Scalable and manageable storage systems

Khalil S. Amiri

December, 2000

CMU-CS-00-178

Department of Electrical and Computer Engineering

Carnegie Mellon University

Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements

for the degree of Doctor of Philosophy

Thesis Committee:

Garth A. Gibson, Chair

Gregory R. Ganger

M. Satyanarayanan

Daniel P. Siewiorek

John Wilkes, HP Laboratories

This research is sponsored by DARPA/ITO, through DARPA Order D306, and issued by Indian Head Division under contract N00174-96-0002. Additional support was provided by generous contributions of the member companies of the Parallel Data Consortium including 3COM Corporation, Compaq, Data General, EMC, Hewlett-Packard, IBM, Intel, LSI Logic, Novell, Inc., Seagate Technology, StorageTek, Quantum Corporation and Wind River Systems. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of any supporting organization or the US Government.

Keywords: storage, network-attached storage, storage management, distributed disk arrays, concurrency control, recovery, function partitioning.

Abstract

Emerging applications such as data warehousing, multimedia content distribution, electronic commerce and medical and satellite databases have substantial storage requirements that are growing at 3X to 5X per year. Such applications require scalable, highly-available and cost-effective storage systems. Traditional storage systems rely on a central controller (file server, disk array controller) to access storage and copy data between storage devices and clients which limits their scalability.

This dissertation describes an architecture, network-attached secure disks (NASD), that eliminates the single controller bottleneck allowing throughput and bandwidth of an array to scale with increasing capacity up to the largest sizes desired in practice. NASD enables direct access from client to shared storage devices, allowing aggregate bandwidth to scale with the number of nodes.

In a shared storage system, each client acts as its own storage (RAID) controller, performing all the functions required to manage redundancy and access its data. As a result, multiple controllers can be accessing and managing shared storage devices concurrently. Without proper provisions, this concurrency can corrupt redundancy codes and cause hosts to read incorrect data. This dissertation proposes a transactional approach to ensure correctness in highly concurrent storage device arrays. It proposes distributed device-based protocols that exploit trends towards increased device intelligence to ensure correctness while scaling well with system size.

Emerging network-attached storage arrays consist of storage devices with excess cycles in their on-disk controllers, which can be used to execute filesystem function traditionally executed on the host. Programmable storage devices increase the flexibility in partitioning filesystem function between clients and storage devices. However, the heterogeneity in resource availability among servers, clients and network links causes optimal function partitioning to change across sites and with time. This dissertation proposes an automatic approach which allows function partitioning to be changed and optimized at run-time by relying only on the black-box monitoring of functional components and of resource availability in the storage system.

This dissertation is dedicated to my mother, *Aicha*, and to my late father, *Sadok*.

Acknowledgements

The writing of a dissertation can be a daunting and tasking experience. Fortunately, I was lucky enough to have enjoyed the support of a large number of colleagues, family members, faculty and friends that have made this process almost enjoyable. First and foremost, however, I am grateful to God for blessing me with a wonderful family and for giving me the opportunity and ability to complete my doctorate.

I would like to thank my advisor, Professor Garth Gibson, for his advice and encouragement during my time at Carnegie Mellon. I am also grateful to the members of my thesis committee, Gregory Ganger, Dan Siewiorek, Satya and John Wilkes for agreeing to serve on the committee and for their valuable guidance and feedback. I am also indebted to John Wilkes for his mentoring and encouragement during my internships at HP labs.

My officemates, neighbors, and collaborators on the NASD project, Fay Chang, Howard Gobioff, Erik Riedel and David Rochberg were great colleagues and wonderful friends. I am also grateful to the members of the RAIDFrame and NASD research groups and to the members of the entire Parallel Data Laboratory, who always proved ready to engage in challenging discussions. In particular, I would like to thank Jeff Butler, Bill Courtright, Mark Holland, Eugene Feinberg, Paul Mazaitis, David Nagle, Berend Ozceri, Hugo Patterson, Ted Wong, and Jim Zelenka. In addition, I would like to thank LeAnn, Andy, Charles, Cheryl, Chris, Danner, John, Garth, Marc, Sean, and Steve.

I am grateful to Joan Digney for her careful proofreading of this document, for her valuable comments, and for her words of encouragement. I am also grateful to Patty for her friendship, support, assistance, and for running such a large laboratory so well. I am equally grateful to Karen for helping me with so many tasks and simply for being such a wonderful person.

During my time at Carnegie Mellon, I had several fruitful collaborations with colleagues within and outside the University. In particular, I would like to thank David Petrou, who worked with me on ABACUS, for always proving ready to engage in challenging discussions or to start any design or

implementation task we thought was useful. I would also like to thank Richard Golding who worked with me on the challenging scalability and fault-tolerance problems in storage clusters. Richard was always a source of good ideas, advice and encouragement, in addition to being great company. I am deeply grateful to the members of the Parallel Data Lab's Industrial Consortium for helping me understand the practical issues surrounding our research proposals. In particular, I am grateful to Dave Anderson and Chris Mallakapali of Seagate, to Elizabeth Borowsky then of HP labs, to Don Cameron of Intel, and to Satish Rege of Quantum.

I can not imagine my six years in graduate school without the support of many friends in the United States and overseas. I would like to thank in particular Adel, Ferdaws, Gaddour, Karim, Najed, Rita, Sandra, Sean, and Sofiane.

Last but not least, I am infinitely grateful to my mother, Aicha, and my late father, Sadok. They provided me with unconditional love, advice, and support. My late father and his father before him presented me with admirable role models of wisdom and perseverance. Despite our long-distance relationship, my brothers and sisters, Charfeddine, Amel, Ghazi, Hajer, Nabeel, and Ines overwhelmed me with more love and support throughout my years in college and graduate school than what I could have ever deserved. Many thanks go to my uncles and aunts and their respective families, for their unwavering support.

Khalil Amiri
Pittsburgh, Pennsylvania
December, 2000

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data sets stored on-line are growing at phenomenal rates, often doubling every year [Emanuel, 1997], and reaching several terabytes at typical e-commerce companies [Lycos, 1999]. Examples include repositories of satellite and medical images, data warehouses of business information, multimedia entertainment content, on-line catalogs, and attachment-rich email archives. The explosive growth of electronic commerce [News, 1998] is generating huge archives of business transaction records and customer shopping histories every week [EnStor, 1997]. It is also reported that NASA's new earth observing satellite will generate data sets up to three times as large as the size of the Library of Congress every year.

Given these rapid growth rates, organizations face the need to incrementally scale their storage systems as demand for their services grows and their data sets expand. For many companies, estimating the growth rate is not an easy task [Lycos, 1999]. This makes the need to incrementally scale systems in response to unpredictable demand a pressing concern in practice. Traditional storage systems rely on file servers to copy data between clients and storage devices. File servers, unlike network switches, are not efficient in moving data between clients and storage nodes because they interpose synchronous control functions in the middle of the data path. As a result, file servers have emerged as a severe scalability bottleneck in storage systems. Consequently, to deliver acceptable bandwidth to clients, file servers have to be custom built or be high-end machines imposing a high cost overhead. Expanding storage beyond a single file server's capacity is also costly because it requires acquiring a new server and because it requires system administrators to explicitly replicate files and balance load and capacity across the servers.

The importance of storage system performance and availability in practice leads to the employment of a plethora of manual and ad-hoc techniques to ensure high-availability and balanced load. Not surprisingly, the cost of storage management continues to be the leading component in the cost

of ownership of computer systems [Gibson and Wilkes, 1996]. The cost of storage management is estimated to be three to six times the initial initial purchase cost of storage [Golding et al., 1995]. This is distressing given that the purchase cost of the storage subsystem dominates the cost of the other components in the system, making up to 75% of the purchase cost of the entire computer system in many enterprise data centers.

This dissertation proposes a novel architecture and novel algorithms that enable storage systems to be more cost-effectively scalable. Furthermore, the dissertation proposes an approach to ensure automatic load balancing across storage system components. Together, the body of these solutions described in this dissertation promises to make storage systems more manageable and cost-effectively scalable.

1.1 The storage management problem

A storage system is a system of hardware and software components that provides a persistent repository for the storage of unstructured streams of bytes. A storage system typically includes the storage devices, such as magnetic disks —used for persistent storage— the storage controllers (processors) responsible for accessing and managing these devices, and the networks that connect the devices to the controllers.

The ideal storage system has good performance regardless of its size (i.e. its data can be stored and retrieved quickly), high availability (i.e. its data can be accessed despite partial faults in the components), and sufficient capacity (i.e. that the system can seamlessly expand to meet growth in user storage requirements). Storage systems today fall short of this ideal in all these aspects.

Currently, these shortcomings are addressed by manual management techniques, which are both costly and of limited effectiveness. They are typically performed by administrators who are rarely equipped to undertake such complex optimization and configuration decisions. Human expertise continues to be scarce and expensive, explaining the exorbitant costs associated with storage management.

1.1.1 The ideal storage system

The ideal storage system can be characterized by four major properties:

- Cost-effective scaling: The ratio of total system cost to system performance (e.g. throughput or bandwidth) should remain low as the system increases in capacity. That is, doubling the

number of components (cost) of the system should double its performance. This requires that system resources be effectively utilized. These resources are often unevenly distributed across the different nodes in the storage system.

- Availability: The storage system should continue to serve data even if a limited number of components fail.
- Flexible capacity through easy and incremental growth: To meet expansion needs, increasing the size should be a simple and almost entirely automatic task.
- Security: The storage system must enforce data privacy, integrity and allow users to define expressive access control policies.

1.1.2 The shortcomings of current systems

The storage systems that are most widely used in medium and high-end servers today are *redundant disk arrays*, multi-disk systems that use disk striping and parity codes to balance load across disks, provide increased bandwidth on large transfers, and ensure data availability despite disk failures [Patterson et al., 1988]. To scale the capacity of a storage system beyond the maximum capacity or performance of a single disk array, multiple disk arrays are used. Load and capacity are often balanced manually by moving volumes between the arrays. Expanding the system by a few disks sometimes requires the purchase of a new array, a major step increase in cost. This occurs when the capacity of the array has reached its maximum or when the load on the controller becomes too high. Management operations such as storage migration and reconstruction are either carried off-line or performed on-line by the central array controller, restricting the system's scalability.

A storage system includes clients, array controllers and (potentially programmable) storage devices. The resources available at these nodes vary across systems and with time. Balancing the load across the system components often requires rewriting filesystems and applications to adjust the partitioning of function between nodes (clients and servers) to take advantage of changes in technology and applications. Filesystems have traditionally dictated that all user applications execute on the client, thereby opting for a "data shipping" approach, where file blocks are shipped to the client and processed there. Recently, however, with the availability of excess cycles in storage devices and servers, research has demonstrated dramatic benefits from exploiting these cycles to perform filtering, aggregation and other application-specific processing on data streams. Function shipping

reduces the amount of data that has to be shipped back to the client and substantially improves performance.

Function shipping is not always the ideal choice, however. Devices and servers can be easily overloaded since their resources are limited compared to the aggregate client resources. Whether data or function shipping is more optimal depends on current load conditions, the workload mix and application characteristics. Currently, optimal function partitioning of function is the responsibility of the programmer and system administrator. This adds to the cost of storage management.

To battle increasing storage system management costs, storage systems should scale cost-effectively and balance load automatically without manual intervention.

This dissertation identifies and addresses three key technical challenges to making storage systems more cost-effectively scalable and manageable.

1.2 Dissertation research

The research in this dissertation consists of three relatively independent parts. The first part introduces a storage system architecture, network attached secure disks (NASD), that enables cost-effective bandwidth scaling and incremental capacity growth. NASD modifies the storage device interface to allow it to transfer data directly to end clients without copying data through a centralized storage controller or file server. The basic idea is to have clients cache the mapping from a high-level file name to an object on a NASD device. The client then uses this cached mapping to map a file access onto a NASD device access. Data is transferred between the client and the NASD device without a copy through the server. Using a scalable switched network to connect the clients and the devices, the storage system can be expanded and its performance scaled by attaching additional NASD devices and clients to the network. This architecture is described in Chapter 3.

The NASD architecture allows clients direct access to NASD devices. By striping across NASD devices, clients can achieve high bandwidths on large transfers. The function that implements this striping across multiple storage devices, and which generally implements the RAID protocols (which maintain redundancy and manage the block layout maps), must therefore be executed at the client so that data travels from source to sink directly without being copied through a store-and-forward node. A NASD system therefore contains multiple controllers (at multiple clients) which can be simultaneously accessing storage at the same time. These controllers must be coordinated so that races do not corrupt redundancy codes or cause clients to read incorrect data.

The second part of the dissertation, namely Chapter 4, presents an approach based on light-

weight transactions which allows storage controllers to be active concurrently. Specifically, multiple controllers can be accessing shared devices while management tasks (such as data migration or reconstruction) are ongoing at other controllers. The protocols distribute the work of ensuring concurrency control and recovery to the endpoints. As a result, they do not suffer from the single controller bottleneck of traditional arrays. Distributed protocols unfortunately suffer from increased implementation complexity and involve higher messaging overhead. Both are very undesirable in storage systems because they increase the chance of implementation bugs and limit performance. This part shows how complexity can be managed by breaking down the problem into subproblems and solving each subproblem separately. The proposed solution parallels previous work on database transaction theory. It relies on simple two-phased operations, called storage transactions, as a basic building block of the solution. This part also proposes an optimistic protocol based on timestamps derived from loosely synchronized clocks that exhibits good scalability, low latency and limited device-side state and complexity. This protocol is shown to work well for random access and contended workloads typical of clustered storage systems. Furthermore, it is shown to have robust performance across workloads and system parameters.

The third part of the dissertation tackles the problem of function partitioning in the context of data-intensive applications executing over distributed storage systems. Rapidly changing technologies cause a single storage system to be composed of multiple storage devices and clients with disparate levels of CPU and memory resources. The interconnection network is rarely a simple crossbar, and is usually quite heterogeneous. The bandwidth available between pairs of nodes depends on the physical link topology between the two nodes and also on the dynamic load on the entire network. In addition to hardware heterogeneity, applications also have characteristics that vary with input arguments and with time.

The research described in this part demonstrates that the performance of storage management and data-intensive applications can be improved significantly by adaptively partitioning their functions between storage servers and clients. Chapter 5 describes a programming system which allows applications to be composed of components that can be adaptively bound to the client or server at run-time. It proposes an approach wherein application components are observed as black-boxes that communicate with each other. An on-line performance model is used to dynamically decide to place and re-place components between cluster nodes. It also quantifies the benefits of adaptive function partitioning through several microbenchmarks.

The thesis of this dissertation can be summarized as follows:

1. *Parallel hosts attached to an array of shared storage devices via a switched network can achieve scalable bandwidth and a shared virtual storage abstraction.*

2. *Using smart devices for distributed concurrency control in storage clusters achieves good scalability while requiring limited device-side state.*

3. *Proper function placement is crucial to the performance of storage management and data-intensive applications and can be decided based on the black-box monitoring of application components.*

1.3 Dissertation road map

The rest of the dissertation is organized as follows. Chapter 2 provides background information useful for reading the rest of the dissertation. It discusses trends in the underlying hardware technologies and reviews the demands that emerging application place on storage systems. It also covers some background on redundant disk arrays. The later part of the chapter summarizes the basic theory of database transactions and summarizes the different concurrency control and recovery protocols employed by transactional storage systems.

Chapter 3 is devoted to the NASD storage architecture which enables cost-effective bandwidth scaling and incremental capacity growth. It reiterates the enabling trends, the changes required at the hosts and the storage device to enable direct transfer, and the proposed storage device interface. It also describes a prototype storage service that aggregates multiple NASD devices into a shared single virtual object space. This Chapter shows that this storage service can deliver scalable bandwidth to bandwidth hungry applications such as data mining.

Chapter 4 presents an approach that allows parallel storage controllers in a NASD system to be actively accessing and managing storage simultaneously. The approach is based on a modular design which uses lightweight transactions, called base storage transactions (BSTs), as a basic building block. The key property of BSTs is serializability. The chapter presents protocols that ensure serializability and recovery for BSTs with high scalability.

Chapter 5 tackles the problem of function partitioning in distributed storage systems. In particular, it shows that data-intensive applications can benefit substantially from the adaptive partitioning of their functions to avoid bottlenecked network links and overloaded nodes. This chapter describes the ABACUS prototype, which was designed and implemented to demonstrate the feasibility of adaptive function placement. ABACUS consists of a programming model that allows applications

to be composed of graphs of mobile component objects. A run-time system redirects method invocation between component objects regardless of their placement (at client or at server). ABACUS continuously collects measurements of object resource consumption and system load and invokes an on-line performance model to evaluate the benefit of alternative placement decisions and adapt accordingly. The later part of the chapter focuses on validating the programming model by describing a filesystem built on ABACUS and by measuring the benefits that the adaptive placement of function enables. Finally, Chapter 6 summarizes the conclusions of the thesis and ends with directions for future work.

Chapter 2

Background

This chapter presents background information useful for reading the rest of the dissertation. It starts by reviewing the trends that motivate the research in the rest of the dissertation. Section 2.1 summarizes the trends in the technologies driving the important storage system components, namely magnetic disks, processors, interconnects and memory. Section 2.2 surveys trends in application capacity and bandwidth requirements, which are growing rapidly.

The second part of this background chapter covers the necessary background for Chapter 4. The solutions proposed in Chapter 4 specialize database transaction theory to storage semantics to implement a provably correct and highly concurrent parallel disk array. Sections 2.3 contains a quick refresher on redundant disk arrays. Section 2.4 reviews database transactions, the traditional technique applied to building concurrent and fault-tolerant distributed and cluster systems.

2.1 Trends in technology

Storage systems contain all the components that go into larger computer systems. They rely on magnetic disks, DRAM, processors – both for on-disk controllers and array controllers – and interconnects to attach the disks to the array controllers and the array controllers to the hosts. Consequently, significant changes in the technologies driving the evolution of these components influence the design and architecture of storage systems.

This section presents both background on how these components function, as well as recent trends in their cost and performance.

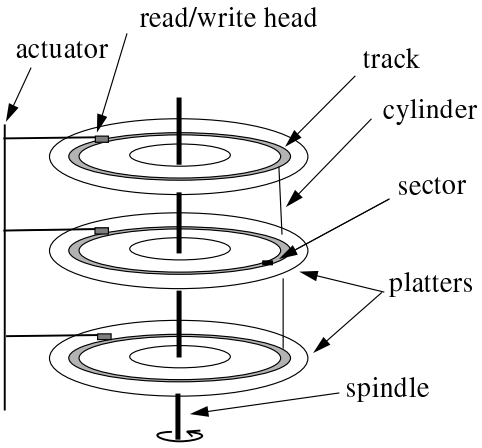


Figure 2.1: The mechanisms in a magnetic disk. A magnetic disk comprises several platters attached to a rotating spindle. Storage on the platters is organized into concentric tracks which increase in circumference and capacity the farther they are from the spindle. Read/write heads are attached to a disk arm which is moved by an actuator to the proper track. Once the head is positioned over the proper track, the disk waits for the target sector to pass under the head. Data is transferred to or from the platter as the sector passes under the head. Only one head on one track is active at a time because the heads are rigidly linked and only one can be properly aligned at a time.

2.1.1 Magnetic disks

Figure 2.1 depicts the mechanisms in a magnetic disk. Data is stored in concentric tracks on parallel platters. A spindle rotates the platters at a fixed rotational speed. An arm moves laterally towards and away from the center of the platters to position the head on a particular track. Sectors correspond to a small angular portion of a track, which often stores 512 to 1024 bytes. Sectors represent the unit of addressability of a magnetic disk. Once the head is positioned on the proper track, the head waits until the sector rotates under it. At that time, data is transferred from the magnetic surface to the read buffer (in case of a read request) or from the write buffer to the surface (in case of a write).

The latency of a disk access can therefore be broken down into three main functions: seek, rotational and transfer latencies. Seek latency refers to the time it takes to position the read/write head over the proper track. This involves a mechanical transitional movement that may require an acceleration in the beginning and a deceleration and a repositioning in the end. As a result, although seek times have been improving, they have not kept up with the rates of improvement of silicon processors. While processing rates have improved by more than an order of magnitude, average seek times have shrunk to only half of their values of a decade ago (Table 2.1).

Disk characteristic	1980	1987	1990	1994	1999
Seek time(ms)	38.6	16.7	11.5	8(rd)/9(wr)	5
RPM	3600	3600	5400	7200	10000
Rotational latency(ms)	5.5	8.3	5.5	4.2	3
Bandwidth (MB/s)	0.74	2.5	3-4.4	6-9	18-22.5
8KB transfer(ms)	65.2	28.3	19.1	13.1	9.6
1MB transfer(ms)	1382	425	244	123	62

Table 2.1: Latency and bandwidth of commodity magnetic disks over the past two decades. The 1980 Disk is a 14 inch (diameter) IBM 3330, the 1987 Disk is a 10.5 inch Fujitsu (Super Eagle) M2316A, the 1990, 1994 disks are 3.5 inch Seagate ST41600, and the 1999 disk is a 3.5 inch Quantum Atlas 10k. Data for 1980 is from [Dewitt and Hawthorn, 1981], while data for 1987 and 1990 is from [Gibson, 1992]. The 1994 data is from [Dahlin, 1995].

The second function, rotational latency, refers to the time it takes to wait for the sector to rotate under the read/write head. This is determined by the rotational speed of the disk. Rotational speeds have improved slowly over the past decade, improving at an average annualized rate of 13%. Higher rotational speeds reduce rotational latencies and improve transfer rates. Unfortunately, they are hard to improve because of electrical and manufacturing constraints. Table 2.1 shows that rotational speeds have almost doubled this past decade.

The third function is transfer time, which is the time for the target sectors to pass under the read/write head. Disk transfer times are determined by the rotational speed and storage density (in bytes/square inch). Disk areal densities continue to increase at 50 to 55% per year, leading to dramatic increases in sustained transfer rates, averaged at 40% per year [Grochowski and Hoyt, 1996].

As shown in Table 2.1, disk performance has been steadily improving with more pronounced gains for large transfer access time than for small accesses. Small accesses are still dominated by seek time, while large transfers can exploit the improvement in the steady increase in sustained transfer rates. The transfer time for a 1MB access is being halved every 4 years, while the transfer time for an 8KB access is being cut by $1/3$ over the same four year period. These trends have different implications for sequential and random workloads since sequential scan based applications benefit more from newer generation disk drives than do random access workloads.

The cost of magnetic storage continues to be very competitive with other mass storage media alternatives. The cost per megabyte of magnetic storage continues to drop at an average rate of 40%

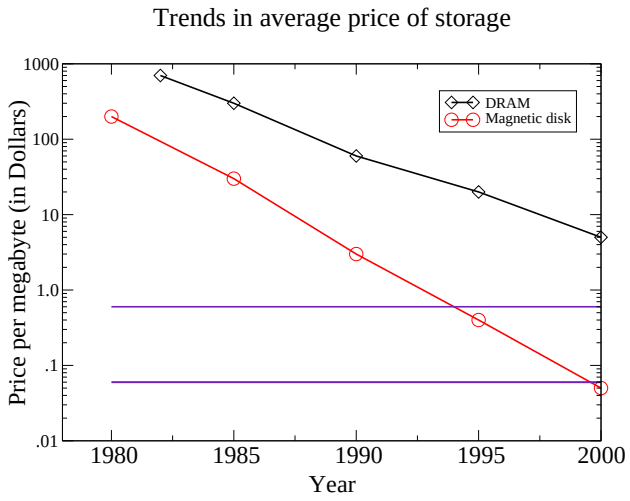


Figure 2.2: The cost per megabyte of magnetic disk storage compared to the cost of DRAM, paper and film. Magnetic disk prices have dropped at 40% per year, faster than DRAM and Flash. The cost of magnetic disk storage is now more than an order of magnitude cheaper than DRAM/Flash. The horizontal band at the bottom of the graph represents the price range of paper and film. For the last few years, magnetic disk storage has been competing head to head with paper and film storage in dollars per megabyte. The graph is a simplified reproduction of the one reported in [Grochowski, 2000].

per year, from $1 in 1993 to less than five cents in 1998. As shown in Figure 2.2, it is becoming cheaper to store information on disk than on paper or on film. Furthermore, magnetic storage continues to be an order of magnitude cheaper than RAM.

2.1.2 Memory

There are three important kinds of memory technology: ROM, Static RAM (SRAM) and Dynamic RAM (DRAM). ROM, and its programmable variants PROM and EEPROM, can only be read whereas RAM can be read and written. ROM is used to store permanent system memory or to store code that need not be updated or is updated very rarely, such a firmware. The contents of PROM chips can be updated (programmed) by using special equipment.

SRAM cells maintain the bit they store as long as current is continuously supplied to them. DRAM cells, however, store charge in semiconductor capacitors and do not flow current continuously. A DRAM cell must be refreshed many times per second, however, to maintain the stored charge. Compared to DRAMs, SRAMs are twenty times or so faster, much more expensive in power consumption and are several times more expensive than DRAM in chip real estate. In general, SRAM technology is used for fast memory banks such as registers and on-chip caches, while

cheaper and denser DRAM chips are used for main memory.

There are two kinds of DRAM technologies, the traditional asynchronous DRAM and the newer synchronous DRAM. Synchronous DRAM chips are clocked and are more suitable in systems where the processor's clock frequency is in the hundreds of Megahertz. The leading type of synchronous DRAM is Rambus [Rambus, 2000]. Rambus memory works more like an internal bus than a conventional memory subsystem. It is based around a high-speed 16-bit bus running at a clock rate of 400 MHz. Transfers are accomplished at the rising and falling edges of the clock, yielding an effective theoretical bandwidth of approximately 1.6 GB/s. Rambus bit-width is narrower than conventional 64 bit system buses. Narrower bit-width enables faster clocking, in fact yielding higher bandwidth.

DRAM capacities have been growing at 60% per year, and their bandwidth have been growing at 35% to 50% per year [Dahlin, 1995]. DRAM and SRAM memory chips are called volatile memory technologies because they lose their contents once powered off. Flash memory is a non-volatile memory technology which does not require power to maintain its storage. Other non-volatile memory technologies rely on battery-backed up RAM, a RAM memory bank with a power supply that can survive power failures by using batteries.

RAM and non-volatile memory technologies are still not cost-competitive with magnetic disks. The dollar per megabyte cost of RAM and Flash, for instance, is still an order of magnitude higher than that of magnetic storage. Figure 2.2 shows the trends in cost per megabyte for DRAM versus magnetic disk technology and more traditional storage media such as paper and film. Dropping memory prices have resulted in larger memory sizes on client and server workstations, and enabling some personal computing applications to fit entirely in memory, considerably improving response time for the end user. However, other emerging applications that use richer content such as video, audio and multimedia and data archival and warehousing applications are still compelled by the cheaper and the more rapidly decreasing costs of magnetic storage to use magnetic disks to store their massive data sets.

2.1.3 Processors

Processing power has been steadily increasing at a rate of 50% to 55% per year, roughly doubling every two years. This persistent exponential growth rate is resulting in substantial processing power on client and server workstations. Storage devices (which utilize processors to perform command processing, caching and prefetching, and other internal control and management operations) have

also benefited from increased processing power. The increase in processing speed of commodity processors and their dropping cost is resulting in the availability of substantial computing power at the on-disk processor for the execution of additional device-independent function. This general trend of processing power "moving" towards peripheral devices is expected to continue as processing becomes more abundant [Gray, 1997].

One result of the rapid sustained growth in processing power is the creation of a new generation of "commodity embedded servers" where the server workstation is integrated with the on-disk processor [Cobalt Networks, 1999]. One implication of such emerging embedded servers is that future large servers will be replaced by a cluster of commodity-priced embedded servers. Although embedded servers have substantial processing power, they are still limited in memory compared to clients.

The sustained rapid growth rate in processor speeds is also leading to an increase in the variability of processor speeds in a cluster. Clusters expand incrementally, which means that machines are purchased regularly, typically every few months. New machines come with faster processors. Machines that were meant to be fast servers may be easily dwarfed by newly purchased desktop computers. Conversely, once resourceful clients may be outstripped by newer and faster "embedded servers."

2.1.4 Interconnects

Two interesting trends can be distinguished in networking technology. The first is the consistent increase in the bandwidth of "machine-room" or "cluster" networks, which are surpassing the rate at which client processors can send, receive and process data. The second is the increase in the heterogeneity of the networks used to provide connectivity past the machine room(s) to the campus and to the wide area. High-bandwidth networks are often deployed within a single machine room or within a building. Networks that span several buildings and provide connectivity within a campus or a wide area are still largely heterogeneous, consisting of several types and generations of networking technologies.

LAN bandwidth has increased consistently over the past two decades. The dramatic increase in network bandwidth came with the introduction of switched networks to replace shared media such as the original Ethernet. Other increases are attributed to faster clock rates, optical links, and larger scales of integration in network adapters and switches enabling faster sending, receiving and switching. However, faster network technologies were not widely adopted in all deployed net-

works, primarily for cost reasons. This has resulted in wide variations in networks and in bandwidth across different environments. Three classes of networks can be distinguished: backbone networks, machine-room or server networks, and client local area networks.

Backbone networks carry a large amount of aggregate traffic, are characterized by a limited number of vendors and carriers and by large profit margins. As a result, backbone networks have been quick to adopt new and faster technologies.

Similarly but to a somewhat lesser degree, machine-room networks used to connect servers together to build highly parallel clusters or to connect servers to storage have embraced new and faster interconnect technologies relatively quickly. There is, however, a large number of server vendors and storage vendors. This requires lengthier standardization processes and implementation delays before a technology becomes available in a server network.

Local area networks typically connect a large number of client machines, dispersed across the campus and manufactured by a variety of vendors. Consequently, the introduction and deployment of new networking technologies in the client network have been a very slow and infrequent process. New technologies also often have stringent distance and connectivity requirements making them inadequate for the wider area. This high "barrier to entry" for new networking technologies into client networks is dictated also by the high cost associated with network upgrades. Client-side network adapter cards, closet equipment used for bridging and network extension as well as physical links throughout the building often must be upgraded together. Thus, to be viable, a new technology has to offer a substantial improvement in bandwidth.

Machine-room or server networks

Originally, networks were based on multiple clients sharing a common medium to communicate. One of the most popular early network technologies is Ethernet [Metcalfe and Boggs, 1976]. Ethernet is a specification invented by Xerox Corporation in the 1970s that operates at 10 Mb/s using a media access protocol known as carrier sense multiple access with collision detection (CSMA/CD). The term is currently used to refer to all all CSMA/CD LANs, even ones that have faster than 10Mb/s bandwidth and those that do not use coaxial cable for connectivity. Ethernet was widely adopted and was later standardized by the IEEE. The IEEE 802.3 specification was developed in 1980 based on the original Ethernet technology. Later, a faster Ethernet, called Fast or 100 Mbit Ethernet, a high-speed LAN technology that offers increased bandwidth, was introduced.

In shared 10 or 100 Mbit Ethernet, all hosts compete for the same bandwidth. The increasing

Networking Technology	Year introduced	Interconnection Equipment	Aggregate bandwidth (8 nodes, Mb/s)
Shared Ethernet	1982	Hub	10
Switched Ethernet	1988	Switch	80
Shared Fast Ethernet	1995	Switch	100
Switched Fast Ethernet	1995	Switch	800
Shared Gigabit Ethernet	1999	Hub	1000
Switched Gigabit Ethernet	1999	Switch	8000

Table 2.2: Bandwidth of currently available Ethernet networks. Bandwidth is increasing, especially in high-end networks. The year of introduction refers to the approximate date when the standard was approved by the IEEE as a standard. Prototype implementations usually precede that date by a few years.

Hub per-port average price	1996	1998	2000
Ethernet (10BaseT)	87	71	15
Fast Ethernet (100BaseT)	174	110	15

Table 2.3: Cost trends for Ethernet Hubs. All prices are in US $. The price for 1996 and 1998 is taken from [GigabitEthernet, 1999]. The price for 2000 is the average price quoted by on-line retailers for major brands (http://www.mircowarehouse.com/)

Switch per-port average price	1996	1998	2000
Ethernet (10BaseT)	440	215	35
Fast Ethernet (100BaseT)	716	432	56
Gigabit Ethernet	-	-	1200

Table 2.4: Cost trends for Ethernet Switches. All prices in US $. The price for 1996 and 1998 is taken from [GigabitEthernet, 1999]. The price for 2000 is the average price quoted by on-line retailers for major brands (http://www.mircowarehouse.com/)

levels of integration in network hardware in the late eighties enabled cost-effective packet switching at the data-link and network layers [OSI Standard, 1986]. A switch replaces the repeater and effectively gives the device full 10 Mb/s bandwidth (or 100 Mb/s for Fast Ethernet) to the rest of the network by acting as a logical crossbar connection. Such switched architectures can enable multiple pairs of nodes to communicate through a switched without a degradation in bandwidth like shared Ethernet. The first Ethernet switch was created in 1988. Today, several switch-based networks exist based on either copper or optical fiber cables including ATM [Vetter, 1995], Fibre-Channel [Benner, 1996], Gigabit Ethernet [GigabitEthernet, 1999], Myrinet [Boden et al., 1995] and ServerNet [Horst, 1995].

Switched networks are increasingly used to connect multiple nodes to form high-end clusters. These same networks are also used to connect storage devices and other peripherals, leading to the merger of interprocessor and peripheral storage interconnects. Single room switched networks are highly reliable and can deliver high-bandwidth data to simultaneously communicating pairs of network nodes. Network bandwidth in high-end cluster networks has improved at an average 40% to 45% per year. Over the period from 1980-94, the aggregate network bandwidth available to a 16 node cluster of high-end, desktop workstations has increased by a factor of 128, from a 10 Mb/s on shared Ethernet in the early 1980's to 1280 Mb/s on switched ATM, where 8 nodes send and 8 receive at the same rate. Six years later, the aggregate bandwidth available to the same 16 node cluster has reached 8 Gb/s on switched Gigabit Ethernet.

The growth in the bandwidth of Ethernet networks, the most cost-effective networking technology, is illustrated in Table 2.2. Table 2.3 and Table 2.4 illustrate the price per port for a hub-based (shared) network or a switch-based network. These tables show that new technologies start expensive and then their prices quickly drop. This is due to increased volumes and commoditization. These trends predict that machine room clusters will soon be able to use switched networks to interconnect storage devices and servers at little or no extra cost to traditional technologies.

Of course, this is the raw available bandwidth in the network, however, and not the observed end to end application bandwidth. Endpoint sender and receiver processing continue to limit the actual effective bandwidth seen by applications to a fraction of what is achievable in hardware. End application bandwidth has increased only by a factor of four between 1984 and 1994 [Dahlin, 1995]. Network interfaces accessible directly from user-level (e.g. U-Net [von Eicken et al., 1995], Hamlyn [Buzzard et al., 1996] and VIA [Intel, 1995]) help address the processing bottlenecks at the endpoints by avoiding the operating system when sending and receiving messages.

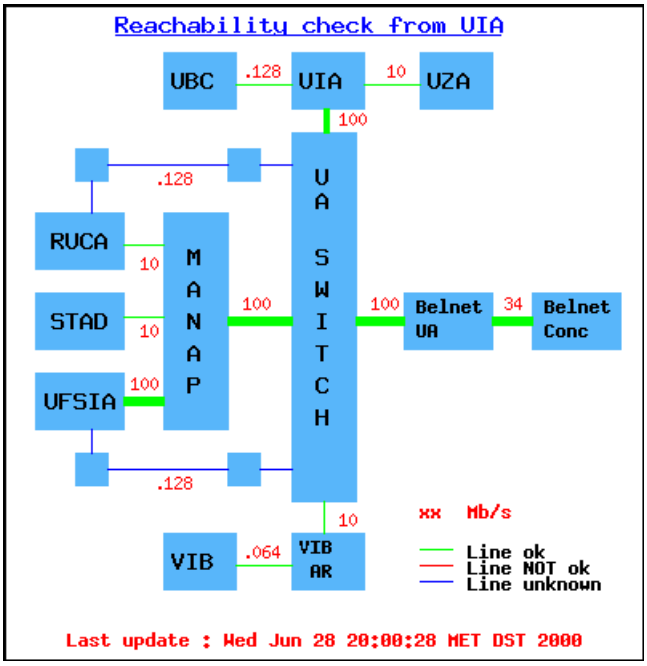


Figure 2.3: Networks that span more than a single room often have a substantial amount of heterogeneity as a result of incremental and unplanned growth. This figure shows the connectivity map of the campus of University of Antwerp (UIA campus) [University of Antwerp Information Service, 2000]. The numbers on the links represent bandwidth in Mb/s. This heterogeneity is very typical of large networks.

Client local and wide area networks

In parallel with developments in high-end single-room networks, the growth of the internet and of intranets have expanded the number of computers connected to networks within a single campus and within the wide area. The networks used to connect computers outside of the cluster or machine room are usually under more cost pressure. Moreover, network upgrades in wider areas occur less often due to the larger costs involved. As a result, while high-end environments use switched networks to connect their clients and servers, much slower networks still dominate many other environments.

