of the hash table. In Section 10.3.2, we discuss these design alternatives in detail.

8.1.3 Independence of Failures

Two separate incidents over the 6 month experiment lead to the breakdown of our assumption of independent failures. The first incident occurred when we ran our throughput scaling benchmarks using all 208 CPUs in the millennium cluster (128 for the hash table, and 80 for load generators). After a few minutes of gathering data, 1/3 of the nodes in the cluster simultaneously lost power. Investigation revealed that the power supply unit for the machine room that housed the cluster was underprovisioned for the number of machines that had been installed. Running the benchmark on the full cluster forced all of the CPUs in the cluster to run at full load, generating a spike in power consumption that eventually caused a fuse to blow in the power supply. The solution to this was of course to properly provision the power supply for the machine room. However, it serves to once again demonstrate that coupling between the nodes in the cluster can lead to correlated performance degradation or failure. In this case, the coupling was the power supply of the nodes.

The second incident occurred when we first attempted the experiment in which we bulk-loaded a hash table with the 1.28 terabytes of data. After about 1 hour of loading, the all of the bricks in the hash table began to fail within seconds or minutes of each other. Investigation revealed that the bricks had run out of memory, and that an extremely subtle, latent, slow memory leak was surfacing. During normal two-phase commit processing, the sequence of events that a brick experiences is:

```
1) receive PREPARE message from two-phase commit coordinator

2) create a state-machine to handle this action, and store the
   state machine in a multiplexing table

3) grab a lock for the requested key, and issue a single-node
   hash table lookup for the requested key

4) receive a completion for that lookup, and send an ``OK''
   message to the two-phase commit coordinator

5) receive a ``COMMIT'' message from the coordinator, issue
   a write to the single-node hash table, and release the lock

6) destroy the state machine for the action
```

However, it is possible (although extremely rare) that another brick participating in the two-phase commit will fail to grab a lock for the requested key, and will send an "FAIL" message to the two-phase commit coordinator. In this situation, the coordinator will issue "ABORT" messages to all bricks instead of the "COMMIT" messages that would normally arrive in step 5 above. If this ABORT arrives between steps 3 and 4 above, our brick implementation would correctly cancel the hash table lookup, but it would fail to destroy the state machine for the action. Thus, in this rare circumstance, the memory associated with that state machine would be leaked. Over the period of an hour, enough of these rare circumstances would occur so that the bricks would run out of memory, and crash. The leak rates were roughly the same across all bricks, and thus the bricks would crash out at approximately the same time.

The solution to this problem was simple: we fixed the bug that caused the memory leak, eliminating the correlated failure. However, this sort of correlated failure demonstrates that the bricks are all indirectly coupled together by their workload alone, and if any memory leak exists in any part of the system (including those that we don't control, such as the operating system), then correlated failures can emerge again. A more robust solution is to accept that these sorts of failures are inevitable no matter how carefully software is built, and to restart bricks proactively (perhaps even rebooting cluster nodes) as preventative maintenance against memory leaks in both our software, libraries we depend on, and the operating system itself. Regardless, we believe it is better to address independence of failures inside the DDS, rather than forcing each application to address it itself.

8.1.4 Failstop

We also observed an incident that led to the failure of our assumption of fail-stop behavior. In this incident, a service that used the hash table was run on a machine that wasn't physically housed in the cluster. This machine was configured to act as a firewall, dropping all packets associated with incoming TCP connection requests. Because of this, any TCP connection attempt made to this machine would block for up to 12 minutes, since standard TCP implementations reattempt connection requests 10 times with an exponentially increasing backoff between attempts. Because of the fact that we used a single thread inside the "RPC" layer of our brick implementation, any connection attempt made by our brick to that machine would cause the brick to lock up and become unresponsive for 12 minutes. Unfortunately, part of the RPC layer included a session layer that would automatically created a connection to services that use the DDS, so as soon as that service first contacted the DDS, any brick that was involved in an operation from this service would become frozen for 12 minutes, but would then wake up and continue to operate.

The side-effect of this behavior was that all of the DDS libraries in the cluster would falsely believe that these frozen bricks had crashed, since all operations to those bricks would inevitably time out. However, 12 minutes later, the bricks would reappear. Fortunately, none of the existing DDS libraries would communicate with these bricks, but any newly instantiated client libraries would do so, since they haven't been informed that the bricks are dead. This would lead to a state of system delusion.

This particular failure mode would not happen in a more closely administered environment; both the fact that the machine housing the service wasn't a part of the cluster, and the fact that this machine was running firewall software violated our stipulation that the DDS and services were in a carefully controlled environment. However, it serves to demonstrate that there are situations in which fail-stop behavior might not occur.

8.1.5 Debugging Workload vs. Operational Workload

One final assumption violation occurred as part of the cs199 undergraduate class efforts. We assumed that tables are large and long-lived. As a result, we didn't spend any effort optimizing the table creation or destruction operations. Also, we assumed that creation and destruction would be "special" events that occur rarely, and when they do occur, they will be done under the careful watch of a single system administrator. Because of this, we failed to synchronize these operations to permit multiple processes to issue them simultaneously.

What we failed to realize is that during the process of designing, implementing, and debugging a service, that service and its associated DDS tables would be instantiated and destroyed many times. In fact, during a typical debugging cycle, it is likely that a service's tables would be created and destroyed every few minutes. Worse, independent programmers may simultaneously attempt to create, destroy, and use the same table, resulting in extremely unpredictable behavior because of our lack of synchronization of creates and destroys. Because of this, we abandoned our assumption that creation and destruction operations are rare events, and reimplemented the table creation and destruction operations to include synchronization and atomicity with respect to other operations.

8.2 Java as a High Performance Systems Platform

We found that Java was an adequate platform from which to build a scalable, high performance subsystem. Certainly, Java's high-level programming model made for extremely rapid development. Java's strong typing also encouraged modularity, and greatly facilitated the abstract interfaces of the I/O core and its event delivery mechanisms. 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 very high 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 influence garbage collection behavior adequately. The presence of garbage collection therefore makes it impossible to accurately predict the performance of a system at any given instance, although the effect of garbage collection is predictable and can be averaged out over long periods of time. As shown in Section 7.3.1, the overhead of the garbage collector can increase as more active heap space is allocated by an application, introducing the potential for thrashing.

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. Accordingly, we have implemented a high-throughput asynchronous I/O completion mechanisms into the JVM using the JNI native interface; because it depends on JNI, this mechanism isn't portable across operating systems or architectures.

Our final observation about Java is that in our experience, our programs run a factor of 3-5 slower than equivalent "C"-based programs. Part of this is because of the relative immaturity of Java compiler technology, but a fundamental part of this is a result of garbage collection and because of the overhead due to the high-level system interfaces offered by Java runtime systems. The large degree of abstraction provided by these interfaces simplifies programming, but at the cost of performance. As processors and I/O gets faster, it is possible that the benefits of this simplification will outweigh the associated performance costs.

8.3 Example Services

In this section, we describe several interesting services that we implemented using our distributed hash table. The services' implementation was greatly simplified by using the DDS, and they trivially scaled by adding more service instances.

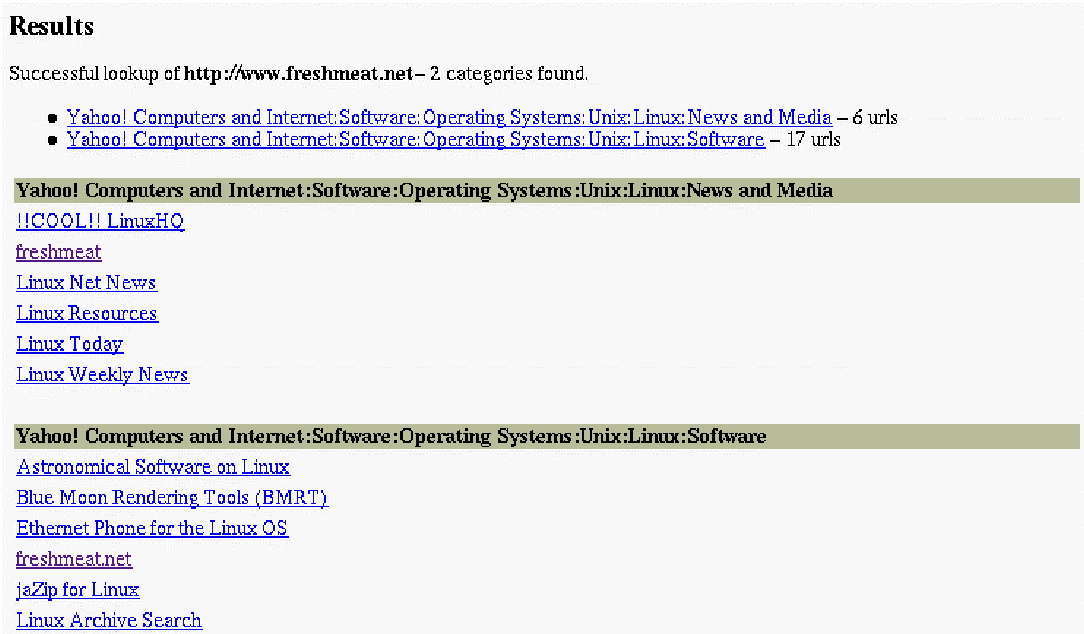


Figure 52: **Parallelisms:** The Parallelisms services uses an inverted index of the Yahoo! web directory (stored in a DDS) to return a list of web pages that are ontologically related to a user-specified URL.

8.3.1 Parallelisms

The Parallelisms service[1], built using the early prototype of the distributed hash table, allows a user to search for web sites that are semantically related to a user-specified URL. Parallelisms uses a statically constructed large table that maps URLs (the user-specified page) to groups of URLs (related sites, grouped by semantic relation). This mapping was obtained by crawling the Yahoo! web directory, which provides a list of sites that is hierarchically structured according to semantic content. To build our related site map, we inverted the Yahoo! directory: given a URL, our inverted table returns a list of Yahoo! ontology paths. We also maintain the normal Yahoo! ontology so that we can convert those ontological paths to URLs. Figure 52 shows Parallelisms in action: given the URL

[1]Parallelisms is periodically available at `http://ninja.cs.berkeley.edu/demos/parallelisms/parallelisms.html`

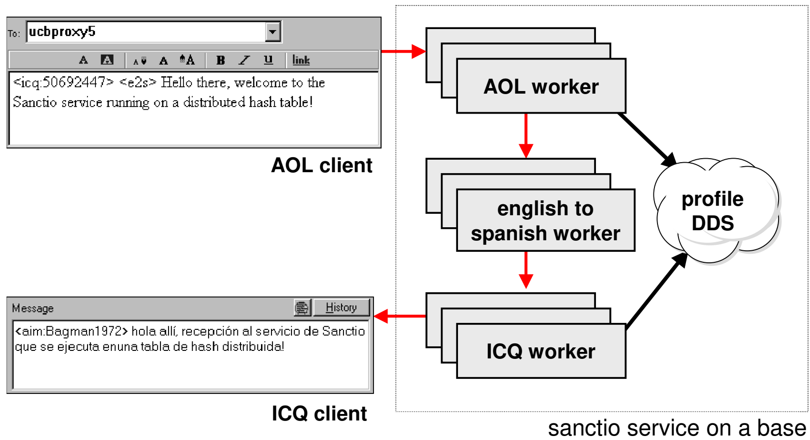


Figure 53: **Sanctio Messaging Proxy:** The Sanctio messaging proxy service is composed of language translation and instant message protocol translation workers in a base. Sanctio allows unmodified instant messaging clients that speak different protocols to communicate with each other; Sanctio can also perform natural language translation on the text of the messages.

`http://www.freshmeat.net`, Parallelisms returns two semantics groups (Linux "News and Media", and Linux "Software") that contain a total of 23 related URLs.