This will remain the case for the foreseeable future for several reasons. First, extremely large switched networks with hundreds or thousands of nodes will remain costly for the foreseeable future. Constructing a huge crossbar to enable totally switched architecture for thousands of clients is still prohibitively expensive compared to a hierarchical topology. Usually, networks have hierarchical topologies which exploit the locality of traffic to reduce cost. For instance, several clients

in a department may share a high-bandwidth switched network, but connect to other departmental networks via a lower-bandwidth link. In this manner, bottlenecks can be avoided most of the time. However, for certain traffic and data distribution patterns, "bottleneck links" can severely limit application response time.

Second, distributed filesystems are expected to expand into private homes to provide users with better and more uniform data access and management software between home and office. Bandwidth to the home continues to make step increases with the introduction of cable modems, ISDN and ADSL. Although these technologies are several times faster than traditional phone-based modems, they are still limited to 1 to 2 Mb/s of bandwidth at best, and still substantially slower than standard LAN technologies such as 10 Mbit and 100 Mbit Ethernet. Figure 2.3 illustrates the bandwidth heterogeneity in an actual deployed network.

2.2 Application demands on storage systems

Traditional as well as emerging applications make three important demands on storage systems. First, application storage requirements are growing at a rapid pace, often reaching 100% per year. Second, continuous data availability remains a pressing demand for most organizations. Third, many emerging applications employ algorithms that require high-bandwidth access to secondary storage.

2.2.1 Flexible capacity

Storage requirements are growing at a rapid pace. The explosive growth of electronic commerce is generating massive archives of transaction records documenting customer behavior and history as well inter-business commerce. Medical applications such as image databases of treated patient cells are generating massive archives of multimedia objects. Scientific applications continue to generate massive repositories of geological and astronomical data. A large and growing number of applications from real-time astronomy, business and web data mining, and satellite data storage and processing are require massive amounts of storage. This storage must be distributed across storage devices to provide the necessary bandwidth and reliability. For example, the storage of audio information from 10 radio stations for one year requires 1 TB of disk space. The accumulation of video information from one station can fill up to 5 TB in a single year. The average database size of a radiology department in a hospital is over 6 TB [Hollebeek, 1997].

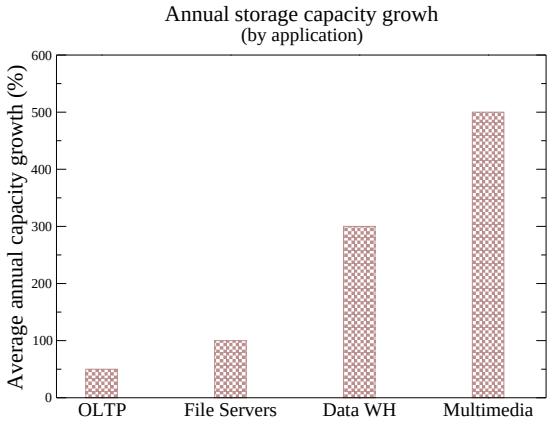


Figure 2.4: Annual average storage capacity growth by application. The storage needs of multimedia and data warehousing applications are increasing at factors of three and five per year. The data is from 1997 [EnStor, 1997].

Figure 2.4 shows that email and data warehousing applications are the most demanding in storage capacity growth. Like many emerging data-intensive applications, these applications often do not use relational databases but instead use filesystems or other custom non-relational data stores. Search, multimedia and data mining represent three important and common data-intensive applications.

2.2.2 High availability

Down-time is increasingly unacceptable in on-line services. Table 2.5 shows the average cost of down-time in dollars per minute for various traditional applications. Costs are expected to be higher with the increasing importance of on-line electronic commerce where customers can immediately turn to the next service provider.

The importance of data availability requires that all storage be replicated or protected by redundant copies. Furthermore, the need to recover site disasters requires that data be remotely replicated or “backed up” regularly. It is crucial that data be accessible at acceptable latencies during these management operations.

2.2.3 High bandwidth

Many web and business applications are fundamentally concerned with extracting patterns from large amounts of data. Whether it is a pattern in customer shopping preferences, or a pattern in

Application	Cost of down-time ($ per minute)
ERP	13,000
Supply Chain Management	11,000
Electronic Commerce	10,000
ATM/EFT/POS	8,500
Internet Banking	7,000
Universal Personal Services	6,000
Customer Service Centre	3,700

Table 2.5: Average cost of down-time for various applications. The actual cost varies depends on the particular organization. ERP stands for enterprise resource planning software, which performs several tasks such as sales and billing, inventory management and human resource related processing. The numbers are reported by Stratus Technology [Jones, 1998].

links among home pages on the web, many emerging applications are concerned with extracting "patterns" or "knowledge" from massive data sets with little structure. This translates into multiple passes over the data to test, refine and validate hypotheses.

Multimedia applications

The growing size of data sets is making search a frequent and important operation for a large fraction of users. From email messages to employee records to research papers and legal documents, search is probably one of the most frequently executed operations. While indexing can help reduce the amount of data that must be read from secondary storage for some applications, it is not effective for searching emerging multimedia and temporal databases, and for data archives without a structured schema.

Image databases, for example, often support queries by image content. Usually, the interesting features (e.g. color, texture, edges...) of the image are extracted and mapped onto feature vectors which represent the "fingerprint" of the image. Each image or multimedia object is associated with a feature vector, which is stored when the image is entered in the database. Feature vectors are used for content-based search. Performing the search in the "feature space" reduces the need to access "raw" objects. A feature vector contains several numerical attributes. The number of attributes in the vector is known as the "dimensionality" of the vector and correspondingly of the database.

The dimensionality of the feature vector is usually large because multimedia objects often have several interesting features such as color, shape, and texture, and because each feature is usually represented by several numerical attributes. For example, color can be represented by the percentage of blue, red and green pixels in the image. Research on indexing multidimensional and multimedia databases has made significant strides to improve access latencies. Grid-based and tree-based schemes such as R-trees and X-trees have been proposed to index multidimensional databases. Tree-based data structures generalize the traditional B-tree index by splitting the database into overlapping regions. However, as the number of attributes that a query is conditioned on increases, the effectiveness of these indexing schemes becomes severely limited. As dimensionality increases, a range or a nearest neighbor query requires accessing a relatively large portion of the data set [Faloutsos, 1996]. Because disk mechanics heavily favor sequential scanning over random accesses, an indexing data structure that requires many random disk accesses often performs slower than sequential scanning of the data set. Thus, while these data structures have proven effective for low dimensionality data, they are of little value in indexing higher dimensionality data. Simple sequential scanning of the feature vectors therefore becomes preferable to index-based search since sequential disk access bandwidths are much higher than random access bandwidths.

The inherent large dimensionality of multimedia objects and the tendency of search queries to condition on many dimensions at once is known as the “dimensionality curse”. Temporal databases, which store sequences of time-based samples of data, such as video frames or daily stock quotes, are also plagued with the dimensionality curse. These databases often support “query by example” interfaces, where the user provides an example sequence and asks for similar ones in the database. For example, a broker may search for all stocks that moved similarly over a given timed period. A stock is often represented by a high dimensionality vector, corresponding to the “fingerprint” of the stock in the time or frequency domain [Faloutsos, 1996].

Data mining

Businesses are accumulating large amounts of data from daily operation that they would like to analyze and “mine” for interesting patterns [Fayyad, 1998]. For instance, banks archive records of daily transactions, department stores archive records of point-of-sale transactions, online-catalog servers archive customer browsing and shopping patterns. This information contains interesting patterns and “knowledge nuggets” about customer profiles that management would like to discover and exploit. Typical data mining operations search for the likelihood of certain hypotheses by

analyzing the data for the frequency of occurrence of certain events. Most data mining algorithms require sequential scanning, often making multiple passes over the data.

One example of a data mining application is one that discovers association rules in sales transactions [Agrawal and Srikant, 1994]. Given customer purchase records, the application extracts rules of the form "if a customer purchases item A and B, then they are also likely to purchase item X." This information enables stores to optimize inventory layout.

Data mining applications comprise several components, some of which are data-intensive (data parallel), and others are more memory and CPU intensive. Frequent sets counting applications, for example, start counting the occurrences of pairs of items, triplets, four tuples, etc. in consecutive passes. The later passes access the secondary storage system less and less, consuming more CPU as their working set fit in main memory.

Summary

Emerging important applications are data-intensive, comprising at least one component which sequentially scans the data set. Such applications require high-bandwidth storage systems. They also place pressure on interconnects by processing large amounts of data. They, therefore, would benefit from techniques that minimize data movement especially over slow and overloaded networks.

2.3 Redundant disk arrays

Emerging applications require massive data sets, high-bandwidth and continuous availability. This dissertation proposes a storage system architecture that provides scalable capacity and bandwidth while maintaining high-availability. This research builds on previous work in high-availability and high-bandwidth storage systems, in particular disk arrays and transactional systems. The following sections provide necessary background on these two topics.

Traditionally, filesystems were contained completely on a single storage device. To allow file systems to be larger than a single storage device, logical volume managers concatenate many storage devices under the abstraction of a single logical device. Logical volume managers perform a part of the storage management function, namely the aggregation of multiple devices to look like a single one while employing a simple round-robin load balancing strategy across the devices. A logical volume manager is typically implemented in software as a device driver, and at most has a small effect on performance.

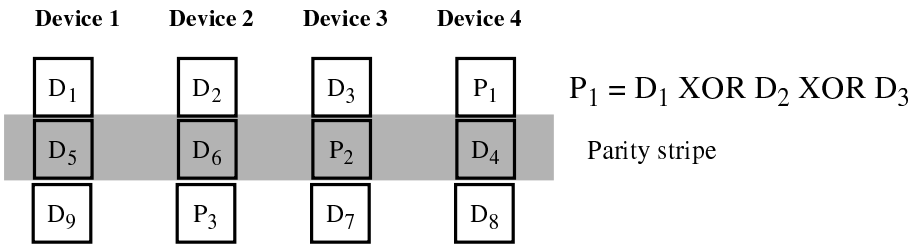


Figure 2.5: Layout of data and parity blocks in a RAID level 5 system. Data blocks within a stripe are protected by a parity block which is the cumulative XOR of the data blocks. The parity block is rotated around the devices in an array. Each write to any of the disks needs to update the parity block. Rotating the parity block balances the parity write traffic across the devices.

In the late 1980s, in order to bridge the access gap between the storage subsystem and the processors, arrays of small, inexpensive disks (RAID) were proposed [Patterson et al., 1988] to replace large expensive disk systems and automate load balancing by striping data [M. Livny and Boral, 1987]. RAID arrays provide the illusion of a single logical device with high small-request parallelism and large-request bandwidth. By storing a partially redundant copy of the data as parity on one of the disks, RAIDs improved reliability in arrays with a high number of components.

2.3.1 RAID level 0

RAID level 0 writes data across the storage devices in an array, one segment at a time. Reading a large logical region requires reading multiple drives. The reads can be serviced in parallel yielding N-fold increases in access bandwidth.

This technique is known as “striping”. Striping also offers balanced load across the storage devices. When a volume is striped across N devices, random accesses to the volume yield balanced queues at the devices. Striping is known to remove hot-spots if the stripe unit size is much smaller than the typical workload hot spot size.

2.3.2 RAID level 1

RAID 0 is not fault-tolerant. A disk failure results in data loss. RAID 1 writes a data block to two storage devices, essentially replicating the data. If one device fails, data can be retrieved from the replica. This process is also called “mirroring.” Mirroring yields higher availability but imposes a high capacity cost. Only half the capacity of a storage system is useful for storing user data.

When replication is used, reads can be serviced from both replicas and load can be balanced

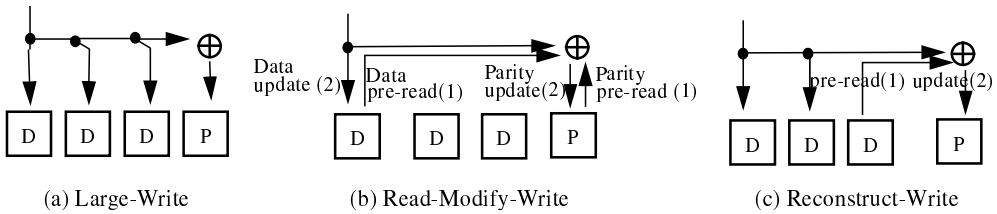


Figure 2.6: RAID level 5 host writes and reads in the absence of faults. An arrow directed towards a device represents a physical device write, while an arrow originating at a device represents a read from the device. The + operation represents bitwise XOR. Different protocols are used depending on the number of units updated: A large write where all units are updated is shown in (a), a read-modify-write where less than half of the data units in a stripe are updated is shown in (b), a reconstruct write where the number is larger than half is shown in (c). Some update protocols, like those shown in (b) and (c), consist of two-phases separated by a synchronization point at the host. I/Os labelled by a (1) are performed in a first phase. At the end of this phase, the new parity is computed and the parity and data are updated in a second phase (arrows labeled by a (2)).

by selecting one copy at random, through round-robin selection, or by selecting the disk with the shortest queue.

2.3.3 RAID level 5

RAID level 5 employs a combination of striping and parity checking. The use of parity checking provides redundancy without the 100% capacity overhead of mirroring. In RAID level 5, a redundancy code is computed across a set of data blocks and stored on an other device in the group. This allows the system to tolerate any single self-identifying device failure by recovering data from the failed device using the other data blocks in the group and the redundant code [Patterson et al., 1988]. The block of parity that protects a set of data units is called a parity unit. A set of data units and their corresponding parity unit is called a parity stripe. Figure 2.5 depicts the layout of blocks in a RAID level 5 array.

Write operations in fault-free mode are handled in one of three ways, depending on the number of units being updated. In all cases, the update mechanisms are designed to guarantee the property that after the write completes, the parity unit holds the cumulative XOR over the corresponding data units. In the case of a large write (Figure 2.6(a)), since all the data units in the stripe are being updated, parity can be computed by the host as the XOR of the data units and the data and parity blocks can be written in parallel.

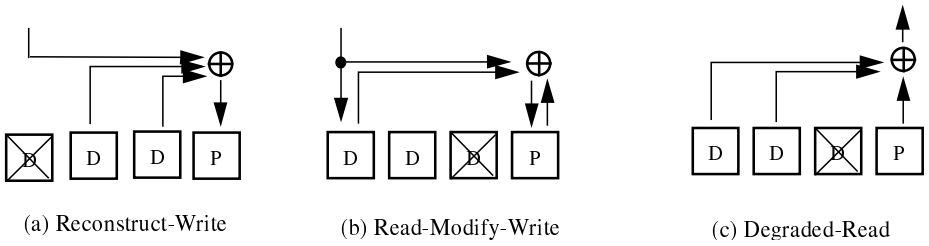


Figure 2.7: RAID level 5 host writes and reads in the presence of a fault. An arrow directed towards a device represents a physical device write, while an arrow originating at a device represents a read from the device. The + operation represents bitwise XOR. A device marked with an X represents a failed device. Different write protocols are used depending on whether the blocks on the failed device are being updated or not. If the failed device is being updated, the blocks on the surviving devices are read in a first phase, and the new parity computed by XORing these blocks with the block that was intended to be written to the failed device. As a result of the parity update, the update of the failed device is reflected in the parity device. This algorithm is shown in (a). If the failed device is not being updated, then the read-modify-write algorithm shown in (b) is used. A read request that touches the failed device is serviced by reading all the blocks in the stripe from the surviving devices and XORing them together to reconstruct the contents of the failed device (c).

If less than half of the data units in a stripe are being updated, the read-modify-write protocol is used (Figure 2.6(b)). In this case, the prior contents of the data units being updated are read and XORed with the new data about to be written. This produces a map of the bit positions that need to be toggled in the parity unit. These changes are applied to the parity unit by reading its old contents, XORing it with the previously generated map, and writing the result back to the parity unit. Reconstruct-writes (Figure 2.6(c)) are invoked when the number of data units is more than half of the number of data units in a parity stripe. In this case, the data units not being updated are read, and XORed with the new data to compute the new parity. Then, the new data units and the parity unit are written. If a device has failed, the degraded-mode write protocols shown in Figure 2.7(a) and Figure 2.7(b) are used. Data on the failed device is recomputed by reading the entire stripe and XORing the blocks together as shown in Figure 2.7(c). In degraded mode, all operational devices are accessed whenever any device is read or written.

2.4 Transactions

This dissertation proposes a scalable storage architecture based on giving clients direct access to storage devices over switched networks. In such a system, multiple clients can be concurrently

accessing shared devices. Proper algorithms must be devised to ensure correctness in the presence of concurrent shared accesses and also to ensure progress in the event of untimely failures of clients and devices. For the storage system to be scalable, these protocols must also scale well with system size avoiding centralized bottlenecks. Transactions were developed for fault-tolerant, concurrent-access database systems, and later applied in the construction of other fault-tolerant distributed systems. Chapter 4 proposes an approach based on a storage-specialized transactions and based on distributed protocols that exploit trends towards increased device intelligence to distribute control work to the endpoints avoiding centralization.

Because Chapter 4 builds on previous work in database transaction theory, a brief review of the main concepts in that field is necessary. This section reviews transaction theory as a background for the discussions in Chapter 4. This section may be skipped by readers comfortable with transactions and with database concurrency control and recovery algorithms.

Database systems were developed to support mission-critical applications requiring highly concurrent and highly available access to shared data by a large number of users. A database system comprises two components that are responsible for this: a concurrency control component and a recovery component. Concurrency control refers to the ability to ensure concurrent users consistent access to the data despite the fact that the execution of their operations may be interleaved by the database engine. Recovery refers to the ability of the database to tolerate software and hardware failures in the middle of an operation.

A central concept in database systems is that of a transaction, a program unit which performs a sequence of reads and writes to items in the database, with the accesses usually separated by some computation. A transaction is guaranteed to have strong correctness properties in the face of concurrency and failures, and is the unit on which the database system provides these properties.

2.4.1 Transactions

Database transactions [Gray et al., 1975, Eswaran et al., 1976] have four core properties, widely known as the ACID properties [Haerder and Reuter, 1983, Gray and Reuter, 1993]:

- **Atomicity**: This is the “all or nothing” property of transactions. A transaction’s effects will either be completed in their entirety or none of them will reach the database, even in the case of untimely failures.

- **Consistency**: This property asserts that transactions must preserve high-level integrity constraints with respect to the data in the database. These constraints depend on the contents and customer use of the particular database.

- **Isolation**: This property concerns the concurrent execution of transactions. It states that intermediate changes of concurrent transactions must not be visible to each other, providing each transaction with the illusion that it is executing in isolation.

- **Durability**: Durability means that the impact of committed transactions cannot be lost.

An example transaction is shown below. The transaction transfers a specified $amount$ from account $source\_ac$ to account $dest\_ac$. The transaction must first make sure that the source account has a sufficient balance. Then, the source account is decremented and the destination account is credited by the same amount. In the event that a failure occurs after the source account has been debited, but before the transaction has completed, the transaction mechanism ensures that the statements between the `T.begin` and the `T.commit` will either execute to their entirety, or none of their effects will reach the database (atomicity). Furthermore, once the transaction completes with a successful commits, its updates to the database can not be undone or lost (durability). This transaction does not result in loss or creation of money to the institution since it transfer the same amount from the source account to the destination account, maintaining the invariant that the sum over all accounts remains fixed (consistency). Note that consistency depends on the semantics of the operations performed by the program and the semantics of the database. However, the atomicity and durability properties can be enforced by the database without knowledge of the particular semantics of the transaction.

```
1 T1.begin
2 if (source_ac.balance > amount)
3    source_ac.balance = source_ac.balance - amount;
4    dest_ac.balance = dest_ac.balance + amount;
5  else
6    T1.abort
7 T1.commit
```

2.4.2 Serializability

Database systems must execute transactions concurrently to meet demanding throughput requirements. This concurrency must not result in incorrect behavior. To reason about the correctness of concurrent transactions, we can abstract the details of the transaction program and focus on the accesses that it makes to the database. A transaction is described simply as a sequence of $read$ and $write$ requests to data items in the database. For example, the account transfer transaction above issues a $read$ to the source account (Line 2) to make sure it has enough funds, then issues a $write$ to decrement the source account by the transferred amount (Line 3) and finally issues a $write$ to the destination account to increment it with the transferred amount (Line 4).

Accesses to data items by a transaction are denoted by $r[x]$ and $w[x]$ where x is the data item being accessed. More precisely, $w_i[x]$ denotes a write by transaction T_i to data item x. C_i and A_i denote a commit and an abort of transaction T_i respectively.

Using this terminology, a transaction that transfers funds from account x to account y then finally commits can be described as:

$$T_1 : \ r_1[x] \ \ w_1[x] \ \ r_1[y] \ \ w_1[y] \ \ C_1$$

while a transaction that eventually aborts can be described as:

$$T_1 : \ r_1[x] \ \ A_1$$

In this case, the transaction aborts after reading the source account x and discovering that it does not have enough funds to cover the amount that needs be transferred to the destination account y.

Concurrent executions of transactions are described by execution histories which show how the accesses performed by the transactions interleave with each other. An example history showing two transactions is H_1:

$$H_1 : \ r_1[x] \ \ w_1[x] \ \ r_1[y] \ \ w_1[y] \ \ C_1 \ \ r_2[x] \ \ w_2[x] \ \ r_2[z] \ \ w_2[z] \ \ C_2$$

H_1 shows two transactions T_1 and T_2. T_1 transfers a given amount from source account x to destination account y. T_2 transfers another amount from source account x to destination account z. H_1 is called a serial history. A serial execution history is one where only one transaction is active at

a time, and a transaction starts and finishes before the next one starts. Indeed, in the history above, T_1 executes to completion and commits before T_2 starts.

The traditional correctness requirement for achieving isolation of database transactions is *serializability* [Papadimitriou, 1979]. A history is said to be *serializable* if its results are *equivalent* to a serial execution. The notion of equivalence is often defined in terms of the "reads-from" relationship. A transaction T_i is said to "read x from" T_j if there is no access to x between $r_i[x]$ and $w_j[x]$. Two executions H_1 and H_2 are equivalent if they contain the same set of transactions, if they establish the same "reads from" relationship between transactions, if the final value of each data item in the database is written by the same transaction in both histories [1], and if the transactions that commit (abort) in one also commit (respectively abort) in the other.

For example, the following execution H_2 is serializable:

$$H_2: \ r_1[x] \ w_1[x] \ r_2[x] \ r_1[y] \ w_1[y] \ C_1 \ w_2[x] \ r_2[z] \ w_2[z] \ C_2$$

H_2 is serializable because it is equivalent to a serial history, namely H_1 above. In both histories, T_1 and T_2 eventually commit. The final value of x and the finally value of y are both written by transaction T_2 in both histories. Finally, both histories establish the same reads-from relationship: in both histories, T_2 reads x from T_1.

Using this definition for serializability, the execution below is not serializable:

$$H_3: \ r_1[x] \ r_2[x] \ w_1[x] \ r_1[y] \ w_1[y] \ C_1 \ w_2[x] \ r_2[z] \ w_2[z] \ C_2$$

H_3 is not serializable because there is no equivalent serial history that satisfies all the conditions of equivalence described above. In particular, T_2 does not read x from T_1 so T_2 must appear before T_1 in an equivalent serial history. At the same time, however, the final value of data item x is written by T_2 so T_2 must appear after T_1 in an equivalent serial history. Hence, the impossibility of the existence of such a serial history.

2.4.3 Serializability protocols

Serializability has been traditionally ensured using one of three approaches: locking, optimistic, and timestamp ordering methods. Each is a set of rules for when an access can be allowed, delayed, aborted or retried.

[1] That is, if T_1 is the last transaction to write to x in one history, it is also the last one to write to x in the other.

Batch locking

There are two widely used locking variants that achieve serializability: batch locking and two-phase locking. Batch locking is the most conservative approach. Locks on shared data items are used to enforce a serial order on execution. All locks are acquired when a transaction begins and released after it commits or aborts. It can be shown that executions allowed by this locking scheme are serializable. Precisely, the executions are equivalent to a serial one where transactions appear in the order of their lock acquisition time.

Batch locking has the advantage that it is simple to implement. Furthermore, it does not lead to deadlocks because all locks are acquired at the same time. Therefore, there is no "hold-and-wait", a necessary condition for deadlocks to occur [Tanenbaum, 1992]. The disadvantage of this scheme is that it severely limits concurrency since locks blocking access to data from other transactions are potentially held for a long period.

Two-phase locking

Significant concurrency can be achieved with two-phase locking [Gray et al., 1975], a locking scheme where locks may be acquired one at a time, but no lock can be released until it is certain that no more locks will be needed. Two-phase locking allows higher concurrency than batch locking, although it is still susceptible to holding locks for a long period of time while waiting for locks held by other transactions. This is because locks are acquired when the item is first accessed and must be held until the last lock that will be needed is finally acquired.

Unlike batch locking, two-phase locking is susceptible to deadlocks. Consider two transactions T_1 and T_2 that both want to write to two data items x and y. Suppose that T_1 acquired a lock on x and then T_2 acquired a lock on y. Now, T_1 requests the lock on y and T_2 requests the lock on x. Both transactions will block waiting for the other one to release the other lock. Because of the two-phase nature of the locking protocols, a transaction can not release any locks until it has no more locks to acquire. So, neither transaction will release the lock needed by the other, resulting in a deadlock.

When two-phase locking is employed, a deadlock avoidance or detection and resolution strategy must also be implemented. Deadlocks can be prevented by ordering all the possible locks and then enforcing a discipline that require that locks are acquired in a given order. This breaks the "circular wait" condition that is a pre-requisite for deadlocks to occur.

Deadlocks can also be detected and resolved by aborting the appropriate blocked transactions.

Deadlocks can be detected by recording the dependencies if all dependencies can be enumerated and if the overhead of maintaining the dependency information is not prohibitive. Distributed systems can exhibit deadlocks involving transactions at multiple nodes requiring dependency information distributed across sites to be collected and merged to discover a deadlock. In this case, a simple timeout based scheme is likely to work better. If a transaction can not acquire a lock within a pre-specified period of time (the timeout), it is suspected of being involved in a deadlock and is therefore aborted and restarted.

Optimistic methods

Locking is also known as a pessimistic approach, since it presumes that contention is common and that it is worthwhile to lock every data item before accessing it. Optimistic methods are so called because they assume conflict is rare and do not acquire locks before accessing shared data but instead validate at commit time that a transaction's execution was serializable [Eswaran et al., 1976, Kung and Robinson, 1981]. If a serializability violation is detected, the transaction is aborted and restarted.

Optimistic protocols are desirable when locking and unlocking overhead is high (e.g. if it involves network messaging), when conflicts are rare or when resources are plentiful and would be otherwise idle (e.g. multiprocessors).

Timestamp ordering

Timestamp ordering protocols select an a priori order of execution using some form of timestamps and then enforce that order [Bernstein and Goodman, 1980]. Most implementations verify timestamps as transactions execute read and write accesses to the database, but the more optimistic variants delay the checks until commit time [Adya et al., 1995].

In the simplest timestamp ordering approach, each transaction is tagged with a unique timestamp at the time it starts. In order to verify that reads and writes are proceeding in timestamp order, the database tags each data item with a pair of timestamps, *rts* and *wts*, which correspond to the largest timestamp of a transaction that read and wrote the data item, respectively. Basically, a read by transaction T with timestamp $opts(T)$ to data item v is accepted if $opts(T) > wts(v)$, otherwise it is (immediately) rejected. A write is accepted if $opts(T) > wts(v)$ and $opts(T) > rts(v)$. If an operation is rejected, its parent transaction is aborted and restarted with a new larger timestamp.

To avoid the abort of one transaction causing additional transactions to abort, a situation known

as cascading aborts, reads are usually not allowed to read data items previously written by active (uncommitted) transactions. In fact, when an active transaction wants to update a data item, it first submits a "prewrite" to the database declaring its intention to write but without actually updating the data. The database accepts a prewrite only if $opts(T) > wts(v)$ and $opts(T) > rts(v)$. A prewrite is committed when the active transaction T commits and a write is then issued for each submitted prewrite. Only then is the new value updated in the database and made visible to readers. A transaction that issued a prewrite may abort, in which case its prewrites are discarded and any blocked requests are inspected in case they can be completed. If a prewrite with $opts(T)$ has been provisionally accepted but not yet committed or aborted, any later operations from a transaction T' with $opts(T') > opts(T)$ are blocked until the prewrite with $opts(T)$ is committed or aborted.

To precisely describe the algorithms executed by the database upon the receipt of a read, prewrite or write request, a few more variables must be defined. To simplify the presentation of the algorithms, we will assume that the database contains a single data item, say v. Each data item has a queue of pending requests. We denote by $minrts(v)$ the smallest timestamp of a queued read to data item v, while $minpts(v)$ represents the smallest timestamp of a queued prewrite to v. The algorithm executed upon the receipt of a prewrite request can be described as follows:

```
prewrite(v, opts)
if (opts < rts(v)) then
     // opts is too far in the past
     return REJECT;
else if (opts < wts(v)) then
     // opts is too far in the past
     return REJECT;
else
     // opts bigger than the timestamp of any
     // transaction that read or wrote this item
     // accept and put it on the prewrite queue
     enqueue(prewrite, v, opts);
     minpts(v) = MIN (minpts(v), opts);
     return(ACCEPT);
```

Writes are always accepted because the corresponding prewrite was already accepted. Writes may be queued until reads with lower timestamps are serviced. When a write is processed, its corresponding prewrite is removed from the service queue. Processing a write could increase $minpts(v)$, resulting in some reads being serviced. By setting a special flag, the write algorithm triggers the database to inspect any requests queued behind the associated prewrite.

```
write(v, opts)
if (opts > minrts(v)  || opts > minpts(v)) then
     // there is a read ahead of us in the queue
     // so we'll wait so we can service it before
     // we update the data item and force it to be rejected
     // or there is a prewrite before us
     enqueue(write, v, opts);
else
     // write data item to stable storage and update wts
     // set the flag so we scan the queue of waiting requests
     write v to store;
     dequeue(prewrite, v, opts);
     wts(v) = MAX (wts(v), opts);
     inspectQueue = true;
```

In general, a read can be serviced either immediately after it is received, or it may be queued and serviced later. When a read is processed after being removed from the service queue, $minrts(v)$ could increase and cause some write requests become eligible for service. When a read is processed, the `inspectQueue` flag is set, which causes the database to inspect the queue and see if any queued (write) requests can be serviced. Similarly, when a write is processed, the `inspectQueue` flag is set so that queued (read and write) requests are inspected for service. Upon inspection of the queue, any requests that can be serviced are dequeued, one at a time, possibly leading to computing new values of $minrts(v)$ and $minpts(v)$. Upon the receipt of a request to read data item v by a transaction with timestamp $opts$, the database executes the following algorithm to decide to service the read immediately, put it on the queue of pending requests or reject it.

```
read(v, opts)
if (opts < wts(v)) then
     return REJECT;      // opts is too far in the past
else if (opts > minpts(v)) then
     // there is a prewrite before opts
     // we can not accept the read until we know
     // the fate of the prewrite
     // if the prewrite is later confirmed, this
     // read should return a value that is more
     // recent than the current value available now
     enqueue(read, v, opts);
     minrts(v) = MIN (minrts(v), opts);
else
     // There is no prewrite in the queue ahead of opts
     // so accept and update timestamp and set the flag
     // so we scan the queue of waiting writes behind this read
     rts(v) = MAX(rts(v), opts);
     inspectQueue = true;
     return committed value of v;
```

Figure 2.8 shows an example scenario illustrating how timestamp ordering works. Multiple concurrent transactions are active. They submit read, prewrite and write requests to the database. All the requests address a single data item in the database, denoted by x. The request queue shown in the figure represents the requests which are pending to that data item. Requests are queued if they cannot be handled immediately by the database as explained above. Initially, x has an rts of 8 and a wts of 12. The initial state of the database also shows that a prewrite with $opts = 14$ has been accepted by the database.

The scenario proceeds as follows. First, the database receives a read with $opts = 10$. This read is rejected because it is supposed to occur before a write which has already occurred. The following read has a timestamp of $opts = 15$, later than any action that has or is going to occur. Since there is a prewrite queued with a lower timestamp, this read cannot be serviced yet or the abort of the transaction doing the write would force the abort of this transaction (because it read a value that should have never been written). At this time, the database cannot know whether the prewrite will be confirmed with an actual write, or will be aborted. The read request is therefore put on the request queue behind the prewrite request. Shortly thereafter, a read with $opts = 13$ is received.

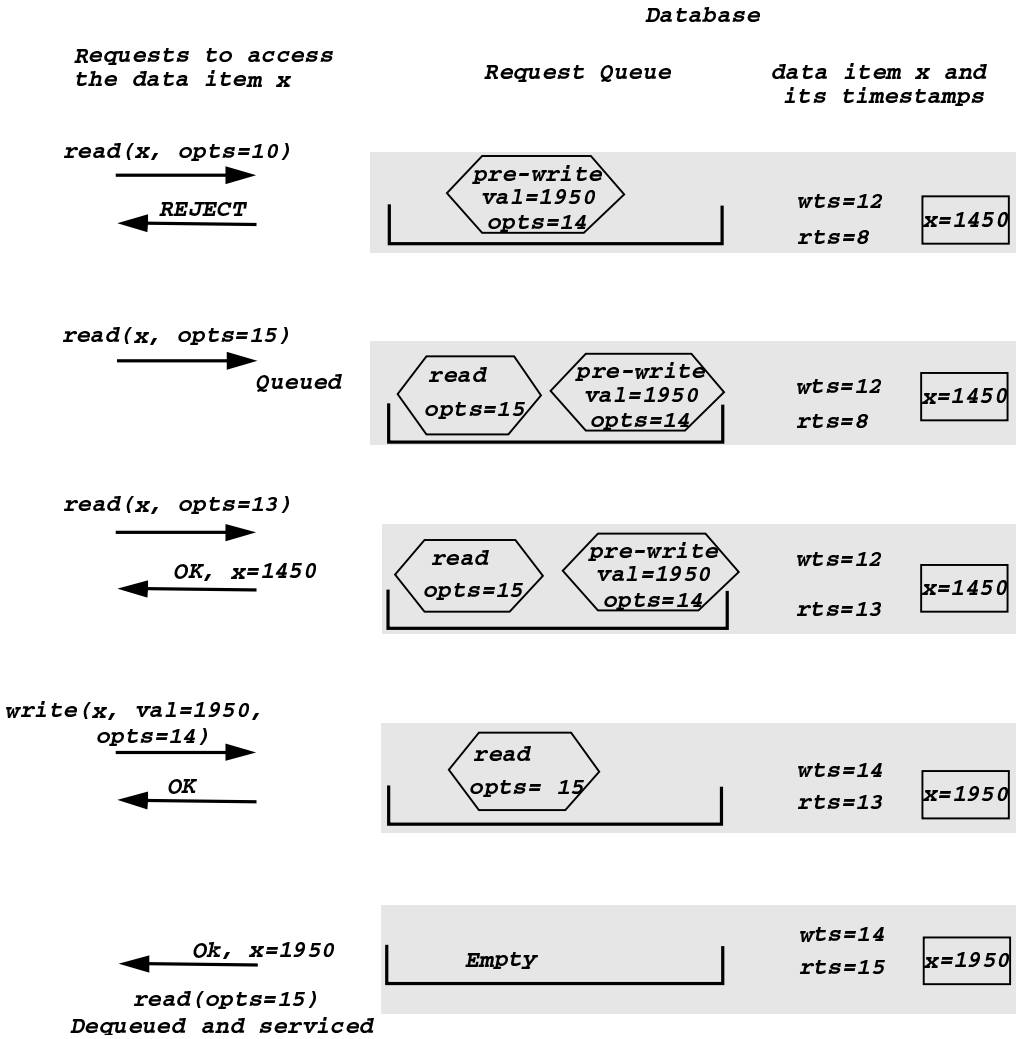


Figure 2.8: A sample scenario of a database using a timestamp ordering protocol. The database is assumed to have a single data item x. At the outset, $rts(x)$ is 8, and $wts(x)$ is 12, and the database has accepted a prewrite with timestamp 14. The figure shows changes to the database state and the request queue as requests are received and handled. The hexagonal shapes represent requests and arrows show when a request arrives to the database and when a response is generated. First, a read request with $opts = 10$ is received and rejected, because wts is 12. The read arrived too late since a transaction with timestamp 12 has already written to x and committed. Later, a read request with $opts = 15$ is received, because a prewrite with a lower timestamp is in the queue, the read is queued. Some time later, a read with $opts = 13$ is received and serviced updating rts. Finally, the write corresponding to the prewrite is received. It is serviced first, then the queue is inspected and the read request with $opts = 15$ is serviced, updating rts and leaving the queue empty. The reader can inspect that the execution allowed by the protocol is indeed serializable. In particular, it is equivalent to the serial execution of $T_{opts=13}$ (read) followed by $T_{opts=14}$ (write) and finally followed by $T_{opts=15}$ (read).

This read is serviced immediately because its $opts$ exceeds $wts = 12$. The committed value of x is returned in response to this read. After the read is serviced, the write corresponding to the prewrite with $opts = 14$ is received. It is serviced first, then the queue is inspected and the read request with $opts = 15$ is serviced, updating rts and leaving the queue empty.

2.4.4 Recovery protocols

The all or nothing recovery property provided by database systems is known as atomicity. The recovery subsystem ensures that all transactions meet the atomicity property in the face of failures. There are three kinds of failures which a database system must handle:

- *Transaction failures.* A transaction may decide to voluntarily abort. This abort is usually induced by the programmer or the user when forward progress cannot be made or might be better achieved by restarting the transaction from the beginning. In the case of a transaction failure, any updates made by the failed transaction to the database must be undone.
- *System failures.* Loss of the contents of volatile memory represent a system failure. In the event of a system failure, transactions that committed must have their effects applied to the database on recovery. Any other active transactions which did not commit at the time of the failure must have their updates removed from the database to preserve the atomicity property.
- *Media failures.* Loss of the contents of non-volatile storage represent a media failure. This requires restoring the database from an archival version.

A database is usually divided into disk pages, which are cached by a buffer manager. Active transactions access the database pages by reading and writing items in the pages cached by the buffer manager. Atomicity is achieved via some form of "logging." Logging is the action of writing a description of the transaction to stable storage, not overwriting the "old" values of the data items that it intends to update. Every time the transaction writes a value in volatile memory, that value is written to the log. When the transaction requests to commit, the values it intends to update are all on the log. A final "commit" record is written to the log to mark the end of the new values written by the committing transaction. Once the commit record is written, the updated values can be written back to stable storage at the buffer manager's leisure. No values can be written before commit time however, unless such updates can be undone at abort time.

One way logging can be used to achieve atomicity is as follows: if a failure occurs before the "commit" record is written to the log, none of the values are updated on the database and the

transaction is considered to have aborted. This achieves atomicity since it enforces the "nothing" effects, where no effects whatsoever from the transaction reach the database.

If a failure occurs after the "commit" record is written to the log (commit point) but before all the changes are applied to the data items, the new values can be applied to the database by accessing the log, thereby achieving the "all" effect and also achieving atomicity.

This logging technique is called Redo logging. It requires that a transaction that did not yet commit never writes to stable storage. This requirement constrains the buffer manager, which can not, for example, evict a dirty page (written by an uncommitted transaction) back to disk by writing it back to its original location. Other techniques such as Undo and Undo/Redo [Gray and Reuter, 1993] also achieve atomicity but dictate different constraints on the buffer manager. An Undo record records in the log the current committed value of the item, allowing the buffer manager to overwrite the database with an uncommitted value and still ensure that if aborted, the effect of the uncommitted transaction can be removed.

Undo/Redo logging places the least amount of constraints on the buffer manager regarding when blocks should be written back to the database at the expense of more logging work.

2.5 Summary

Storage requirements are growing at a rapid pace, often exceeding 100% per year. This rapid growth is fueled by the explosion in the amount of business data which must be archived for later analysis, by the need to store massive data sets collected from ubiquitous sensors and from satellites, and by the popularity of new data types that inherently contain more information such as audio and video.

Storage systems are still predominantly based on magnetic disks, whose sequential transfer rates have been consistently improving over the past decade at an average of 40% per year. Random access times, on the other hand, dominated by mechanical seek positioning, improved much more slowly.

Processors have maintained impressive rates of improvement, doubling in speed every 18 months. Disk technology did not keep up resulting in a severe "I/O gap." To bridge the gap between processing and I/O subsystems, disk arrays were introduced. Data is striped across multiple disks yielding higher bandwidth from parallel transfers. Disk arrays rely on a single controller to implement the striping and RAID functions.

The point-to-point bandwidth available to a communicating pair of nodes in a high-end cluster network have improved at an average of roughly 45% per year from 10 Mb/s in 1980 to 1 Gb/s in

2000. Furthermore, the aggregate bandwidth available to an small 8 node cluster has jumped from 10 Mb/s in the 1980s to 8 Gb/s in 2000, thanks to switching technology that enables simultaneous communication from independent pairs of nodes without degradation in bandwidth. Machine room networks often use reliable switched networks such as Fibre-Channel, Myrinet and Gigabit Ethernet to build high-performance servers from clusters of commodity PCs and devices.

The increasing transfer rates of storage devices place strong pressure on the single disk array controller to deliver high-bandwidth to streaming applications. The convergence of peripheral storage and inter-processor networks point to an architecture that eliminates the single controller bottleneck and allows direct access from multiple hosts to shared storage devices over switched networks. Such a shared storage system promises scalable bandwidth but introduces new challenges in ensuring correctness and availability in the presence of sharing and concurrency. Chapters 3 and 4 describe such an architecture and the algorithms required to ensure correctness and scalability.

The consistent increases in processing rates have resulted in processors of variable speeds coexisting in the same computer system. Storage clients, smart devices, and disk array controllers all tend to have disparate amounts of processing and memory which vary between sites. Networks continue to be quite heterogeneous in their bandwidth because of complex topologies, cost pressures and the coexistence of several technology generations. This makes the static partitioning of function across nodes of a distributed storage system without regard to communication patterns and resource availability undesirable in practice. Chapter 5 proposes an approach to automatically decide on optimal function placement in a distributed storage system.

Chapter 3

Network-attached storage devices

This chapter reviews the argument made by the NASD group at Carnegie Mellon [Gibson et al., 1997b, Gibson et al., 1998], and which states that current storage architectures do not scale cost-effectively because they rely on file server machines to copy data between storage devices and clients. Unlike networks, file server machines must perform critical functions in the midst of the data path and therefore must be custom-built (and expensive) to meet the demands of bandwidth-hungry applications. Gibson *et al*. [Gibson et al., 1997b, Gibson et al., 1998] demonstrated that separating filesystem control messaging from data transfers improves scalability and reduces cost by eliminating the file server from the data access path. Based on this architecture, this chapter proposes a storage service where storage striping, aggregation and fault-tolerance functions are also distributed to the storage clients and the storage devices, thereby eliminating the need for the synchronous involvement of a centralized server machine.

The chapter is organized into two parts. The first part restates the principles developed by the NASD group working at Carnegie Mellon from 1995 to 1999. In particular, Section 3.1 summarizes the trends that mandate changing the current server-attached disk (SAD) architecture. Section 3.2 describes two network-attached storage architectures, network-attached SCSI (NetSCSI) and network-attached secure disks (NASD), while Section 3.3 discusses the NASD architecture in particular. A validation of the potential benefits of the NASD architecture was reported elsewhere [Gibson et al., 1997b, Gibson et al., 1998] and is only briefly reviewed in this chapter.

The second part of the chapter reports on a striping storage service on top of NASDs. NASD enables clients to benefit from striped parallel transfers from multiple devices over switched networks. Section 3.4 describes a prototype storage service, Cheops, which delivers on that promise. Section 3.5 reports on the performance of bandwidth-hungry applications on top Cheops/NASD. Section 3.6 describes alternative storage architectures that have been proposed to provide scalable

storage. Section 3.7 summarizes the chapter.

3.1 Trends enabling network-attached storage

Network-attached storage devices enable direct data transfer between client and storage without invoking the file server in common data access operations. This requires relatively important changes both to hosts and storage devices. Such changes are becoming possible and compelling, however, thanks to the confluence of several overriding factors: (1) the cost-ineffective scaling of current storage architectures; (2) the increasing object sizes and data rates in many applications; (3) the availability of new attachment technologies; (4) the convergence of peripheral and interprocessor switched networks, and; (5) an excess of on-drive transistors.

3.1.1 Cost-ineffective storage systems

Distributed filesystems [Sandberg et al., 1985, Howard et al., 1988] have been widely used to allow data to be stored and shared in a networked environment. These solutions rely on file servers as a bridge between storage and client networks. In these systems, the server receives data on the storage network, encapsulates it into client protocols, and retransmits it on the clients' network. This architecture, referred to as server-attached disk (SAD), is illustrated in Figure 3.1. Clients and servers share a network and storage is attached directly to general-purpose workstations that provide distributed file services. This is costly both because the file server has to have enough resources to handle bandwidth-hungry clients and because it often requires system administrators to manage capacity and load balancing across the servers when multiple servers are used.

While microprocessor performance is increasing dramatically and raw computational power would not normally be a concern for a file server, the work done by a file server is data- and interrupt-intensive and, with the poorer locality typical of operating systems, faster processors will provide much less benefit than their cycle time trends promise [Ousterhout, 1990, Chen and Bershad, 1993].

Typically, distributed file systems employ client caching to reduce this server load. For example, AFS clients use local disk to cache a subset of the global system's files. While client caching is essential for high performance, increasing file sizes, computation sizes, and work-group sharing are all inducing more misses per cache block [Ousterhout et al., 1985, Baker et al., 1991]. At the same time, increased client cache sizes are making these misses more bursty.

When the post-client-cache server load is still too large, it can either be distributed over mul-

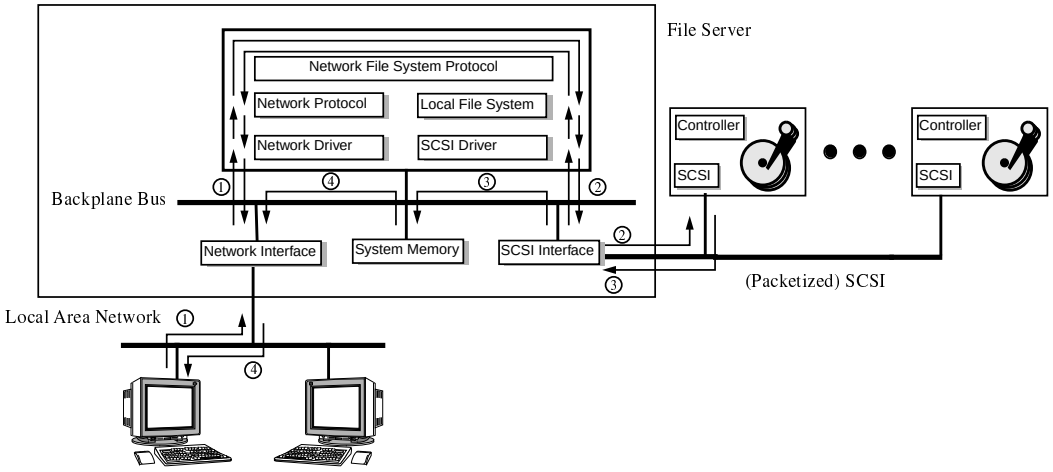


Figure 3.1: Server-attached disks (SAD) are the familiar local area network distributed file systems. A client wanting data from storage sends a message to the file server (1), which sends a message to storage (2), which accesses the data and sends it back to the file server (3), which finally sends the requested data back to the client (4). Server-integrated disk (SID) is logically the same except that hardware and software in the file server machine may be specialized to the file service function.

tiple servers or satisfied by a custom-designed high-end file server. Multiple-server distributed file systems attempt to balance load by partitioning the namespace and replicating static, commonly used files. This replication and partitioning is too often ad-hoc, leading to the "hotspot" problem familiar in multiple-disk mainframe systems [Kim, 1986] and requiring frequent user-directed load balancing.

Not surprisingly, custom-designed high-end file servers more reliably provide good performance, but can be an expensive solution [Hitz et al., 1990, Drapeau et al., 1994]. Since file server machines often do little other than service distributed file system requests, it makes sense to construct specialized systems that perform only file system functions and not general-purpose computation. This architecture, called server-integrated-disk (SID), is not fundamentally different from SAD. Data must still move through the server machine before it reaches the network, but specialized servers can move this data more efficiently than general-purpose machines.

Since high performance distributed file service benefits the productivity of most users, this server-integrated disk architecture occupies an important market niche [Hitz et al., 1990, Hitz et al., 1994]. However, this approach binds storage to a particular distributed file system, its semantics, and its performance characteristics. For example, most server-integrated disks provide NFS file service, whose inherent performance has long been criticized [Howard et al., 1988]. Furthermore, this ap-

proach is undesirable because it does not enable distributed file system and storage technology to evolve independently. Server striping, for instance, is not easily supported by any of the currently popular distributed file systems. Binding the storage interface to a particular distributed file system hampers the integration of such new features [Birrel and Needham, 1980].

3.1.2 I/O-bound large-object applications

Traditionally, distributed file system workloads have emphasized small accesses and small files whose sizes are growing though not dramatically [TPC, 1998, Baker et al., 1991]. However, new workloads are much more I/O-bound. Examples include evolving data types such as multimedia, audio and video. In addition to richer content, storage bandwidth requirements continue to grow rapidly due to rapidly increasing client performance and more data-intensive algorithms. Applications such as data mining of retail transaction records or telecommunications call records to discover historical trends employ data-intensive algorithms.

Traditional server-based architectures are fundamentally not scalable. They can support these demanding bandwidth requirements only at a high cost overhead.

3.1.3 New drive attachment technology

The same factors that are causing disk densities to improve by 60% per year are resulting in yearly improvements in disk bandwidths of 40% per year [Grochowski and Hoyt, 1996]. This is placing stricter constraints on the physical and electrical design of drive busses (e.g. SCSI or IDE) and often dramatically reducing bus length. As a result, the storage industry has moved to drives that attach to scalable networks, such as Fibre-Channel, a serial, switched, packet-based peripheral network. Fibre-Channel [Benner, 1996] allows long cable lengths, more ports, and more bandwidth. Fibre-Channel-attached hosts and drives communicate via the SCSI protocol [ANSI, 1986, ANSI, 1993], encapsulating SCSI commands over the Fibre-Channel network.

3.1.4 Convergence of peripheral and interprocessor networks

Scalable computing is increasingly based on clusters of workstations. In contrast to the special-purpose, topologically regular, highly reliable, low-latency interconnects of massively parallel processors such as the SP2 and Paragon, clusters typically use Internet protocols over commodity LAN routers and switches. To make clusters effective, low latency network protocols and user-level access to network adapters have been proposed, and a new adapter card interface, the Virtual

Interface Architecture, is being standardized [Maeda and Bershad, 1993, von Eicken et al., 1995, Buzzard et al., 1996, Intel, 1995].

3.1.5 Excess of on-drive transistors

Disk drives have heavily exploited the increasing transistor density in inexpensive ASIC technology to both lower cost and increase performance by developing sophisticated special purpose functional units and integrating them onto a small number of chips. For example, Siemen's TriCore integrated microcontroller and ASIC chip contains a 100 MHz 3-way issue superscalar 32-bit datapath with up to 2 MBytes of on-chip DRAM and customer defined logic in 1998 [TriCore News Release, 1997].

3.2 Two network-attached storage architectures

Eliminating the file server from the data path so that data can be transferred directly from client to storage device would improve bandwidth and eliminate the server as a bottleneck. This requires attaching storage devices to the network and making them accessible to clients. Simply attaching storage to a network leaves unspecified the role of the network-attached storage device in the overall architecture of the distributed file system. The following subsections present two architectures that separate control messaging from data transfers but demonstrate substantially different functional decompositions between server, client and device.

The first case, the simpler network-attached disk design, network SCSI, minimizes modifications to the drive command interface, hardware and software. The second case, network-attached secure disks, leverages the rapidly increasing processor capability of disk-embedded controllers to restructure the drive command interface and offload even more work from the file server.

3.2.1 Network SCSI

NetSCSI is a network-attached storage architecture that makes minimal changes to the hardware and software of SCSI disks. This architecture allows direct data transfers between client and device while retaining as much as possible of SCSI, the current dominant mid- and high-level storage device protocol. This is the natural evolution path for storage devices; Seagate's Barracuda FC is already providing packetized SCSI through Fibre-Channel network ports to directly attached hosts.

File manager software translates client requests into commands to disks, but rather than returning data to the file manager to be forwarded, the NetSCSI disks send data directly to clients,

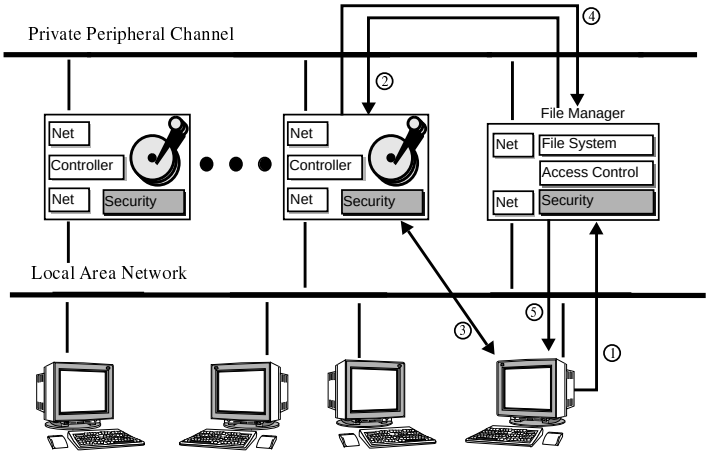


Figure 3.2: Network SCSI (NetSCSI) is a network-attached disk architecture designed for minimal changes to the disk's command interface. However, because the network port on these disks may be connected to a hostile, broader network, preserving the integrity of on-disk file system structure requires a second port to a private (file manager-owned) network or cryptographic support for a virtual private channel to the file manager. If a client wants data from a NetSCSI disk, it sends a message (1) to the distributed file system's file manager which processes the request in the usual way, sending a message over the private network to the NetSCSI disk (2). The disk accesses data, transfers it directly to the client (3), and sends its completion status to the file manager over the private network (4). Finally, the file manager completes the request with a status message to the client (5).

similar to the support for third-party transfers already supported by SCSI [Drapeau et al., 1994]. The efficient data transfer engines typical of fast drives ensure that the drive's sustained bandwidth is available to clients. Further, by eliminating the file manager from the data path, its workload per active client decreases. However, the use of third-party transfer changes the drive's role in the overall security of a distributed file system. While it is not unusual for distributed file systems to employ a security protocol between clients and servers (e.g. Kerberos authentication), disk drives do not yet participate in this protocol.

There are four interesting levels of security within the NetSCSI model [Gibson et al., 1997b]: (1) accident-avoidance with a second private network between file manager and disk, both locked in a physically secure room; (2) data transfer authentication with clients and drives equipped with a strong cryptographic hash function; (3) data transfer privacy with both clients and drives using encryption and; (4) secure key management with a secure processor.

Figure 3.2 shows the simplest security enhancement to NetSCSI: a second network port on each disk. Since SCSI disks execute every command they receive without an explicit authoriza-

tion check, without a second port even well-meaning clients can generate erroneous commands and accidentally damage parts of the file system. The drive's second network port provides protection from accidents while allowing SCSI command interpreters to continue following their normal execution model. This is the architecture employed in the High Performance Storage System [Watson and Coyne, 1995]. Assuming that file manager and NetSCSI disks are locked in a secure room, this mechanism is acceptable for the trusted network security model of NFS.

Because file data still travels over the potentially hostile general network, NetSCSI disks are likely to demand greater security than simple accident avoidance. Cryptographic protocols can strengthen the security of NetSCSI. A strong cryptographic hash function, such as SHA [NIST, 1994], computed at the drive and at the client would allow data transfer authentication (i.e., the correct data was received only if the sender and receiver compute the same hash on the data).

For some applications, data transfer authentication is insufficient, and communication privacy is required. To provide privacy, a NetSCSI drive must be able to encrypt and decrypt data. NetSCSI drives can use cryptographic protocols to construct private virtual channels over the untrusted network. However, since keys will be stored in devices vulnerable to physical attack, the servers must still be stored in physically secure environments.

If NetSCSI disks are equipped with secure coprocessors [Yee and Tygar, 1995], then keys can be protected and all data can be encrypted when outside the secure coprocessor, allowing the disks to be used in a variety of physically open environments.

3.2.2 Network-Attached Secure Disks (NASD)

With network-attached secure disks, the constraint of minimal change from the existing SCSI interface is relaxed. Instead, the focus is on selecting a command interface that reduces the number of client-storage interactions that must be relayed through the file manager, offloading more of the file manager's work without integrating file system policy into the disk.

Common, data-intensive operations, such as reads and writes, go straight to the disk, while less-common ones, including namespace and access control manipulations, go to the file manager. As opposed to NetSCSI, where a significant part of the processing for security is performed on the file manager, NASD drives perform most of the processing to enforce the security policy. Specifically, the cryptographic functions and the enforcement of manager decisions are implemented at the drive, while policy decisions are made in the file manager.

Authorization, in the form of a time-limited capability applicable to the file's map and con-

tents, is provided by the file manager which still maintains control over storage access policy [Gobioff, 1999]. In order to service an access to a file, the logical access must be mapped onto physical sectors. This mapping can be maintained by the file manager and can be provided dynamically as in Derived Virtual Devices (DVD) [Van Meter et al., 1996]. It can also be maintained by the drive. In this latter case, the filesystem authors must surrender detailed control over the layout of the files they create. With the "mapping metadata" that controls the layout of files maintained at the drive, a NASD drive exports a namespace of file-like objects. Files in the directory tree are mapped onto NASD objects by the file manager, but the block management and allocation with a NASD object is the responsibility of the NASD device.

In summary, both NetSCSI and NASD allow direct data transfer. However, NASD decomposes more of the file server's function and delegates part of it to the storage device. This is attractive because it reduces file server load [Gibson et al., 1997b] allowing higher scalability. It also allows computation resources to scale nicely with capacity. The following section discusses the NASD architecture and its properties in more detail.

3.3 The NASD architecture

Like NetSCSI, Network-Attached Secure Disks (NASD) repartition file server function among client, device and residual file server. However, NASD delegates more functions to the client and device to minimize the amount of client-server interactions in the common case. NASD does not advocate that all functions of the traditional file server need to be or should be migrated into storage devices. NASD devices do not dictate the semantics of the highest levels of distributed file system function – global naming, access control, concurrency control, and cache coherence. Nevertheless, NASD devices, as discussed in the next chapter, can provide mechanisms to support the efficient implementation of these high-level functions. High-level policy decisions such as access control, naming and quota management, are still reserved to the residual file server, which is called the file manager.

It is the large market for mass-produced disks that promises to make NASD cost-effective. This mass production requires a standard interface that must be simple, efficient, and flexible to support a wide range of file system semantics across multiple technology generations. The NASD architecture can be summarized in its four key attributes: 1) direct transfer to clients; 2) asynchronous oversight by file managers; 3) secure interfaces supported by cryptography; and 4) the abstraction of variable-length objects.

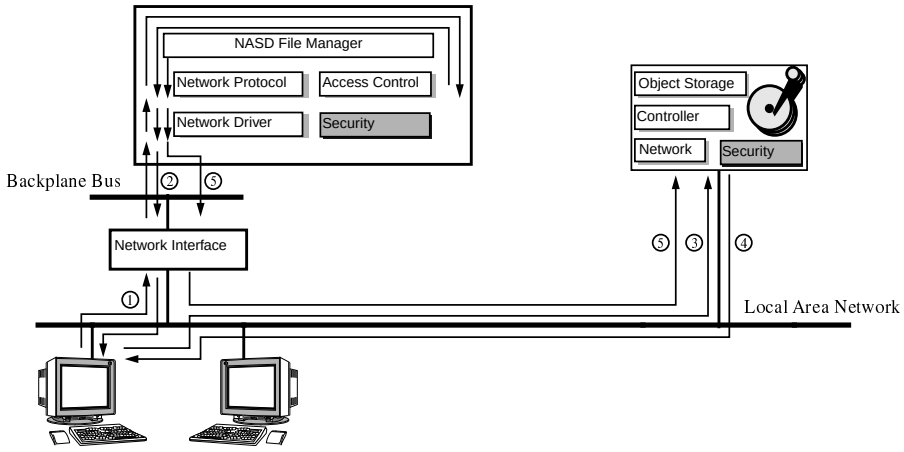


Figure 3.3: Network-attached secure disks (NASD) are designed to offload more of the file system's simple and performance-critical operations. For example, in one potential protocol a client, prior to reading a file, requests access to that file from the file manager (1), which delivers a capability to the authorized client (2). So equipped, the client may make repeated accesses to different regions of the file (3, 4) without contacting the file manager again unless the file manager chooses to force reauthorization by revoking the capability (5).

3.3.1 Direct transfer

Data accessed by a filesystem client is transferred between NASD drive and client without indirection (store-and-forward) through a file server machine. Because control of naming is more appropriate to the higher-level file system, pathnames are not understood at the drive, and pathname resolution is split between the file manager and client. A single drive object can store the contents of a client file, although multiple objects may be logically linked by the file system into one client file. Such an interface provides support for banks of striped files [Hartman and Ousterhout, 1993] or logically-contiguous chunks of complex files [de Jonge et al., 1993]. In any case, the mapping from high-level pathname to an object is performed infrequently and the results of this mapping cached at the client. This allows the client to access the device directly in the common case.

As an example of a possible NASD access sequence, consider a file read operation depicted in Figure 3.3. Before issuing its first read of a file, the client authenticates itself with the file manager and requests access to the file. If access is granted, the client receives the network location of the NASD drive containing the object and a time-limited capability to access the object and for establishing a secure communications channel with the drive. After this point, the client may directly request access to data on NASD drives, using the appropriate capability [Gobioff, 1999].

3.3.2 Asynchronous oversight

Access control decisions made by a file manager must be enforced by a NASD drive. This enforcement implies authentication of the file manager's decisions. The authenticated decision authorizes particular operations on particular groupings of storage. Because this authorization is asynchronous, a NASD device may be required to record an audit trail of operations performed, or to revoke authorization at the file manager's discretion. For a small number of popular authentication systems, Kerberos [Neuman and Ts'o, 1994] for example, NASD drives could be built to directly participate, synchronously receiving a client identity and rights in an authenticated message from a file manager during its access control processing. Derived Virtual Devices use this approach, which is simplified by the availability of authenticated RPC packages, also used by filesystems such as AFS [Satyanarayanan, 1990].

An alternative that does not depend on the local environment's authentication system and that does not depend on the synchronous granting of rights by the file manager, is to employ capabilities similar to the ICAP [Gong, 1989] or Amoeba [Mullender et al., 1990] distributed environments. Capabilities are transported to the device via a client but are only computable/mutable by file manager and NASD drive. Filesystem policy decisions, such as what operations a particular client can perform on a particular set of stored data, are encoded into capabilities by the file manager. These capabilities are given by the file manager to clients. A client presents these capabilities to the NASD drives, who enforce the access control policy by decrypting and examining the contents of the sealed capability, without synchronous recourse to the file manager.

The problem of ensuring security in a NASD-based storage system is the topic of a recent dissertation [Gobioff, 1999]. The key observation here is that ensuring security in a NASD system does not require a central entity that is involved in the critical path of data transfer. File managers are contacted only infrequently and do not interfere with common-case data transfers. Moreover, the NASD device does not dictate how access control is enforced. It provides a mechanism for enforcing authorization which can be used by different types of filesystems.

Efficient, secure communications defeats message replay attacks by uniquely timestamping messages with loosely synchronized clocks and enforcing uniqueness of all received timestamps within a skew threshold of each node's current time. Although all that is needed in a NASD drive is a readable, high-resolution, monotonically increasing counter, there are further advantages to using a clock whose rate is controlled by a network time protocol such as NTP [Mills, 1988]. Such a clock, for example, can be exploited to develop efficient timestamp ordering protocols for concur-

rency control. Such a protocol is described in Chapter 4.

3.3.3 Object-based interface

The set of storage units accessible to a client as the result of an asynchronous access control decision must be named and navigated by client and NASD. Because a NASD drive must enforce access control decisions, a file manager should describe a client's access rights in a relatively compact fashion. While it is possible for the file manager to issue one capability granting a client access to a block or block range, this approach results in a large number of capabilities. Furthermore, it requires a new capability to be issued every time a new block is allocated on a NASD device.

It is therefore advantageous for the capability to refer to compact names that allow a client to simplify its view of accessible storage to a variable length set of bytes, possibly directly corresponding to a file. Such names may be temporary. For example, Derived Virtual Devices (DVD) use the communications port identifier established by the file manager's decision to grant access to a group of storage units [Van Meter et al., 1996]. In the prototype interface, a NASD drive partitions its allocated storage into containers which we call objects.

The NASD object interface enhances the ability of storage devices to manage themselves. Previous research has shown that storage subsystems can exploit detailed knowledge of their own resources to optimize on-disk block layout, prefetching and read-ahead strategies and cache management [English and Stephanov, 1992, Chao et al., 1992, de Jonge et al., 1993, Patterson et al., 1995, Golding et al., 1995].

Magnetic disks are fixed-sized block devices. Traditionally, filesystems are responsible for managing the actual blocks of the disks under their control. Using notions of the disk drive's physical parameters and geometry, filesystems maintain information about which blocks are in-use, how blocks are grouped together into logical objects, and how these objects are distributed across the device [McKusick et al., 1984, McVoy and Kleiman, 1991].

Current SCSI disks offer virtual or logical fixed-sized blocks named in a linear address space. Modern disks already transparently remap storage sectors to hide defective media and the variation in track densities across the disk. By locating blocks with sequential addresses at the location closest in positioning time (adjacent where possible), SCSI supports the locality-based optimizations being computed by the filesystems' obsolete disk model. More advanced SCSI devices exploit this virtual interface to transparently implement RAID, data compression, dynamic block remapping, and representation-migration [Patterson et al., 1988, Wilkes et al., 1996, Holland et al., 1994].

Operation	Arguments	Return values	Description
`CreatePartition()`	partition	status	create a new partition (zero-sized)
`RemovePartition()`	partition	status	remove partition
`ResizePartition()`	partition, new size	status	set partition size
`CreateObj()`	partition, initial attributes	new identifier, attributes, status	create a new object on partition, optionally set its attributes
`RemoveObj()`	partition, identifier	status	remove object from partition
`GetAttr()`	partition, identifier	attributes, status	get object attributes
`SetAttr()`	partition, identifier, new attributes	new attributes, status	change attributes, retrieving resulting attributes when complete
`ReadObj()`	partition, identifier, regions	data, length, status	read from a strided list of scatter-gather regions in an object
`WriteObj()`	partition, identifier, regions, data	length, status	write to a strided list of scatter-gather regions in an object

Table 3.1: A subset of the NASD interface. Storage devices export multiple partitions, each a flat-object space, accessed through a logical read-write interface. Objects are created within a specific partition. Each object has attributes that can be queried and changed. Some attributes are maintained by the NASD device, such as last access time. Some attributes are "filesystem-specific" and are updated by the filesystem code through a `SetAttr()` command. These attributes are opaque to the NASD device.

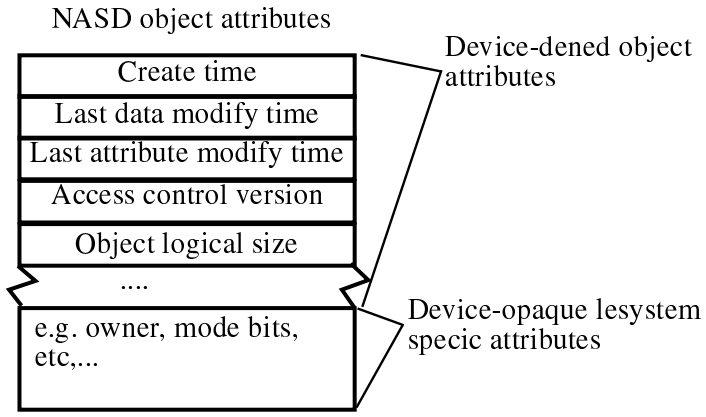


Figure 3.4: Structure of a NASD object's attributes. Some attributes are maintained by the device such as create time, object size, and last data modify time. In addition to NASD maintained attributes, each object has an access control version variable and a large filesystem-specific attribute space (256 bytes in our prototype) which is used by the filesystem for its own purposes.

The NASD interface abandons the notion that file managers understand and directly control storage layout. Instead, NASD drives store variable-length, logical byte streams called objects. Filesystems wanting to allocate storage for a new file request one or more objects to hold the file's data. Read and write operations apply to a byte region (or multiple regions) within an object. The layout of an object on the physical media is determined by the NASD drive. To exploit the locality decisions made by a file manager, sequential addresses in NASD objects should be allocated on the media to achieve fast sequential access. For inter-object clustering, NASD breaks from SCSI's global address space and adopts a linked list of objects where proximity in the list encourages proximity on the media, similar to the Logical Disk model [de Jonge et al., 1993].

In addition to being data stores, NASD objects maintain associated metadata, called object attributes. Figure 3.4 depicts an example of NASD object attributes. Some attributes such as create time, the last time the data was written, or the last time the attributes were modified are maintained by the device and can not be directly manipulated by the external systems. These attributes are updated indirectly when the object is created, when the object's data is written to, or when any attributes are modified respectively. In addition to the NASD maintained attributes, listed fully in [Gibson et al., 1997a], each NASD object has a large "filesystem specific" attribute space (256 bytes in our prototype). These attributes are opaque to the NASD device which treats them as a byte string. These are updated and manipulated by the filesystem for its own purposes. For instance, a UNIX like filesystem may use these attributes to store the mode bits or the owner and group identifiers.

The important subset of the NASD interface is summarized in Table 3.1; a more complete and detailed description of the NASD interface is provided in Gibson *et al*. [Gibson et al., 1997a].

3.4 The Cheops storage service

A filesystem using NASD devices allows clients to make direct read and write accesses to the storage devices. Simulation studies reported in [Gibson et al., 1997b] demonstrated that an NFS file server can support an order of magnitude more clients if NASD based devices are used in lieu of server-attached devices. AFS file servers were shown to support three times more clients.

Although NFS and AFS were later ported to use NASD to validate the NASD interface and gain further experience, scalability experiments with large client populations were not carried out. The design of AFS and NFS over NASD is reported in [Gibson et al., 1998]. In their NASD ports, both NFS and AFS over NASD mapped a single file or directory in the directory structure onto a single NASD object on a single device. Large file transfer could not benefit from parallel striped transfers. Furthermore, RAID across devices was not supported.

To exploit the high bandwidth possible in a NASD storage architecture, the client-resident portion of a distributed filesystem needs to make large, parallel data requests across multiple NASD drives and to minimize copying, preferably bypassing operating system file caches. Cheops is a storage service that can be layered over NASD devices to accomplish this function. In particular, Cheops was designed to provide this function transparently to higher level filesystems.

3.4.1 Cheops design overview

Cheops implements storage striping and RAID functions but not file naming and other directory services. This maintains the traditional "division of concerns" between filesystems and storage subsystems, such as RAID arrays. Cheops performs the function of a disk array controller in a traditional system. One of the design goals of Cheops was to scale to a very large numbers of nodes. Another goal was for Cheops to export a NASD interface, so that it can be transparently layered below filesystems ported to NASD.

Cheops allows higher-level file systems to manage a single logical object that is served by the Cheops storage management system [Gibson et al., 1998]. Cheops exports the illusion of a single virtual NASD device with the aggregate capacity of the underlying physical NASD devices. To the clients and file managers, there appears to be a single NASD device. This device is accessed via

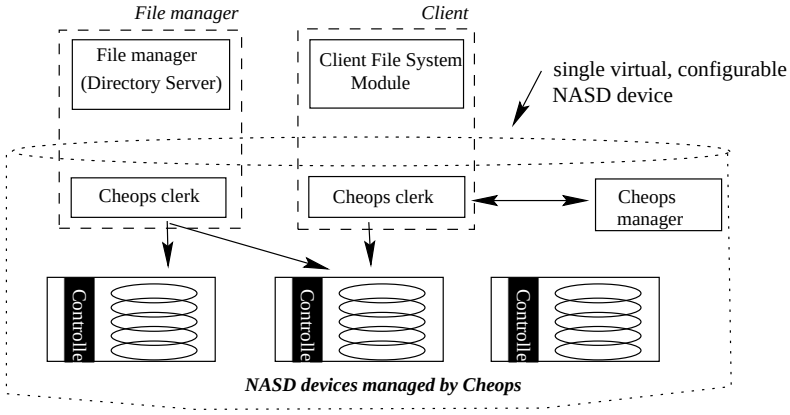
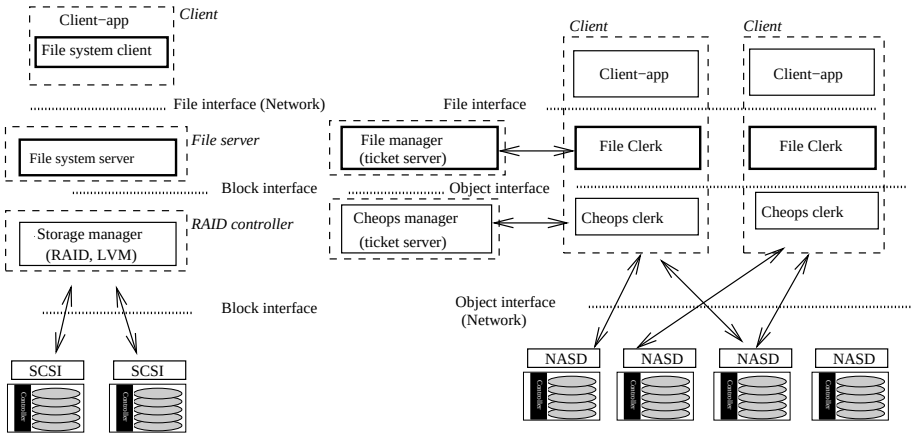


Figure 3.5: Recursing the NASD interface. Cheops exports a virtual NASD interface while managing multiple physical NASDs underneath. Cheops can be regarded as a logical volume manager and a software RAID controller that is specialized to NASD objects.

a local surrogate, called the Cheops clerk. Figure 3.5 illustrates how Cheops clerks export to the client machines the illusion of a single virtual NASD device.