Both the inverted and regular Yahoo! ontologies are stored in a single distributed hash table. This table (which currently contains over 4 million entries, averaging roughly 250 bytes each in length) was populated through repeated insertions from the Yahoo! crawl log. Because all of this data is maintained in the distributed hash table, the Parallelisms service itself is quite simple: it contains approximately 400 lines of C code, 130 of which is service-specific logic, and 270 of which is dedicated to thread management, network socket management, and HTTP/HTML parsing and generation.

8.3.2 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 (Figure 53). Sanctio acts as 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 (robust) 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.

8.3.3 Web Server

We also implemented a rudimentary 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 unoptimized Java, 750 of which deals with HTTP parsing and URL resolution, and only 50 of which deals with interacting

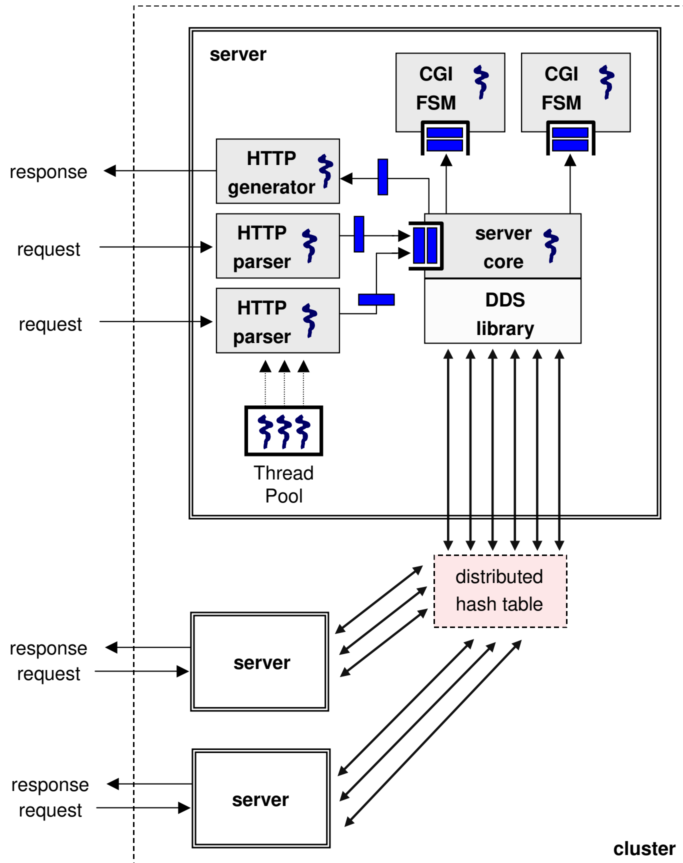


Figure 54: **Scalable Web Server:** The scalable web server consists of a number of stateless web server front ends, all of which rely on a distributed hash table to access the served web pages.

with the hash table DDS.

The web server architecture, shown in Figure 54, consists of a number of stateless web server instances, all of which use a distributed hash table to access the served web pages. Each server instance maintains many concurrent HTTP connections; each connection is served by a dedicated thread (dispatched from a thread pool) that parses the incoming request. Parsed requests are placed on a single request queue, which is served by a "core" thread that either routes the request to a CGI handler module (which is separated from the rest of the server by a thread boundary), or issues a request to the hash table for the appropriate page. Completions from the hash table or CGI handlers are handed back to the dedicated connection threads, which write back a response and then close the connection.

Using the workload generator described in [12], we benchmarked our web server. Each server instance can sustain 127 requests/s for 4KB pages; for each request that a server handles, a single distributed hash table lookup is issued. Using the hash table benchmarks in Section 7.3, we conclude that in order to saturate a 2-way replicated hash table with N brick processes, we would need to run $4N$ web server instances (assuming that the web server workload fits in the physical memory of the bricks). Using the millennium cluster, we were able to scale to a 16 node distributed hash table, 60 web server instances, and 82 load generators. With this configuration, our web servers were able to sustain an aggregate throughput of 7476 requests/s. The extremely high overhead of our HTTP parsing within the web server was the bottleneck of our system; significant performance improvements could be obtained by optimizing this code.

8.3.4 Others

We have built many other services as part of the Ninja project[2]. For example, we implemented a collaborative filtering engine for a digital music jukebox service [61]; this engine stores users' music preferences in a distributed hash table. We have also implemented a reusable private key store service and a composable user preference service, both of which use the distributed hash table for persistent state management.

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

Part IV

Related and Future Work

Chapter 9

Related Work

This dissertation contains work that is most directly related to the following three areas of systems research:

1. Efficient, robust, and reusable programming models and abstraction libraries for high-performance or highly concurrent systems;

2. Scalable high-performance storage systems, networked file systems, and databases;

3. Platforms for simplifying the construction of scalable, highly available Internet services or for the construction of cluster-based parallel applications.

In this chapter, we describe the relevant related work in these three areas, and compare it to that in our dissertation.

9.1 Programming Models for Concurrent Systems

The ADAPTIVE Communications Environment (ACE) [42] is a C++-based object-oriented framework that provides platform-independent implementations of a number of design patterns for distributed systems software. ACE includes patterns for IPC, shared memory management, object brokerage, connection management, naming, and logging, but most relevant to this thesis are its patterns associated with event demultiplexing and concurrency. In particular, the Reactor pattern [116] and Proactor pattern [76] abstract away mechanisms for non-blocking I/O channel availability and asynchronous I/O completions; the JAWS web server [75] is an example application that uses the Proactor pattern to simplify its development. Reactor and Proactor are similar to the I/O core's mechanisms for asynchronous I/O completion and application-specific event handlers. However, the I/O core further advocates a code structure on top of this event-delivery mechanism, describes how this code structure can be used to condition applications against load, and identifies composition operations and additional patterns that can be used to compose multiple components resulting in a service with availability and scalability.

In [104], Ousterhout gave a now-infamous presentation on the tradeoffs between threaded and event-driven programming styles. In this presentation, he claimed that the disadvantages associated with threaded programming (debugging reentrant code, deadlock elimination, performance overhead) imply that event-driven programming should be selected whenever possible. In this thesis, we agree with his conclusions, but only for unstructured threaded programs. The "wrap" design pattern and thread boundaries described in this dissertation, if used appropriately, can eliminate the programming-related disadvantages of threaded programming while bounding the performance overhead associated with it. We believe that event-driven programming is necessary for maximum performance from high concurrency systems.

Significant effort has been put into improving the mechanisms associated with both threaded and event-driven systems; here, we highlight a few notable projects. In [110], Pu et al. describe a mechanism for incrementally specializing operating system functionality based on known invariants. Using this specialization, threads themselves can carry synthesized kernel calls with them that vastly improve the latency and decrease the overhead associated with operations such as I/O, queue manipulation, signal processing, and context switching. In [5], the distinction between kernel and user threads is blurred through an API and mechanism for unifying kernel-level and user-level schedulers. This mechanism provides notification to the user-level scheduler of kernel events; the resulting system has the performance of user-level threads with the consistent behavior and true concurrency of kernel threads. In [13], Banga et al. advocate changes to the UNIX system call interface to more efficiently support event notification to event-driven applications, and in [14], modifications to the `select()` UNIX system call are proposed to result in higher throughput and scalability for realistic concurrent workloads.

The Click modular packet router [101] uses a software architecture which is similar to our design framework; packet processing stages are implemented by separate code modules with their own private state. Click modules communicate using either queues or function calls, so threads can cross module boundaries. Click is a domain-specific system for obtaining high concurrency in a packet router, and as such is less general and lower-level than the framework presented here. Click uses both *push* and *pull* semantics for flow control; that is, packet processing modules can send data downstream or request that data be pushed upstream to it. The rationale for pull processing is that push semantics require packets to be queued up before stages which are not ready to process a packet (for example, when a network port is busy). The push/pull distinction is important when threads can cross module boundaries, as is the case in Click. Our framework always imposes a queue between modules, so threads push data downstream (to other queues) and pull data upstream (from their incoming queue). Click could be implemented using our framework, by creating a thread pool boundary (using the *Wrap* pattern) where a queue exists between two Click modules. Click modules that communicate through function calls would operate within a single task handler using our framework.

Many high-performance web server and proxy cache implementations use events for achieving high throughput and high concurrency. For example, the Harvest web cache [26] contains a single thread, and uses non-blocking I/O mechanisms for passing completions through the various layers of the system. Similarly, the Flash web server [105] uses events for network I/O, and a thread pool to simulate events for disk I/O.

High speed I/O systems have also been the focus of much study. On the networking side, much progress has been made in the realm of fast and zero-copy network transports [128, 127, 30, 102] and operating system structure to support fast networking [43, 77, 106, 44]. Work has also gone into obtaining balanced I/O streaming in cluster environments [10, 9]. Our work is complementary to this research, and can be layered on top of these existing techniques or mechanisms.

9.2 Scalable Storage Systems

Litwin et al.'s scalable, distributed data stores (SDDS) such as RP^* and LH^* [83, 90, 91, 92] helped to motivate our own work. Their work focuses primarily on defining the theoretical and algorithmic properties of a family of scalable data structures, demonstrating properties such as the smooth incremental scalability and fault tolerance in the presence of failures. Their work does not, however, consider the systems issues of implementing an SDDS that satisfies the concurrency, availability, and incremental scalability needs of Internet services; most of their proposed data structures have not been implemented or tested with the same operational rigour as our distributed hash table. In addition, the consistency model offered by their data structure is unspecified.

There are a variety of projects that provide programmatically accessible storage in the infrastructure. The Linda project [3] provides a collaborative storage space to be used for both data persistence, process synchronization, and event posting; Linda became quite sophisticated, incorporating notions of transactions and query processing. Linda imposed its own storage and access methods on applications, and didn't allow them to select from a number of different data structures, which is the ultimate goal of our work. Furthermore, Linda didn't focus on making its storage available and scalable, and it was not implemented in a clustered environment. Recent spinoffs of Linda include the IBM TSpaces project [133] and Sun's JavaSpaces [98], both of which provide Linda-like functionality to the Java environment, but in a non-scalable way.

The single-node hash table, although not the primary focus of our work, is related to the design and implementation of single-node file systems [96, 111] and databases [11, 25, 58, 63, 65, 95, 117, 118]. Our current design is relatively naive in comparison to this research. For example, if a service currently receives write traffic with little locality, then the single-node hash table design will become a bottleneck due to seeks. To ameliorate this, we could adopt a log-structured approach similar to [111] to merge the many writes into a single log; writing this log would result in sequential traffic, eliminating expensive seeks.

Our work has a great deal in common with distributed and parallel databases. The problems of partitioning and replicating data across shared-nothing multicomputers has been studied extensively in these communities [46, 64, 89, 120]. 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. If we were to strengthen the consistency model of our hash table to include transactions, the resulting system would look very similar to a distributed relational storage system (RSS), to use the terminology of [11].

We also have much in common with distributed and parallel file systems

[6, 33, 48, 84, 113, 126]. 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 [86], 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 [47] 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 [49] attempt to transparently add 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.

There have been many projects that explored wide-area replicated, distributed services [45, 99]. 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.

9.3 Clusters and Internet Service Platforms

Much work has gone into researching issues associated with clusters of workstations. Some of this work focuses on providing run-time environments that attempt to provide a single, unified system image of the cluster, and on harvesting idle resources in that cluster [51, 60, 93, 94, 123, 129]. Other systems have attempted to glue wide-area computing systems into a world-wide computational grid [28, 54, 73]. Research has also gone into issues such as load balancing in a cluster environment, using techniques like implicit load balancing [8] and locality-aware request distribution [50].

A number of projects have explored the use of clusters of workstations specifically as a general-purpose platform for building Internet services [2, 7, 35, 55]. To date, these platforms rely on file systems or databases for persistent state management, or they implement their own service-specific storage layer. 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 [112] 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.