The basic design goal of Cheops is to enable clients to perform direct parallel accesses on the NASD devices. This requires the striping function to be implemented in the client. Precisely, the Cheops clerk is responsible for mapping a logical access to the virtual NASD device onto a parallel access to the physical NASD devices. Figure 3.6 depicts how NASD, Cheops, and filesystem code fit together to enable direct parallel transfers to the client while maintaining the NASD abstraction to the filesystem. The figure contrasts this architecture to the traditional server-attached disk (SAD) architecture. In both architectures, an application request is first received by the client's representation of the filesystem (file clerk in Figure 3.6). In the SAD system, this request is forwarded to the file server as a file-level request. This request is processed by the server-side filesystem and mapped onto a block-level request to the local disk array controller or logical volume manager. The RAID controller maps this logical block access onto one or more physical block access to the server-attached storage devices.

In the NASD system, the application request is received by the local client filesystem clerk. The access is mapped onto a NASD object access by contacting the local "object cache" or the file manager in case of a cache miss. Once the object is identified, the file access translates to an object access on the virtual NASD device exported by Cheops. The local Cheops clerk takes this object access and maps it onto one or more physical object accesses possibly consulting the Cheops manager if a virtual to physical metadata mapping is not cached. These physical accesses are sent



(a) Server−attached disk architecture (b) NASD architecture

Figure 3.6: Function decomposition in NASD compared to traditional file server based storage. At each client, a file clerk performs file caching and namespace mapping between high-level filenames and virtual storage objects. The storage clerk on the client receives logical accesses to a virtual storage object space, and maps the accesses onto physical accesses to the NASD objects. Parallel transfers are then carried out from multiple NASDs into the storage clerk.

out on the network. The Cheops clerk and managers at this layer implement the function of the RAID controller in the SAD system.

Cheops managers manage the metadata maps and initiate management tasks such as reconstruction, backup and migration. Upon receipt of a lookup request from a clerk, the Cheops manager returns the mapping of the higher level object onto a set of underlying NASD objects and the appropriate capability list. This mapping and capability list is used by the clerk to perform the accessed to the NASD devices. Cheops-exported objects have attributes as required by the NASD interface. These attributes can be embedded in one of the physical NASD objects or in a separate object. The first solution does not require allocating additional physical NASD objects but can disrupt the alignment of client accesses. The second solution allocates one NASD object per partition or per group of objects to store the attributes of the virtual objects in that partition. This is more desirable since it does not disrupt the alignment of client accesses at the expense of slightly more implementation complexity.

This layered approach does not adversely affect the overall file manager's control because it is already largely asynchronous. For example, the file manager revokes a high-level capability in a physical NASD by updating the attributes of the object on the NASD device. Because the

device compares attributes encoded in the capability with these on the device during every access, modifying the attributes amounts to revoking the capability. This is implemented in Cheops as follows. The file manager invokes a `SetAttr()` to the virtual NASD device (through the local clerk) to modify the access version number attribute. This request is relayed by the local clerk to the Cheops manager. The Cheops manager revokes access to each physical object by updating their attributes before acknowledging a response to the file server. This is implemented by the Cheops manager as follows. Recall that the clerks at many clients may have cached the "virtual to physical" mapping and the capabilities that allow them to perform access to the underlying physical object. The manager can disallow these clients from accessing storage immediately by modifying the attributes of the underlying physical objects. Thus, to revoke a capability, the file manager sends a `SetAttr()` to the local clerk, the surrogate of all virtual objects managed by Cheops. This request is forwarded by the clerk to the Cheops manager. The manager in turns sends `SetAttr()` commands to the underlying physical NASD objects to modify their attributes, invalidating all outstanding physical capabilities.

3.4.2 Layout protocols

Cheops clerks cache the layout maps for recently accessed virtual objects locally. The layout map describes how a virtual object is mapped onto physical NASD objects, and provides capabilities to enable access to each physical NASD. Furthermore, the layout maps describe how the object should be read and written. For example, a map may specify that an object is striped across four NASD objects with a stripe unit size of 64KB according to a RAID level 0 layout.

A Cheops clerk uses the layout map to read and write the objects. In the current prototype implementation, the Cheops clerk as well as the manager contain code that allow them to access a statically defined set of possible layouts and their associated access algorithms. Chapter 5 describes a framework that enables the Cheops clerk to be dynamically extended to perform accesses to new architectures introduced by a Cheops manager.

The layout cached by Cheops clerks may change as a result of a failure or a management operation. Disk failures require the use of different access protocols to read and write a virtual object. Furthermore, the layout map may change because of a storage migration initiated by a manager. To maintain the coherence of maps, layouts maps are considered valid for a limited period of time. After that period, called the lease period, passes, the layout map is considered invalid and must be refreshed by the clerk. The clerk refreshes the map by contacting a storage manager to find out if

the map has changed.

When a Cheops manager desires to change a layout map, for example because storage has migrated to a new device, it sends explicit messages to the clients to invalidate their cached maps. Messages must be also sent to the devices to revoke access, by invalidating the capabilities cached at the clients. Because the layout maps expire after a certain period, crashed clients and network partitions are handled easily in the same way. If any client does not acknowledge the invalidation, the manager waits until the lease expires and then performs its layout change. Fault recovery and concurrency control in Cheops deserves a deeper discussion, which is the subject of the following chapter.

3.4.3 Storage access protocols

Cheops clerks contain statically linked code that allows them to perform read and write accesses to several RAID architectures, namely RAID levels 0, 1 and 5. In Cheops, it is possible for multiple storage access from many clients to be ongoing concurrently at the same time. Furthermore, a device or host failure may occur in the middle of the execution of a storage access. Correctness must be ensured in the midst of this concurrency and in the case of untimely failures. The following chapter addresses this concurrency control and recovery problem. For the rest of this chapter, we will focus only on normal case performance.

3.4.4 Implementation

The Cheops clerk is implemented as a library which can be linked in with user-level filesystems and applications. It uses multiple worker threads from a statically sized thread pool to initiate multiple parallel RPCs to several NASD devices. Data returned by the NASD devices is copied only once from the communication transport's buffers to the application space. The Cheops clerk does not induce an extra copy. Instead, it deposits data returned by each device directly in the application buffer at the proper offset.

Cheops clerks maintain a least-recently-used cache of virtual-to-physical mappings. The clerks also maintain a cache of NASD device tables. These tables map a NASD device id onto an IP address and a port and a communication handle if a channel is still open between the clerk and the device. Cheops used initially DCE RPC over UDP/IP as the communication transport. A later implementation used a lightweight RPC package with better performance [Gibson et al., 1999].

Cheops builds on several earlier prototypes in network filesystems and in striping storage sys-

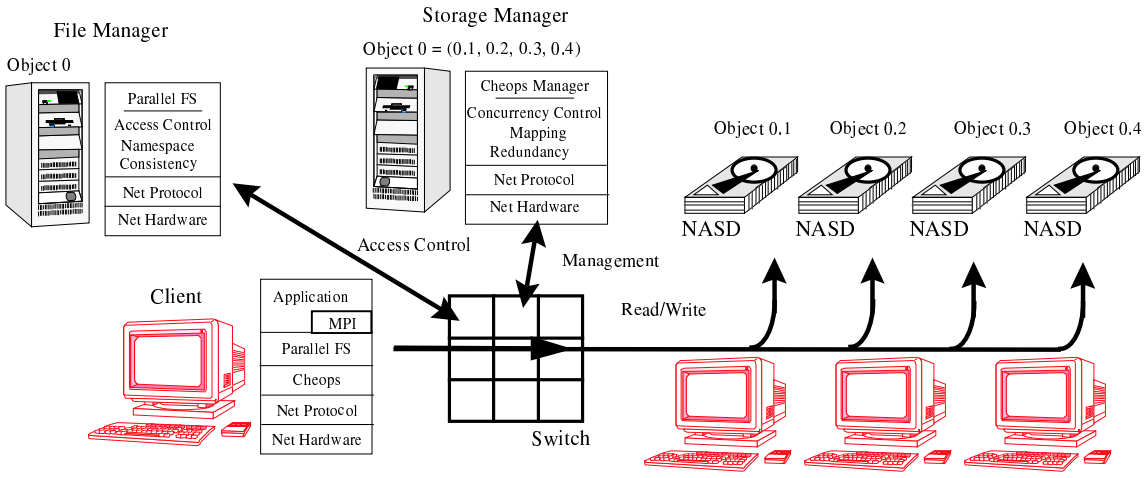


Figure 3.7: A NASD-optimized parallel filesystem. NASD PFS is Cheops with a simple filesystem layer on top of it. This filesystem layer performs data and name caching. NASD PFS is used in conjunction with MPI for parallel applications in a cluster of workstations. The filesystem manages objects which are not locally backed by data. Instead, they are backed by a storage manager, Cheops, which redirects clients to the underlying component NASD objects. Our parallel filesystem extends a simple Unix filesystem interface with the SIO low-level interface [Corbett et al., 1996] and inherits a name service, directory hierarchy, and access controls from the filesystem.

tems [Cao et al., 1994, Cabrera and Long, 1991, Lee and Thekkath, 1996]. In its RAID level 0 implementation, it serves to demonstrate that the NASD interface can be effectively and efficiently virtualized, so that filesystems do not have to know about the underlying storage management that is ongoing inside Cheops.

A parallel file system was used to demonstrate the bandwidth scaling advantages of NASD arrays, as depicted in Figure 3.7. Such a file system was used because high bandwidth applications are often parallel applications. MPICH version 2 [Forum, 1995] for parallel program communications was used to implement a simple parallel file system, NASD PFS. NASD PFS is implemented in a library which in turns links to the Cheops clerk library.

Because the NASD prototype used DCE on UDP for data transfer, the Cheops clerk uses DCE as well. Parallel cluster applications often employ low-latency system area networks, which, by offering network adapter support for protocol processing, can dramatically shorten the code overhead for bulk data transfer. Our prototype did not have such on-board protocol processing [Gibson et al., 1998]. Nevertheless, the application described in this Chapter are so data-intensive that they can accommodate the computational overhead of DCE RPC over UDP and still exhibit significant speed-ups.

3.5 Scalable bandwidth on Cheops-NASD

To evaluate the scalability of NASD/Cheops, a basic prototype implementation of Cheops to work with the prototype NASD device. The scalability of NASD/Cheops was compared to that of a traditional NFS file server with disks directly attached to the server. The hardware configurations of both systems were carefully constructed to provide a fair comparison.

Both synthetic benchmarks (read/write), as well as I/O intensive applications such as search and data mining were designed, implemented and evaluated on SAD and NASD architectures.

3.5.1 Evaluation environment

Our experimental testbed consists of clients, file managers and NASD drives connected by a network. The NASD prototype is described in [Gibson et al., 1998] and implements its own internal object access, cache, and disk space management modules (a total of 16,000 lines of code) and interacts minimally with Digital UNIX. For communications, the prototype uses DCE RPC 1.0.3 over UDP/IP.

The scalability of NASD/Cheops was compared to that of a traditional NFS file server with disks directly attached to the server. The hardware for the NASD/Cheops system was composed of the following:

- NASDs: A NASD consists of SCSI disks (Seagate Medallist ST52160) attached to an old workstation. Two SCSI disks, with an average transfer rate of 3.5-4MB/sec each, were used as the magnetic storage media. Object data is striped across both disks yielding an effective sequential transfer rate of up to 7.5 MB/sec. The workstation is a DEC Alpha 3000/400 (133 MHz, 64 MB, Digital UNIX 3.2g). The SCSI disks are attached to the workstation via two 5 MB/s SCSI busses. The performance of this five year old machine is similar to what is available in high-end drives today and to what is predicted to be available in commodity drive controllers soon. We use two physical drives managed by a software striping driver to approximate the 10 MB/s rates we expect from more modern drives.

- Clients: Digital AlphaStation 255 machines (233Mhz, 128MB, Digital UNIX 3.2g-3) were used as clients.

- File manager: One Alpha 3000/500 (150 MHz, 128 MB, Digital UNIX 3.2g-3) was used as a file manager.

- Network: The clients and the NASDs were connected by a 155 Mb/s OC-3 ATM network (Digital Gigaswitch/ATM).

The traditional NFS server configuration is referred to as server-attached-disk (SAD) configuration. The much more effective hardware used for the SAD system was composed of the following:

- File Server: A digital AlphaStation 500/500 was used as a "traditional file server" (500 MHz, 256 MB, Digital UNIX 4.0b). The file server machine was connected via two OC3 ATM links to a Digital Gigaswitch/ATM, with half of the clients using each link.
- Storage devices: Eight SCSI disks (Seagate ST34501W, 13.5 MB/s) were attached to the file server workstation via two 40 MB/s Wide UltraSCSI busses, so that the peripheral SCSI links into the workstation can deliver the bandwidth of the disks.
- Clients: As in the NASD case, AlphaStation 255 machines were used as clients.
- Network: As in the NASD case, the clients and the server were connected by a 155 Mb/s an ATM network (Digital Gigaswitch/ATM).

3.5.2 Raw bandwidth scaling

The parallel application employed in this demonstration is a data mining application that discovers association rules in sales transactions [Agrawal and Schafer, 1996] with synthetic data generated by a benchmarking tool. This application "discovers" rules of the form "if a customer purchases item A and B, then they are also likely to purchase item X" to be used for store layout or inventory decisions. It does this in several full scans over the data, first determining the items that occur most often in the transactions (the 1-itemsets) and then using this information to generate pairs of items that occur often (2-itemsets) and larger groupings (k-itemsets).

Our parallel implementation avoids splitting records over 2 MB boundaries and uses a simple round-robin scheme to assign 2 MB chunks to clients. Each client is implemented as four producer threads and a single consumer. Producer threads read data in 512 KB requests (which is the stripe unit for Cheops objects in this configuration) and the consumer thread performs the frequent sets computation, maintaining a set of itemset counts that are combined at a single master client. This threading maximizes overlapping and storage utilization.

Figure 3.8 shows the bandwidth scalability of the most I/O bound of the phases (the generation of 1-itemsets) processing a 300 MB sales transaction file. A single NASD provides 6.2 MB/s per

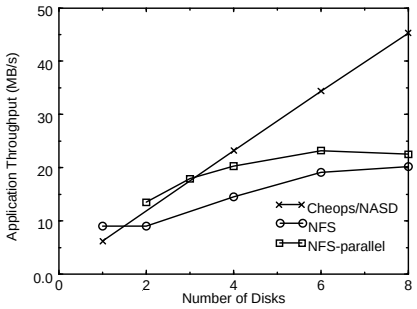


Figure 3.8: Scaling of a parallel data mining application. The aggregate bandwidth computing frequent sets from 300 MB of sales transactions is shown. The NASD line shows the bandwidth of N clients reading from a single NASD PFS file striped across N NASD drives and scales linearly to 45 MB/s. All NFS configurations show the maximum achievable bandwidth with the given number of disks, each twice as fast as a NASD, and up to 10 clients spread over two OC-3 ATM links. The comparable NFS line shows the performance of all the clients reading from a single file striped across N disks on the server and bottlenecks near 20 MB/s. This configuration causes poor read-ahead performance inside the NFS server, so we add the NFS-parallel line where each client reads from a replica of the file on an independent disk through the one server. This configuration performs better than the single file case, but only raises the maximum bandwidth from NFS to 22.5 MB/s.

NASD drive and our array scales linearly up to 45 MB/s with 8 NASD drives. In comparison, we also show the bandwidth achieved when NASD PFS fetches from a single higher-performance traditional NFS file instead of a Cheops NASD object. The NFS file server is an AlphaStation 500/500 (500 MHz, 256 MB, Digital UNIX 4.0b) with two OC-3 ATM links (half the clients communicate over each link), and eight Seagate ST34501W Cheetah disks (13.5 MB/s) attached over two 40 MB/s Wide UltraSCSI busses. Using optimal code, this machine can internally read as much as 54.1 MB/s from these disks through the raw disk interface, doing nothing with this data.

The graph of Figure 3.8 shows two application throughput lines for this server. The line marked NFS-parallel shows the performance of each client reading from an individual file on an independent disk on the one NFS file server and achieves performance up to 22.5 MB/s. The results show that an NFS server (with 35+ MB/s of network bandwidth, 54 MB/s of disk bandwidth and a perfect sequential access pattern on each disk) loses much of its potential performance to CPU and interface limits. In comparison, each NASD is able to achieve 6.2 MB/s of the raw 7.5 MB/s available from its underlying dual Medallists. Finally, the line marked NFS is actually the most comparable configuration to the NASD experiment. It shows the bandwidth when all clients read from a single NFS file striped across N disks. This configuration, at 20.2 MB/s, is slower than NFS-parallel

because its prefetching heuristics fail in the presence of multiple request streams to a single file.

In summary, NASD PFS on Cheops delivers nearly all of the bandwidth of the NASD drives, while the same application using a powerful NFS server fails to deliver half the performance of the underlying Cheetah drives. The difference in performance is expected to widen with larger sizes, as Cheops continues to exploit the available network bandwidth for aggregate parallel transfers. Server-based filesystems are limited, on the other hand, by the performance of the server machine. In very large systems, multiple file servers must be used. This complicates administration. Cheops can deliver a single virtual storage abstraction from a large collection of storage devices. Nevertheless, large systems bring challenges for both architectures. The upcoming chapter discusses how a shared storage array, like Cheops, can ensure scalable concurrency control and fault-tolerance in very large systems.

3.6 Other scalable storage architectures

Scalable storage is not a new goal. Several storage systems have been designed with the goal of delivering high-bandwidth to clients. Several others have focussed on scaling to large sizes. This section briefly reviews the alternative architectures that have been proposed in the literature and how they compare to Cheops/NASD.

Previous storage systems reviewed in this section are organized into three categories. The first corresponds to systems which removed the file server machine from the data transfer path, thereby decoupling control from data transfer. The second corresponds to systems that striped data across multiple storage nodes to achieve high I/O data rates, while the third corresponds to systems that distributed the storage management and access functions to the nodes while tolerating the node failures.

3.6.1 Decoupling control from data transfer

The availability of high-bandwidth disk arrays have recently highlighted the fileserver as the chief bottleneck in storage access. To eliminate the file server from the data path, several systems proposed decoupling control messaging in filesystems from actual data transfer. Examples include HPSS [Watson and Coyne, 1995] and RAID II [Drapeau et al., 1994] which focussed on moving the file server machine out of the data transfer path. These systems still relied on synchronous oversight of the file server on each access operation. However, data is transferred directly from storage

to the client network without being copied through the file server machine.

RAID II

After building the initial redundant disk array prototype in 1989, the RAID group at U.C. Berkeley discovered that parallel transfers from a disk array can achieve higher bandwidth than can be delivered by the memory system of the host workstation. The second RAID prototype, RAID II [Drapeau et al., 1994], was designed to address this bottleneck and deliver as much as possible of the array's bandwidth to file server clients.

A custom-built crossbar memory system was used to connect disks directly to a high-speed network to which clients are attached. This allowed parallel transfers from the disks to the clients to use the high-speed crossbar and network without going through the server's memory. The RAID II prototype was used by a log-structured filesystem to deliver high read and write bandwidths to data-intensive applications. The RAID II system was shown to deliver up to 31 megabytes per second for sequential read operations, and 20 megabytes for large random accesses.

Mass Storage Reference Model

The Mass Storage System Reference Model, an early architecture for hierarchical storage subsystems, has advocated the separation of control and data paths for almost a decade [Miller88, IEEE94]. The Mass Storage System Reference Model was implemented in the High Performance Storage System (HPSS) [Watson and Coyne, 1995] and augmented with socket-level striping of file transfers over the multiple network interfaces found on mainframes and supercomputers.

HPSS preserved the fixed block interface to storage and continues to rely on synchronous oversight of commands by the file manager. It constitutes a first step in totally removing the file server from the data path. Cheops over NASD builds on this idea of decoupling control from data transfers. It more aggressively avoids the file manager, however, by allowing clients to cache long-term mappings which can be used to directly translate a high-level file access to a storage device access without recourse the file manager.

3.6.2 Network-striping

To allow client applications high-bandwidth access to storage, several storage systems such as Swift [Cabrera and Long, 1991], Zebra [Hartman and Ousterhout, 1993] and Tiger [Bolosky et al., 1996] introduced the idea of striping across multiple storage servers across a network.

Swift

To obtain cost-effective scalable bandwidth on a local area network, data must be striped across the network and across multiple servers. Swift [Cabrera and Long, 1991] is an early striping storage system that partitioned client data across multiple storage servers to provide high aggregate I/O rates to its clients. A "storage manager" decided file layout, stripe unit size and reliability mechanism. Users provided preallocation info such as size, reliability level, data rate requirements.

Cheops is similar to Swift in its architecture. Cheops differs from Swift in that it is designed to work with the NASD object interface. Furthermore, Swift like Zebra, did not focus on the problem of ensuring highly concurrent write access to shared RAID storage. The protocols in the next chapter are devoted to addressing the concurrent RAID update problem which arises in distributed RAID systems.

Zebra

Another network filesystem that implemented direct client access to striped storage is the Zebra striped network filesystem [Hartman and Ousterhout, 1993], which is a log-structured file system striped over network storage nodes. Zebra stripes client logs of recent file system modifications across network storage servers and uses RAID level 4 to ensure fault-tolerance of the each log. By logging many recent modifications before initiating a parallel write to all storage servers, Zebra avoids the small write problem of RAID level 4.

As in the log-structured file system [Rosenblum, 1995], Zebra uses stripe cleaners to reclaim free space. Zebra assumes clients are trusted; each time a client flushes a log to the storage servers, it notifies the file manager of the new location of the file blocks just written through a message, called a "delta" which is post-processed by the manager to resolve conflicts with the cleaner. Zebra lets each clients write to the storage servers without going through the server and coordinates the clients and the cleaners optimistically with file server post-processing. By making clients responsible for allocating storage for new files across the storage servers, Zebra effectively delegates to the clients the responsibility of low-level storage management.

While Zebra is a filesystem which integrates storage management and directory services, Cheops is a storage system that does not implement any namespace or directory services. Cheops was designed to provide the abstraction of virtual NASD objects from a collection of physical NASD devices.

Berkeley xFS

The limitations of using a single central fileserver have been widely recognized. xFS attempted to address some of the problems of a central file server by effectively replicating or distributing the file server among multiple machines [Anderson et al., 1996]. To exploit the economics of large systems resulting from the cobbling together of many client purchases, the xFS file system distributes code, metadata and data over all clients, eliminating the need for a centralized storage system [Dahlin95]. This scheme naturally matches increasing client performance with increasing server performance. Instead of reducing the server workload, however, it takes the required computational power from another, frequently idle, client.

xFs stripes data across multiple client machines and uses parity codes to mask the unavailability of clients. The xFS approach uses a model where an additional machine does store-and-forward, rather than allowing the clients to directly communicate with storage.

Microsoft Tiger Video Fileserver

Tiger is a distributed fault-tolerant fileserver designed for delivering video data with real-time guarantees [Bolosky et al., 1996]. Tiger is designed to serve data streams at a constant rate to a large number of clients while supporting more traditional filesystem operations.

Tiger uses a collection of commodity computers, called cubs, networked via an ATM switch to deliver high-bandwidth to end clients. Data is mirrored and striped across all cubs and devices for fault-tolerance. Storage devices are attached to the cubs which buffer the bursty disk transfers and deliver smooth constant-rate streams to clients.

The main problem addressed by the Tiger design is that of efficiently balancing user load against limited disk, network and I/O bus resources. Tiger accomplishes this by carefully allocating data streams in a schedule that rotates across the disks. Cheops/NASD, like Tiger, uses switched networks and striped transfer to deliver high-bandwidth. Cheops function is provided by the intermediate computers in Tiger, where it is specialized for video service.

3.6.3 Parallel and clustered storage systems

Several systems distributed storage across nodes. In Swift, for example, storage is distributed across multiple nodes or “agents”. However, a single storage manager is responsible for allocation and management. This leads to a single point of failure. Systems such TickerTAIP [Cao et al., 1994] and Petal [Lee and Thekkath, 1996] attempted to distribute storage access and management functions

to the participating storage nodes. Both systems avoid the single point of failure of the storage manager.

TickerTAIP

TickerTAIP is a parallel disk array architecture which distributed the function of the centralized RAID contoller to the storage devices that make up the array [Cao et al., 1994]. TickerTAIP avoids the performance bottleneck and single point of failure of single disk array controllers. A host can send a request to any node in the disk array. The RAID update protocols are executed by the nodes. To ensure proper fault-tolerance in the event of a node failure, nodes log updates to other nodes before writing data to the platters.

Cheops/NASD is similar to TickerTAIP in that accesses can be initiated by multiple nodes. Cheops clerks can be concurrently accessing shared devices. The TickerTAIP node controllers can be concurrently accessing the storage devices. Cheops however allows any client on the network with the proper access authorization to access storage, while TickerTAIP requires hosts to send the request to one of the array nodes, which in turn performs the access on behalf of the host. Moreover, while TickerTAIP distributes function to the array nodes, Cheops involves a larger number of client and storage nodes. The protocols in the next chapter focus on how synchronization protocols for RAID updates can scale to such a large number of nodes.

Petal

Petal is a distributed storage system that uses closely cooperating commodity computers to deliver a highly-available block-level storage servers [Lee and Thekkath, 1996]. Petal nodes consist of commodity computers with locally attached disks. User data is striped across the nodes and mirrored for fault-tolerance. Petal allows blocks to be migrated between nodes by exporting a virtual block abstraction. Petal nodes collectively maintain the mapping from virtual to physical blocks and migrate blocks to balance load and use newly added devices.

Petal is similar to Cheops in that it introduces a level of indirection to physical storage to allow management operations to be undertaken transparently. Furthermore, Petal similarly uses striping across nodes. Although Petal uses commodity computers as storage servers, the functions executed by a Petal node can conceivably be implemented in the storage device controller. Hence, Petal can enable direct storage access like Cheops/NASD.

Cheops/NASD differs from Petal in several aspects. First, Cheops/NASD offers an object inter-

face which can be used to refer to a collection of related blocks. In this way, a filesystem can delegate the responsibility of storage management and allocation to the Cheops/NASD system. Petal, however, exports a block-level interface. This requires filesystems to perform traditional block allocation and de-allocation. Furthermore, Cheops/NASD assumes untrusted clients and therefore uses a capability-based security protocol for access control. Finally, Cheops supports RAID across devices. RAID has major implications on storage system design and performance. A study of the performance of RAID protocols in a Cheops-like system is conducted in simulation and reported on in the next chapter.

3.7 Summary

Storage bandwidth requirements continue to grow due to rapidly increasing client performance, new, richer content data types such as video, and data intensive applications such as data mining. All high-bandwidth solutions to date incur a high overhead cost due to existing storage architectures' reliance on file servers as a bridge between storage and client networks.

With dramatic performance improvements and cost reductions in microprocessors anticipated for the foreseeable future, high-performance personal computers will continue to proliferate, computational data sets will continue to grow, and distributed filesystem performance will increasingly impact human productivity and overall system acquisition and operating cost.

This chapter reported on a Network-Attached Secure Disks (NASD) architecture that enables cost-effective bandwidth scaling. NASD eliminates the server bottleneck by modifying storage devices so they can transfer data directly to clients. Further, NASD repartitions traditional file server functionality between the NASD drive, client and server. NASD does not advocate that all functions of the traditional file server need to be or should be migrated into storage devices. This chapter describes how legacy filesystems such as NFS and AFS can be ported to NASD.

NASD enables clients to perform parallel data transfers to and from the storage devices. This chapter also describes a storage service, Cheops, which implements such function. Real applications running on top of a Cheops/NASD prototype demonstrate that NASD can provide scalable bandwidth. This chapter reports on experiments with a data mining application for which we achieve about 6 MB/s per client-drive pair in a system of up to 8 drives, for a total aggregate bandwidth of 45 MB/s compared to the 22.5 MB/s achievable with NFS.

For the client to conduct parallel transfers directly to and from the NASD devices, it must cache the stripe maps and capabilities required to resolve a file-level access and map it onto accesses to

the physical NASD objects. The Cheops approach is to virtualize storage layout in order to make storage look more manageable to higher-level filesystems. Cheops avoids reinlisting servers to synchronously resolve the virtual to physical mapping by decomposing and distributing its access functions and management functions such that access function is executed at the client where the request is initiated. Cheops managers are responsible for authorization and oversight operations so that the participating clients always do the right thing. This essentially "recurses" the NASD interface by decomposing the Cheops functions similarly to that of a filesystem ported to NASD.

Despite the encouraging scaling results, a few challenges remain. First, RAID level 5 requires synchronization (stripe locking) to avoid corrupting shared parity codes when concurrent accesses to the same stripe are ongoing at the same time. More generally, Cheops requires synchronization protocols to coordinate access to shared devices by clerks when storage is being migrated or reconstructed. Moreover, the transitions when failures are detected have to be handled correctly. All these protocols must scale to the ambitious sizes that the NASD architecture allows. This problem is discussed in the following chapter.

Chapter 4

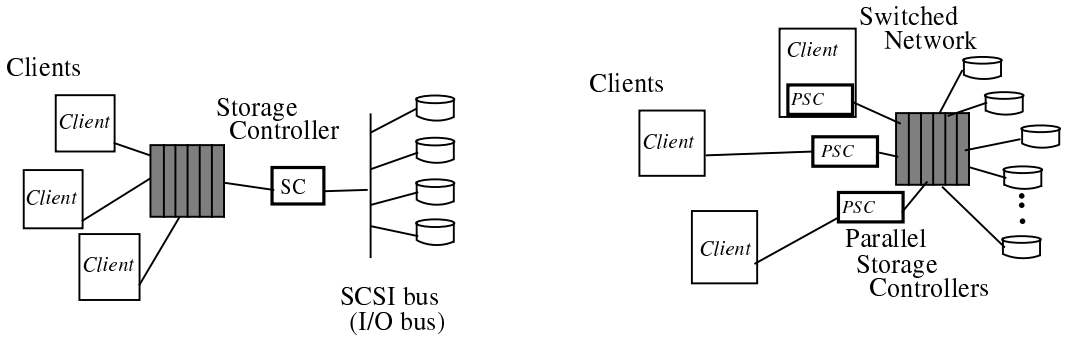
Shared storage arrays

Network attached storage enables parallel clients to access potentially thousands of shared storage devices [Gibson et al., 1998, Lee and Thekkath, 1996] over cost-effective and highly-reliable switched networks [Horst, 1995, Benner, 1996]. Logically, this gives rise to a shared storage array with a large number of devices and concurrent storage controllers. Cheops, described in the previous chapter, is an example of a shared storage array. Cheops distributes the intelligence to access storage to the clients. In general, in a shared array, each client acts as the storage controller on behalf of the applications running on it, achieving scalable storage access bandwidths.

In such systems, clients perform access tasks (read and write) and management tasks (storage migration and reconstruction of data on failed devices). Each task translates into multiple phases of low-level device I/Os, so that concurrent clients accessing shared devices can corrupt redundancy codes and cause hosts to read inconsistent data. This chapter is devoted to the problem of ensuring correctness in a shared storage array. The challenge is guaranteeing correctness without compromising scalability.

The chapter is organized as follows. Section 4.1 motivates this research by describing some hazards that can occur as a result of races or untimely failures in shared arrays. Section 4.2 defines a shared storage array in general terms. This general definition allows the solutions described in this chapter to be applied to any system that allows sharing of storage devices, regardless of the storage device interface (NASD or SCSI). Section 4.3 describes an approach which is based on breaking down the tasks that storage controllers perform into short-running two-phased transactions, called base storage transactions, or BSTs for short.

The task of ensuring correctness is then reduced to guaranteeing that BSTs have a few desirable properties. Section 4.4 describes these properties and why they are needed. One key property of BSTs is that their execution is serializable. Section 4.5 focuses on protocols that ensure this



(a) Traditional storage systems

(b) Scalable storage arrays

Figure 4.1: Traditional storage systems (a) use a single controller. Shared arrays (b) use parallel cooperating controllers to access and manage storage. These storage controllers will be equivalently referred to as controllers, clients or hosts in this discussion. The term NASD or storage device will be used to refer to the device and its on-board controller.

serializability property. Section 4.6 discusses extensions of these protocols to shared arrays that perform distributed caching at the controllers. The other key property of BSTs is that they ensure array consistency upon recovery from untimely failures. Section 4.7 is devoted to a discussion of recovery protocols that ensure this consistency property. Section 4.8 contains a discussion of how the proposed protocols can be optimized or generalized. Section 4.9 summarizes the chapter.

4.1 Ensuring correctness in shared arrays

Traditionally, when a system includes multiple storage devices, a storage controller, layered beneath the filesystem, is used to manage them. The controller can be a hardware device such as a RAID controller, or a software pseudo-device such as a logical volume manager. In both cases, a single *storage controller* is used to coordinate access to the storage devices, as depicted in Figure 4.1(a). In addition to performing storage access on behalf of clients, the storage controller also performs other "management" tasks. Storage management tasks include migrating data to balance load or utilize new devices [Lee and Thekkath, 1996], adapting storage representation to access pattern [Wilkes et al., 1996], backup, and the reconstruction of data on failed devices.

Figure 4.1(b) depicts a shared storage array. In shared arrays, each client acts as the storage controller on behalf of the applications running on it, achieving scalable storage access bandwidths. Unfortunately, such shared storage arrays lack a central point to effect coordination. Because data

is striped across several devices and often stored redundantly, a single logical I/O operation initiated by an application may involve sending requests to several devices. Unless proper concurrency control provisions are taken, these I/Os can become interleaved so that hosts see inconsistent data or corrupt the redundancy codes. These inconsistencies can occur even if the application processes running on the hosts are participating in an application-level concurrency control protocol, because storage systems can impose hidden relationships among the data they store, such as shared parity blocks.

Scalable storage access and online storage management are crucial in the storage marketplace today [Golding et al., 1995]. In current storage systems, management operations are either done manually after taking the system off-line or rely on a centralized implementation where a single storage controller performs all management tasks [Wilkes et al., 1996]. The Petal system distributes storage access and management tasks but assumes a simple redundancy scheme [Lee and Thekkath, 1996]. All data in Petal is assumed to be mirrored. Other RAID levels are not supported.

The paramount importance of storage system throughput and availability has, in general, lead to the employment of ad-hoc management techniques, contributing to annual storage management costs that are many times the purchase cost of storage [Golding et al., 1995].

4.1.1 Concurrency anomalies in shared arrays

Large collections of storage commonly employ redundant codes transparently with respect to applications so that simple and common device failures can be tolerated without invoking expensive higher level failure and disaster recovery mechanisms. The most common in practice are the RAID levels (0, 1 and 5) described in Chapter 2.

As an example of a concurrency anomaly in shared arrays, assume data is striped according to the very common RAID level 5 scheme. These conflicts need not correspond to application write-write conflicts, something many applications control using higher level protocols. Consider two clients writing to two different data blocks that happen to be in the same parity stripe, as shown in Figure 4.2. Because the data blocks are different, application-level concurrency control is satisfied, but since both data blocks are in the same parity stripe of the shared storage system, the parallel controllers at the hosts both pre-read the same parity block and use it to compute the new parity. Later, both hosts write data to their independent blocks but overwrite the parity block such that it reflects only one host's update. The final state of the parity unit is therefore not the cumulative XOR of the data blocks. A subsequent failure of a data disk, say device 2, will lead to reconstruction that

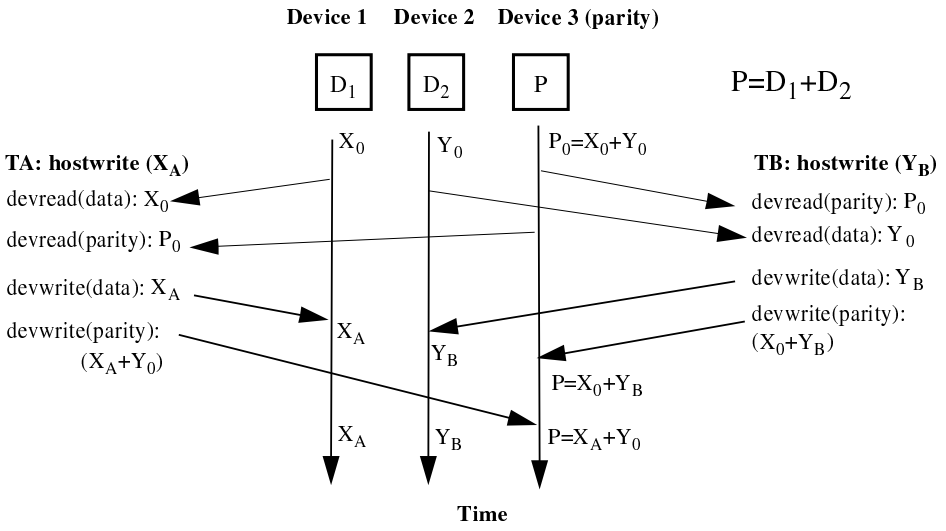


Figure 4.2: A time-line showing two concurrent Read-Modify-Write operations in RAID with no concurrency control provisions taken. Initially, the data units on device 1 and device 2 contain X_0 and Y_0, respectively, and the parity unit contains their XOR $(X_0 + Y_0)$. Although host A is updating device 1 and host B is updating device 2, they both need to update the parity. Both read the same version of the parity, but Host A writes parity last overwriting B's parity write, and leaving parity inconsistent.

does not reflect the last data value written to the device (Y_B).

In general, storage-level races can occur between concurrent accesses, or between concurrent access and management tasks, such as migration or reconstruction. These races may not be visible to application concurrency control protocols because the point of conflict is introduced by the underlying shared storage service.

4.1.2 Failure anomalies in shared arrays

In addition to anomalies that occur as a result of races, additional anomalies arise as a result of untimely failures. Consider a client performing a small RAID 5 write to a stripe. Assume that a host or power failure occurs after the data block is written but before the parity block is updated with the new value. Upon recovery, the parity stripe will be inconsistent because the data block was updated but the update was not reflected in the parity.

Figure 4.3 depicts an example scenario where an untimely failure leaves the parity inconsistent. If this is not detected and fixed, the array will remain in an inconsistent state and a subsequent failure and reconstruction will recover incorrect data. The reconstructed data, computed from the surviving disks and the parity disk, will be different from the data that was last written to the the failed device.

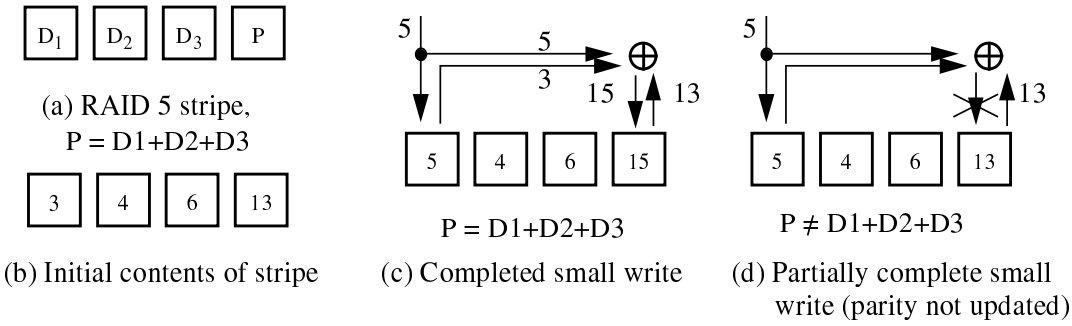


Figure 4.3: An example of an inconsistency that can arise when an untimely failure causes a RAID 5 write not to complete. Only the data block is updated. The figure shows a RAID stripe (a), with initial values of the data and parity blocks, displayed in decimal (b). The parity block is initially the sum of the values in the data blocks (In a real array, the sum is done in base 2 and is performed using the bitwise XOR of the blocks). The figure also shows two scenarios. In the first scenario, a small write completes successfully (c), updating both data and parity blocks. In the second (d), only the data block is updated. A failure occurs before the parity is written. As a result, the array is left inconsistent.

Proper recovery procedures must be followed upon such untimely failures to detect such incompletions and reinstate the consistency of the parity code. The combination of concurrency and untimely failures can induce inconsistencies in the array and in the data read by end hosts, making ensuring correctness a challenging task in a shared storage array. Fortunately, transaction theory as well as recent work on mechanizing error recovery in centralized disk arrays [Courtright, 1997] provide a good foundation from which a scalable solution to this problem can be devised.

4.2 System description

The discussion in this chapter does not assume that the storage devices are NASD disks. As such, the devices do not have to necessarily export an object interface. The solutions presented in this chapter apply to any storage system where clients are allowed access to a shared pool of storage devices. These devices can export a NASD or a traditional SCSI (block-level) interface. However, this chapter assumes that the storage devices store uniquely named blocks and act independently of each other.

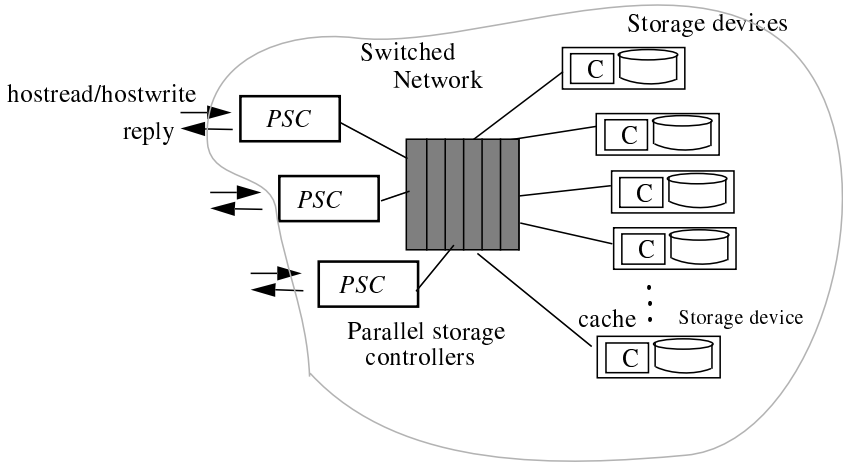


Figure 4.4: In a basic storage array, parallel storage controllers receive host requests and coordinate access to the shared storage devices. Data blocks are cached at the device where they are stored. No caches are maintained by the controllers.

4.2.1 Storage controllers

This chapter will use the terms *storage controller* or *host* to refer to the piece of software that executes on the client and is responsible for storage access. In Cheops, the clerk is the storage controller. This corresponds to the Parallel Storage Controller (PSC) in Figure 4.1(b) and in Figure 4.4. Figure 4.4 depicts the architecture of a shared storage array. Storage controllers perform four main tasks, which can be divided into *access tasks* and *management tasks*. The access tasks are reads and writes (*hostread* and *hostwrite*). These tasks provide the semantics of reading or writing a disk drive or array. The management tasks are reconstruction and migration (*reconstruct* and *migrate* respectively).

The *hostread* and *hostwrite* tasks are addressed to *virtual objects*, which may be files or whole volumes. Blocks within a virtual object may be mapped onto physical block ranges on one or more storage devices in a variey of ways: for example, Chapter 3's Cheops [Gibson et al., 1998], generalized shared storage [Amiri et al., 2000] or Petal's distributed virtual disks [Lee and Thekkath, 1996]. In all mapping schemes, the representation of a virtual object can be described by a *stripe map* which specifies how the object is mapped, what redundancy scheme is used, and how the object is accessed.

For maximal scalability, stripe maps should be cached at the clients to exploit their processors for performing access tasks. Management functions will occasionally change the contents of a stripe map — for example, during migration or reconstruction. However, copies of the maps cached by

the hosts must be kept coherent. There are several ways to maintain the coherence of cached data, such as leases [Gray and Cheriton, 1989] or invalidation callbacks [Howard et al., 1988].

Leases are preferrable to callbacks because they allow the system to make forward progress in the case of faults. For example, a client may acquire a lock and then become inaccessible. The rest of the nodes in the system may not know whether the client has crashed or whether a part of the network has become unavailable. Leases allow the rest of the nodes to make progress by associating a duration with a lock, known as the lease duration. If leases are used, the system can assume that a lock acquired by a client is no longer valid once the lease duration has expired, unless the client explicitly requests it to be refreshed.

Clients of a shared storage array use leases to maintain coherence of maps and simplify fault-tolerance. A client receives a stripe map from the storage manager. This stripe map is guaranteed to be coherent until a specified expiration time, after which it must refreshed before it is used. Before making a change to a stripe map, a manager must invalidate all outstanding leases at the clients. For efficiency, a callback may be used to invalidate a cache faster than by waiting for lease expiration. However, if a manager cannot contact one of the clients, it must wait until the expiration of the lease before making a change to the map.

4.2.2 Storage managers

This work does not discuss how storage managers choose to manage virtual objects. It assumes that multiple managers exist and a protocol is used for load balancing and fail-over. A scalable solution to the dynamic election of storage managers is presented in [Golding and Borowsky, 1999]. The only assumption made here is that, at any point in time, there is a unique storage manager associated with a virtual object.

In a shared storage array, any client can access any virtual object. As a result, access tasks can be concurrently executing at multiple clients. Storage management tasks are slightly different, however. As already mentioned, storage management tasks include: reconstruction and migration. Reconstruction refers to the re-computation of data on a failed device from redundant data on other devices in its stripe groups. Migration refers to moving data between devices, potentially changing the stripe unit size or the RAID level. Storage migration is useful for load and capacity balancing, and to enact performance optimizations. For example, when storage devices are added, objects stored on loaded servers are usually copied to the newly added devices to balance load and utilize the newly added resources. This is often known as “data migration.”

Another kind of migration is "representation migration" which can be used to optimize performance by adapting data layout to access pattern. As was demonstrated by the AutoRAID system [Wilkes et al., 1996], adapting storage representation to workload access patterns can improve performance. For instance, the ideal RAID level depends on the workload access pattern and the degree of storage utilization. Both RAID level 5 and level 1 representations are fault-tolerant, but behave differently under read and write workloads. A RAID level 5 small write results in four I/Os, while a RAID level 1 write results in two, making the latter preferable for write-intensive workloads. RAID level 1 has a 100% redundancy overhead, however, as opposed to 25% for RAID level 5 with a stripe width of 5 disks. This makes RAID level 5 preferable when capacity is short. Migrating between the two levels can adapt to changes in the workload.

Storage management tasks are initiated by the storage manager for the virtual object. The storage manager can delegate "pieces" of the task (of migration or reconstruction) to clients or to other managers by giving them an exact description of the "operation plan." An example plan would specify reconstructing the first 10 blocks on a failed device and writing it to a replacement device.

4.3 Storage access and management using BSTs

Concurrency is an essential property of shared storage. In large clustered systems, for example, many clients often use stored data at the same time. In addition, because the reconstruction or copying of large virtual objects can take a long time, it is inevitable that most systems will want to allow concurrent access. As discussed in section 4.1, however, it is hard to ensure correctness when concurrent storage controllers share access to data on multiple devices. Device failures, which can occur in the midst of concurrently executing tasks, complicate this even further.

Transaction theory, which was originally developed for database management systems, handles this complexity by grouping primitive read and write operations into transactions that exhibit ACID properties (Atomicity, Consistency, Isolation and Durability) [Haerder and Reuter, 1983]. Databases, however, must correctly perform arbitrary transactions whose semantics are application-defined. Storage controllers, on the other hand, perform only four tasks, the semantics of which are well known. This a priori knowledge enables powerful specialization of transaction mechanisms for storage tasks [Courtright, 1997].

The following discussion establishes the correctness of storage controller tasks in terms of a durable serial execution, where tasks are executed one at a time and where failures only occur while the system is idle (in the gaps between task executions). The discussion then addresses how

concurrency and failures are handled.

As a start, let's break down each storage controller task into one or more simple operations, called base storage transactions (BSTs). BSTs are transactions specialized to storage controller tasks. Informally, BSTs can be thought of as operations that do not interleave with each other even when they are executing concurrently. They appear to have executed one at a time, in a serial order. Furthermore, even if a failure occurs while many BSTs are active, the array is found in a consistent state the next time it is accessed after the failure. Thus, for the rest of this section, BSTs can be assumed to be executing serially and failures can be assumed not to happen during operations. The properties of BSTs will be precisely described in the next section.

What BSTs enable is a simplified view of the world where operations and failure anomalies do not simultaneously occur. This allows one level of complexity to be removed from storage controller software. This section assumes the existence of BSTs and builds a solution based on them. The focus in this section is on how BSTs can be used to allow management tasks to be ongoing in the presence of access tasks at the clients.

The approach proposed here allows management tasks to be performed on-line, while clients are accessing virtual objects. It breaks down management tasks into short-running BSTs. This reduces the impact on time-sensitive host access tasks and enables a variety of performance optimizations [Holland et al., 1994, Muntz and Lui, 1990]. Furthermore, it requires clients accessing a virtual object to "adapt" their access protocols depending on the management task that is being performed.

Each virtual object is in one of four modes: Fault-free (the usual state), Degraded (when one device has failed), Reconstructing (when recovering from a failure) or Migrating (when moving data). The first two modes are access modes, where only access tasks are performed, and the second two are management modes, where both management and access tasks are allowed. Clients therefore perform four main high-level tasks: *hostread*, *hostwrite*, *migrate* and *reconstruct*. *Hostread* and *hostwrite* tasks can be performed under any of the modes. The *migrate* task occurs only in the migrating mode, and the *reconstruct* task occurs only in the reconstructing mode.

Different BSTs are used in different modes, partly to account for device failures and partly to exploit knowledge about concurrent management and access tasks.

	Fault-Free	Degraded	Migrating	Reconstructing
Read	Read	Degraded-Read	Read-Old	Degraded-Read
Write	Large-Write, Read-Modify-Write, Reconstruct-Write	Read-Modify-Write, Reconstruct-Write	Multi-Write	Replacement-Write
Migrate	—	—	Copy-Range	—
Reconstruct	—	—	—	Rebuild-Range

Table 4.1: BSTs used by access and management tasks. The BST used by a read or a write task depends on the mode that the object is in. In the migrating mode, a read task invokes the Read-Old BST which reads the old or original copy of the block as shown in Figure 4.7. A write task invokes the Multi-Write BST which writes to both the old and the new copies. In this mode, a special task (the migrate task) is ongoing in the background. This task invokes the Copy-Range BST to copy data from original to new location. In the reconstructing mode, a read task invokes the Degraded-Read BST to recompute the data on the failed disk from the redundant copy in the parity. A write task writes to the degraded array and also updates the replacement copy. In the background, the reconstruct task is in progress in this mode. This task invokes the Rebuild-Range BST to rebuild data from the degraded array and write it to the replacement copy.

4.3.1 Fault-free mode

This discussion focuses on RAID level 5 because it is the most complex and general RAID level. The solutions described in the following sections can be readily applied to RAID levels 1 and 4, for example.

Table 4.1 shows the BSTs used to perform each allowed task in each of these modes. The BSTs for Fault-free mode are straightforward, and are shown in Figure 4.5.

Tasks are mapped onto BSTs as follows. An access task, hostread or hostwrite, is executed using one BST. Which BST is chosen depends on how much of the stripe is being updated.

4.3.2 Degraded mode

In degraded mode, only access tasks are allowed. The BST used depends on whether the failed device is being accessed or not. If the device is not accessed, the BST used is similar to the one used in Fault-Free mode. If the failed device is being read or written, all the surviving stripe units must be accessed as shown in Figure 4.6.

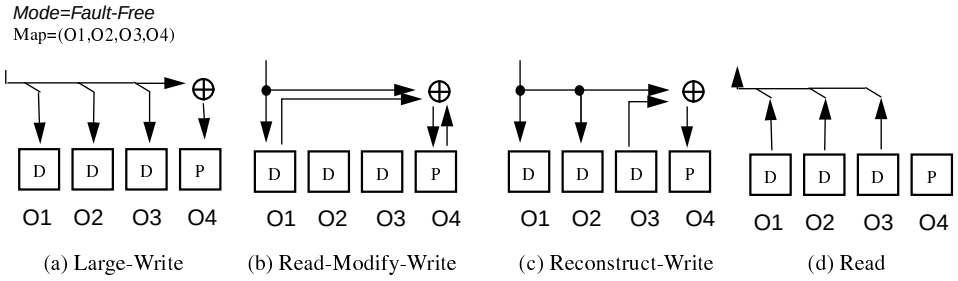


Figure 4.5: Storage transactions used in fault-free mode. These BSTs are the basic Fault-Free write protocols of RAID level 5 described in Chapter 2.

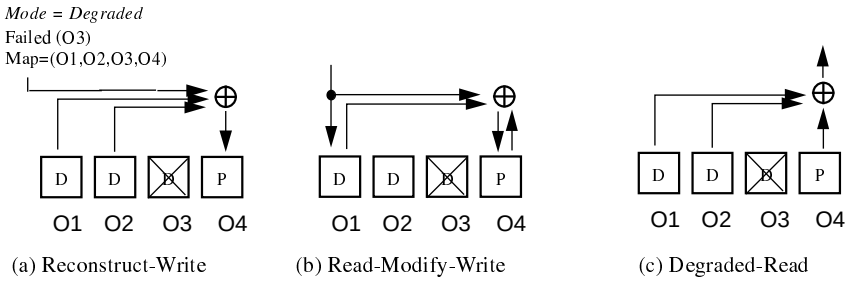


Figure 4.6: Storage transactions used in degraded mode. These BSTs are the basic degraded mode write protocols of RAID level 5 described in Chapter 2.

4.3.3 Migrating mode

To simplify exposition, assume a non-redundant virtual object as shown in Figure 4.7. Recall that migrating an object requires two steps: copying the data to a new location and then updating the map to point to the new location.

In the migrating mode, the stripe map for the virtual object specifies the old and new physical locations. Hostwrite tasks update the old and new physical locations by invoking a Multi-Write BST. Thus, at any point during the migration, the target physical blocks are either empty (not yet written to) or contain the same contents as their associated source physical blocks. Hostread tasks invoke the read BST, which reads the physical blocks from their old locations. The read BST does not access the target physical blocks because they may be still empty. The migrate task can be ongoing in parallel using the copy BST.

Because the execution of BSTs is serializable, the Read-Old BST is correct in that it always returns the most recent data written to a block. To see why serializability is sufficient to guarantee correctness, consider any serial history composed of Multi-Write, Read-Old and Copy-Range BSTs.

The Read-Old BST reads the contents of the original location, which is not updated by the Copy-Range BST, so Copy-Range BSTs can be ignored. By inspection, since the Multi-Write BST is the only BST that updated the original blocks and since a Multi-Write BST is invoked for each hostwrite task, it follows that a following hostread task invoking a Read-Old BST will read the contents that are last written to the block.

Furthermore, when the migrate task completes after having copied the entire original object in the target location, both the original and target objects are guaranteed to have the same contents. To see why serializability guarantees that at the conclusion of the migrate task both copies contain the same contents, consider this intuitive argument based on a serial execution of Multi-Write and Copy-Range BSTs. Read-Old BSTs can be ignored because they never update the contents of storage. In particular, and without loss of generality, consider a block on the target location. For any such block in the target location, there are exactly two cases: Either the last transaction that wrote the block is a Multi-Write BST or is a Copy-Range BST. If the last transaction that wrote to the block is a Copy-Range BST, then it must be that the contents of this block are the same as its counterpart in the original location. This follows from the semantics of the Copy-Range BST which copies data from original to target. If the last transaction that wrote to the block is a Multi-Write BST (in this case the Copy-Range was followed by a host update), then this transaction must have written the same contents in both copies, also by virtue of its semantics. Both cases lead us to the desirable conclusion that both copies contain the same contents.

Thus, when the migrate task completes, the storage manager can revoke all outstanding leases, change the stripe map to point to the new location and discard the old location.

4.3.4 Reconstructing mode

This mode is used when recovering from a disk failure (Figure 4.8). The system declares a new block on a new disk to be the replacement block, then uses the reconstruct task to recover the contents of that block. This can occur in parallel with hostread and hostwrite tasks. All these tasks are aware of both the old and new mappings for the stripe, but the read BSTs use the “original array,” ignoring the replacement block altogether. Hostwrite tasks use BSTs that behave as if the original array were in Degraded mode, but also update the replacement block on each write to the failed block.

The reconstruct task rebuilds the data on the replacement block using the Rebuild-Range BST, which reads the surviving data and parity blocks in a stripe, computes the contents of the failed data block and writes it to the replacement disk. When the reconstruct task is done, the replacement

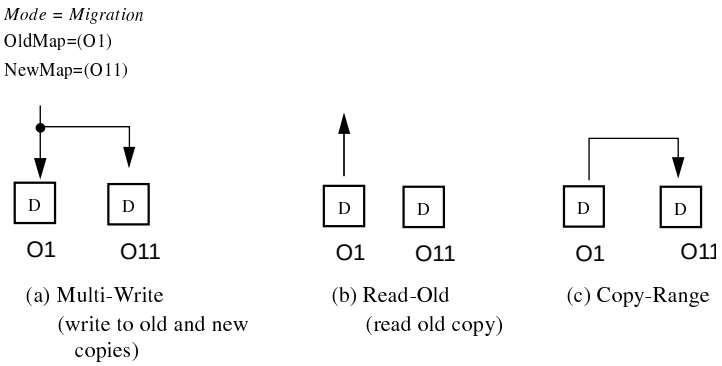


Figure 4.7: Storage transactions used in the migrating mode. The Multi-Write BST updates both the old location and the new location of the block. The Read-Old BST reads the old location, ignoring the new location. The migrate task invokes the Copy-Range BST to copy the contents of the blocks from the old location to the new one.

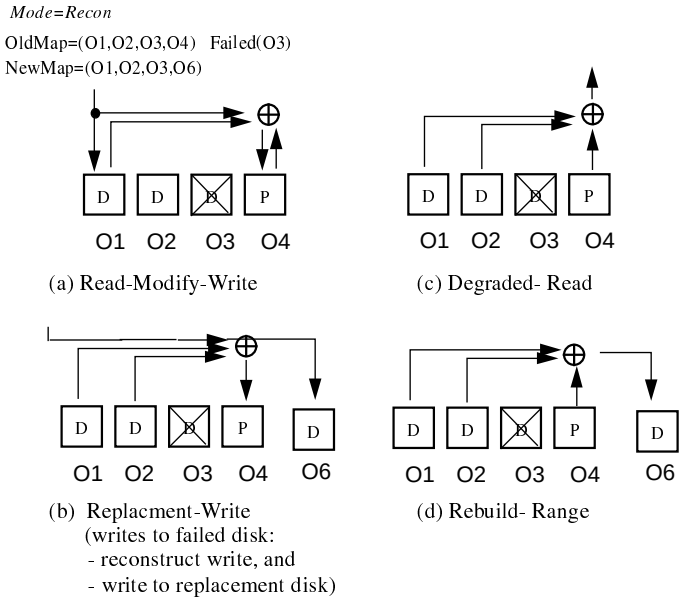


Figure 4.8: Storage transactions used in reconstructing mode. Write tasks that do not address the failed device use the Read-Modify-Write BST (a) as in degraded mode. Writes that address the failed device must update both the old location, the degraded array, and the new location, the replacement device. The write BST invoked in this mode is called Replacement-Write (b). The reconstruction task invokes the Rebuild-Range (d) to reconstruct the contents of the failed device and write them to the replacement. Read tasks invoke the Degraded-Read BST (c) as in degraded mode to read the data from the old location, the degraded array.

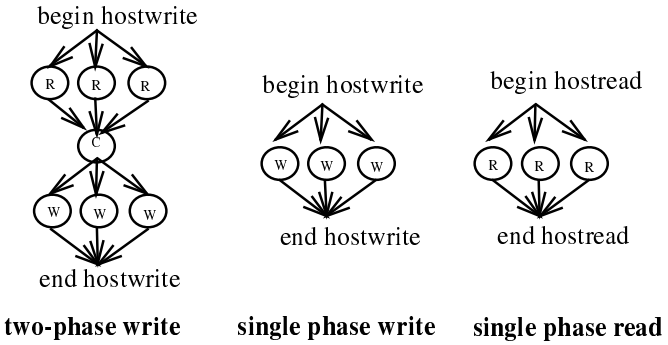


Figure 4.9: The general composition of storage-level host operations. Operations are either single phase or two-phase. Two-phase operations have a read phase followed by a write phase. A computation (C) separates the two I/O phases. Host operations are denoted by *hostread* and *hostwrite*, while disk-level requests are denoted by *devread* and *devwrite*.

block will reflect the data from the failed block, parity will be consistent, and the stripe enters Fault-free mode. Note that reconstruction concurrent with host accesses may result in unnecessary, but still correct, work.

4.3.5 The structure of BSTs

As shown in the previous section, all the required BSTs share a common structure. Each BST maps onto one or more low-level read and write requests to the NASD devices. A low-level request is a read or write request to blocks within NASD objects. These are often referred to as device-level I/Os or *devread* and *devwrite*. One property that will be later exploited by the concurrency control protocols is that storage transactions often have a two-phase nature. All base storage transactions are composed of either single phase or two-phased collections of low-level requests, as shown in Figure 4.9.

A single phase *hostread* (*hostwrite*) breaks down into parallel *devread* (*devwrite*) requests, while a two-phase *hostwrite* breaks down into a first read phase (where *devreads* are sent to the disks) followed by a write phase (where *devwrites* are issued). In fact, all the RAID architectures (including mirroring, single failure tolerating parity and double failure tolerating parity [Patterson et al., 1988, Blaum et al., 1994] as well as parity declustering [Holland et al., 1994], which is particularly appropriate for large storage systems, share the common characteristic that all logical reads and writes map onto device-level I/Os either into a single-phase or into a read phase followed by a write phase.

4.4 BST properties

BSTs specialize general transaction ACID properties to the limited tasks of shared storage controllers. Before describing the protocols that guarantee the key properties of BSTs in the next section, it is necessary to precisely state the properties of BSTs first. BSTs' ACID properties have a specific meaning specialized to shared arrays.

4.4.1 BST Consistency

In the context of shared redundant storage, consistency means that redundant storage blocks contain data that correctly encodes the corresponding data block values. For RAID level 5, this means that after each BST, the value of the parity block (P) is the XOR of all the values of the last writes to each of the corresponding data blocks (D).

Each of the BSTs described before has the property that, provided storage is consistent when they start and does not fail while they are executing, and provided that BSTs execute one at a time, then storage is consistent after the completion of the BST. This storage specialization of transaction consistency guides much of the rest of the work in this chapter.

4.4.2 BST Durability

The durability property states that storage regions written by a successful BST maintain the last written data and never switch back to older data values. Durability must be preserved even after a device failure and a subsequent reconstruction.

Because the primitive operations of BSTs transform stable storage (typically magnetic disks), durability of changes is not difficult. Because BSTs guarantee the consistency property, RAID level 5 reconstruction will yield the correct data (the last written values). Thus consistency of the array and a serial application of BSTs guarantees durability. This property will not be discussed further.

4.4.3 No atomicity

This is the property that a transaction, once started, either completes entirely or terminates without changing any storage values. In the absence of knowledge of the function of a transaction, database systems must provide full atomicity by logging values to be changed before making any changes. The logs are preserved until all the changes are made, and if a failure occurs, the log of committed changes are re-applied during recovery [Haerder and Reuter, 1983].

However, storage subsystems (disks and disk arrays) have traditionally not provided this property. In current storage systems, if a client fails after some devices have been updated but before data has been sent to all devices, atomicity will be violated because some blocks have been written, and the values that should have been written to the others have been lost with the loss of the host's memory. All existing storage systems have this problem, although database systems built on these non-atomic storage systems achieve atomicity using database-level logging. In agreement with current systems, BSTs are therefore not required to have and, in this work, do not have the "all-or-nothing" atomicity property.

The consistency property of BSTs, however, states that upon such a failure, partially changed stripes must be detected and new parity recomputed to reflect the possibly incomplete changes applied to the data. The consistency property is upheld by detecting the changed stripes and by invoking a Rebuild-Range BST to make the parity consistent with the data block (even if these data blocks are not complete or correct with respect to the application). Note that to detect a failure and take these actions, storage system code not on the failed host must know which BST was active. This is accomplished through the protocols described in Section 4.7.

TickerTAIP [Cao et al., 1994] is a parallel disk array which exploited knowledge of storage semantics to similarly achieve this level of parity consistency without full write-ahead logging.

4.4.4 BST Isolation

In showing that the consistency and durability properties are met, BSTs were required to appear as if they executed serially, one at a time. This is the isolation property of BSTs. Precisely, this property states that the execution of concurrent BSTs is serializable, that is, yields identical effects as if the concurrent BSTs executed in at least one of many possible serial execution sequences [Papadimitriou, 1979]. The effect is defined in terms of the values read by other BSTs.

Serializability ensures that the concurrency anomalies described in Section 4.1 do not occur. Furthermore, it enables hosts access and management tasks to be ongoing simulataneously without leading to incorrectness. This property must be ensured at low overhead, however, because of the stringent performance demands on storage systems by higher-level applications. Furthermore, the protocols ensuring serializability must scale well to the sizes expected and desired in large network-attached storage arrays.

Traditional protocols ensuring serializability rely on a central node to order and serialize concurrent host accesses. This central node can become a performance bottleneck in large systems.

Simulation parameter	Value
System Size	30 devices, 16 hosts, 20 threads per host, RAID level 5, 64 KB stripe unit Stripe width = 4 data + parity, 20000 stripe units per device.
Host workload	Random think time (normally distributed, mean 80 ms, variance 10 ms), 65% reads, host access address is uniform random, access size varies uniformly between 1 and 6 stripe units
Service time	Disk service time random (normally distributed with mean positioning of 8 ms, variance 1 ms), 15 MB/sec transfer rate Network bandwidth is 10 MB/s and has a random per-message overhead of 0.5-1ms. Mean host/lock server message processing time, request service time: 750 microseconds.

Table 4.2: Baseline simulation parameters. Host data is striped across the devices in a RAID level 5 layout, with 5 devices per parity group. Host access sizes vary uniformly between 1 and 6 stripe units, exercising all the possible RAID write algorithms discussed in Section 4.3. The region accessed on each device is about 1.22 GB, representing a 25% "active" region of a 5 GB disk. The sensitivity of the protocols to the size of region accessed, read/write ratio, and network latency variability are explored later in this section.

The following section analyzes four serializability algorithms for BSTs: two traditional centralized protocols, and two device-based distributed protocols that exploit trends towards increased device intelligence to achieve higher scalability.

4.5 Serializability protocols for BSTs

To provide serializability for storage transactions, three approaches are considered: centralized locking, distributed device-embedded locking, and timestamp ordering using loosely synchronized clocks. Each protocol is described and evaluated in simulation. The evaluation workload is composed of a fault-free random access workload applied to RAID level 5 storage. All the presented protocols guarantee serializability for all *hostread* and *hostwrite* operations, but exhibit different latency and scaling characteristics.

4.5.1 Evaluation environment

The protocols were implemented in full detail in simulation, using the Pantheon simulator system [Wilkes, 1995]. The cluster simulated consists of hosts and disks connected by a network. Table 4.2

shows the baseline parameters of the experiments. Although the protocols were simulated in detail, the service times for hosts, controllers, links and storage devices were derived from simple distributions based on observed behavior of Pentium-class servers communicating with 1997 SCSI disks [Technology, 1998] over a fast switched local area network (like Fibre-Channel). Host clocks were allowed to drift within a practical few milliseconds of real-time [Mills, 1988].

The performance of the protocols is compared under a variety of synthetically generated workloads and environmental conditions. The baseline workload represents the kind of sharing that is characteristic of OLTP workloads and cluster applications (databases and file servers), where load is dynamically balanced across the hosts or servers in the cluster resulting in limited locality and mostly random accesses. This baseline system applies a moderate to high load on its storage devices yielding about 50% sustained utilization.

4.5.2 Centralized locking

With server locking, a centralized lock server provides locking on low-level storage block ranges. A host acquires an exclusive (in case of a *devwrite*) or a shared (in case of a *devread*) lock on a set of target ranges by sending a lock message to the lock server. The host may then issue the *devread* or *devwrite* low-level I/O requests to the devices. When all I/O requests complete, the host sends an unlock message to the lock server. The lock server queues a host's lock request if there is an outstanding lock on a block in the requested range. Once all the conflicting locks have been released, a response is returned to the host. However, server locking introduces a potential bottleneck at the server and delays issuing the low-level I/O requests for at least one round trip of messaging to the lock server.

Server locking is an example of batch locking, described in Chapter 2. As mentioned in that chapter, batch locking achieves serializability. Furthermore, because all locks are acquired in a single message, latency is minimized and deadlocks are avoided.

Callback locking [Howard et al., 1988, Lamb et al., 1991, Carey et al., 1994] is a popular variant of server locking, which delays the unlock message, effectively caching the lock at the host, in the hope that the host will generate another access to the same block in the near future and be able to avoid sending a lock message to the server. In the case of a cache hit, lock acquistion messaging is avoided and access latency is reduced. If a host requests a lock from the lock server that conflicts with a lock cached by another host, the server contacts the host holding the conflicting lock (this is the callback message), asking it to relinquish the cached lock so the server can grant a lock to the

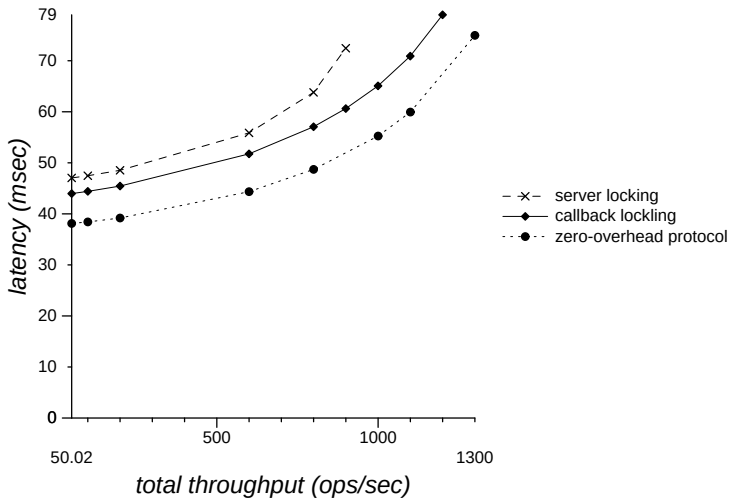


Figure 4.10: Scaling of server and callback locking. The bottom-most line represents the zero-overhead protocol which has no concurrency control and provides no serializability guarantees. Centralized locking schemes bottleneck before delivering the maximum achievable throughput. The applied load was increased by increasing the number of active hosts in the system until saturation. The baseline workload (16 hosts) corresponds to the fifth point from the left in the graph.

new owner. In this case, the client requesting the lock incurs longer delays until the client(s) holding the conflicting locks are contacted by the lock server.

One common optimization to callback locking is to have locks automatically expire after a specified period, known as the *lease duration*, so that callback messaging is reduced. Our implementation of centralized callback locking uses a lease duration of 30 seconds, during which consecutive lock acquisitions by different clients suffer the callback messaging latency overhead.

Figure 4.10 highlights the scalability limitation of centralized locking protocols. It plots the host request-to-response latency of the protocols against throughput (number of hosts). Server locking bottlenecks substantially before delivering the maximum throughput that is attainable with the zero-overhead protocol. This is caused by the fact that the server's CPU is bottlenecked with handling network messaging and processing lock and unlock requests.

Callback locking reduces lock server load and lock acquisition latencies, provided that locks are commonly reused by the same host multiple times before a different host requests a conflicting lock. At one extreme, a host requests locks on its private data and never again interacts with the lock server until the lease expires. At the other extreme, each lock is used once by a host, and then

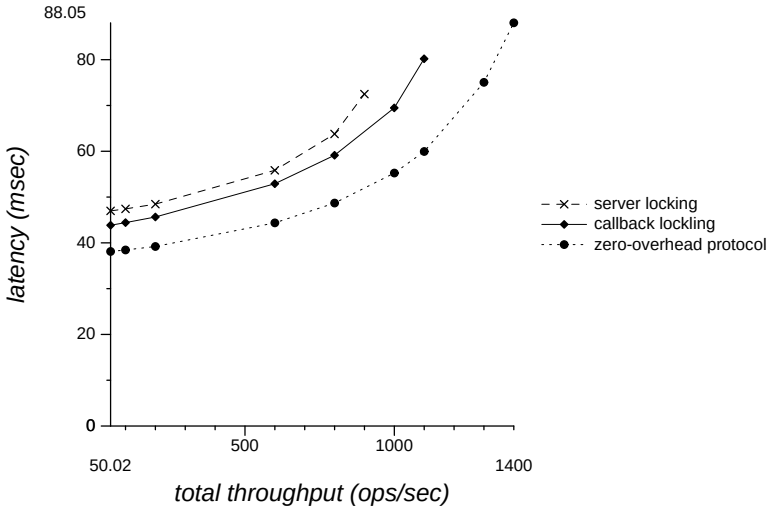


Figure 4.11: Scaling of server and callback locking under increased contention. In this experiment, the region accessed by the hosts is restricted to only 40% of the region used in the baseline workload. Callback locking yields smaller benefits under this high-contention workload than under the baseline workload. Both centralized protocols bottleneck before delivering the full throughput of the storage system. The baseline workload (16 hosts) corresponds to the fifth point from the left in the graph.

is called back by a conflicting use. This will induce the same number of messages as simple server locking, but the requesting host must wait on two other machines, one at a time, to obtain a lock that must be called back, potentially doubling latency before the disk I/O can occur. This pre-I/O latency can be even worse if a read lock is shared by a large number of hosts since all locks need to be recalled.

Figure 4.10 shows that at the baseline workload, callback locking reduces latency relative to simple locking by over 10% but is still 15% larger than the zero-overhead protocol. This benefit is not from locality as the workload contains little of it, but from the dominance of read traffic which allows concurrent read locks at all hosts, until the next write. While more benefits would be possible if the workload had more locality, the false sharing between independent accesses that share a parity block limits the potential benefits of locality.