A recently emerging product category in the commercial world is that of application servers, including BEA WebLogic [15], ObjectSpace Voyager [103], and IBM WebSphere [78]. An application server is a middleware platform that bridges the gap between web server front ends and database back ends; an application server seeks to support scalable, concurrent applications using industry-standard programming interfaces, such as Enterprise Java Beans [121] and Java Servlets [79]. Although these systems provide a variety of application programming interfaces, their internal structure is generally thread based. Threads, network connections, and database connections are usually pooled to limit resource consumption on the server; replication is used across several servers to provide scalability and fault isolation.

Kaashoek et al. [82] and Mogul [100] propose specializing operating system architectures for server applications, using Internet services as a specific example. Although this work focuses on low-level aspects of O/S performance for servers (such as disk and network access overhead), it also realizes the benefit of an event-driven concurrency model. In [82], application-specific handlers [130] are used to install application-level event handlers in the kernel for added performance. This approach complements our design framework by providing novel kernel-level functionality to improve I/O performance.

Chapter 10

Discussion and Future Directions

In the course of experimenting with the programming model of Part II and the distributed hash table implementation described in Part III, we uncovered a number of topics that warrant additional exploration and design reconsideration. In this chapter, we touch upon the most interesting of these topics.

10.1 Programming Model

10.1.1 Debugging

A known weakness of event-driven models is the increased difficulty of debugging them. The control flow of a task in an event-driven model is no longer equivalent to the control flow of a single thread, since tasks can span threads by crossing thread boundaries, or they can have no execution context at all while sitting idle in queues. Because of this, it is very hard for programmers to trace the chain of events that result from an occurrence such as task reception. Traditional debuggers only allow programmers to inspect variable state and to trace the execution of threads; such debuggers have no notion of causality from events, and accordingly it is impossible for them to provide an event tracing facility.

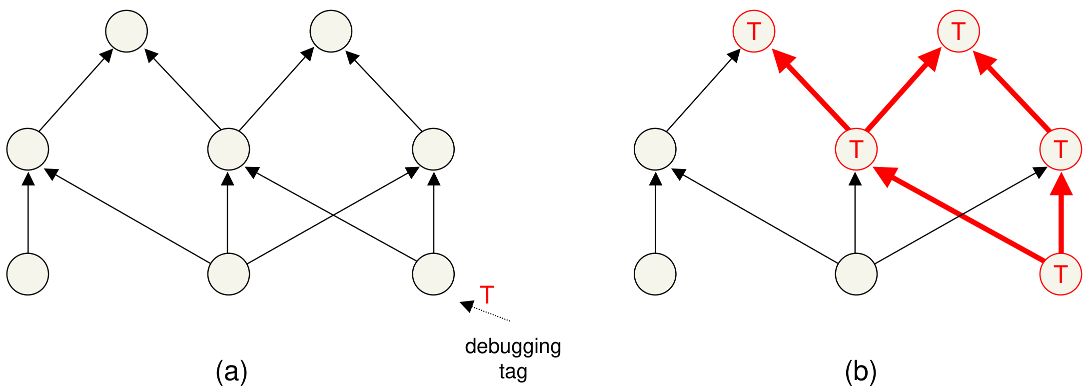


Figure 55: **Debugging event flow:** In (a), we depict a graph of event handlers and the event flow between them. A hypothetical debugging "tag" is injected into the event flow graph. In (b), the resulting event flow paths and affected handlers are highlighted; this highlighted graph would be output by the debugger, along with timing information.

We see an opportunity to introduce "event aware" debugging capabilities to programs that make use of the primitives provided by the I/O core; this is possible because events, event reception, and event dispatch are all strongly typed primitives in our programming model. The debugging tool that we envision would allow a programmer to interrupt program execution, inspect all current events in the system, and tag any event in order to initiate the monitoring of its flow. The debugger would then log the path that this event and any events that are causally related to this event take through the system, and allow the programmer to inspect this log, as well as all associated events currently in the contour generated by the chain of causality (Figure 55).[1]

Even with the strongly typed event and event dispatch mechanisms of the I/O core, it is still impossible for a debugger to identify all chains of causality. For example, a program may aggregate multiple event receptions over time to produce a new, single aggregate event; a debugger has no way to automatically recognize that this has happened. To overcome

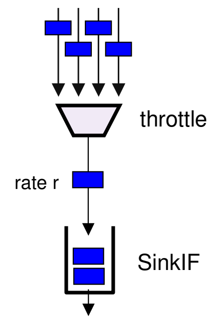
[1]To use a physical metaphor, this is the programming equivalent of an intravenous pylogram (IVP) X-ray commonly used to inspect patients kidneys; to perform this procedure, doctors inject a special material into a patient's bloodstream that appears distinctly on the X-ray, making it possible to trace the flow of blood into the kidney.

this limitation, programmers could instrument their code to provide debugging hints to the infrastructure regarding such causal events. In the previous examples, the programmer could explicitly add all of the tags from the incoming events to the outgoing aggregate event.

10.1.2 Conditioning Beyond Load

One of the benefits of the programming framework that we presented in Chapter 4 is that the explicitly exposed queues and event-driven style of control flow permit the use of patterns such as "wrap" to condition application code. The "wrap" pattern imparts robustness to load, and limits the concurrency of the code to keep it within an efficient operating regime. The "replicate" pattern imparts robustness in the face of failure, assuming that the replicated code has some adequate strategy for replicated state management.

An interesting future avenue of work would be to investigate whether or not there are other mechanisms that could be adopted to further condition code. For example, it would be extremely useful to be able to allow untrusted third parties to dynamically inject services into a running Internet service platform. Providing this mechanism would allow individual programmers to build, debug, and deploy scalable, robust services without requiring them to purchase and manage the cluster and machine room that houses the service. This would encourage the kind of distributed innovation that was responsible for making the web so successful: anybody could contribute to the Internet service landscape for relatively little cost. For this to be realistic, uploaded services must be conditioned to cause isolation from each other: a service must not be able to unreasonably disrupt the operation of another service. Isolation would involve imposing and enforcing limits on memory consumption, disk consumption, disk and network channel rates, as well as sandboxing code for the sake of security.



As an example of how to approach this additional conditioning, the sink abstraction discussed in Chapter 5 could be exploited to simplify the enforcement of rate limits on disk and network channels. Because all sinks implement the same `SinkIF` abstract interface, we could dynamically interpose a "throttle" element with this same interface on the sink, but have the throttle element perform rate limiting on its output flow. There could be many classes of throttle implementations that experiment with different rate limiting policies, such as leaky-bucket algorithms, fair queueing [39, 107], or class-based queueing [52].

In addition to conditioning individual stages, we could also apply conditioning across entire services that encompass multiple stages. Our programming framework considers services to be directed graphs of composed stages and patterns. These graphs look very similar to computer networks, in that they are composed of computational elements with queues, and data that flows between these elements in discrete "packets". It is possible that many of the same techniques used in the network community to induce good behavior on networks could be used to condition services, for example imposing flow control, congestion control, and using routing algorithms to induce a balance of data flow across the computational elements.

10.1.3 Demultiplexing and Layering

A challenge we encountered while building large, layered systems using our programming framework was being able to demultiplex events that arrive at a high-level layer from a lower-level layer. Because of our concurrency model, if the high-level layer needs to perform a long-latency operation using the lower layer, the high-level layer will bundle up the state associated with the executing task, and then asynchronously dispatch the op-

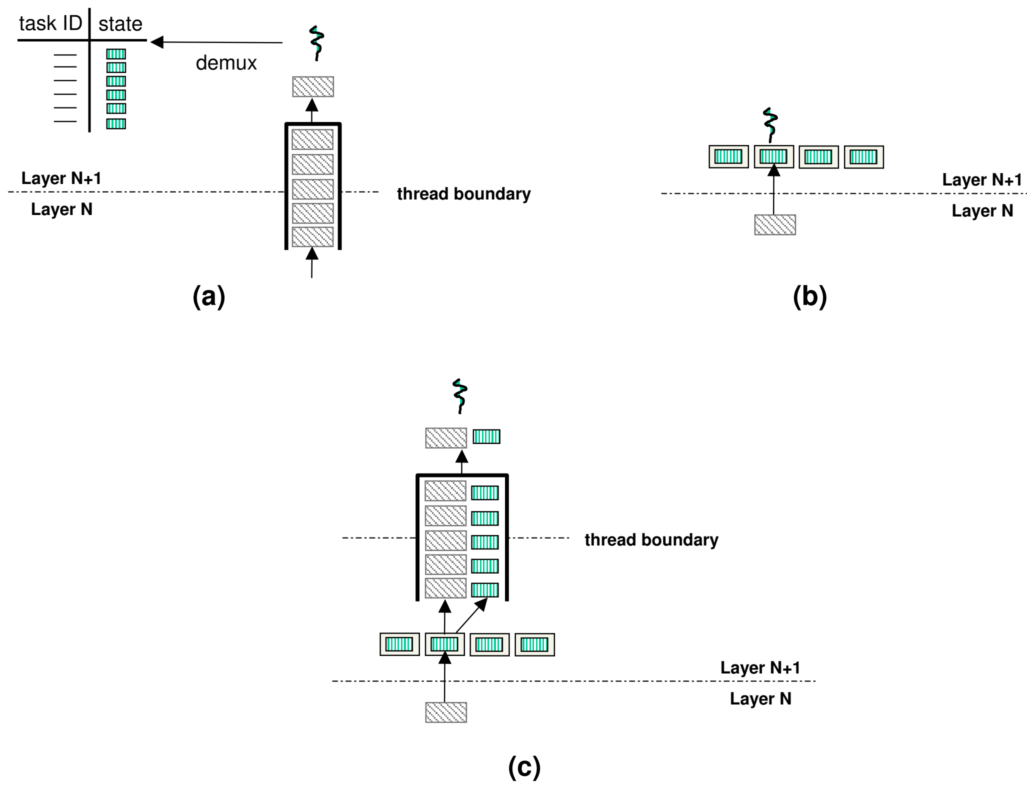


Figure 56: **Demultiplexing:** (a) Shows the demultiplexing challenge in our current code: given an incoming completion event, the event handler thread must match that completion with the task state associated with the completion. In (b), we show how we could route completions directly to handler interfaces wrapped around task state, eliminating demultiplexing altogether, but also eliminating thread boundaries. In (c), we show how to eliminate demultiplexing while preserving thread boundaries.

eration on the lower layer. Eventually, a completion event will flow from the lower layer and arrive at the higher layer. The demultiplexing challenge is in being able to find the appropriate bundled state for the task to receive the completion, based only on information in the completion itself.

As mentioned in Section 5.1.1, we augmented our APIs to allow callers to specify completion handlers for every I/O request that they issued. This was done to allow multiple applications to share lower layers, allowing completions to be automatically routed to the correct application. However, once the completion arrives at the application, there is still demultiplexing that must be done between concurrent tasks (Figure 56a).

We believe that some additional syntactic sugar can eliminate even this demultiplexing. Instead of storing an opaque state bundle to store tasks that are awaiting completions, we could wrap each of these bundles with fine-grained, dynamically created completion handler object. We could then specify this handler as the destination for completions for any requests generated by the handler's task. Completion events would thus arrive as method invocations on a task object, instead of arriving as elements on an application's queue and needing further demultiplexing (Figure 56b). A wrinkle with this approach is that it effectively removes the thread boundary between the layers, but this can easily be reintroduced by having the wrapper reenqueue the completion and the destination task on a queue (Figure 56c).

10.2 Distributed Data Structures

10.2.1 Indexes and Embeddable Data Structures

As we were building services using the distributed hash table DDS, it became clear that many services would benefit from the ability to embed complex structures inside a hash table. For example, in the collaborative filtering engine for the jukebox, the engine would store one hash table record for every song that a given user had listened to. Whenever a user played a song, the engine would simply look up the record for that song, and modify it. Occasionally, however, the engine would need to gather all records for a given user. In order to be able to do this, the engine manually maintained an index block for each user that included the keys of all records for that user. To fetch all records, the engine would retrieve this block, and afterwards retrieve (in parallel) all records in that index. The unfortunate consequences of this are that the index blocks grow with the size of the users' record sets, and that each addition of a record must be accompanied with a modification of an index block. Both consequences contributed to poor performance.