Figure 4.11 shows the performance of the centralized protocols when the target region accessed by the hosts is restricted to 40% of the original region. The graph shows that the benefit of callback locking is reduced as the percentage of locks revoked by other conflicting clients before being reused increases with contention. This increases the pre-I/O latency and increases the load on the

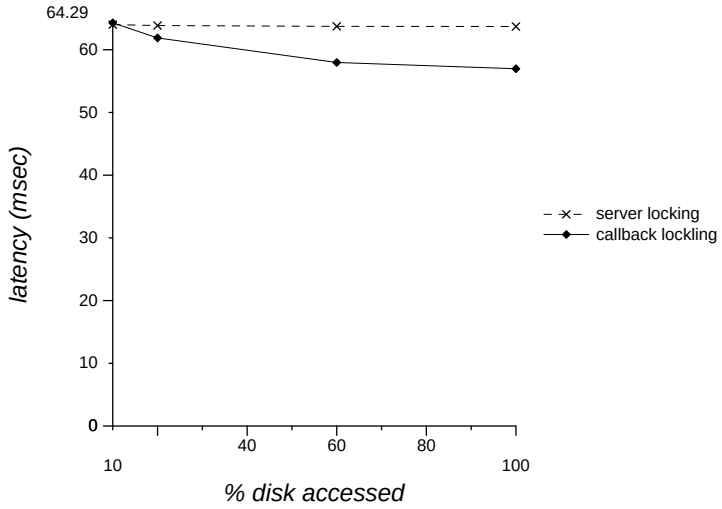


Figure 4.12: The effect of contention on host latency for centralized locking protocols. Latency increases under callback locking as the fraction of disk targeted by host accesses decreases. Contention increases from right to left, as the % of disk accessed by the workload decreases. Callback locking reduces latencies compared to server locking under moderate contention even for a random-access workload because the baseline workload is mostly (65%) reads and shared locks are cached at each host. The graph shows that under high contention (small fraction of disk accessed), the latency of callback locking increases relative to that of server locking because caching brings little benefit as locks are revoked by conflicting clients before they are reused.

lock server. As the system size and load increases, the performance of both centralized callback locking and server locking become limited by the central server bottleneck.

Figure 4.12 shows the effect of contention on the relative performance of the locking protocols. When contention is high, closer to the origin on the x-axis of the graph, there is little benefit from lock caching. In that case, the percentage of the disk that the clients are accessing is relatively small, resulting in frequent lock callbacks and hand-offs between clients. When contention is low, callback locking shows a noticeable reduction in response time as locks are reused by clients before they are revoked.

4.5.3 Parallel lock servers

The scalability bottleneck of centralized locking protocols can be avoided by distributing the locking work across multiple parallel lock servers. This can be achieved in two ways: a host can send a single request to one of the lock servers which coordinate among themselves (ensuring proper concurrency control and freedom from deadlock) and return a single response to the host, or hosts

can directly send parallel lock requests to the servers based on a partitioning of locks. As the first approach simply displaces the responsibility at the cost of more messaging, it is not discussed further. Instead, the focus in the remaining discussion is placed on partitioned locks.

Because locks are partitioned across locks servers, whether by a static or dynamic scheme, there will always be some pairs of locks that are on different servers. When a host attempts to acquire multiple locks managed by multiple servers, deadlocks can arise. This can be avoided if locks are statically ordered and all hosts acquire locks in the same prescribed order, but this implies that lock requests are sent to the lock servers one at a time — the next request is sent only after the previous reply is received — and not in parallel, increasing latency and lock hold time substantially. Alternatively, deadlocks can be detected via time-outs. If a lock request can not be serviced at a lock server after a given time-out, a negative response is returned to the host. A host recovers from deadlock by releasing all the acquired locks, and retrying lock acquisition from the beginning. While parallel lock servers employing a deadlock avoidance or detection scheme do not have the bottleneck problem, they suffer from increased messaging and usually longer pre-I/O latency. This induces longer lock hold time, increasing the chance of potential conflict compared to an unloaded single lock server.

4.5.4 Device-served locking

With the paramount importance of I/O system scalability and the opportunity of increasing storage device intelligence, it seemed promising to investigate embedding lock servers in the devices. The goal is to reduce the cost of a scalable serializable storage array, and by specialization, increase its performance. The specializations exploit the two-phase nature of BSTs to piggy-back lock messaging on the I/O requests, thereby reducing the total number of messages, and latency. In device-served locks, each device serves locks for the blocks stored on it. This balances lock load over all the devices, enhancing scalability. Like any parallel lock server scheme, simple device-served locking increases the amount of messaging and pre-I/O latency. Often, however, the lock/unlock messaging can be eliminated by piggy-backing these messages on the I/O requests because the lock server and the storage device are the same. Lock requests are piggy-backed on the first I/O phase of a two-phase storage transaction. To make recovery simple, this scheme required that a host not issue any *devwrite* requests until all locks have been acquired, although it may issue *devread* requests. Therefore, restarting an operation in the lock acquisition phase does not require recovering the state of the blocks (since no data has been written yet).

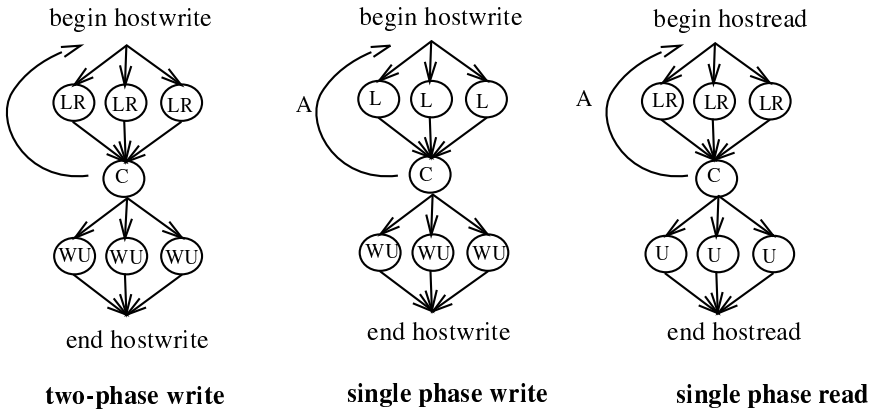


Figure 4.13: The breakdown of a host operation with device-served locking and the piggy-backing optimization. A node represents a message exchange with a device. An 'L' node includes a *lock* operation, 'U' stands for an *unlock* operation, 'LR' represents the *lock-and-devread* operation, while 'WU' stands for *devwrite-and-unlock*. The edges represent control dependencies. A 'C' node represents a synchronization point at the host, where the host blocks until all preceding operations complete, restarting from the beginning if any of them fail. Lock operations can fail if the device times out before it can grant the lock ('A').

In the case of single-phase writes, a lock phase must be added, preceding the writes, since the lock and write requests cannot be bundled together without risking serializability. However, unlock messages can be piggy-backed onto write I/O requests as shown in Figure 4.13.

In the case of single-phase reads, lock acquisition can be piggy-backed on the reads, reducing pre-I/O latency, but a separate unlock phase is required. The latency of this additional phase can be hidden from the application since the data has been received and the application is no longer waiting for anything. In the case of two-phase writes, locks can be acquired during the first I/O phase (by piggy-backing the lock requests on the *devread* requests) and released during the second I/O phase (by piggy-backing the unlock messages onto the *devwrite* requests), totally hiding the latency and messaging cost of locking.

This device-supported parallel locking is almost sufficient to eliminate the need for leased callback locks because two-phase writes have no latency overhead associated with locking and the overhead of unlocking for single phase reads is not observable. Only single phase writes would benefit from lock caching.

Device-served locking is more effective than the centralized locking schemes, as shown in Figure 4.15 and Figure 4.16. With the baseline workload of 16 hosts, device-served locking causes latencies only 6% larger than minimal. Despite its scalability, device-served locking has two dis-

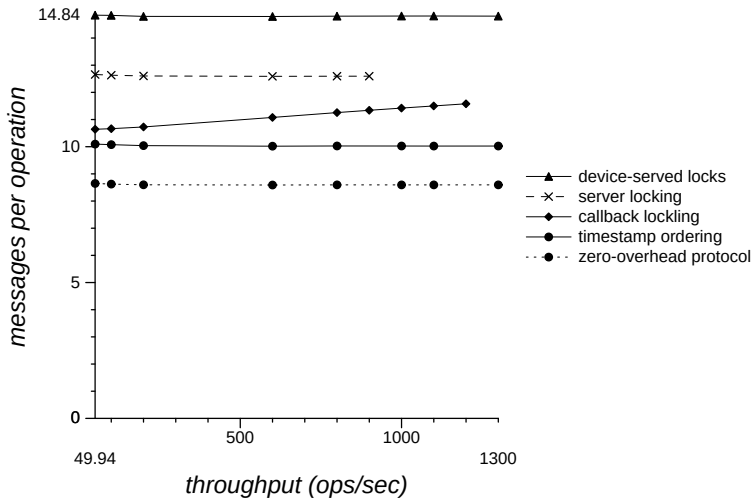


Figure 4.14: The messaging overhead of the various protocols. Device-served locks require an explicit unlock message for each block accessed. In the absence of retries, timestamp ordering requires one message per read I/O request, resulting in lower messaging overhead.

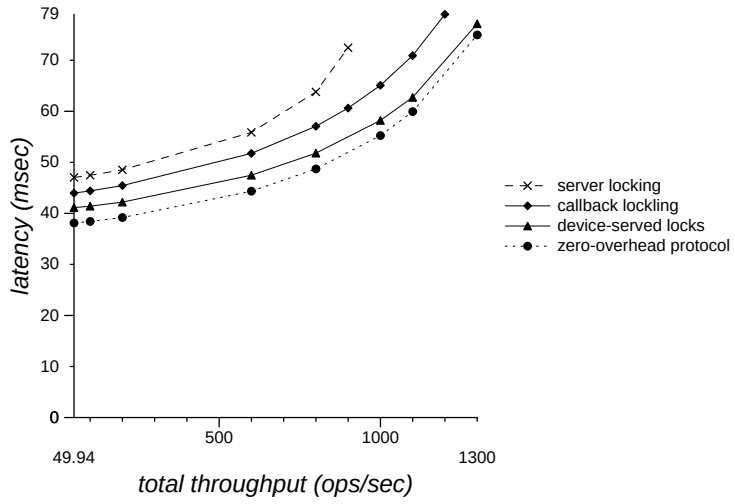


Figure 4.15: Scalability of device-served locking compared to the centralized variants. Device served locking comes within a few percent of delivering the full throughput of the storage system.

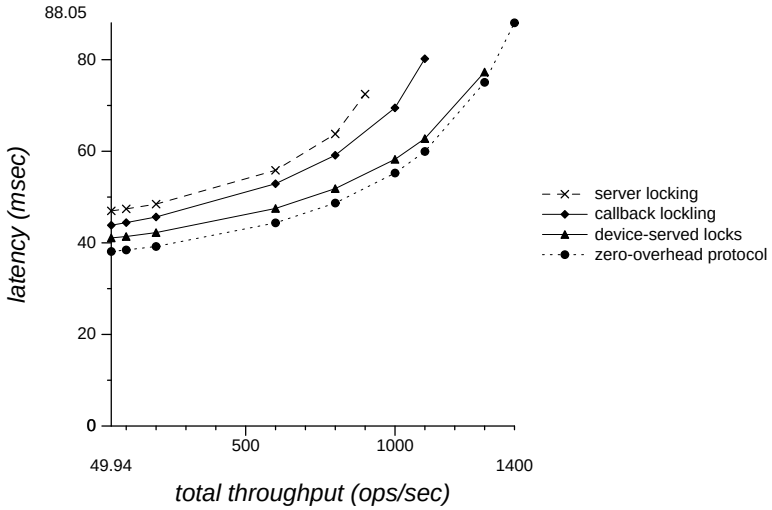


Figure 4.16: Scalability of device-served locking compared to the centralized variants under increased contention. In this experiment, host accesses are restricted to 40% of the region targeted by the baseline workload. Device served locking comes within a few percent of delivering the full throughput of the storage system.

advantages: it uses more messages than centralized locking protocols, as shown in Figure 4.14, and it has performance vulnerabilities under high contention due to its susceptibility to deadlocks. A component of the deadlock vulnerability disadvantage is the difficulty of configuring time-outs for device-served locks. Note that although deadlocks may be uncommon, they have two second order effects: they cause a large number of requests to be queued behind the blocked (deadlocked) requests until time-outs unlock the effected data; and, when deadlocks involve multiple requests at multiple devices, time-outs lead to inefficiencies because they restart more operations than necessary to ensure progress. This results in an increased messaging load on the network and wasted device resources for more message and request processing.

4.5.5 Timestamp ordering

Timestamp ordering protocols are an attractive mechanism for distributed concurrency control over storage devices since they place no overhead on reads and are not susceptible to deadlocks. As in the database implementation, discussed in Section 2.4.2, timestamp ordering works by having hosts independently determine the same total order in which high-level operations should be serialized. By providing information about that order (in the form of a timestamp) in each I/O request, the intelligent storage devices can enforce the ordering and thereby serializability. Since I/O requests

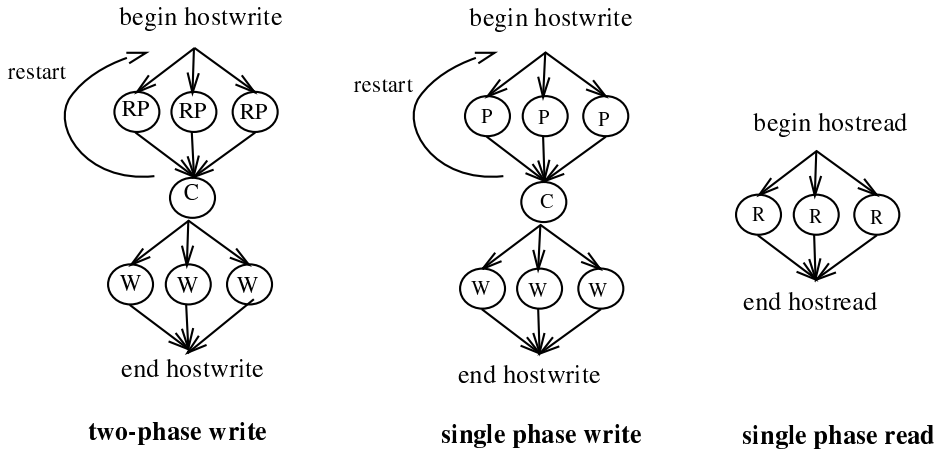


Figure 4.17: The composition of host operations in the optimized timestamp ordering protocol. *devread*, *devwrite*, and *prewrite* requests are denoted by 'R', 'W' and 'P' nodes respectively. 'RP' denotes a *read-and-prewrite* request. A 'C' node represents a synchronization barrier at the host, where the host blocks until all preceding operations complete, restarting from the beginning if any of them fail. Some of the prewrites or reads may be rejected by the device because they did not pass the timestamp checks. In a two-phase write, the prewrite requests are piggy-backed with the reads. Much like device-served locks of Section 4.5.4, single phase writes still require a round of messages before the *devwrite* requests are issued.

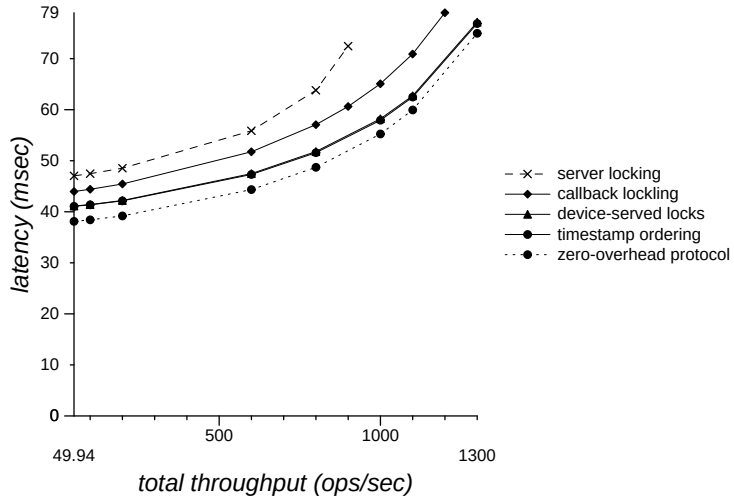


Figure 4.18: Scalability of device-based protocols compared to the centralized variants. Both device-based locking and timestamp ordering closely approximate the performance of the zero-overhead protocol, coming within a few percent of the maximal throughput and minimal latency achievable under the zero-overhead protocol.

are tagged with an explicit order according to which they have to be processed (if at all) at each device, deadlocks can not occur and all allowed schedules are serializable. Instead of deadlock problems, out-of-order requests will be rejected, causing their parent high-level operation to be aborted and retried with a later timestamp.

The use of timestamp ordering based on loosely synchronized clocks for concurrency control and for efficient timestamp management has been demonstrated by the Thor client-server object-oriented database management system [Adya et al., 1995]. As in the database case, since each device is performing a local check, a write request may pass the check in some devices, but the high-level operation may abort due to failed checks in other devices. This can lead to partial effects of a host write operation being applied to the storage system. While atomicity in the event of failures is not a requirement of BSTs, it is not acceptable to allow non-failing storage systems to damage data routinely. To avoid partial updates and the associated complex and expensive undo operations they require, multi-device writes need consensus before allowing changes. By splitting the write protocol into a prewrite phase followed by a write phase, the protocol ensures that all the devices take the same decision. This is consistent with timestamp ordering as implemented in database and distributed systems. The cluster of hosts share a loosely-synchronized clock that is used to generate timestamps. New timestamps are generated uniquely at a host by sampling the local clock, then appending the host's unique identifier to the least significant bits of the clock value. Each device maintains two timestamps associated with each block (rts and wts). The device similarly maintains a queue of requests, reads, writes and prewrites which are awaiting service (e.g. blocked behind a prewrite). $minpts$ denotes the smallest timestamp of a prewrite that has been accepted in a specific block's request queue. A refresher on timestamp ordering protocol with example scenarios and algorithms is given in Section 2.4 of the background chapter.

The read-modify-write protocol will now be discussed as an example since it employs the piggybacking optimization and is of reasonable complexity. This protocol reads data and parity in a first phase, uses this data together with the "new data" to compute the new parity, then updates both data and parity. The host starts by generating a new timestamp, $opts$, then sends low-level I/O read requests to the data and parity devices, tagging each request with $opts$, and bundling each request with a prewrite request as shown in Figure 4.17.

The device receiving a "read-and-prewrite" request performs the necessary timestamp checks both for a read and a prewrite, accepting the request only if both checks succeed; that is $opts > rts$ and $opts > wts$. An accepted request is queued if $opts > minpts$ because there is an outstanding

prewrite with a lower timestamp, otherwise data is returned to the host and rts is updated provided that $opts > rts$. When the host has received all the data from the accepted "read-and-prewrite" requests, it computes the new parity and sends the new data and parity in low-level write I/O requests also tagged with $opts$. The devices are guaranteed not to reject these writes by the acceptance of the prewrites. In addition to doing the write, each device will update wts, discard the corresponding prewrite request from the queue, and possibly increase $minpts$. The request queue is then inspected to see if any read or read-and-prewrite requests can now be completed. Under normal circumstances, the read-modify-write protocol does not incur any overhead, just like piggy-backed device-based locking.

Several optimizations can be applied to the implementation of timestamp ordering to achieve greater efficiency. These optimizations are described next. They were implemented in the simulation and all graphs include their effects.

Minimizing buffering overhead

The protocol as presented can induce a high space overhead to buffer written data when there is high overwrite and read activity to overlapping ranges. If the storage device has accepted multiple prewrites to a block, and their corresponding writes are received out of order, the writes with largest timestamps have to be buffered and applied in timestamp order to satisfy any intermediate reads in order. Our approach for avoiding excessive data write buffering is to apply writes as soon as they arrive, rendering some later writes unnecessary and some later reads with a lower timestamp impossible. As a result, these later reads with a lower timestamp will have to be rejected. Although readers can starve in this protocol if there is a persistent stream of writes, this is unlikely at the storage layer. An important advantage of this immediate write processing approach is that storage devices can exploit their ability to stream data at high data rates to the disk surface.

Avoiding timestamp accesses

The reader may have already noticed that the protocols require that the pair of timestamps rts and wts associated with each disk block be durable, read before any disk operation, and written after every disk operation. A naive implementation might store these timestamps on disk, nearby the associated data. However, this would result in one extra disk access after reading a block (to update the block's rts), and one extra before writing a block (to read the block's previous wts). Doubling the number of disk accesses is not consistent with the desired goal of achieving "high-performance".

As long as all clocks are loosely synchronized (to differ by a bounded amount) and message

	Host latency (msec)	**Throughput (ops/sec)**	**Messages per operation**
Centralized locking	63.8	800	12.6
Callback locking	57.0	800	11.2
Device-served locking	51.8	800	14.8
Timestamp ordering	51.6	800	10
Zero-overhead protocol	48.7	800	8.6

Table 4.3: Summary performance results of the various protocols under the baseline workload. The zero-overhead protocol does not perform any control work, simply issuing I/Os to the devices and, hence, it does not provide any correctness guarantees.

delivery latency is also bounded, a device need not accept a request timestamped with a value much smaller than its current time. Hence, per-block timestamp information older than T seconds, for some value to T, can be discarded and a value of $NOW - T$ used instead (where NOW stands for current time). Moreover, if a device is re-initiated after a "crash" or power cycle, it can simply wait time T after its clock is synchronized before accepting requests, or record its initial synchronized time and reject all requests with earlier timestamps. Therefore, timestamps only need volatile storage, and only enough to record a few seconds of activity.

In our implementation, a device does not maintain a pair of timestamps for each block on the device. Instead, it maintains per-block read and write timestamps only for those blocks that have been accessed in the past T seconds. These recent per-block timestamps are maintained in a data structure, known as the timestamp log. Periodically, every T seconds, the device truncates the log such that only timestamps that are within T seconds of current time are maintained. T is known as the log truncation window. If an access is received to a block but the block's timestamp is not maintained in the log, the block is assumed to have a timestamp of $NOW - T$, where NOW stands for current time.

To understand why log truncation does not result in unnecessary rejections, recall that host clocks are loosely synchronized to within tens of milliseconds and message latency is bounded, so a new request arriving to a device will have a timestamp that is within tens of milliseconds of the device's notion of current time. The device rejects a request if its timestamp $opts$ does not exceed the maintained values of rts and wts. In particular, the device will *unnecessarily* reject the request when $opts$ exceeds the "real" read and write timestamps, $realrts$ and $realwts$, but when $opts$ fails

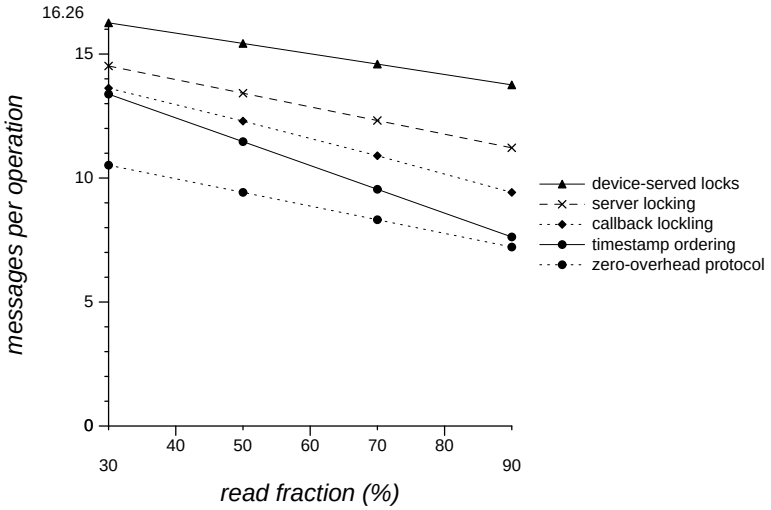


Figure 4.19: The effect of (read/write) workload composition on messaging overhead for the various protocols. Under read-intensive workloads, timestamp ordering comes close to the zero-overhead protocol. Its messaging overhead increases as the fraction of writes increases. However, it still performs the least amount of messaging across the range of workloads.

to exceed the truncated values used by the device, rts and wts. Considering only rts, this means that:

$$realrts < opts < rts$$

Replacing rts by its truncated value of $NOW - T$, the inequation becomes:

$$realrts < opts < NOW - T$$

Thus, for a request to be rejected unnecessarily, it must be timestamped with a value $opts$ that is more than T seconds in the past, where T is the log truncation window. This can be made impossible in practice by selecting a value of T that is many multiples of the clock skew window augmented by the network latency. A T of a few seconds largely satisfies this need. This minimizes the chance that a request will be received and rejected because of aggressive reduction of the number of accurately maintained timestamps kept in the device cache.

In addition to being highly scalable as illustrated in Figure 4.18, another advantage of timestamp ordering is that it uses the smallest amount of messaging compared to all the other protocols (Figure 4.19). It has no messaging overhead on reads, and with the piggy-backing optimization applied,

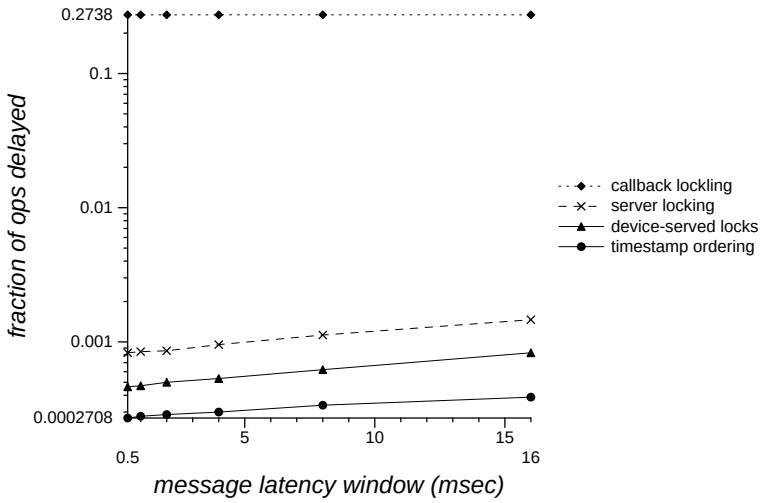


Figure 4.20: The effect of the variability in network message latencies on the fraction of operation delayed or retried for the various protocols. Message latency is varied uniformly between 500 microseconds and a maximum value, called the window size. This is the window of variability in network message latencies. Higher window sizes model a network that has highly unpredictable (variable) message latencies. The graph plots the fraction of operation delayed or retried against the size of the variability window size.

it can also eliminate the messaging overhead associated with read-modify-write operations.

4.5.6 Sensitivity analysis

This section reports on the performance of the protocols over unpredictable network latencies (causing interleaved message deliveries) and using faster disks.

Sensitivity to network variability

When several operations attempt to access a conflicting range, the succeeding operations are delayed until the first one completes. The probability of delay depends on the level of contention in the workload. But even for a fixed workload, the concurrency control protocol and environmental factors (e.g. network reordering of messages) can result in different delay behavior for the different protocols. As shown in Figure 4.20 and Figure 4.21, the fraction of operations delayed is highest for callback locking because it has the highest window of vulnerability to conflict (lock hold time). Moreover, its lock hold time is independent of message time variability because it is based on lease hold time.

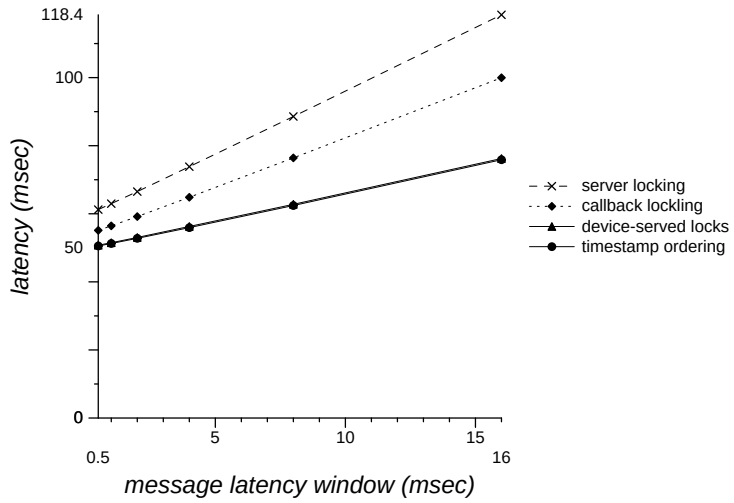


Figure 4.21: The effect of the variability in network message latencies on host latency for the various protocols. Message latency is varied uniformly between 500 microseconds and a maximum window size. The graph plots latency against the size of the variability window size.

Distributed device-based protocols both do better than callback locking and server locking because they exploit piggy-backing of lock/ordering requests on the I/Os, thereby avoiding the latency of communicating with the lock server before starting the I/O and shortening the window of vulnerability to conflict. Both device-based protocols, however, are potentially more sensitive to the message transport layer, or more precisely, to message arrival skew. Message arrival skew can cause deadlocks and restarts for device-served locks, and rejections and retries for timestamp ordering because concurrent multi-device requests are serviced in a different order at different devices. Restarts and retries are also counted as delays and are therefore accounted for in Figure 4.20.

To investigate the effect of message skew on the delay and latency behavior of the protocols, an experiment was conducted where message latency variability was changed and the effects on performance measured. Message latency was modeled as a uniformly distributed random variable over a given window size, extending from 1 to *ws* milliseconds. A larger window size implies highly variable message latencies and leads to a higher probability of out-of-order message arrival. Figures 4.20 and 4.21 graph the fraction of operations delayed and host end-to-end latency against the network delay variability window size *ws*. All schemes suffer from increased variability because it also increases the mean message delivery time. However, timestamp ordering and device-based locking slow down little more than the zero-overhead protocol. But high message variability plagues

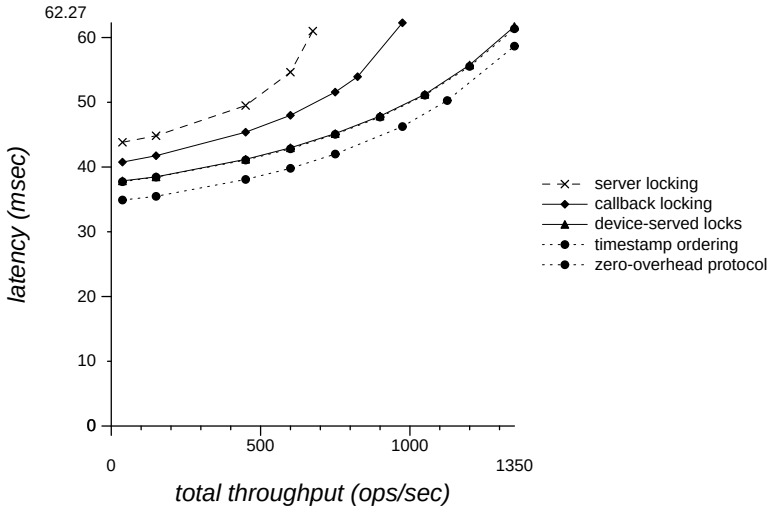


Figure 4.22: The scalability of callback locking, device-served locking and timestamp ordering under a 40% disk cache hit rate. The bottom-most line represents the performance of the zero-overhead protocol. While timestamp ordering and device-served locking continue to approximate ideal performance at higher hit rates, callback locking bottlenecks at a fraction of the achievable throughput.

the centralized locking variants significantly more since they perform more pre-I/O messaging.

Sensitivity to faster disks

Finally, one experiment was carried out to try to anticipate the performance of the protocols when faster disks are used. Disk drive access performance is expected to keep growing as a result of evolutionary hardware technology improvements (e.g. higher densities leading to higher transfer rates), the introduction of new technologies (e.g. solid-state disks leading to reduced access times) or device economics (dropping memory prices and increased device functionality [Gibson et al., 1998] leading to larger on-disk cache memory sizes and thereby reduced access times).

To simulate the effect of faster disks, an experiment was carried out where the hit rate of the disk cache was increased and scalability of the protocols under these seemingly faster disks measured. As shown in Figure 4.22, callback locking does not keep up with the throughput of the faster disk drives. Device-served locks and timestamp ordering, on the other hand, continue to approximate ideal scaling behavior.

4.6 Caching storage controllers

In the discussion so far, we assumed that data caching is performed on the device side. The client-side controllers did not cache data or parity blocks. This design implies that a read by a BST executing at a storage controller invokes a message to the device storing that data. The device services the read and returns the data to the requesting controller. The controller passes this data to the host or uses it to compute the new parity, but in either cases discards it after the BST completes.

Such a cache-less controller design is acceptable when the storage network connecting devices and controllers has relatively high-bandwidth and low latency. In such a case, the time taken by a controller to read a block from a device's cache is comparable to a local read from the controller's own memory. If this holds, there is benefit from avoiding double-caching the data blocks at the storage controller. Controller-side caching is wasteful of memory since a block would be soon replicated in the device's cache and in the controller's cache. It also induces unnecessary coherence traffic when blocks are cached at multiple controllers have to be kept up-to-date [Howard et al., 1988, Lamb et al., 1991, Carey et al., 1994]. This coherence traffic can have a negative effect on performance under high contention.

False sharing occurs when higher level software is writing to objects that are smaller than the block-size of the controller's cache, called the "cache block size." In this case, two applications or application threads running on two clients can be writing to two objects which happen to fall in the same "storage block." Such a scenario induces coherence traffic between the controllers to keep their copies of the block up-to-date even though there may be no overlapping accesses. Another kind of false sharing arises from contention over parity blocks when two controllers write to the same stripe. Because false sharing is expected at the storage layer, it is generally undesirable to cache at the parallel controller or host unless the network is substantially slower than local accesses. Another reason against caching is the fact that higher level system software, e.g. filesystems and databases, have their own caches, which will absorb "locality induced" reads. Replicating these blocks in the host's filesystem buffer cache and in the controller's cache is wasteful of memory.

So, if the filesystem or database cache absorbs most application reads, when is caching at the parallel controller useful at all? Caching at the controller can yield performance benefits in a number of situations. First, caching at the controller avoids the network and reduces the load on the devices. When the network is slow, this can translate into dramatic benefits. Furthermore, offloading the devices improve scaling. For example, controller-side caching can eliminate the "read phase" in two-phased BSTs. Many BSTs described in Section 4.3 pre-read a data or a parity block before

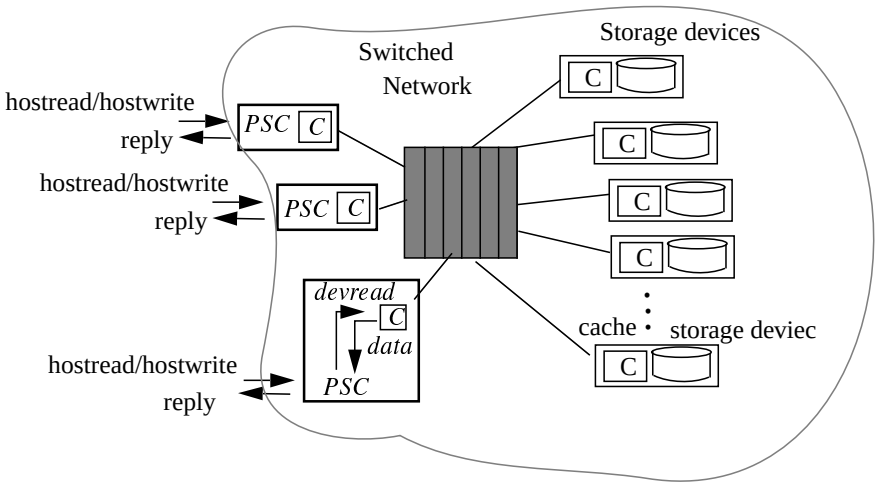


Figure 4.23: In a caching storage array, parallel storage controllers receive host requests and coordinate access to the shared storage devices. Data blocks are cached at the device where they are stored and also on the storage controllers. Consequently, some reads in a BST are serviced from the cache on the controller, avoiding a network message to the device.

writing it. Caching these blocks on the controller side can avoid network transfers from the device caches. This can translate into dramatic reductions in latency when the network is overloaded, and when there is little write contention across controllers for blocks. Low contention increases the chance that a block cached at controller A is not updated by another controller before controller A accesses the same block again, making caching it worthwhile.

Caching at the parallel controller can also prove useful when the application working set does not fit in the filesystem cache. In this case, the controller's cache can be used to cache blocks that have been evicted from higher-level caches (the filesystem buffer cache for example). The controller in this case must coordinate its cache replacement policy with the higher level cache. Recent work has shown that such an approach can yield sizeable benefits for certain applications [Wong and Wilkes, 2000].

Figure 4.23 depicts a shared array where data blocks are cached at the parallel storage controllers as well at the storage devices. When a storage controller caches data and parity blocks, pre-reads can be satisfied from its local cache. Precisely, a read by a BST executing locally can be serviced from the local cache.