A better alternative to this index block would be to allow services to embed small data structures in the hash table. For example, the collaborative filtering engine could maintain a linked list across user records by having each record contain the key of the next record in the list. A challenge with doing this is maintaining the consistency of the embedded data structure without using transactions. Fortunately, because of the fact that we have two-phase commit already implemented in the hash table, we can exploit it to add additional atomic operations, as long as they only affect a single element. For example, we could add an atomic `compare-and-swap` operation, and given this, it is possible to build a consistent embedded linked list.

A second alternative to the currently implemented index block for the collaborative filtering engine would be to provide additional data structures that can serve as an index. For example, if users' records were stored in a B-tree instead of a hash table, enumerating all of the records would be an operation that is directly supported by the B-tree.

10.2.2 Transactions vs. Atomic Actions

As mentioned in Section 7.2.1, adding transactional support to the distributed hash table would introduce significant complexity and would warrant a substantial reconsideration of its design. However, many authors that implemented services using our hash table expressed a desire to be able to perform multiple operations atomically. This desire was originally expressed as a need for transactions, but upon closer discussion, it became clear that in most cases, authors needed to be able to perform atomic read-modify-write operations on single elements, rather than multiple operations that span elements (transactions).

These sorts of atomic operations permit the use of non-blocking synchronization primitives, as discussed in [67]. Non-blocking synchronization has a number of important benefits. It allows synchronized code to be executed in asynchronous event handlers, without fear of introducing deadlock. It also fully decouples process scheduling and synchronization; priority inversion is impossible with non-blocking synchronization. Finally, and most importantly, it completely insulates systems from deadlock due to processes failing while holding locks. For all three of these reasons, we believe that adding suitable non-blocking synchronization primitives can completely obviate the need for a lock manager.

Most non-blocking synchronization primitives (such as `test-and-set`, `compare-and-swap`, or `load-linked-store-conditional`) affect only a single element. However, more sophisticated primitives (such as `double-compare-and-swap`) exist that can vastly simplify non-blocking data structure implementations, but these primitives require the atomic modification of multiple elements. We believe that we can extend our two-phase commit to handle these multi-element atomic operations, but an interesting alternative would be to perform these operations on multiple bytes within the same hash table element. This would subtly introduce additional typing into the hash table interface, as specific bytes within elements would have meaning, instead of elements being completely opaque byte arrays.

10.2.3 Caching and Code Shipping

Our current distributed hash table implementation only makes use of caching within the bricks: disk blocks are accessed through the single-node hash table's buffer cache. Because of this, every read or write operation from a service results in network communication between DDS libraries and bricks, and a read or write being performed on one or more bricks' single-node hash tables. However, if there are workloads with considerable read or write locality, the distributed hash table would greatly benefit from distributed or cooperative caching by the DDS libraries. The mechanisms and policies that would be necessary for such caching are well-explored by previously research, such as [38]. Scalably maintaining cache consistency, and maintaining data availability in the face of cache failures would be the major design challenges; we believe that the correct path to doing so would be to use callback style cache consistency [48], in which bricks would notify libraries upon invalidation of hash table entries, as this style of cache consistency has proven to be scalable in previous projects.

Similarly, the current distributed hash table implementation is completely focused on data-shipping: any computation that is done on hash table elements must be done by services after having retrieved the elements, incurring the overhead associated with network communication. We can conceive of many services that would benefit from the ability to perform code-shipping, i.e., causing computation to be performed at the bricks that store the data. For example, if a service wished to select a hash table element based on its content, we would currently require that service to enumerate through all hash table elements, in essence copying the entire distributed hash table over the network and pushing it through a service filter. A much more efficient way to do this would be to ship the selection criteria in parallel to all bricks, and have the bricks perform a local search. This way, only the result of the selection would be transferred over the network.

One method of closely approximating true code-shipping is to give services insight into the toplogy of the distributed hash table, by granting them access to the current DP and RG maps. A service could then arrange to execute instances of itself on appropriate bricks' nodes, and directly manipulate single-node hash tables. Of course, by doing so, services would give up the consistency and transparent incremental scalability that the distributed hash table libraries provide. Accordingly, we expect that this technique would only be useful for read-only operations, such as selection.

10.2.4 Administration

This dissertation did not broach the topic of distributed hash table administration. A distributed hash table cannot be completely maintenance free; there are a number of events that can occur that require human intervention. If the contents of a hash table grow to exceed the aggregate capacity of the hash table bricks, an administrator must add more nodes to the cluster, and arrange for "split" operations to occur to incorporate this additional disk space. Similarly, if a node fails, an administrator must add a replacement node to the cluster, and arrange for recovery to initiate. We also believe that a DDS should be instrumented to report current operating conditions, such as request throughput and latency, and the distribution of requests across keys (to look for potential hotspots).

Our position is that the infrastructure can provide hooks for carrying out administrative tasks such as recovery, node addition, etc., but that human operators should initiate these tasks, rather than relying purely on automatic code to handle them. Including a well-trained human in the administrative chain helps to ensure that reasonable administrative decisions occur. The infrastructure can help with these decisions, for example by enforcing the constraint that no two replicas of the same partition are assigned to the same brick, or by suggesting additional replication to ease hotspots.

10.3 Future Directions

As is often the case, the research in this dissertation has generated as many (or perhaps even more) questions than it answered: there are several unexplored areas of fertile research that are offshoots of this work. In this section, we highlight several of these.

10.3.1 Layered Distributed Data Structures

Given a DDS such as our distributed hash table, it would be interesting to consider whether or not it is possible to layer higher-level data structures on top of it. For example, given a hash table and a B-tree, a useful aggregate data structure to have would be an indexed hash table through which content could be located both by key (via the hash table) or by searching/filtering on content (via the B-tree). If it were a relatively simple matter for service authors to build higher-level data structures, then they could implement structures that are particularly well suited to their specific service.

We expect that there will be significant challenges with building layered data structures. Ideally, authors would be able to build higher layers without having to explicitly worry about preserving the scalability, consistency, or availability characteristics of the lower-layer "foundation" data structures. However, it is not at all clear how to do this without resorting to heavyweight mechanisms such as nested transactions, or having to reveal the partitioning and replication schemes of the foundation structures so that the higher-level structures can exploit them.

An alternative to layering higher-level data structures on top of "foundation" structures such as hash tables and trees is to provide a DDS construction kit that contains a library of reusable essential elements of all distributed data structures. This library would include two-phase commit code, group membership maps, buffer caches, single-node data structure implementations, communications primitives, and perhaps split-phase RPC stub compilers. Given these pieces, it would be significantly easier to implement a service-specific distributed data structure than writing one completely from scratch. However, even with these pieces, the details of partitioning, load and content balancing, recovery, and even just getting the interfaces correct are subtle and complex. For these reasons, we suspect that the layering approach would be more readily adopted.

10.3.2 A Distributed Lock Manager

The experience we gained with using the distributed hash table as a lock manager (Section 8.1.2) demonstrated that although there is demand for a lock manager, the current DDS implementation is unsuitable for use as such. Two DDS implementation choices lead to this unsuitability: the use of parallel PREPARE messages in the two-phase commit, and the use of spinning retries if an operation fails.

A better lock manager implementation would keep a list of waiting tasks that have requested access to a held lock; when the lock is released, one of these waiting tasks would be notified. Notification obviates spinning, and fits in well with our upcall-based event-driven programming framework. For the lock manager to be highly available, the list of waiting process must be replicated. However, in order to guarantee that the replicas have the same list order, some sort of two-phase commit (or other total-ordering protocol) must be used on insert into the list. If a two-phase commit is used, then we must avoid using parallel PREPARE messages in the first phase, or risk suffering the same performance degradation under contention as experienced in Section 8.1.2.

There are two implementation paths to building a lock manager. The easiest path would be to embed the list of waiting processes inside a distributed hash table, relying on the non-blocking synchronization primitives that we discussed earlier in Section 10.2.2 in order to manage this embedded data structure. However, with this implementation, the service library that releases a lock would be responsible for notifying the next waiting process in the lock's list; we would need to design a scheme by which waiters periodically poll the lock list in case the service library fails before issuing this notification. Unless the two-phase commit were modified for lock operations (which is entirely possible given the modularity of the hash table implementation), the resulting lock manager may suffer degraded performance under high contention, but far less so than in the implementation in Section 8.1.2, since once a process makes it onto the waiter list, it no longer actively contends for access to the lock.

A second path would be to implement a new distributed lock manager data structure. Doing so would require significant implementation effort, but the resulting implementation could be highly tuned for the lock manager workload.

10.3.3 A Better Single-Node Hash Table

As previously mentioned, our current single-node hash table implementation is relatively naive. It is designed for workloads with significant read and write locality; if the offered workload doesn't have enough locality, the hash table performance is disk seek dominated. In addition, the hash table makes no effort to keep its on-disk image consistent, so in the case of failure, the entire hash table must be recovered from a peer. Improving disk performance in the case of traffic with poor locality could be addressed by using log-structured file system techniques [111] (in which all writes are directed to a log, resulting in sequential write traffic). Maintaining high performance while preserving on-disk consistency could be achieved by using techniques from journaling file systems, which keep a log of operations that can be replayed after a crash for quick recovery, or by using a technique similar to soft updates [97], which carefully orders data and metadata writes so that the on-disk image is always consistent.

10.3.4 Additional Data Structures

We would like to implement a few other additional data structures. We believe that an adequate set of structures to build a very large set of interesting services includes our existing distributed hash table, but also a distributed administrative log and some form of a distributed tree. Administrative logging (i.e., using a log to report interesting events,

Service	Data structures used
Web Server	**read-mostly hash table:** (static documents, caching popular documents, caching dynamically generated documents), **log:** (tracking accesses)
Web Proxy Cache	**hash table:** (caching documents), **log:** (tracking accesses)
Search Engine	**hash table:** (caching popular queries), **log:** (spooling crawl data), **hash table:** (word to document relation tables), **tree:** (index over documents), **log:** (tracking accesses)
PIM Server	**hash table:** (document repository), **hash table, tree:** (indexes over repository), **log:** (tracking accesses), **log:** (write-ahead logging for atomic, durable refiling operations). PIM stands for **P**ersonal **I**nformation **M**anagement, i.e. applications such as email and shared calendars.
Chat Engine	**hash table:** (client binding cache), **tree, hash table:** (indexes over client list), **log:** (tracking accesses, archiving chat sessions)

Table 10.1: **Examples of Services:** This table shows how one could hypothetically build some common Internet services out of a distributed hash table, a distributed tree, and a distributed log.

in the style of a web server log) is common to most services. The consistency model of an administrative log is weaker than a hash table: all that matters is that all events are logged in roughly the order they were generated, and that no events are lost. A tree provides the notion of ordering, which would allow the construction of indexes and the issuance of range queries. Table 10.1 shows how to hypothetically implement a number of common modern Internet services using these three data structures.

10.3.5 Alternative Languages to Java

The current performance bottleneck of the distributed hash table (assuming some locality in traffic) is related to our choice to use Java as our implementation language. Garbage collection, expensive class libraries, memory copies arising from the lack of pointers and also due to Java's network and disk abstractions, and the overall degradation in performance due to immature compiler implementations all contributed to relatively poor performance. Although optimal performance was never an explicit goal, and although we were successful in building a scalable implementation, we nonetheless feel that switching to a compiled language such as C++ would permit much more detailed exploration of issues such as damping the variability of throughput and latency, and more carefully balancing cluster resources such as CPU, bus bandwidth, disk throughput, and network throughput. The use of Java simply takes away too much of our ability to control memory usage, processor usage, and memory copies, preventing us from being able to explore such issues.

By switching to C++, we would also be able to more accurately compare our implementation to related abstractions such as parallel file systems or databases. Currently, the performance overhead introduced by Java makes such comparison unfair. For example, because of the economy of mechanism in our distributed hash table, we feel that an optimized implementation should achieve equivalent or higher throughput than modern relational databases. Also, we would be able to compare the performance of our scalable web server application that uses the distributed hash table to other high performance or scalable web server implementations, whereas currently the poor performance of our Java-based web server implementation prevents us from achieving comparable throughput.

10.3.6 Different Consistency Models

Our current hash table implementation offers only one consistency model, although this model and its implications are precisely described and well understood. There are many other consistency models that could be explored, such as transactional consistency, release-oriented consistency, or weak consistency on writes for read-mostly workloads. In addition to the obvious implications of performance and service expressability, we believe that the assumptions we made for our hash table implementation (described in Section 7.1) could be reexamined and perhaps weakened for these alternate consistency models.

Chapter 11

Conclusion

In this thesis, we have demonstrated that there is a programming model, a set of I/O abstractions, and a design framework that are much better suited to high concurrency, I/O centric scalable Internet services than the traditional thread-per-task, synchronous I/O model. We described this programming model and design framework in terms of design patterns that can be applied to code in order to layer it, condition it against load and concurrency, and replicate and pipeline it to give it high performance and availability. Services built using this framework behave well when scaled up, and the explicit queues in the model naturally impart the service property of graceful degradation.

Using this programming model, we demonstrated that it is possible to build a scalable storage platform specifically designed for the needs of Internet services. This storage platform contains all of the service properties; if services relinquish all of their persistent state management to this platform, then they inherit these service properties from the storage platform. This platform, called a distributed data structure, exports a conventional single-site data structure interface, but transparently partitions and replicates the data across a cluster. We made use of clusters' properties in order to tune the design of the DDS to best handle the workloads of Internet services and the failure properties of well-managed clusters.

Given these two elements (the Internet service design framework and the scalable persistent state management platform), we showed that it is possible to easily build a large and interesting class of Internet services that possess all of the service properties. We described several services that we implemented, including a scalable web server, and an instant messaging translation gateway.

There is a rich body of related work to this dissertation, primarily from three research communities: efficient programming models and primitives, scalable storage systems (including file systems and databases), and platforms for Internet service construction. We believe that our own work represents a synthesis and evolution of ideas from many of these communities, resulting in a important new design framework and storage abstraction particularly suitable for the emerging world of infrastructural services.

Bibliography

[1] Amazon.com Inc. The Amazon.com Web Service. `http://www.amazon.com`.

[2] 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.

[3] B. Anderson and D. Shasha. Persistent Linda: Linda + Transactions + Query Processing. In *Springer-Verlag Lecture Notes in Computer Science 574*, Mont-Saint-Michel, France, June 1991.

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

[5] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[6] 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.

[7] 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.

[8] Andrea C. Arpaci-Dusseau, David E. Culler, and Alan Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proceedings of the 1998 SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 233–243, Madison, Wisconsin, USA, June 1998.

[9] Remzi Arpaci-Dusseau, Andrea Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. The Architectural Costs of Streaming I/O: A Comparison of Workstations, Clusters, and SMPs. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, Las Vegas, Nevada, USA, February 1998.

[10] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David A. Patterson, and Katherine Yelick. Cluster I/O with River: Making the Fast Case Common. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems (IOPADS '99)*, May 1999.

[11] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, and Vera Watson. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[12] Gaurav Banga and Peter Druschel. Measuring the Capacity of a Web Server. In

Proceedings of the First USENIX Symposium on Internet Technologies and Systems (USITS '97), Monterey, CA, December 1997.

[13] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A Scalable and Explicit Event Delivery Mechanism for UNIX. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999.

[14] Guarav Banga and Jeffrey C. Mogul. Scalable Kernel Performance for Internet Servers under Realistic Loads, June 1998.

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

[16] T. Berners-Lee, R. Cailliau, A. Loutonen, H.F. Nielsen, and A. Secret. The World Wide Web. *Communications of the ACM*, 37(8):76–82, August 1994. Also see `http://info.cern.ch/hypertext/WWW/TheProject.html`.

[17] A. Bhushan, B. Braden, W. Crowther, E. Harslem, J. Heafner, A. McKenzie, J. Melvin, B. Sundberg, D. Watson, and J. White. The File Transfer Protocol. RFC 172, June 1971.

[18] David Bindel, Yan Chen, Patrick Eaton, Dennis Gees, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Christopher Wells, Ben Zhao, and John Kubiatowicz. OceanStore: An Extremely Wide-Area Storage System. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, Cambridge, MA, November 2000.

[19] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Call. *ACM Transactions on Computing Systems*, 2(1):39–59, February 1984.

[20] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, Michael F. Schwartz, and Duane P. Wessels. Harvest: A Scalable Customizable Discovery and Access System. Technical Report CU-CS-732-94, Department of Computer Science, University of Colorado, Boulder, March 1995.

[21] R. Braden. Requirements for Internet Hosts – Communication Layers. RFC 1122, October 1989.

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

[23] P. Buonadonna, J. Coates, S. Low, and D.E. Culler. Millennium Sort: A Cluster-Based Application for Windows NT using DCOM, River Primitives and the Virtual Interface Architecture. In *Proceedings of the 3rd USENIX Windows NT Symposium*, Seattle, WA, July 1999.

[24] P. Buonadonna, A. Geweke, and D. Culler. An Implementation and Analysis of the Virtual Interface Architecture. In *Proceedings of SC'98*, November 1998.

[25] Michael J. Carey, Joseph M. Hellerstein, and Michael Stonebraker. Designing the B-1: A Universal System for Information. Personal Communication, February 1999.

[26] 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.

[27] Mike Chen, Rob von Behren, and Nikita Borisov. Getting Started with vSpace v2. `http://www.cs.berkeley.edu/~mikechen/vspace/`.

[28] Bernd O. Christiansen, Peter Cappello, Mihai F. Ionescu, Michael O. Neary, Klaus E. Schauser, and Daniel Wu. Javelin: Internet-Based Parallel Computing Using Java. UC Santa Barbara Dept. of Computer Science, 1997.

[29] B.N. Chun and D.E. Culler. REXEC: A Decentralized, Secure Remote Execution Environment for Clusters. In *Proceedings of the 4th Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing*, Toulouse, France, January 2000.

[30] Brent Chun, Alan M. Mainwaring, and David E. Culler. Virtual Transport Protocols for Myrinet. In *Proc. Hot Interconnects 97*, 1997.

[31] F. J. Corbato, M. Merwin-Daggett, and R. C. Daley. An Experimental Time-Sharing System. In *AFIPS Proceedings of the 1962 Spring Joint Computer Conference*, pages 335–344, 1962.

[32] Fernando J. Corbató and Victor A. Vyssotsky. Introduction and Overview of the Multics System. In *AFIPS Conference Proceedings*, 1965. Available at `http://www.lilli.com/fjcc1.html`.

[33] Thomas H. Cormen and David Kotz. Integrating Theory and Practice in Parallel File Systems. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 64–74, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[34] HotMail Corporation. Hotmail. `http://www.hotmail.com`.

[35] Inktomi Corporation. The Inktomi Technology Behind HotBot, May 1996. `http://www.inktomi.com/products/network/traffic/tech/clustered.html`.

[36] Myricom Corporation. Myrinet Performance Measurements. `http://www.myri.com/myrinet/performance/index.html`.

[37] Mark E. Crovella and Azer Bestavros. Explaining World Wide Web Traffic Self-Similarity. Technical Report TR-95-015, Computer Science Department, Boston University, Oct 1995.

[38] Michael D. Dahlin, Clifford J. Mather, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. A Quantitative Analysis of Cache Policies for Scalable Network File Systems. In *Proceedings of the SIGMETRICS '94 Annual Conference on Measurement and Modeling of Computer Systems*, Nashville, Tennessee, May 1994. Association for Computing Machinery.

[39] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 1–12, Austin, TX, September 1989.

[40] Alan Demers, Karin Petersen, Mike Spreitzer, Doug Terry, Marvin Theimer, and Brent Welch. The bayou architecture: Support for data sharing among mobile users. In *Proceedings of the 1994 Workshop on Mobile Computing Systems and Applications*, December 1994.

[41] UC Berkeley CS Division. The Millennium Project (home page), 1999. `http://millennium.berkeley.edu`.

[42] Douglas C. Schmidt and Donal F. Box and Tatsuya Suda. ADAPTIVE: A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment. *Concurrency Practice and Experience*, 5(4), June 1993.

[43] Peter Druschel and Gaurav Banga. Lazy Receiver Processing: A Network Subsystem Architecture for Server Systems. In *Proceedings of the USENIX 2nd Symposium on Operating System Design and Implementation (OSDI '96)*, Seattle, WA, USA, October 1996.

[44] Peter Druschel and Larry Peterson. Fbufs: a High-Bandwidth Cross-Domain Transfer

Facility. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1993.

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

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

[47] 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.

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

[49] 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.

[50] 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.

[51] R.E. Ewing, R.C. Sharpley, D. Mitchum, P. O'Leary, and J. Sochacki. Distributed Computation of Wave Propagation Models Using PVM. In *Proceedings of the IEEE*

Supercomputing '93 Conference, Portland, OR, November 1993. IEEE Computer Society and ACM SIGARCH.

[52] Sally Floyd and Van Jacobson. Link-Sharing and Resource Management Models for Packet Networks. *IEEE/ACM Transactions on Networking*, 3(4), August 1995.

[53] National Laboratory for Applied Network Research. The Squid Internet Object Cache. `http://squid.nlanr.net`.

[54] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[55] 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.

[56] Armando Fox and Eric A. Brewer. Harvest, Yield, and Scalable Tolerant Systems. In *Proceedings HotOS-VII*, Rio Rico, Arizona, March 1999.

[57] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to Network and Client Variability via On-Demand Dynamic Distillation. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Cambridge MA, October 1996.

[58] Michael J. Franklin. Concurrency Control and Recovery. In *The Handbook of Computer Science and Engineering, A. Tucker, ed., CRC Press*, Boca Raton, 1997.

[59] Nicolas D. Georganas. Self-Similar ("Fractal") Traffic in ATM Networks. In *Proceedings of the 2nd International Workshop on Advanced Teleservices and High-Speed Communications Architectures (IWACA '94)*, pages 1–7, Heidelberg, Germany, September 1994.

[60] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, Amin M. Vahdat, and Thomas E. Anderson. GLUnix: A Global Layer Unix for a Network of Workstations. In *Software Practice and Experience*, volume 28:9, July 1998.

[61] 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, Colorado, USA, October 1999.

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

[63] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(3):73–170, June 1993.

[64] 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.

[65] J. N. Gray, R. A. Lorie, and G. F. Putzolu. Granularity of Locks in a Large Shared Database. In *Proceedings of the Conference on Very Large Data Bases*, Framingham, MA, USA, September 1995.

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

[67] Michael Greenwald and David Cheriton. The Synergy Between Non-Blocking Synchronization and Operating System Structure. In *Proceedings of the USENIX 2nd Symposium on Operating System Design and Implementation (OSDI '96)*, Seattle, WA, USA, October 1996.

[68] 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.

[69] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation (OSDI 2000)*, San Diego, California, USA, October 2000.

[70] Steven D. Gribble, Matt Welsh, Eric A. Brewer, and David Culler. The MultiSpace: an Evolutionary Platform for Infrastructural Services. In *Proceedings of the 1999 Usenix Annual Technical Conference*, Monterey, California, USA, Jun 1999.

[71] Steven D. Gribble, Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, , and B. Zhao. The Ninja Architecture for Robust Internet-Scale Systems and Services. *Computer Networks*, 2000. Special Issue on Pervasive Computing, To Appear, `http://www.cs.berkeley.edu/~gribble/papers/ninja.ps.gz`.