The distributed serializability protocols discussed in the previous section do not readily apply to this architecture. Both device-served locks and timestamp ordering rely on the storage device

receiving requests to perform concurrency control checks before servicing them. Going to the device for serializability checks when the controller has the data in its cache seems to defeat the purpose of client-side caching. This section focuses on the two distributed protocols which were shown in the previous sections to have nice scaling properties. It outlines how they can be extended to allow effective controller-side caches. We assume that for both protocols, storage controllers cache data blocks in a local cache, called the controller cache. A *devwrite* writes data through the cache to the storage device, updating both the local copy and the storage device. A *devread* is serviced by checking the cache first. If a valid copy exists, the block is read locally, otherwise a it is retrieved from the storage device.

4.6.1 Device-served leases

As in device-served locking in Section 4.5.4, locks are acquired from the devices. The protocol is also two-phase, guaranteeing serializability: All locks needed for the execution of a BST are acquired before any of them is released. The key difference between device-served locking and device-served leases is that in the latter, the locks are not released immediately after they are acquired in the first phase. Instead, locks are cached at the host. Consequently, both a block's contents and the associated lock are cached by the controller. A *devread* is serviced exclusively from the controller's local cache if both the block and a lock for it are found in the controller's cache.

Like the callback locking approach of Section 4.5.2, locks expire after a specified time period, hence the name "lease". This time period is known as the lease duration. A lock is valid until the lease duration expires or until it is explicitly revoked by the device in a *revokeLease* message (which is analogous to a callback). When a lease expires or is revoked by the device that granted it, the block in the cache is considered invalid and is logically discarded from the cache. A BST *devread* would then have to be forwarded to the device. The *devread* is piggy-backed with a lock request as in device-served locking. The controller sends a *lock-and-devread* message to the device, which responds with the data after the lock can be granted to the requesting controller.

Figure 4.24 depicts how these BSTs break down into basic operations. The piggy-backing optimization is still applied. If a block is not found in the cache, a *lock-and-devread* request is sent to the device where the block is stored. However, locks are not released in the post-read phase, but are instead cached locally. On the device-side, requests to lock, read and write are serviced as in device-served locking, except for one difference. In device-served locking, locks are not cached and are released immediately after the BST completes. Thus a device that can not grant a lock due to

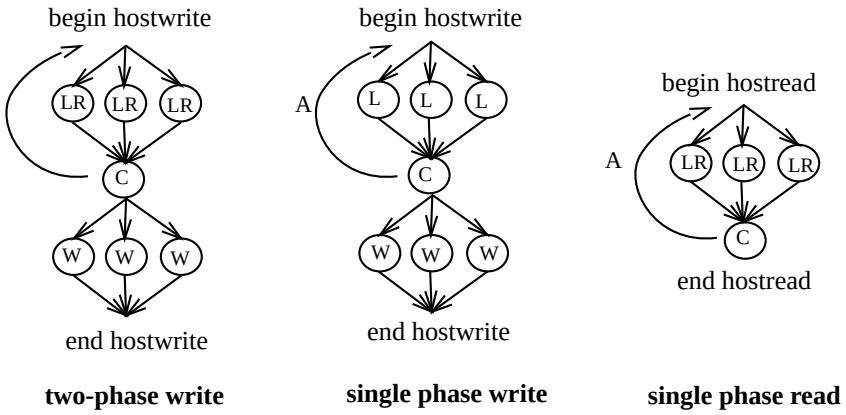


Figure 4.24: The breakdown of a host operation with device-served leasing with piggy-backing. An 'L' node represents a *lock* operation, 'LR' represent the *lock-and-devread* operation, while 'W' stands for *devwrite*. A lock ('L') request is satisfied locally if the lock is cached and the lease has not yet expired. A 'LR' is satisfied locally if the block is in the cache and the associated lease is valid. If the lease is invalid, then a message is sent to the device to refresh the lock ('L' request) or to refresh the lock and read the block ('LR' request). The edges between the nodes represent control dependencies. A 'C' node represents a synchronization point at the host as before.

a conflicting outstanding lock queues the request until the conflicting locks are released. However, in device-served leasing, the device does not wait for the controller to release the lock, since the controller is caching it and will not spontaneously release it. Thus, the device explicitly revokes all valid conflicting locks granted (locks that have not expired) by sending *revokeLease* messages to the caching controllers.

To simplify recovery, it is required that a controller not update any block until the commit point, when all needed locks have been successfully acquired. However, in device-served leasing, locks can eventually expire leading to some tricky complications. It is possible that a controller acquires the first lock, but waits to acquire the remaining locks so long that the first lock becomes invalid (the lease period passes). In this case, the first lock is re-acquired. The commit point is reached only after all the locks have been acquired and are all valid.

On recovery from a soft fault (e.g. power-fail), the device can not grant a lock to a controller if a valid conflicting lock exists at another. If information about outstanding leases at the device is maintained in volatile memory, the device must wait until a safe period after restart before it can grant locks again. This safe period can be as short as the lease period but not shorter.

The recovery protocols discussed in Section 4.7 require, for performance reasons, that the device

be able to identify the set of leases that may be valid and outstanding at the controllers at the event of a failure. This identification does not have to be accurate and can over-approximate this set, as long as it is a relatively small fraction of the blocks on the disk.

4.6.2 Timestamp ordering with validation

Device-served leasing distributes control work across the devices and hosts. It does not have a central scalability bottleneck. Nevertheless, it has two serious disadvantages. First, it can suffer from degraded performance under contention. Lock acquisition can have substantial latency due to revocation messaging. Furthermore, spurious restarts under contention can cause degraded performance. Second, it is relatively complex to implement. Timestamp ordering has the scalability advantages of device-served leasing without the vulnerability to contention or to spurious restarts (since it is deadlock-free). Timestamp ordering, however, does not readily support caching and it has no locks which can be easily overloaded to ensure cache coherence.

To extend timestamp ordering to support controller-side caching in a straight-forward manner, we will replace cache hits with version test messages to the devices. This will only have a negative on read hit latency if all data needing to be read is in the controller's local cache. The controller caches data blocks together with their write timestamps, wts. Each block in a controller's cache is associated with an wts, which is the write timestamp publicized by the device for that block when the controller read the block into its cache. Intuitively, this wts can be thought of as a version number. If another controller writes its copy of the block, the wts for that block is updated on the device, logically making the version cached by other controllers invalid. It suffices to compare the local wts in the controller cache with that maintained at the device to verify whether the device and controller blocks are the same.

The controllers and devices behave as in the basic timestamp ordering, except for the read phase of a single or two-phase BST. In this case, the controller services the read from its local cache if a local block exists. To validate that the local version of the block is not stale and thereby is safe to read, a *readIfInvalid* [1] message is sent to the device for each block. This message contains the timestamp of the BST, $opts$, and the write timestamp of the block read from the cache, wts. If the wts of the cached block matches the wts maintained by the device, then the read performed by the

[1] This means read if the local copy cached at the host is invalid. The device either returns an OK response validating that the local copy cached at the host is valid, or, if the validation fails, returns the new contents of the block and the new associated wts.

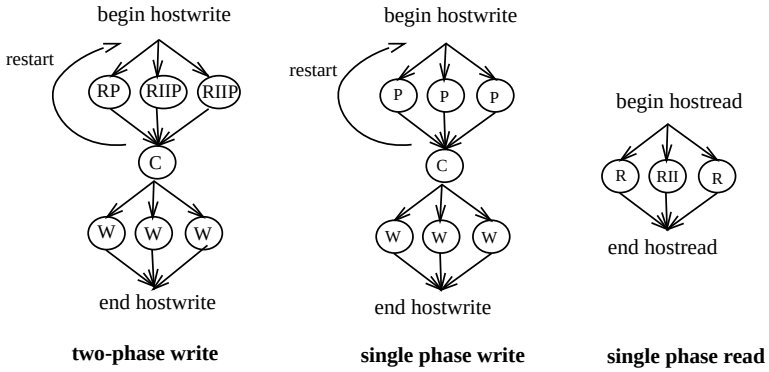


Figure 4.25: The composition of host operations in the TSOV protocol. *devread*, *devwrite*, and *prewrite* requests are denoted by 'R', 'W' and 'P' nodes respectively. A 'RII' denotes a *readIfInvalid* request which is issued in lieu of a *read* if the block is found in the cache. This request includes the *wts* of the cached block and instructs the device to read and return the block only if the cached version is invalid. If the cache block is valid, the *rts* is updated at the device and a positive reply is returned. A 'RP' denotes a *read-and-prewrite* request. An 'RP' request is issued if the block is not in the cache. If the block is found in the cache, a *readIfInvalid-and-prewrite* ('RIIP') is issued. A 'C' node represents a synchronization barrier at the host as before.

controller from its own cache is valid. In this case, the device updates the rts of the block to $opts$ and returns an OK response. If the wts of the cached block is different from that maintained at the device, then the read is not valid. In this case, two things can happen. If $opts > wts$, the block is returned and rts is updated to be $opts$ if $opts$ exceeds rts. If $opts < wts$, the request is rejected and the client will retry with a higher timestamp. This protocol is called timestamp ordering with validation (TSOV).

Figure 4.25 depicts how BSTs break down into basic device operations. In TSOV, the device checks prewrites and writes as in the basic TSO protocol. Reads are handled differently as explained above. In the case of a two-phase BST, the read-and-prewrite becomes a *read-and-prewrite* if the block is not found in the cache or a *readIfInvalid-and-prewrite* request if it is. If validation succeeds, no data is returned by the device and the block in the local cache is used.

The device maintains its timestamp log by rounding up all wts's that are more than T seconds in the past to the value of $NOW - T$. As explained in Section 4.5.5, this does not compromise the correctness of basic timestamp ordering. In the case of TSOV, the effect of this rounding up can result, however, in *readIfInvalid* requests failing validation despite the fact that the cached copy at the host is valid. When the write timestamp associated with a block is truncated, i.e. increased, the next *readIfInvalid* request from a host will fail because the wts maintained by the host will be

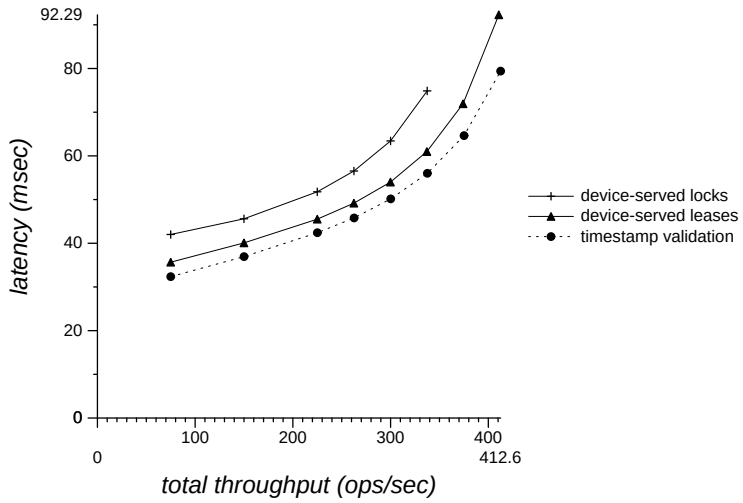


Figure 4.26: The scalability of device-served leases and timestamp ordering with validation. The storage controllers have a local cache that is big enough such that no blocks are evicted because of limited cache capacity. The lease duration is 240 seconds, and the timestamp log truncation window is 300 seconds. The graph also shows the performance of device-served locking (which is also representative of that of basic timestamp ordering) when the controllers do not cache any blocks.

smaller than the truncated value maintained by the device. This induces an unnecessary read of the data from the device. Therefore, in TSOV, the timestamp log truncation window must be longer than a few seconds. For the implementation evaluated in this section, a value of 5 minutes was used.

The following subsections compare and contrast the performance of the cache coherent distributed protocols. In particular, the evaluation criteria are taken to be the latency of completing a BST, the amount of network messaging performed, the fraction of operations delayed or blocked, the size of device-side state, and the implementation complexity of the protocols (both at the devices and at the controllers).

4.6.3 Latency and scaling

Recall that a caching storage controller forwards *devwrites* to the device under both protocols. *Devreads*, however, are handled differently. Under TSOV, a control message is sent to the device to validate a locally cached block and local contents are read if the validation succeeds. If the validation fails, the device returns the new contents of the block or the request is rejected (due to an unacceptable timestamp). Consider the case of low contention, where blocks cached at one con-

troller are not soon written by another. In this case, the validation will often succeed. The difference between TSO and TSOV is that the latter converts the pre-reads or reads in case of a cache hit to a control message exchange with the device. Device-served leasing (DLE) completely eliminates messaging in case of a cache hit. The block is read from the cache if the locally cached lease is valid. However, when a lock is not cached locally at the host, one must be acquired from the device. The device must recall all the conflicting and outstanding leases cached by all hosts before responding to the requesting host. This induces latency when an exclusive lock is requested by a controller for a block that has been cached at many hosts in shared mode.

The major vulnerability of device-served leasing is that this work is performed by the device and not by the controller which can load the device in large systems. Under timestamp ordering with validation, a host A writes to the devices by sending a prewrite message in a first phase followed by a write message in a second phase. Other hosts that have a cached copy of the block written to by host A are not notified. They will discover that the block has been updated when and if they try to validate the read later. The work to keep the caches coherent is delayed until (and if) an access is performed.

When each host accesses a different part of the device, leasing works well. However, when controllers share access to the same devices and objects, device-served leasing can be expected to suffer because of increased messaging and device load. Shared arrays are not characterized by separate locality at each controller, however. First, because cluster applications running on top of them often balance load dynamically across the hosts resulting in blocks being accessed by multiple hosts within a short period of time. Second, even if applications have locality in the logical address space, RAID can map two logically different objects to the same stripe set. This induces contention not only on the parity blocks but also on the data blocks themselves due to RAID write optimizations which sometimes require reading blocks that are not being updated. Under high contention, validation messages of TSOV will often fail, and the device will then return the new contents of the block. TSOV thus reduces in this case to basic timestamp ordering with no caching. DLE under high contention also generates a lot of lease revocations which also make it perform as basic device-served locking. The *revokelease* message in DLE is the equivalent of the *unlock* message in device-served locking. However, a device under DLE is expected to suffer from blocking longer because its locks are distributed across hosts and revocations will tend to be queued and wait for service at more nodes. More importantly, DLE is more vulnerable to deadlocks than basic device-served locking because longer blocking times cause more spurious deadlock detection time-outs, each initiating

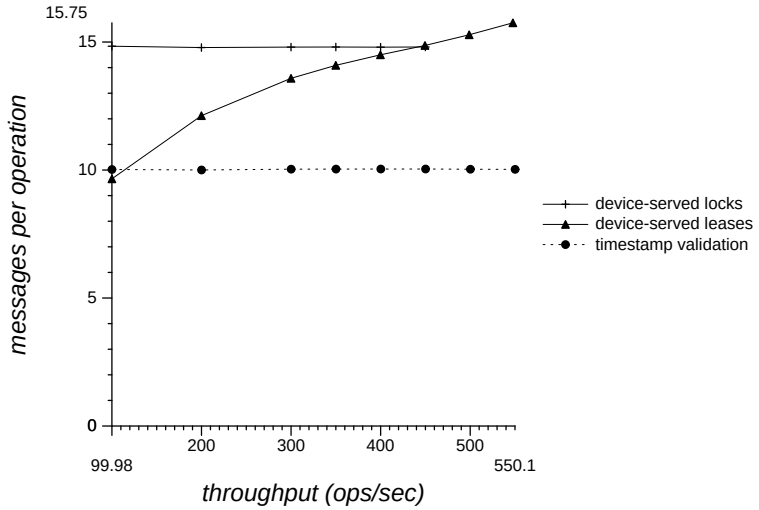


Figure 4.27: The messaging overhead of device-served leases and timestamp ordering with validation, compared to that of device-served locking. The latter assumes no controller-side caches. Note that under high load, device-served leasing (DLE) induces more messging than device-served locking. This is because the fraction of operations retried (to avoid a likely deadlock) under high load is larger under DLE than under device-served locking. Deadlock-induced timeouts are more likely under DLE because lock hold times are longer under DLE (lease duration) than under device-served locking (duration of the operation).

restarts which further loads the devices.

The baseline workload and simulation parameters described in Table 4.2 are used to evaluate the caching protocols, except for the system being third as large (10 devices and 8 hosts). Simulating the more complex caching protocols requires more resources making large scale simulations impractical. All the graphs in Section 4.6 and later correspond to 8 hosts and 10 devices.

Figure 4.26 plots latency versus throughput of TSOV, DLE and device-served locks. Device-served locking corresponds to the performance in the absence of host-side caching. Suprisingly, the graphs shows that timestamp ordering exhibits lower latencies than DLE. Both TSOV and DLE reduce latencies compared to device-served locking without host-side caching. However, the caching benefit of DLE is somewhat offset by the increased load on the devices for lease management and recall as well as by the increased messaging leases induce when hosts contend for a shared storage space.

4.6.4 Network messaging

TSOV does not reduce the number of messages sent on the network over TSO although it converts some data transfers to control messages, reducing the total number of bytes transferred. DLE, on the other hand, eliminates pre-reads and reads altogether when a request hits in the cache but requires revocation messaging when data is shared. Figure 4.27 plots the average number of messages per operation (that is, per BST) for each protocol. TSOV has a relatively constant number of messages per BST, and equal to that of basic TSO. Similarly, device-served locking has a relatively constant messaging overhead. DLE starts with the lowest messaging overhead when the number of hosts is limited (2) and few lease revocations occur. As the number of hosts in the system increases, the amount of messaging required to revoke leases increases. At the same time, leases are revoked more often, requiring hosts to re-acquire them more frequently. Under high throughput, this degrades to worse than the performance of device-served locking because of the large number of operations retried under DLE. This is due to the fact that DLE is more vulnerable to the deadlock detection time-outs.

When a BST is started at a host, it often partially hits in the cache such that some blocks and their leases are found in the cache while some others are not. The blocks that are not in the cache or for which no valid leases are cached must be re-fetched from the device. This hold-and-wait condition of holding some locks locally and attempting to acquire the rest (from multiple devices) opens the possibility of deadlocks. Both device-served locking and DLE have a similar time-out based deadlock detection mechanism at the devices. However, DLE suffers much more timeout-induced restarts. This is because DLE holds locks longer by caching them and therefore is more vulnerable to deadlocks with many more BSTs that start while the locks are locally cached. Furthermore, because the lease revocation work at some devices can take considerably long, deadlocked detection time-outs can often expire in the meantime.

4.6.5 Read/write composition and locality

Under a predominantly read workload, where blocks are rarely invalidated by writes from other hosts, device-served leasing yields similar latencies to timestamp ordering with validation as shown in Figure 4.28. Figure 4.29 graphs the messaging overhead of the protocols as a function of the read traffic ratio in the baseline workload. Under a predominantly read workload, device-served leases induces lower messaging overhead due to the large fraction of local cache hits. However, as the fraction of writes increase, and because hosts access a shared storage space uniformly randomly in

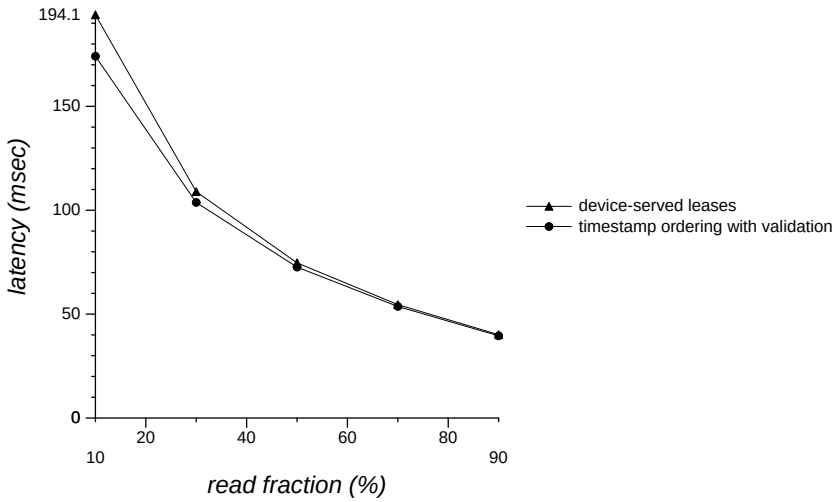


Figure 4.28: The effect of the read/write workload mix on host latency for device-served leases and timestamp ordering with validation.

this workload, the number of callbacks increase. This makes the messaging overhead of device-served leases higher than that of timestamp ordering.

When each host has its own "working set" of blocks that no other hosts can access, then acquiring a lock and caching it becomes more appealing. Under such a workload, DLE should of course exhibit lower latencies than TSOV because it eliminates many read messages while TSOV converts them into control messages. While such a workload is not typical of clusters and of shared arrays, it is valuable to quantify and bound the benefit of DLE over TSOV. Under baseline parameters and perfect locality (no revocations from other hosts), DLE was found to exhibit 20% lower latencies than TSOV.

In the reported experiments, the lease time was 240 seconds, this lease time was configured to give device-served leasing the best performance for this workload. Nevertheless, under such a shared uniformly random workload, timestamp ordering with validation is still preferrable and more robust to changes in contention, read/write composition and network message latency variability. The sensitivity of DLE to lease duration is explored below.

4.6.6 Sensitivity to lease duration

Lease duration impacts both the concurrency control and recovery protocols. Shorter leases make recovery faster as discussed in Section 4.7. The duration of the lease can be good or bad for the

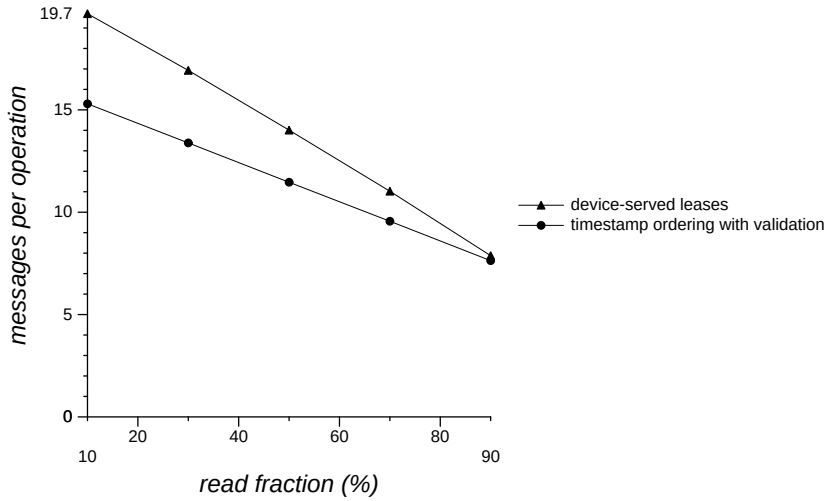


Figure 4.29: The effect of the read/write workload mix on messaging overhead for device-served leases and timestamp ordering with validation. Messaging overhead for both protocols decreases as the ratio of reads increases. When write traffic dominates, timestamp ordering with validation induces a lower messaging overhead than device-served leasing, which suffers from revocation messaging and time-out induced retries.

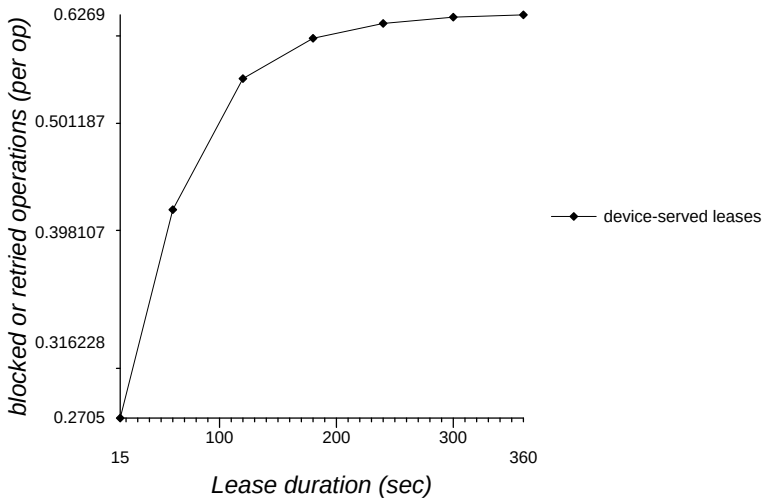


Figure 4.30: The effect of lease duration on the fraction of operation delayed under device-served leases. The longer the lease duration, the higher the likelihood that an operation is delayed at the device waiting for conflicting granted leases to be revoked.

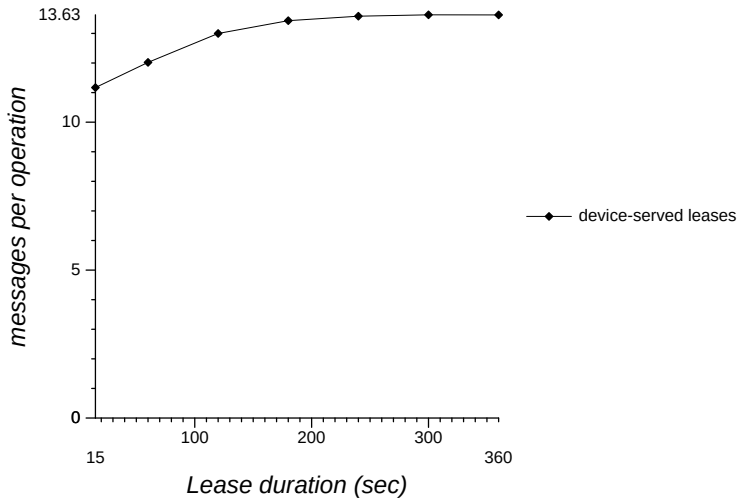


Figure 4.31: The effect of lease duration on the messaging overhead for device-served leases. Shorter leases result in lower lock cache hit rates at the controllers, but reduce the amount of revocation messaging needed when sharing occurs. Longer leases, on the other hand, reduce the need to refresh, but increase the likelihood of revocation messaging when sharing occurs. Their combined effect shows that the overall messaging overhead is minimized with shorter leases, although the difference is not daramtic.

performance of the concurrency control, depending on the workload. If the workload has high locality and low contention, then longer leases are better because they allow one read to satisfy more accesses before the lease is refreshed. Under high-contention, however, shorter leases are better because they minimize delays due to revocations. When a short lease is requested from a device, a good fraction of the previously acquired leases by other hosts would have already expired and so few revocations would result. This further reduces messaging overhead and device load contributing to observably lower latencies.

To investigate the effect of lease duration on DLE, the end-to-end latency, messaging overhead and the fraction of operations delayed were measured under different lease durations for the baseline configuration (8 hosts and 10 devices). Figure 4.31 shows the effect of lease duration on messaging overhead. Short leases require less revocations but also must be refreshed more often. Long leases induce more revocations but do not require refreshes unless they are revoked. Medium-length leases are the worst under the baseline workload because the sum of both effects is larger for them. Figure 4.30 demonstrates that longer leases cause operations to be delayed more often while conflicting outstanding leases are being revoked. Figure 4.32 summarizes the net effect of lease duration on

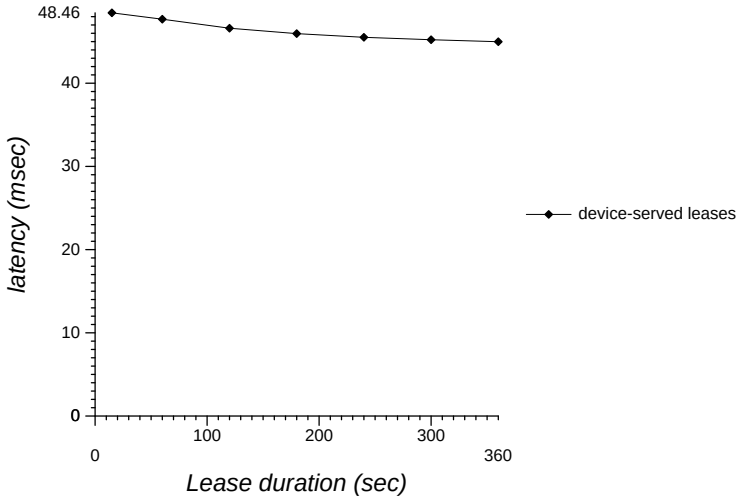


Figure 4.32: The effect of lease duration on host latency. Very short leases result in relatively higher latencies. Increasing lease durations beyond 240 seconds does not result in noticeably lower latencies.

end-to-end latency. It shows that under the baseline random workload, which has moderate load and contention, a lease exceeding few minutes is advisable.

4.6.7 Timestamp log truncation

Under timestamp ordering, devices maintain a log of recently modified read and write timestamps for the recently accessed blocks. To bound space overhead, the device periodically executes a log truncation algorithm. This truncation algorithm deletes all timestamps older than T, measured in seconds, which is the log size parameter. All blocks for which no timestamps (rts or wts) are found in the log are assumed to have an $rts = wts = NOW - T$ by the timestamp verification algorithms. For basic TSO, the log can be very small, holding only the last few seconds of timestamps as described before.

The timestamp log must be stored on non-volatile storage to support the recovery protocols discussed in Section 4.7. Because the timestamp log must be stored on non-volatile storage (e.g. NVRAM), it must be truncated frequently to maintain it at a small size. Section 4.5.5 argued that a very small log size is sufficient for basic timestamp ordering because clock skew and message delay are bounded. For TSOV, the log can not be very small because cache validation requests would be more likely to fail. If the log is truncated every few seconds then a host issuing a validation (RII or RIIP) several seconds after reading a block will have its validation failed, forcing the device to re-

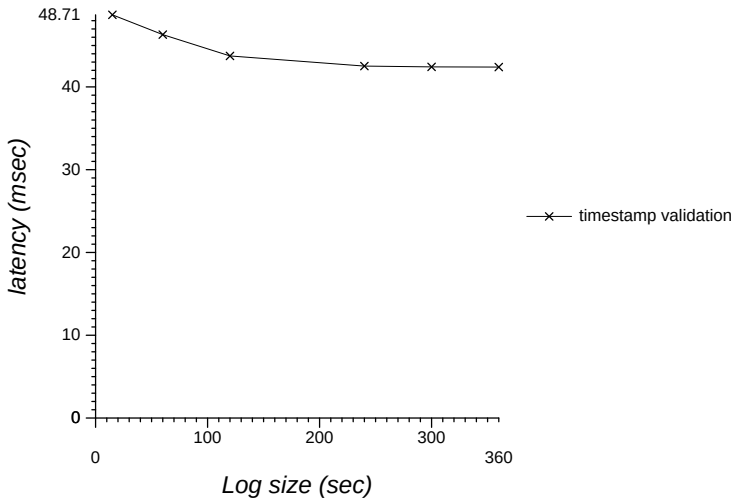


Figure 4.33: The effect of timestamp log size on host latency for timestamp ordering with validation. The log size is measured in seconds of activity. A log size of 200 seconds implies that the device truncates all timestamps older than 200 seconds ago to "current time - 200 seconds". The graph shows that a timestamp log of a few minutes is sufficient to achieve good performance for the baseline workload.

transfer the block to the host. Naturally, the timestamp cache "hit rate" (the fraction of accesses for which validations succeed) increases for TSOV with larger timestamp logs, or equivalently bigger values of T.

For the graphs in this section, the timestamp log size at each device was set to accomodate 5 minutes of timestamp records. To estimate the size of this log, recall that a typical disk device services a maximum of 100 disk operations per second, or 30000 ops in 5 minutes. That assumes that all requests miss in the device's cache and are therefore serviced from the platters. If 32 bytes are used to store the (block, timestamp) pair in a log data structure, this five minute log needs to 960 KB. Smaller log sizes offer good performance also. Note that if the cache hit rate is high, then the traffic is likely to have more locality. In this case, a more localized working set is likely to need a smaller number of timestamp records. Figure 4.33 supports the argument that small log sizes are sufficient, showing that a log size of only 200 seconds is sufficient to provide good performance for the random workload of Table 4.2.

	TSO	DLOCK	TSOV	DLE
Storage device (Lines of code)	1942	1629	1975	1749
Caching support at device (Lines of code)	—	—	33 (=1975-1942)	120 (=1749-1629)
Storage controller (Lines of code)	1821	1810	2008	2852
Caching support at controller (Lines of code)	—	—	187 (=2008-1821)	1042 (=2852-1810)

Table 4.4: Simulation prototype implementation complexity in lines of code. The numbers concern the basic concurrency protocols excluding the recovery part, which is largely shared across all protocols. The second and fourth rows of the table show the additional lines of code added to the device and to the storage controller to support block caching in each case. This is simply the additional lines of code added to TSO to make it TSOV and added to device-served locking (DLOCK) to make it DLE.

4.6.8 Implementation complexity

Although device-served leasing and timestamp ordering have similar performance, device-served leasing is relatively more complex to implement. Table 4.4 shows the lines of code needed to implement each of the protocols in detailed simulation. Except for more robust error handling, the protocols can be readily transplanted into a running prototype. The lines of code may therefore be representative of their real comparable implementation complexity. The table shows that while timestamp ordering and device-served locking are of relative complexity, their caching counterparts are quite different. While it took only 180 lines to add caching support for timestamp ordering at the storage controller, a thousand lines of code were needed to do the same for device-served locking.

The reason behind this difference in complexity is that DLE deals with the additional complexity of lease expiration and lease renewal, deadlock handling code, and lease reclamation logic. A lease held by a host can be reclaimed while an access is concurrently trying to acquire the locks. A lease from one device can expire because a lock request to another device touched by the same hostwrite was queued for a long time. All of these concerns are absent from the implementation timestamp ordering with validation. In the latter, only the write timestamp of the block in the cache is recorded and sent in a validation message to the device. No deadlocks can occur, no leases can expire, and no callbacks are received from the device.

4.7 Recovery for BSTs

Various kinds of failures can occur in a shared storage array. Devices and controllers can crash due to non-hard software or hardware bugs or due to loss of power. Correctness must be maintained in the event of such failures. This section discusses how a shared array can recover from such untimely failures. In particular, it describes the protocols that ensure the consistency property for BSTs, discussed in Section 4.4.

4.7.1 Failure model and assumptions

A shared storage array consists of four components: storage controllers, storage devices, storage managers and network links. From the perspective of this discussion, failures can be experienced by all four components. Network failures include operator, hardware or software faults that cause links to be unavailable, that is incapable of transmitting messages between nodes. This discussion assumes that all network failures are transient. Moreover, it assumes that a reliable transport protocol, capable of masking transient link failures, is used. This discussion also makes the simplifying assumption that a storage manager failure is always transient and masked. That is, the storage manager will eventually become available such that a communication with it will always succeed. Other work describes a technique to implement this in practice [Golding and Borowsky, 1999]. Therefore, this discussion will focus only on failures in clients and devices.

A device can undergo a permanent failure resulting in the loss of all data stored on it. A device can also undergo a transient failure, or *outage*, causing it to lose all the state stored in its volatile memory. Similarly, a controller can undergo a permanent failure which it does not recover from. It can also undergo a transient failure, or *outage*, after which it eventually restarts but causing it to lose all state in its volatile memory. For the rest of the discussion, a *failure* of a device or a controller will designate a permanent failure while an *outage* will designate a transient failure.

Figure 4.34 shows the protocols used in a shared storage array. The storage controllers use BST-based access protocols to read and write virtual storage objects. Access protocols involve controllers and devices. The layout maps used by the access protocols are fetched by the controller from the storage manager. Layout maps are associated with leases specifying the period of time during which they are valid. After this lease period expires, the controller must refresh the layout map by contacting the storage manager. A storage manager can invalidate a layout map by contacting the storage controller caching that map.

The storage manager changes a virtual object's mode or layout map only when no storage con-

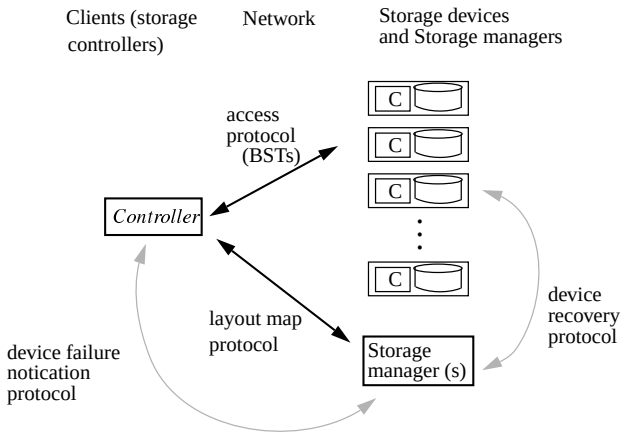


Figure 4.34: A shared storage array consists of devices, storage controllers, and storage managers. Storage controllers execute BSTs to access the devices. The BSTs require the controller to know the object's layout across the devices. A layout map protocol between the storage controller and storage manager allows controllers to cache valid layout maps locally. If a storage controller discovers a device failure, it takes appropriate local actions to complete any active BSTs then notifies the storage manager through the device failure notification protocol. Similarly, a device may encounter a controller failure after a BST has started but before it is finished. Such a device notifies the storage manager. This latter executes the proper recovery actions (device recovery protocol).

trollers are actively accessing that object. Precisely, a storage manager must make sure no controller has a valid layout map in its cache when it performs a change to the layout map. Layout maps are stored (replicated) on stable storage. A storage manager synchronously updates all replicas of a layout map when it switches a virtual object to a different mode or when it changes where storage for the object is allocated. This is acceptable given that virtual object maps are changed infrequently when objects are migrated or when devices fail.