[72] Robert Grimm, Tom Anderson, Brian Bershad, and David Wetheral. A System Architecture for Pervasive Computing. In *Proceedings of the 9th ACM SIGOPS European Workshop*, Kolding, Denmark, September 2000.

[73] Andrew S. Grimshaw, William A. Wulf, James C. French, Alfred C. Weaver, and Paul F. Reynolds Jr. Legion: The Next Logical Step Toward a Nationwide Virtual Computer. Technical Report CS-94-21, University of Virginia, Department of Computer Science, Jun 1994.

[74] Joseph M. Hellerstein. Designing the Telegraph Storage Manager. In *Proceedings of the Eight International Workshop on High Performance Transaction Systems (HPTS '99)*, 1999.

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

[76] James C. Hu, Irfan Pyarali, and Douglas C. Schmidt. Applying the Proactor Pattern to High-Performance Web Servers. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems*, October 1998.

[77] Norman C. Hutchinson and Larry L. Peterson. The x-Kernel: an Architecture or Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1), January 1991.

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

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

[80] Yahoo! Inc. The Yahoo! Directory and Web Services. `http://www.yahoo.com`.

[81] Arun Iyengar, Jim Challenger, Daniel Dias, and Paul Dantzig. High-Performance Web Site Design Techniques. *IEEE Internet Computing*, 4(2), March 2000.

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

[83] 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.

[84] 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.

[85] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[86] 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.

[87] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the Self-Similar Nature of Ethernet Traffic (Extended Version). *IEEE/ACM Transactions on Networking*, 2(1), February 1994.

[88] Nikolai Likhanov, Boris Tsybakov, and Nicolas D. Georganas. Analysis of an ATM Buffer with Self-Similar ("Fractal") Input Traffic. In *Proceedings of IEEE INFOCOM '95*, Boston, MA, April 1995. IEEE.

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

[90] 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.

[91] Witold Litwin, Marie-Anne Neimat, and Donovan A. Schneider. LH* - A Scalable, Distributed Data Store. *ACM Transactions on Database Systems (TODS)*, 21(4):480–525, December 1996.

[92] Witold Litwin and Thomas Schwarz. A High-Availability Scalable Distributed Data Structure using Reed Solomon Codes. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Dallas, TX, 2000.

[93] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report 1346, University of Wisconsin-Madison Computer Sciences Technical Report, Apr 1997.

[94] Miron Livny and Mike Litzkow. Making workstations a friendly environment for batch jobs. In *The Third Workshop on Workstation Operating Systems*, Apr 1992.

[95] L. F. Mackert and G. M. Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 84–95, Washington, D.C., USA, May 1986.

[96] McKusic, Joy, Leffler, and Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3), 1984.

[97] Marshall Kirk McKusick and Gregory R. Ganger. Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. In *Proceedings of the 1999 Annual Usenix Technical Conference*, June 1999.

[98] Sun Microsystems. JavaSpaces White Paper. `http://java.sun.com/products/javaspaces/whitepapers/index.html`.

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

[100] Jeffrey C. Mogul. Operating Systems Support for Busy Internet Servers. In *Proceedings HotOS-V*, Orcas Island, Washington, May 1995.

[101] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaasho ek. The Click modular router. In *Proceedings of the 17th ACM Symposium on Operating System s Principles (SOSP '99)*, pages 217–231, Kiawah Island, South Carolina, December 1999.

[102] Myricom Corporation. Myrinet: A Gigabit Per Second Local Area Network. In *IEEE Micro*, February 1995.

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

[104] John K. Ousterhout. Why Threads Are A Bad Idea (for most purposes). Presentation given at the 1996 Usenix Annual Technical Conference, January 1996.

[105] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the 1999 Annual Usenix Technical Conference*, June 1999.

[106] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. In *Proceedings the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, LA, USA, February 1999.

[107] Abhay K. Parekh and Robert G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks—The Single Node Case. In *Proceedings of INFOCOM '92*, Florence, Italy, May 1992.

[108] Vern Paxson and Sally Floyd. Wide-area Traffic: the Failure of Poisson Modeling. In *ACM SIGCOMM '94 Conference on Communications Architectures, Protocols and Applications*, London, UK, August 1994.

[109] Apache Server Project. Project home page. `http://www.apache.org`.

[110] Calton Pu, Tito Autrey, and Andrew Black. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, USA, December 1995.

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

[112] 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.

[113] 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.

[114] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical Network Support for IP Traceback. In *Proceedings of the 2000 ACM SIGCOMM Conference*, Stockholm, Sweden, August 2000.

[115] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Race Detector for Multi-Threaded Programs. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, October 1997.

[116] Douglas C. Schmidt. Reactor: An Object-Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. In *Proceedings of the 6th USENIX C++ Technical Conference*, Cambridge, MA, April 1994.

[117] Selinger, Astrahan, Chamberlain, Lorie, and Price. Access Path Selection in a Relational Database Management System. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 1979.

[118] Margo I. Seltzer and Ozan Yigit. A New Hashing Package for UNIX. In *Proceedings of the USENIX Winter 1991 Technical Conference*, Dallas, TX, January 1991.

[119] Junehwa Song, Eric Levy, Arun Iyengar, and Daniel 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, April 2000.

[120] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: a Wide-Area Distributed Database System. *The VLDB Journal*, 5(1):48–63, January 1996.

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

[122] Sun Microsystems Inc. Java Native Interface Specification. `http://java.sun.com/products/jdk/1.2/docs-guide-jni/index.html`.

[123] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The PVM Concurrent Computing System: Evolution, Experiences, and Trends. *Parallel Computing*, 20(4):531–545, April 1994.

[124] Cisco Systems. Local director. `http://www.cisco.com/warp/public/751/lodir/index.html`.

[125] Computer Emergency Response Team. Advisory CA-2000-01 Denial-of-Service Developments. `http://www.cert.org/advisories/CA-2000-01.html`, January 2000.

[126] 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.

[127] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, USA, Decemeber 1995.

[128] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *19th Annual International Symposium on Computer Architecture*, 1992.

[129] David W. Walker. The Design of a Standard Message-Passing Interface for Distributed Memory Concurrent Computers. *Parallel Computing*, 20(4):657–673, April 1994.

[130] Deborah A. Wallach, Dawson R. Engler, and M. Frans Kaashoek. ASHs: Application-specific Handlers for High-Performance Messaging. In *Proceedings of the ACM SIGCOMM '96 Conference: Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 40–52, Stanford, CA, August 1996.

[131] Helen J. Wang, Bhaskaran Raman, Chen nee Chuah, and et al. ICEBERG: An Internet-core Network Architecture for Integrated Communications. In *IEEE Personal Communications*, August 2000.

[132] Matt Welsh and David Culler. Jaguar: Enabling Efficient Communication and I/O in Java. *Concurrency: Practice and Experience*, 2000. Special Issue on Java for High-Performance Network Computing, To appear, `http://www.cs.berkeley.edu/~mdw/papers/jaguar-journal.ps.gz`.

[133] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. TSpaces. *IBM Systems Journal*, 37(3), April 1998.

Appendix A

Source Code: SignServer

This appendix lists the source code to the various flavors of the `SignServer` programs described in Section 4.4, as well as the source code for the benchmark client.

A.1 Common Utility Functions

```
/*
 * Author: Steve Gribble <gribble@cs.berkeley.edu>
 * Inception Date: August 30th, 2000
 * File: HashSigner.java
 */

package ninja2.personal.gribble.ex;

import java.io.*;
import java.lang.*;
import java.util.*;
import java.security.*;
import ninja2.personal.gribble.ex.*;

/**
 * This class allows you to sign things using a hard-coded
 * keyed hash.
 *
 * @author Steve Gribble
 */
public class HashSigner {

   public static byte[] hashkey1 = {
```

```
    -84, -19, 0, 5, 115, 114, 0, 34, 115, 117, 110, 46
  };

  public static byte[] hashkey2 = {
    21, 106, 12, 77, 77, 35, -65, -18, 32, -111, 2, 21
  };

 MessageDigest md = null;

 public HashSigner() {
   try {
     md = MessageDigest.getInstance("MD5");
   } catch (NoSuchAlgorithmException nsae) {
     nsae.printStackTrace();
     System.exit(-1);
   }
 }

 public byte[] sign1(byte[] signme) {
   byte[] retB = null;

   md.reset();
   md.update(hashkey1);
   md.update(signme);
   retB = md.digest();

   return retB;
 }

 public byte[] sign2(byte[] signme) {
   byte[] retB = null;

   md.reset();
   md.update(hashkey2);
   md.update(signme);
   retB = md.digest();

   return retB;
 }

 // a little performance test
 public static void main(String args[]) {

   byte[] testB = new byte[5000];
```

```
    for (int i=0; i<5000; i++)
      testB[i] = (byte) (i%256);

    HashSigner hs = new HashSigner();

    System.out.println("doing loop");
    long before;
    before = System.currentTimeMillis();
    before = System.currentTimeMillis();
    for (int i=0; i<10000; i++) {
      byte[] hashr = hs.sign1(testB);
    }
    long after = System.currentTimeMillis();

    long diff = (after - before) / 10;
    System.out.println("hash: " + diff + " us");
  }
}


/*
 * Author: Steve Gribble <gribble@cs.berkeley.edu>
 * Inception Date: August 30th, 2000
 * File: Serializer.java
 */

package ninja2.personal.gribble.ex;

import java.io.*;
import java.lang.*;
import java.util.*;


/**
 * This class exists only to define some convenience routines for
 * casting serializable objects to byte arrays, and vice versa.
 *
 * @author Steve Gribble
 */
public class Serializer {

  /**
   * Given a byte array, create an object out of it through the magic
   * of Java deserialization.
```

```
   *
   * @param     arr the byte array containing a serialized object
   * @return    the deserialized object, if all is well
   * @exception IOException if something bad happens with the streams
   * @exception ClassNotFoundException if it isn't a serialized object
   *            after all
   */
  public static Object ByteArrToObject(byte[] arr) throws IOException,
    ClassNotFoundException {

    ByteArrayInputStream bi = new ByteArrayInputStream(arr);
    ObjectInputStream is =
      new ObjectInputStream(new BufferedInputStream(bi));

    Object o = is.readObject();
    is.close();
    bi.close();
    return o;
  }

  /**
   * Given a byte array, create an object out of it through the magic
   * of Java deserialization.
   *
   * @param     o the (serializable) object to be serialized
   * @return    a bytearray containing the serialized object
   * @exception IOException if something bad happens with the streams,
   *            or if the object isn't serializable after all
   */
  public static byte[] ObjectToByteArr(Object o) throws IOException {
    ByteArrayOutputStream bs = new ByteArrayOutputStream();
    ObjectOutputStream os =
      new ObjectOutputStream(new BufferedOutputStream(bs));
    os.writeObject(o);
    os.flush();
    byte[] rv = bs.toByteArray();
    os.close();
    return rv;
  }
}
```

A.2 Threaded SignServer

```
/*
 * Author: Steve Gribble <gribble@cs.berkeley.edu>
 * Inception Date: August 30, 2000
 * File: SignServer.java
 *
 */

package ninja2.personal.gribble.ex;

import ninja2.core.io_core.interfaces.*;
import ninja2.core.io_core.interfaces.network.*;
import ninja2.core.io_core.tcp_network.*;
import ninja2.core.io_core.fs_disk.*;
import ninja2.core.io_core.core.*;
import ninja2.core.io_core.thread_pool.*;
import ninja2.core.io_core.util.*;
import ninja2.personal.gribble.ex.*;
import java.io.*;
import java.net.*;
import java.util.*;

/**
 * The SignServer receives a packet from the network, computes two
 * keyed MD5 hashes of the packet (using two different keys), and
 * then returns the two hashes to the client.  This uses no design
 * pattern, but rather uses the thread-per-task model.
 *
 * @author   Steven Gribble
 */
public class SignServer implements ThreadPool_TaskIF, UpcallHandlerIF {
  private ThreadPool tp = null;
  private Hashtable  writers = null;
  private NetworkIF  nif = null;
  private MemAllocatorIF alloc = null;
  private NinjaLinkedList nll = new NinjaLinkedList();
  private SinkIF disk_sink = null;
  private MemRegionIF logentry = null;

  // create the NetworkIF to receive tasks from network clients
  public SignServer(String self, int num_threads) {
    ThreadPool vizpools = new ThreadPool(15,15,15);
    for (int i=0; i<num_threads+2; i++) {
      nll.add_to_tail(new HashSigner());
    }
```

```
  writers = new java.util.Hashtable();
  alloc = (MemAllocatorIF) new ByteArrayMemAllocator();
  tp = new ThreadPool(num_threads, num_threads, num_threads);
  try {
    nif = (NetworkIF) new TCPNetworkVizier(vizpools, self);
  } catch (UnknownHostException uhe) {
    System.err.println("Unknown host in server address " + self);
    System.exit(1);
  } catch (NumberFormatException nfe) {
    System.err.println("Badly formatted port num in server address " + self);
    System.exit(1);
  }

  // open up the disk sink
  FSDiskSegmentVizier fdsv = new FSDiskSegmentVizier(vizpools);
  try {
    disk_sink = (SinkIF) fdsv.openForWrite("/var3/gribble/bigfile", 0);
    logentry = alloc.allocateRegion(128);
  } catch (Exception e) {
    e.printStackTrace();
    System.exit(-1);
  }

  // register ourselves as the packet arrival handler
  nif.register_upcall(this);
  System.out.println("SignServer: ready to receive.");
}

// here's where packet arrives come in - this is the UpcallHandlerIF
public void enqueue_many(QueueElementIF[] els) {
  for (int i=0; i<els.length; i++)
    enqueue(els[i]);
}
public void enqueue(QueueElementIF el) {
  // dispatch the arrivals to the thread pool
  tp.enqueue_task(this, el);
}

// the thread pool dispatches arrive here
private int num_done = 0;
public void go_for_it(Object task_argument) {
  NetworkReadFinished nrf = (NetworkReadFinished) task_argument;
  NetworkWriterIF nw = null;
```

```
HashSigner signer = null;
synchronized(nll) {
  signer = (HashSigner) nll.remove_tail();
}

if (++num_done > 1000) {
  System.out.println("tick");
  num_done = 0;
}

// try to read something from a disk
Queue compQ = new Queue();
GenericSinkElement gse2 = new GenericSinkElement(logentry,
                                                 logentry.getSize(),
                                                 0, compQ);
try {
  disk_sink.sync_enqueue(gse2, null);
} catch (SinkDeadException sde) {
  sde.printStackTrace();
  System.exit(-1);
}
compQ.blocking_dequeue(0);

// grab the appropriate network sink to send the reply to, or
// create it if it doesn't already exist
nw = (NetworkWriterIF) writers.get(nrf.peer);
if (nw == null) {
  synchronized(writers) {
    nw = (NetworkWriterIF) writers.get(nrf.peer);
    if (nw == null) {
      // doesn't exist, so create it
      try {
        System.out.println("Connecting to " + nrf.peer);
        nw = nif.openForWrite(nrf.peer);
        writers.put(nrf.peer, nw);
      } catch (IOException ioe) {
        ioe.printStackTrace();
        System.exit(-1);
      }
    }
  }
}

// hash1
```

```
    byte[] sign1 = signer.sign1(nrf.region.fetchEntireByteArray());
    // hash2
    byte[] sign2 = signer.sign2(nrf.region.fetchEntireByteArray());
    MemRegionIF mr = null;
    try {
      mr = alloc.allocateRegion(sign1.length + sign2.length);
      mr.copyByteArrayInto(sign1, 0, 0, sign1.length);
      mr.copyByteArrayInto(sign2, 0, sign1.length, sign2.length);
    } catch (OutOfMemoryError ome) {
      ome.printStackTrace();
      System.exit(-1);
    }

    synchronized(nll) {
      nll.add_to_tail(signer);
    }

    // send the response here
    GenericSinkElement gse = new GenericSinkElement(
      mr, mr.getSize(), 0, null
      );
    try {
      if (!nw.async_enqueue(gse, null)) {
        System.err.println("enqueue failed");
        System.exit(1);
      }
    } catch (SinkDeadException sde) {
      sde.printStackTrace();
      System.exit(1);
    }
  }

  // this method handles the command line arguments passed in to the
  // SignServer.
  public static void main(String args[]) {
    if (args.length != 2) {
      System.err.println("usage: SignServer self num_threads");
      System.exit(1);
    }
    int numt = 0;
    try {
      numt = Integer.parseInt(args[1]);
    } catch (NumberFormatException nfe) {
      nfe.printStackTrace();
```

```
      System.exit(1);
    }
    SignServer as = new SignServer(args[0], numt);
  }
}
```

A.3 Wrapped SignServer

```
/*
 * Author: Steve Gribble <gribble@cs.berkeley.edu>
 * Inception Date: August 30, 2000
 * File: W_SignServer.java
 */
package ninja2.personal.gribble.ex;

import ninja2.core.io_core.interfaces.*;
import ninja2.core.io_core.interfaces.network.*;
import ninja2.core.io_core.interfaces.disk.*;
import ninja2.core.io_core.tcp_network.*;
import ninja2.core.io_core.fs_disk.*;
import ninja2.core.io_core.core.*;
import ninja2.core.io_core.thread_pool.*;
import ninja2.core.io_core.util.*;
import ninja2.personal.gribble.ex.*;
import java.io.*;
import java.net.*;
import java.util.*;

/**
 * The WP_SignServer receives a packet from the network, computes two
 * keyed MD5 hashes of the packet (using two different keys), and
 * then returns the two hashes to the client.  It, however, uses
 * the WRAP design pattern to condition itself against load.  It uses
 * a single thread across a thread boundary.
 *
 * @author   Steven Gribble
 */
public class W_SignServer {
  private ThreadPool tp = null;
  private Hashtable  writers = null;
  private NetworkIF  nif = null;
  private Queue      task_queue = null;
  private MemAllocatorIF alloc = null;
  private SinkIF disk_sink = null;
```

```
private GenericSinkElement logentry = null;
private NinjaLinkedList nll_gse = new NinjaLinkedList();
private NinjaLinkedList nll_sink = new NinjaLinkedList();

private HashSigner signer = new HashSigner();

// create the NetworkIF to receive tasks from network clients
public W_SignServer(String self) {
  writers = new java.util.Hashtable();
  alloc = (MemAllocatorIF) new ByteArrayMemAllocator();
  task_queue = new Queue();

  tp = new ThreadPool(15, 15, 15);   // just for handling TCP connections
  try {
    nif = (NetworkIF) new TCPNetworkVizier(tp, self);
  } catch (UnknownHostException uhe) {
    System.err.println("Unknown host in server address " + self);
    System.exit(1);
  } catch (NumberFormatException nfe) {
    System.err.println("Badly formatted port num in server address " + self);
    System.exit(1);
  }

  // register the queue as the completion handler - this is WRAP
  nif.register_upcall(task_queue);
  System.out.println("W_SignServer: ready to receive.");

  // open up the disk sink
  FSDiskSegmentVizier fdsv = new FSDiskSegmentVizier(tp);
  try {
    disk_sink = (SinkIF) fdsv.openForWrite("/var3/gribble/bigfile", 0);
    MemRegionIF mr = alloc.allocateRegion(128);
    logentry = new GenericSinkElement(mr, mr.getSize(), 0, task_queue);
  } catch (Exception e) {
    e.printStackTrace();
    System.exit(-1);
  }
}

// here's where the wrap thread waits for arrivals
public void go_for_it() {
  int num_done = 0;

  while(true) {
```

```
QueueElementIF els[] = task_queue.blocking_dequeue(0);
if (els != null) {
  for (int i=0; i<els.length; i++) {
    if (els[i] instanceof DiskSegmentWriteDrainedEvent) {
      process_disk_write((DiskSegmentWriteDrainedEvent) els[i]);
    } else {
      NetworkReadFinished nrf = (NetworkReadFinished) els[i];
      NetworkWriterIF nw = null;

      if (++num_done > 1000) {
        System.out.println("tick");
        num_done = 0;
      }
      // grab the appropriate network sink to send the reply to, or
      // create it if it doesn't already exist
      nw = (NetworkWriterIF) writers.get(nrf.peer);
      if (nw == null) {
        // doesn't exist, so create it
        try {
          System.out.println("Connecting to " + nrf.peer);
          nw = nif.openForWrite(nrf.peer);
          writers.put(nrf.peer, nw);
        } catch (IOException ioe) {
          ioe.printStackTrace();
          System.exit(-1);
        }
      }

      // hash1
      byte[] sign1 = signer.sign1(nrf.region.fetchEntireByteArray());
      // hash2
      byte[] sign2 = signer.sign2(nrf.region.fetchEntireByteArray());
      MemRegionIF mr = null;
      try {
        mr = alloc.allocateRegion(sign1.length + sign2.length);
        mr.copyByteArrayInto(sign1, 0, 0, sign1.length);
        mr.copyByteArrayInto(sign2, 0, sign1.length, sign2.length);
      } catch (OutOfMemoryError ome) {
        ome.printStackTrace();
        System.exit(-1);
      }

      // line up the response here
      GenericSinkElement gse = new GenericSinkElement(
```

```
            mr, mr.getSize(), 0, null
            );

          nll_sink.add_to_tail(nw);
          nll_gse.add_to_tail(gse);

          // do the disk write here
          try {
            disk_sink.sync_enqueue(logentry, null);
          } catch (SinkDeadException sde) {
            sde.printStackTrace();
            System.exit(-1);
          }
        }
      }
    }
  }
}

private void process_disk_write(DiskSegmentWriteDrainedEvent dwde) {
  SinkIF nw = (SinkIF) nll_sink.remove_head();
  GenericSinkElement gse = (GenericSinkElement) nll_gse.remove_head();
  try {
    if (!nw.async_enqueue(gse, null)) {
      System.err.println("enqueue failed");
      System.exit(1);
    }
  } catch (SinkDeadException sde) {
    sde.printStackTrace();
    System.exit(1);
  }
}

// this method handles the command line arguments passed in to the
// W_SignServer.
public static void main(String args[]) {
  if (args.length != 1) {
    System.err.println("usage: W_SignServer self");
    System.exit(1);
  }
  W_SignServer as = new W_SignServer(args[0]);
  as.go_for_it();
}
}
```

A.4 Wrapped, Pipelined SignServer

```
/*
 * Author: Steve Gribble <gribble@cs.berkeley.edu>
 * Inception Date: August 30, 2000
 * File: WR_SignServer.java
 */

package ninja2.personal.gribble.ex;

import ninja2.core.io_core.interfaces.*;
import ninja2.core.io_core.interfaces.network.*;
import ninja2.core.io_core.interfaces.disk.*;
import ninja2.core.io_core.tcp_network.*;
import ninja2.core.io_core.fs_disk.*;
import ninja2.core.io_core.core.*;
import ninja2.core.io_core.thread_pool.*;
import ninja2.core.io_core.util.*;
import ninja2.personal.gribble.ex.*;
import java.io.*;
import java.net.*;
import java.util.*;

/**
 * The WP_SignServer receives a packet from the network, computes two
 * keyed MD5 hashes of the packet (using two different keys), and
 * then returns the two hashes to the client.  This server uses the WRAP
 * pattern to condition against load, and the PIPELINE pattern to
 * achieve functional parallelism across CPUs in a multiprocessor.
 *
 * @author   Steven Gribble
 */
public class WP_SignServer implements java.lang.Runnable {
  private ThreadPool tp = null;
  private Hashtable  writers = null;
  private NetworkIF  nif = null;
  private Queue      task_queue1 = null, task_queue2 = null;
  private MemAllocatorIF alloc = null;
  private HashSigner signer1 = new HashSigner();
  private HashSigner signer2 = new HashSigner();
  private SinkIF disk_sink = null;
  private GenericSinkElement logentry = null;
```

```
private NinjaLinkedList nll_gse = new NinjaLinkedList();
private NinjaLinkedList nll_sink = new NinjaLinkedList();

private static class Continuer implements QueueElementIF {
  public NetworkReadFinished nrf;
  public byte[] sign1;
}

// create the NetworkIF to receive tasks from network clients
public WP_SignServer(String self) {
  writers = new java.util.Hashtable();
  alloc = (MemAllocatorIF) new ByteArrayMemAllocator();
  task_queue1 = new Queue();
  task_queue2 = new Queue();

  tp = new ThreadPool(15, 15, 15);   // just for handling TCP connections
  try {
    nif = (NetworkIF) new TCPNetworkVizier(tp, self);
  } catch (UnknownHostException uhe) {
    System.err.println("Unknown host in server address " + self);
    System.exit(1);
  } catch (NumberFormatException nfe) {
    System.err.println("Badly formatted port num in server address " + self);
    System.exit(1);
  }

  // register the queue as the completion handler - this is WRAP
  nif.register_upcall(task_queue1);
  System.out.println("WP_SignServer: ready to receive.");

  // open up the disk sink
  FSDiskSegmentVizier fdsv = new FSDiskSegmentVizier(tp);
  try {
    disk_sink = (SinkIF) fdsv.openForWrite("/var3/gribble/bigfile", 0);
    MemRegionIF mr = alloc.allocateRegion(128);
    logentry = new GenericSinkElement(mr, mr.getSize(), 0, task_queue2);
  } catch (Exception e) {
    e.printStackTrace();
    System.exit(-1);
  }
}

// here's where the two stage threads are created
public void go_for_it() {
```

```
  Thread t1 = new Thread(this, "1");
  Thread t2 = new Thread(this, "2");
  t1.start();
  t2.start();
}
public void run() {
  String myName = Thread.currentThread().getName();
  if (myName.equals("1"))
    q1_go();
  else
    q2_go();
}

// HERE IS WHERE THE FIRST STAGE WRAPPED THREAD STARTS
public void q1_go() {
  int num_done = 0;

  while(true) {
    QueueElementIF els[] = task_queue1.blocking_dequeue(0);
    if (els != null) {
      for (int i=0; i<els.length; i++) {
        Continuer cnr = new Continuer();
        cnr.nrf = (NetworkReadFinished) els[i];

        if (++num_done > 1000) {
          System.out.println("tick");
          num_done = 0;
        }

        // hash 1 here
        cnr.sign1 = signer1.sign1(cnr.nrf.region.fetchEntireByteArray());
        task_queue2.enqueue(cnr);
      }
    }
  }
}

// HERE IS WHERE THE SECOND STAGE WRAPPED THREAD STARTS
public void q2_go() {
  while(true) {
    QueueElementIF els[] = task_queue2.blocking_dequeue(0);
    if (els != null) {
      for (int i=0; i<els.length; i++) {
        if (els[i] instanceof DiskSegmentWriteDrainedEvent) {
```

```
      process_disk_write((DiskSegmentWriteDrainedEvent) els[i]);
    } else {
      Continuer cnr = (Continuer) els[i];
      NetworkReadFinished nrf = cnr.nrf;
      NetworkWriterIF nw = null;

      // grab the appropriate network sink to send the reply to, or
      // create it if it doesn't already exist
      nw = (NetworkWriterIF) writers.get(nrf.peer);
      if (nw == null) {
        // doesn't exist, so create it
        try {
          System.out.println("Connecting to " + nrf.peer);
          nw = nif.openForWrite(nrf.peer);
          writers.put(nrf.peer, nw);
        } catch (IOException ioe) {
          ioe.printStackTrace();
          System.exit(-1);
        }
      }

      // hash2 here
      byte[] sign2 = signer2.sign2(nrf.region.fetchEntireByteArray());
      MemRegionIF mr = null;
      try {
        mr = alloc.allocateRegion(cnr.sign1.length + sign2.length);
        mr.copyByteArrayInto(cnr.sign1, 0, 0, cnr.sign1.length);
        mr.copyByteArrayInto(sign2, 0, cnr.sign1.length, sign2.length);
      } catch (OutOfMemoryError ome) {
        ome.printStackTrace();
        System.exit(-1);
      }

      // line up the response here
      GenericSinkElement gse = new GenericSinkElement(
        mr, mr.getSize(), 0, null
        );

      nll_sink.add_to_tail(nw);
      nll_gse.add_to_tail(gse);

      // do the disk write here
      try {
        disk_sink.sync_enqueue(logentry, null);
```

```
          } catch (SinkDeadException sde) {
            sde.printStackTrace();
            System.exit(-1);
          }
        }
      }
    }
  }
}

private void process_disk_write(DiskSegmentWriteDrainedEvent dwde) {
  SinkIF nw = (SinkIF) nll_sink.remove_head();
  GenericSinkElement gse = (GenericSinkElement) nll_gse.remove_head();

  try {
    if (!nw.async_enqueue(gse, null)) {
      System.err.println("enqueue failed");
      System.exit(1);
    }
  } catch (SinkDeadException sde) {
    sde.printStackTrace();
    System.exit(1);
  }
}

// this method handles the command line arguments passed in to the
// WP_SignServer.
public static void main(String args[]) {
  if (args.length != 1) {
    System.err.println("usage: WP_SignServer self");
    System.exit(1);
  }
  WP_SignServer as = new WP_SignServer(args[0]);
  as.go_for_it();
}
}
```

A.5 SignServer Client

```
/*
 * Author: Steve Gribble <gribble@cs.berkeley.edu>
 * Inception Date: August 30th, 2000
 * File: SignClient.java
```

```
 */

package ninja2.personal.gribble.ex;

import ninja2.core.io_core.interfaces.*;
import ninja2.core.io_core.interfaces.network.*;
import ninja2.core.io_core.tcp_network.*;
import ninja2.core.io_core.core.*;
import ninja2.core.io_core.thread_pool.*;
import ninja2.core.io_core.util.*;

import java.io.*;
import java.net.*;
import java.util.*;
import java.lang.*;

/**
 * The SignClient forges connections to one or more SignServers,
 * and starts pipelines of requests to those servers.  The
 * total throughput across all servers is measured.  In other
 * words, this client embodies the REPLICATE pattern if more than
 * one target server is specified on the command line; load-balancing
 * composition is used because of the nature of the closed-loop pipelines.
 *
 * @author   Steven Gribble
 */
public class SignClient {

  private QueueIF compq = null;
  private SinkIF  peer_sinks[] = null;
  private String  peers[] = null;
  private Hashtable sink_index = new java.util.Hashtable();
  private int     winlen = 0;
  private int     packetsize = 5000;
  private int     measuresize = 1500;
  private int     latsize = 1247;
  private GenericSinkElement gse, gse_l;

  private boolean gotnor = false;
  private byte norbyte = 0;

  // forge connections to all the SignServers
  public SignClient(String self, int winlen, String[] peers) {
    this.peers = peers;
```

```
ThreadPool tp = new ThreadPool(15,15,15);
this.winlen = winlen;
peer_sinks = new SinkIF[peers.length];
NetworkIF nif = null;
try {
  nif = (NetworkIF) new TCPNetworkVizier(tp, self);
} catch (UnknownHostException uhe) {
  System.err.println("Unknown host in host string " + self);
  System.exit(1);
} catch (NumberFormatException nfe) {
  System.err.println("Badly formatted port num in host string " + self);
  System.exit(1);
}
compq = new Queue();
nif.register_upcall(compq);
for (int i=0; i<peers.length; i++) {
  System.out.println("Connecting to " + peers[i]);
  while (peer_sinks[i] == null) {
    try {
      peer_sinks[i] = nif.openForWrite(peers[i]);
      sink_index.put(peers[i], peer_sinks[i]);
    } catch (IOException ioe) {
    }
    try {
      Thread.currentThread().sleep(200);
    } catch (InterruptedException ie) {
    }
  }
}

// warm up pipelines to the servers
System.out.println("starting volley");
ByteArrayMemAllocator bama = new ByteArrayMemAllocator();
MemRegionIF volley = bama.allocateRegion(packetsize);
gse = new GenericSinkElement(volley, packetsize, 0, null);
for (int i=0; i<packetsize; i++)
  volley.putByte((byte) 0, i);
MemRegionIF lat_sampler = bama.allocateRegion(packetsize);
gse_l = new GenericSinkElement(lat_sampler, packetsize, 0, null);
for (int i=0; i<packetsize; i++)
  lat_sampler.putByte((byte) 1, i);
for (int i=0; i<winlen; i++) {
  for (int j=0; j<peers.length; j++) {
    try {
```

```
        if (!peer_sinks[j].async_enqueue(gse, null)) {
          System.err.println("enqueue failed");
          System.exit(1);
        }
      } catch (SinkDeadException sde) {
        sde.printStackTrace();
        System.exit(1);
      }
    }
  }

  // enter reactive closed-loop phase
  System.out.println("entering closed-loop");
  int comps = 0;
  long before, after;
  long lat_req, lat_rep;
  before = System.currentTimeMillis();
  while (true) {
    QueueElementIF els[] = compq.blocking_dequeue(0);
    int ellen = els.length;
    for (int i=0; i<ellen; i++) {
      NetworkReadFinished nrf = (NetworkReadFinished) els[i];
      handle_nrf(nrf);
      comps++;
    }
    if (comps >= measuresize) {
      after = System.currentTimeMillis();
      printTime(before, after, comps);
      comps = 0;
      before = after;
    }
  }
}

private static void printTime(long long1, long long3, int int5) {
  long long6 = long3 - long1;
  double double8 = (double) int5 / (double) long6;
  double double10 = double8 * 1000.0;

  System.out.println( int5 + " iterations in " + long6 +
                      " milliseconds = " + double10
                      + " iterations per second" );
}
```

```
private long lat_before = 0;
private long lat_tries = 0;
private void handle_nrf(NetworkReadFinished nrf) {
  GenericSinkElement outse;
  lat_tries++;

  if ((lat_tries >= latsize) && (lat_before == 0)) {
    outse = gse_l;
    lat_tries = 0;
    lat_before = System.currentTimeMillis();
  } else
    outse = gse;

  if (gotnor == false) {
    norbyte = nrf.region.getByte(0);
    gotnor = true;
  }
  if (nrf.region.getByte(0) != norbyte) {
    long curt = System.currentTimeMillis();
    // is the magic sampler byte
    if (lat_before != 0) {
      System.out.println("Latency sample:  " +
                         (curt-lat_before) + " ms");
    }
    lat_before = 0;
  }

  // close the pipeline loop- send another packet
  SinkIF outSink = (SinkIF) sink_index.get(nrf.peer);
  if (outSink == null) {
    System.err.println("Unknown peer: " + nrf.peer);
    System.exit(-1);
  }
  try {
    if (!outSink.async_enqueue(outse, null)) {
      System.err.println("enqueue failed");
      System.exit(1);
    }
  } catch (SinkDeadException sde) {
    sde.printStackTrace();
  }
}

public static void main(String args[]) {
```

```
    if (args.length < 3) {
      System.err.println("usage: SignClient self windowlen peer [peer*]");
      System.exit(1);
    }
    int pipeline = 0;
    try {
      pipeline = Integer.parseInt(args[1]);
    } catch (NumberFormatException nfe) {
      nfe.printStackTrace();
      System.exit(1);
    }

    String[] peers = new String[args.length-2];
    for (int i=2; i<args.length; i++)
      peers[i-2] = args[i];

    SignClient as = new SignClient(args[0], pipeline, peers);
  }
}
```