For example, to switch a virtual object from fault-free mode to a migrating mode, the storage manager can wait until all outstanding leases expire. Alternatively, it can send explicit messages to invalidate the outstanding layout maps cached by storage controllers. At the end of this invalidation phase, the controller is assured that no storage controller is accessing storage since no valid layout maps are cached anywhere. At this time, the storage manager can switch the layout map of the virtual object and move it to a migrating mode. After this, the new map can be served to the storage controllers which will use BSTs specified in the map and corresponding to the migrating mode.

A storage controller can discover a failed device or can experience a device outage after a BST has started and before it has completed. Such exceptional conditions often require manager action

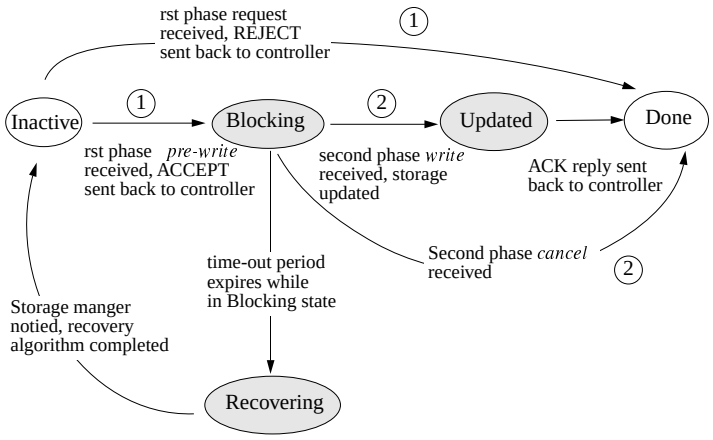


Figure 4.35: The states of a device during the processing of a BST. The device starts in the **Inactive** state. The transition to the **Blocking** state occurs with the receipt and the successful processing of a first phase *prewrite* request from a storage controller. If a second phase *write* message is received confirming the *prewrite*, the device updates storage and transitions to the **Updated** state. A reply is sent to the controller and a final transition to the **Done** state from the **Updated** state is performed. If the second phase message is a *cancel*, the device discards the *prewrite* and transitions to the **Done** state. If a time-out period passes while the device is in the **Blocking** state and without the receipt of any message, the device transitions to the **Recovering** state. The arrows labeled with a "1" ("2") refer to transitions that are caused by the receipt and processing of a first (second) phase message from the controller.

to restore consistency and properly complete the BST. A controller-manager protocol allows controllers to report such failure conditions to the storage manager (Figure 4.34). Similarly, a storage device can block in the middle of executing a BST waiting for a controller message that never arrives. The controller may have failed or restarted and lost all its state as a result. In this case, the device must notify the storage manager to properly complete the BST and ensure consistency. A device-manager protocol (Figure 4.34) is defined to allow devices to report incomplete BSTs to the storage manager.

4.7.2 An overview of recovery for BSTs

The discussion assumes that all storage manager failures and network failures are masked, leaving four remaining kinds of failures of interest: device failures, device outages, controller failures and controller outages. The amount of work required to recover properly from a failure or outage depends largely on when the event occurs. Figure 4.35 shows the states of a device involved in a BST. The device starts in the **Inactive** state. This discussion assumes a timestamp ordering protocol, but

the case of device-served locking is quite similar. Both protocols essentially export a lock to the data block after the first phase request is accepted at the device. This exclusive lock is released only after the second phase message is received. In the **Inactive** state, the device receives a first phase request, a *prewrite* or a *read-and-prewrite* request. If the request is rejected, a reply is sent back to the controller and the operation completes at the device and the device moves to the **Done** state. A new operation must be initiated by the controller to retry the BST.

If the request is accepted, the device transitions to the **Blocking** state. In this state, the block is locked and no access is allowed to it until a second phase message is received. The second phase message can be a *cancel* message or a *write* message containing the new contents of the block. If a *cancel* message is received, the device discards the *prewrite* and transitions to the **Done** state. If a *write* message is received confirming the *prewrite,* the device transitions to the **Updated** state once the data is transferred successfully to stable storage. The device then formulates a reply acknowledging the success of the write and sends it to the controller. Once the message is sent out, the device transitions to the **Done** state and the BST completes.

If a time-out period passes and no second phase message is received, the device transitions to the **Recovering** state. From this state, the device notifies the storage manager of the incomplete BST and awaits the manager's actions. The manager restores the array's consistency before allowing the device to resume servicing requests to the virtual object. Similarly, if the device experiences a failure in the midst of processing a *write* message such that storage is partially updated, the device transitions to the **Recovering** state and notifies the storage manager.

Figure 4.37 shows the states of a controller executing a BST. The figure shows a write BST. In the first phase, *prewrite* messages possibly combined with *read*s are sent out. These messages, marked with a "1" in Figure 4.35, cause the device to transition from the **Inactive** to the **Blocking** state if the *prewrite* is accepted or to the **Done** state if the *prewrite* is rejected. Once all the replies to these first phase messages are collected by the controller, a second phase round of message is sent out to confirm the write or to cancel it. These messages, marked by a "2" in Figure 4.35, cause the device to transition from **Blocking** to **Done** (in case of a *cancel*) or from **Blocking** to **Updated** (in case of a successfully processed *write*).

The state diagram of Figure 4.35 is helpful in understanding the implications of a device or controller failure or outage while a BST is in progress. A controller failure or outage before any phase 1 messages are sent out is *benign*, that is, it does not require the involvement of the storage manager to ensure recovery. Recovery can be achieved by local actions at the nodes. In this case,

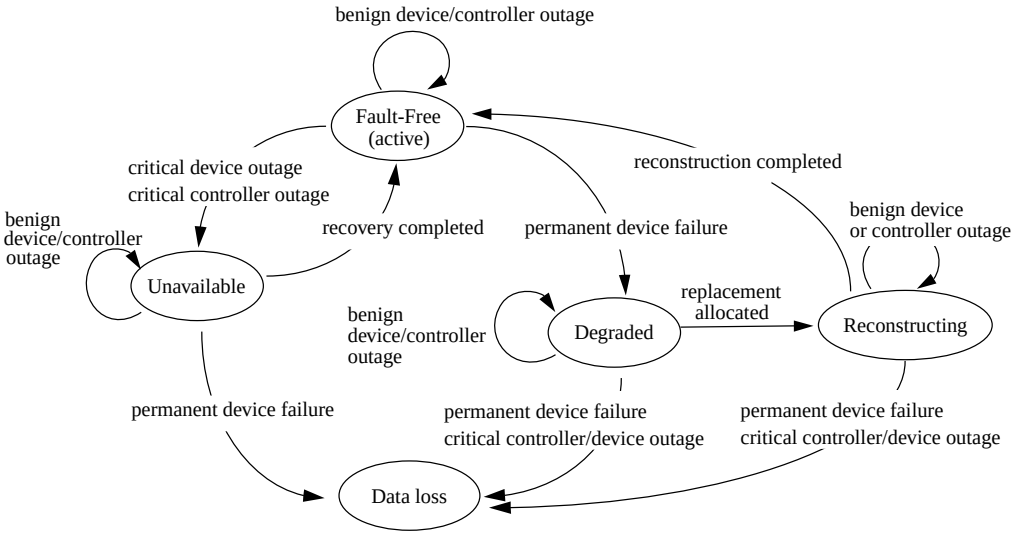


Figure 4.36: The modes of a virtual object and the events that trigger modal transitions. Benign failures and outages do not cause the object to switch modes. Critical device outages, device failures, and critical controller failures and outages cause the object to switch modes. A second critical failure while the object is in degraded, reconstructing or recovering modes is catastrophic and results in data unavailability.

upon restart, the controller must simply re-fetch the layout maps to begin access to storage. In this case, no device has transitioned to the **Blocking** state and no locks are held by the BST. If a controller experiences a failure or outage after all the second phase messages are sent out and successfully received by the devices, then the devices will update the blocks on persistent storage and complete the BST. The devices will transition from the **Blocking** to the **Updated** and unilaterally to the **Done** state. Such a failure is also benign. However, a controller failure or outage after at least one device has transitioned to the **Blocking** state and before all the second phase messages are sent out to the devices is considered *critical*; that is, will require special action by the storage manager to restore the array's consistency and/or to ensure progress.

Similarly, a device failure or outage before any device has reached the **Updated** state can be handled easily by the controller. Since no storage has been updated anywhere, a storage controller facing a permanent device failure or a device outage (inaccessible device) can simply abort the BST by multicasting a cancel message in the second phase to all the surviving devices. The device failure or outage can be considered to have occurred before the BST started. The storage controller notifies the storage manager of the problem and the storage manager moves the object to the proper mode.

However, if a device failure or outage occurs after *some* devices have received and processed

the second phase message and transitioned from the **Blocking** to **Updated** state, recovery is slightly more complicated. The storage controller completes the second phase by writing to all the surviving devices. This failure is considered *critical* because it requires special action by the storage manager on recovery. The manager must establish whether the device has permanently failed or has experienced an outage (restart). If the device has permanently failed, the failure can be considered to have occurred right after the device was updated. The BST is considered to have succeeded but the virtual object must be moved to a degraded mode and reconstruction on a replacement device must be soon started. If the device has experienced an outage, the device should not be made accessible immediately upon restart since it still contains the old contents of the blocks. The storage manager must ensure that the data that the BST intended to write to the device is on stable storage before re-enabling access. This data can be reconstructed from the redundant copy written by the BST.

A virtual object can be in one of several modes: Fault-Free, Migrating, Degraded, Reconstructing, Unavailable or Data-loss. The first four modes were already introduced. The last two were not because they do not pertain to the concurrency control discussion. In the last two states (Unavailable and Data-loss), host accesses to the virtual object are not allowed. In the Data-loss mode, the object is not accessible at all. In the Unavailable mode, the storage manager is the only entity which can access the object. In this mode, the storage manager restores the consistency of the array. Once the array is consistent, the manager moves the object to an access mode and re-enables access to the object.

Figure 4.36 represents a sketch of the different modes of a virtual object. The object starts out in Fault-Free mode. A permanent device failure causes a transition to the Degraded mode. The allocation of a replacement device induces a transition to the Reconstructing mode from the Degraded mode. A second device failure or a critical outage (of a device or controller in the midst of a BST) while the object is in degraded or reconstructing modes results in data loss. The object transitions to the Data-loss mode and its data is no longer available. Critical outages of controller or device while the object is in Fault-Free mode cause a transition to the Unavailable mode where the object is not accessible until the array's consistency is restored. In this mode, no storage controller can access the object because no new leases are issued and previous leases have expired. The storage manager consults a table of actions which specifies what recovery procedure to take for each kind of failure and BST. A compensating BST is executed to achieve parity consistency, the object is potentially switched to a new mode, and then new leases can be issued. Benign failures and outages, on the other hand, do not cause the object to change modes. They do not require any

recovery work besides possibly a local action by the node experiencing the outage upon restart.

4.7.3 Recovery in fault-free and migrating modes

Under Fault-Free and Migrating modes, all devices are operational. The discussion will focus on a single failure or outage occurring at a time. The pseudo-code executed by the devices is given in Section 4.7.6.

Controller outage/failure. Controller failures and outages are essentially similar. A permanent controller failure can be regarded as a long outage. Because a storage controller loses all its state during an outage, it makes no difference to the storage system whether the controller restarts or not. Any recovery work required to restore the array's consistency must proceed without the assistance of the failed storage controller. A controller outage amounts to losing all the layout maps cached at the client. After restart, the storage controller must re-fetch new layout maps from the storage manager to access virtual objects. Upon restart, no special recovery work is performed by the controller.

When the virtual object is in Fault-Free/Migrating mode and the storage controller experiences an outage while no BSTs are active, the outage is benign. The virtual object does not change modes as a result of the controller failure or outage. Upon restart of the storage controller, access to the virtual object can begin immediately.

Critical controller failures and outages can occur in the midst of a BST's execution. The controller can crash after some devices have accepted its *prewrite* request. These devices will move to the **Blocking** state and wait for a second-phase message. This second phase message will never come because the controller has crashed. A storage device associates a time-out with each accepted *prewrite* request. If the corresponding second phase message (*cancel* or *write*) is not received within the timeout period, the storage manager is notified.

The storage manager must restore the consistency of the array because it may have been corrupted as a result of the controller updating some but not all of the devices. The storage manager restores the consistency by recomputing parity. The storage manager does not guarantee that the data on the devices reflect the data the controller intended to write. To complete the write, the application must re-submit the write. This is correct since the semantics of a hostwrite, like a write to a disk drive today, is not guaranteed to have completed and reached stable storage until the controller responds with a positive reply. In this case, the controller did not survive until the end of the BST and could not have responded to the application with a positive completion reply.

Device outage. A device outage when the device is in the **Inactive** state does not require much

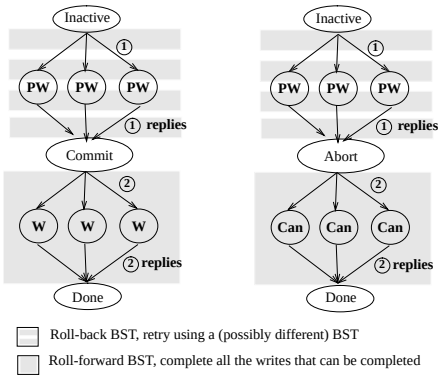


Figure 4.37: The algorithm followed by the controller when executing a BST. Ovals correspond to the state of the BST at the controller. Circles correspond to the processing of a message at the device. An arrow from an oval to a circle is a message from a controller to the device. An arrow from a circle to an oval is a reply from the device to the controller. The controller starts in the inactive state, issues a first round of prewrite messages, and once the replies are collected decides to commit or abort the BST. Once a decision is reached, second phase messages are sent out to the devices. Once the replies are received, the BST completes. The point right before sending out second phase *write* messages is called the commit point. The controller experiencing a device failure or outage before the commit point decides to Abort the BST. If all devices accept the prewrite, the BST is committed and writes are sent out to the devices that remain available. All failures in the second phase are simply reported to the storage manager.

recovery work besides careful initialization procedures upon restart from the outage. Under timestamp ordering, the device must wait period of T seconds upon restart before starting to service new requests. The device can establish that no BSTs were in progress during the outage by inspecting its queue of accepted *prewrite*s stored in NVRAM. If the queue is empty, the device waits for T seconds and starts accepting requests. If the queue contains some entries, the device enters the Recovering mode and notifies the storage manager.

Notice that all BSTs used in a shared storage array are represented as directed acyclic graphs (Figure 4.37), in which it is possible to ensure that no device write begins until after all device reads are complete [Courtright, 1997]. This point, right before the storage controller sends out second phase *write* requests is called the commit point. A storage controller may encounter a device outage before the commit point, that is before any second phase messages are sent out. In this case, the storage controller notifies the storage manager that it suspects a device failure, after canceling its first phase requests at the other devices that have responded to it during the first phase. In any case, recovery is simply enacted by retrying the BST later when the device is back on-line. No special recovery work is required at the device or storage manager.

If the storage controller has crossed the commit point but did not complete the writes everywhere, then the surviving devices will update storage (move to the **Updated** and **Done** states) while the device experiencing the outage will not. This device has accepted the *prewrite* but is not available to process the corresponding write. In this case, recovery proceeds as follows. Upon restart, the device inspects its queue of accepted prewrites stored in NVRAM and discovers that it has failed after accepting a prewrite and before processing the corresponding *write*. Such a device notifies the storage manager to perform recovery.

Device failure. A permanent device failure can occur (or be discovered) when the system is idle or when a BST is active. If the failure occurs during an idle period, the storage manager is notified. The storage manager revokes all leases to the virtual objects that have been allocated storage on that device. The virtual objects are moved to the degraded or reconstructing mode, marking the device as failed. The object is moved to a reconstructing mode if the storage manager can reallocate storage space on a functional device to replace the space on the failed device. Given that the device is no longer available, data stored on the device must be reconstructed from redundant copies. The storage manager changes the layout map to reflect the new allocation and moves the object to a reconstructing mode. A task to reconstruct the contents of the failed device and write it to the replacement space is started. If there is no space available to allocate as a replacement, the object is moved to a degraded mode. In degraded mode, writes addressed to the failed device are reflected in the parity. In any case, storage controllers that fetch the new map will be required to use the degraded mode or the reconstructing mode BSTs as specified in the map.

A permanent device failure can also be discovered after a BST has started. In this case, the storage controller must be careful in how to complete the BST. Before reaching the commit point, any BST encountering a permanent device failure (during a read) simply terminates, and its parent task reissues a new BST in degraded mode. This occurs as follows. The storage controller discovering the permanent failure aborts the current BST, discards the current layout map and notifies the storage manager of the failure. The storage manager verifies that the device has failed and then moves the virtual objects to degraded or reconstructing mode. The storage controller fetches the new map and restarts the access using a degraded mode BST.

After the commit point, a BST can encounter a permanent device failure. A storage controller encountering a single device failure after one or more writes are sent to the devices simply completes. This is correct because an observer cannot distinguish between a single failure occurring after the commit point and the failure of that device immediately after the BST completes. The

recovery protocol proceeds as follows. After the write completes, the storage manager is notified of the failed device. The storage manager moves the object to a degraded or reconstructing mode before re-enabling access to the object.

Lease expiration. A storage controller can experience a special kind of outage due to the expiration of a lease. Note that a lease can expire when the client is in the middle of executing a BST. If the expiration occurs before the commit point, then the second phase is not started. Instead, requests are sent to the devices to cancel the first phase and release the locks. The map is then refreshed by contacting the appropriate storage manager effectively extending the lease. The access protocol at the storage controller checks the lease expiration time before starting the second phase to establish that the lease will remain valid for the duration of the second phase. Because all lock requests have been acquired in the first phase, the second phase will not last for a long time. Once all replies have been received from the devices, the BST is considered to have completed successfully.

A storage manager may enter an object into a recovery mode during this second phase. This occurs if the second phase lasts for a long enough time that the lease expires. Since no client accesses are accepted in recovery mode and all locks are reset, the storage device will respond to the controller's second phase message with a error code. The BST is considered failed by the storage controller and the write must be retried to make sure that the contents of storage reflect the values in the write buffer.

4.7.4 Recovery in degraded and reconstructing modes

Under this mode, a device has failed in the array and therefore the array is not fault-tolerant. A second device failure or an untimely outage can result in data loss.

Controller outage/failure. As in the Fault-Free and Migrating modes, a controller outage amounts to losing all the layout maps cached at the client. After restart, the storage controller must re-fetch new layout maps from the storage manager to access virtual objects. Benign failures that occur when no BST is active or in the issuing state of a BST are straightforward to handle. Upon restart of the crashed controller, access can begin immediately also.

Controller failures and outages that occur in the midst of a BST can often lead to data loss because they corrupt the parity code making the data on the failed device irreconstructible. If the outage occurs before the commit point but after some devices have reached the **Blocking** state, then no device has been updated. The devices will eventually time-out and notify the storage manager. The storage manager, however, may not be able to ascertain whether the controller did or did not

cross the commit point and must in this case assume the controller may have partially updated the stripe and consequently declare data loss. Under a few fortunate cases, the storage manager will be able to establish that no device has been updated and cancel (roll-back) the BST without moving the object to the Data-loss state.

If the storage manager finds that all the devices participating in the BST started by the failed controller are in the **Blocking** state, then it can conclude that no device has updated storage. In this case, the storage manager can cancel the *prewrites* and re-enable access to the object. If on the other hand, the storage manager finds that at least one device has accepted a prewrite or write with a higher timestamp than that of the incomplete BST, it can not establish whether that device rejected the *prewrite* of the incomplete BST or whether it accepted it. It is possible that this device has accepted the *prewrite*. The storage controller could have sent a *write* message to the device and then crashed before updating the remaining devices. In this case, the device would have updated storage and completed the BST locally. It could have later accepted another prewrite or write with a higher timestamp. In this case, the storage manager must assume the conservative option and declare that the BST updated some devices but not all of them declaring data loss.

In order for the storage manager to accurately determine the fate of the incomplete BST at all the devices, it must have access to the history of writes serviced in the past. This can be achieved if the device maintains in NVRAM not only the accepted *prewrites* but also the recently serviced *write*s to the block.

Device outage. A critical device outage occurs when a device participates in the first phase of a BST and then experiences an outage before receiving the second phase message. In this case, recovery depends on when the outage is discovered. If the outage occurs during the Issuing or Blocked states, the storage controller will fail to contact the device and will therefore cancel the requests accepted at the other devices and inform the storage manager of the device's inaccessibility. In this case, the access is simply retried later when the device is back on-line. No special recovery work is required at the device or storage manager.

If the storage controller has written to the storage devices it intended to update except for a device that experienced the outage after responding to the first phase message, then the storage controller completes the write to the surviving devices and writes the data intended for the crashed device to a designated scratch space. Then, it notifies the storage manager. The storage manager revokes all leases and moves the object to the recovery mode. When the device restarts, the data is copied from the scratch space to the device and access is re-enabled. If the outage was experienced

by the replacement device, then the incomplete BST is compensated for by copying the data which was successfully written to the degraded array to the replacement device. In this case, the controller need not write to a scratch space since it is already stored in the degraded array.

Device failure. A permanent device failure in degraded mode amounts to data loss. The object is moved to the Unavailable mode and its data is no longer accessible.

4.7.5 Storage manager actions in the recovering mode

The storage manager receives notifications of device outages and failures from controllers. The storage manager takes a simple sequence of actions upon the receipt of such notifications. It revokes all leases for the virtual object being updated, then executes the proper compensating BST to restore the array's consistency then moves the object to a different mode if necessary and finally re-enables access to the virtual object.

Upon a general power failure, all the devices and clients experience a transient failure and must restart fresh, losing all the state in their volatile memory. In this case, there is no surviving entity to notify the storage manager of the outage. All the devices and controllers experience a simultaneous outage. Controllers have no state and upon restart can not assist the storage manager in restoring the array's consistency. They can not tell the storage manager what BSTs were in progress during the outage.

The storage manager relies on the devices to efficiently perform recovery after a general outage. Upon a restart, a device notifies the storage manager(s) that allocated space on it so that storage managers can carry out any recovery work before the device is allowed to start servicing client requests. The storage manager must determine whether and which nodes experienced a critical outage; that is, were actively processing a BST during the outage. This is achieved as follows. Upon restart from an outage, a storage device inspects its queue of accepted prewrites. This information is stored in NVRAM on the device and therefore survives transient outages. If the queues are found empty, the device is made accessible to serve requests to the storage controllers. If the queue contains outstanding prewrites, then the storage manager knows that a BST was in progress during the outage. It can then execute the proper compensating action.

Table 4.5 summarizes the recovery actions taken by the storage manager upon a critical outage or failure. The table shows the compensating transaction executed by the storage manager to complete the BST to achieve consistency. It also shows the new mode that the object will transition to (from the Unavailable mode) after recovery is completed.

Active BST	Object mode	Type of failure	Compensating	New object mode
Write	Fault-Free	Critical outage	Rebuild-Range	Fault-Free
Write	Fault-Free	Device failure	None	Degraded
Multi-Write	Migrating	Critical outage	Rebuild-Range	Migrating
Multi-Write	Migrating	Device failure (degraded array)	Rebuild-Range	Migrating
Multi-Write	Migrating	Replacement device failure	Rebuild-Range	Migrating
Copy-Range	Migrating	Critical outage	Copy-Range	Migrating
Copy-Range	Migrating	Device failure (degraded array)	Copy-Range	Degraded
Copy-Range	Migrating	Replacement device failure	Restart copy task on new device	Migrating
Write	Degraded	Critical outage	None	Data Loss
Write	Degraded	Device failure	None	Data Loss
Write	Reconstructing	Critical outage	None	Data Loss
Write	Reconstructing	Device failure (degraded array)	None	Data Loss
Write	Reconstructing	Replacement device failure	Restart recon task on new device	Reconstructing
Rebuild-Range	Reconstructing	Critical outage	Rebuild-Range	Reconstructing
Rebuild-Range	Reconstructing	Device failure (degraded array)	None	Data Loss
Rebuild-Range	Reconstructing	Replacement device failure	Restart recon task on new device	Reconstructing

Table 4.5: The recovery actions taken upon a failure or outage. The table shows for each critical outage or failure the compensating BST executed by the storage manager to restore the BST's consistency as well as the new mode that the object is moved to after recovery is completed by the storage manager. "Write" is used to refer to all the write BSTs available in the specified mode.

Compensating storage transactions in degraded and reconstructing modes

In degraded mode, one of the data disks has already failed. A second disk or client failure in the middle of an access task before the data on the first disk is reconstructed is considered fatal. Similarly, in the reconstructing mode, one of the data disks has already failed. A second disk or client failure in the middle of an access task before the data on the first disk is reconstructed is considered fatal. However, if the failure occurs in the middle of a reconstruct task (the Rebuild-Range BST), then the BST can be simply restarted. The degraded array is not corrupted by the failure, although the replacement block being written to may have been only partially updated. The BST can be restarted and the value of the failed data block recomputed and written to the replacement device.

Compensating transactions in Fault-Free and Migrating modes

Once the failed BST is detected and reported to the storage manager, the storage manager waits until all leases to the virtual object expire and then executes a compensating transaction to restore the array's consistency. Once consistency is restored, the mode of the virtual object is updated if necessary and access re-enabled by granting layout maps to the requesting controllers. Each BST has a compensating BST which is executed exclusively by the storage manager in recovery mode whenever a BST fails.

Under fault-free mode, once the suspect ranges are identified, consistency can be effectively re-established by recomputing the parity block from the stripe of data blocks. Under this mode, all devices are operational, so the Rebuild-Range BST can be used to recompute the parity block.

In migrating mode, a BST is invoked either by an access task or by a migrate task. For BSTs that are invoked by hostwrite tasks, the compensating transactions are similar to the ones in Fault-Free mode. In the case of a copy BST that is invoked by a migrate task, the compensating transaction is the transaction itself. That is, the storage manager simply reissues the copy transaction.

4.7.6 The recovery protocols

This section presents the pseudo-code for the actions taken by the devices and storage controllers. Storage controllers are relatively simpler. They are stateless, do not have to execute any special algorithms on restart or recovery from an outage. The important task implemented by the storage controller is the execution of a BST. The pseudo-code on the following page shows the algorithm abstracting the details of the concurrency control algorithm and focusing on the recovery actions

taken by the controller while in the midst of executing a BST. The storage controller can experience a device outage or failure of simply a device that is not responding. Upon encountering such an event, the storage controller cancels the BST if it did not cross the commit point. Otherwise, it completes the BST. As soon as the BST is completed or canceled, the storage controller reports the problem to the storage manager which in turns takes the recovery actions described above.

Storage devices are more complicated, because they maintain state across outages and restarts of controllers and of their own. Upon recovery, storage devices execute the function InitUponRestart(). During normal operation, requests are handled by invoking the HandleRequest function. If a time-out is reached while a second phase message is not received, the HandleTimeout function is invoked. The pseudo-code reflects the steps already discussed for device recovery and time-out handling.

```
 //Device Actions
01 /* Device-side pseudo-code: Handle request req from controller contr */
02 HandleRequest (req, contr)
03     if ( req.type == read)
04          resp = checktimestamp(req.opts, req.blockid);
05          if (resp==OK)
06               send (data[req.blockid], OK) to contr;
07          else
08               send (REJECT) to contr;
09     endif
10      if ( req.type == prewrite or read-and-prewrite)
11          resp = checktimestamp(req.opts, req.blockid);
12          if (resp == OK)
13               enqueue (req, NOW + TIMEOUT) to timeoutq;
14               send (OK, data) to contr;
15          else
16               send (REJECT) to contr;
17          endif
18     endif
19     if ( req.type == write)
20          /* discard request with timestamp opts from queue */
21          write req.buf to req.blockid on stable storage;
22          timeoutq.discard(req.opts);
23          send (OK) to contr;
24     endif
```

```
01 /* Device-side pseudo-code to handle a timeout */
02 HandleTimeout (req)
03     send (req, BST_NOT_COMPLETED) to storage manager;
04 end
```

```
01  /* Device-side pseudo-code to execute after restart from an outage */
02 InitUponRestart()
03       if (prewriteq is empty)
04          wait T seconds;
05          return (OK); /* upon return, requests can be accepted */
06     else
07          send(NULL, BST_NOT_COMPLETED) to storage manager
08          return (RECOVERING);
09     endif
10 end
```

```
// Controller actions

01 /* Execute a two phase write bst to object with map objmap */
02 ExecuteTwoPhaseBST (bst, objmap)
03     for dev = 1 to bst.numdevices
04          send (bst.device[i]);
05     endfor
06     deadline = NOW + TIMEOUT;
07      while ( replies < numdevices and time < deadline)
08          receive(resp);
09          replies ++;
10     endwhile /* continued on next page ...*/
```

```
11    if (replies < numdevices)
12         /* some devices did not respond */
13         for dev = 1 to bst.numdevices
14              send (CANCEL) to bst.device[i];
15         endfor;
16         /* notify manager of the not-responding devices */
17         for each dev not in replies
18              send (dev, NOT_RESPONDING) to storage manager;
19         endfor;
20         Discard(objmap); /* discard layout map, must be re-fetched later */
21         return (DEVICE_OUTAGE_OR_FAILURE);
22    else if (numoks in replies < numdevices)
23         /* rejection at one or more devices, send CANCELs */
24         for dev = 1 to bst.numdevices
26              send (CANCEL) to bst.device[i];
26         endfor;
27         return (RETRY);
28    else if (numoks in replies == numdevices)
29         for dev = 1 to bst.numdevices
30              send (OK, bst.data[i] to bst.device[i])
31         endfor
32          replies = 0; deadline = NOW + TIMEOUT;
33          while (replies < numdevices and time < deadline)
34              receive (resp);
35         endwhile
36         if (replies < numdevices) /* some devices did not respond */
38              slist = null; /* list of devices suspected of failure */
37              for dev = 1 to bst.numdevices
38                   if (dev did not respond)
39                        add dev to slist;
40              /* notify manager of the not-responding devices */
42              send (bst, BST_NOT_COMPLETED, slist) to storage manager;
44              Discard(objmap);
45              return (DEVICE_OUTAGE_OR_FAILURE);
46         endif
47    endif
```

4.8 Discussion

The discussion so far focussed on protocols that ensure serializability for all executing BSTs. Serializability is a "sufficient" guarantee since it makes a shared array behave like a centralized one, in which a single controller receives BSTs from all clients and executes them one at a time. This guarantee, however, can be too strong if certain assumptions hold regarding the semantics and structure of high-level software. Furthermore, the discussion has focussed on RAID level 5 layouts. However, a large number of disk array architectures have been proposed in the literature.

This section highlights the generality of the presented protocols by showing how they can readily generalized to adapt to and exploit different application semantics and underlying data layouts. It also includes a discussion of the recovery protocols.

4.8.1 Relaxing read semantics

Many applications do not send a concurrent *hostread* and *hostwrite* to the same block. For example, many filesystems such as the Unix Fast File System [McKusick et al., 1996], do not send a write to the storage system unless an exclusive lock is acquired to that block, which prohibits any other client or thread from initiating a read or write to that block until the write completes. Consequently, it never occurs that a read is initiated while a write is in progress to that block.

This property of many applications precludes the need to ensure the serializability of reads because they never occur concurrently with writes. In fault-free operation, where reads do not need to access parity blocks, a *hostread* accesses only data blocks the higher level filesystem has already acquired (filesystem-level) locks for. It follows that the only concurrent writes to the same stripe must be updating other data blocks besides the ones being accessed by the *hostread*. It is therefore unnecessary to acquire a lock to the data block before a read.

Recall that in fault-free mode, only *hostread* and *hostwrite* tasks are allowed. Thus, if the higher-level filesystem ensures no read/write conflicts, *hostread*s can simply be mapped onto direct *devreads* with no timestamp checks or lock acquire/release work. This can speed up the processing at the devices and reduce the amount of messaging on the network. Note that this read optimization can not be applied in degraded mode. Serializability checking is required in degraded mode and reconstructing modes because contention can occur over the same data block even if the hosts do not issue concurrent *hostread* and *hostwrites* to the same block. The performance evaluation results and conclusions do not change much even if concurrency control is not performed on fault-free mode reads. In this case, all the protocols will not perform any control messaging on behalf of reads

and therefore have the minimal latency possible for *hostreads*. Concurrency control is still required for *hostwrites* which can conflict on parity as well as data blocks.

Note that regardless of what the higher-level software is doing, concurrency control must be ensured for two concurrent writes to the same stripe. This is because two writes to different data blocks can contend over the same data or parity blocks, something that is totally invisible to higher-level software (consider, for example, a Read-Modify-Write BST and a Reconstruct-Write BST, both shown in Figure 4.5, occurring concurrently in Fault-free mode). Serializability of such write BSTs is essential so that parity is not corrupted. Similarly, the serializability of copy BSTs with ongoing writes is also required for correctness regardless of what the synchronization protocol used by higher-level software.

Finally, the deadlock problem associated with device-served locking variants can be eliminated by requiring clients to acquire locks on entire stripes. This breaks the "hold and wait" condition because clients do not have to acquire multiple locks per stripe. Only a single lock is acquired. Stripe locking substantially reduces concurrency, however, especially with large stripe sizes. This in turn degrades I/O throughput and increases access latencies for applications that perform a lot of small writes to shared large stripes.

4.8.2 Extension to other RAID levels

The approach discussed in this chapter can be extended in a straightforward manner to other RAID levels, including double-fault tolerating architectures. The reason is that all the read and write operations in all of the RAID architectures known to the author at the time of the writing of this dissertation [Blaum et al., 1994, Holland et al., 1994] consist either of a single (read or write) phase or of a read phase followed by a write phase. Thus, the piggy-backing and timestamp validation approach described in the previous sections apply directly to these architectures as well.

4.8.3 Recovery

Multiple component failures can easily lead to an object ending in the Unavailable state. In practice, a poorly designed system can be vulnerable to the correlated failures and outages of several components. For example, if disks share the same power source or cooling support, then multiple disks can experience faults at the same time. The likelihood of more than a single failure can be substantially reduced by designing support equipment so that it is not shared by the same disks in the same stripe group [Schulze et al., 1989]. Another scheme is to use uninterruptible power supplies (UPS). Mul-

tiple or successive failures can cause several transitions between modes. The following discussion will focus on a single failure or transition at a time.

While the concurrency control protocol work is largely distributed, the recovery work heavily relies on the storage manager. This is not a serious problem because recovery is not supposed to be common. Furthermore, there need not be a single storage manager in the system. There can be many storage managers as long as, at any given time, there is a unique manager serving the layout map for a given virtual object. Thus, the storage manager's load can be easily distributed and parallelized across many nodes.

When a manager fails, however, another storage manager must take over the virtual objects that it was responsible for. Other work has investigated how storage managers can be decided dynamically by the storage devices upon a failure so that no static assignment of managers to devices is necessary [Golding and Borowsky, 1999]. This work is complementary to the solutions discussed in this dissertations and solves a complementary problem of ensuring fast parallel recovery when the system restarts after a general failure. The protocols described in [Golding and Borowsky, 1999] handle network partitions as well as device and manager outages.

TickerTAIP [Cao et al., 1994] is a parallel disk array architecture that distributed the function of the disk array controller to the storage nodes in the array. One of the design goals of TickerTAIP was to tolerate node faults. Hosts in TickerTAIP did not directly carry out RAID update protocols. RAID update protocols were executed by one storage node on behalf of the host. Host failure, therefore, was not a concern. The protocols discussed in this dissertation generalize the recovery protocols of TickerTAIP to the case where the RAID update algorithms involve both the clients and the devices.

4.9 Summary

Shared storage arrays enable thousands of storage devices to be shared and directly accessed by hosts over switched storage-area networks. In such systems, storage access and management function are often distributed to enable concurrent accesses from clients and servers to the base storage. Concurrent tasks can lead to inconsistencies for redundancy codes and for data read by hosts. This chapter proposed a novel approach to constructing a scalable distributed storage management system that enables high concurrency between access and management tasks while ensuring correctness. The proposed approach breaks down the storage access and management tasks performed by storage controllers into two-phased operations (BSTs) such that correctness requires ensuring only the

serializability of the component BSTs and not of the parent tasks.

This chapter presented distributed protocols that exploit technology trends and BST properties to provide serializability for BSTs with high scalability, coming within a few percent of the performance of the ideal zero-overhead protocol. These protocols use message batching and piggybacking to reduce BST latencies relative to centralized lock server protocols. In particular, both device-served locking and timestamp ordering achieve up to 40% higher throughput than server and callback locking for a small (30 device) system. Both distributed protocols exhibit superior scaling, falling short of the ideal protocol's throughput by only 5-10%.

The base protocols assume that within the shared storage array, data blocks are cached at the storage devices and not at the controllers. When controllers are allowed to cache data and parity blocks, the distributed protocols can be extended to guarantee serializability for reads and writes. This chapter demonstrates that timestamp ordering with validation, a timestamp based protocol, performs better than device-served leasing especially in the presence of contention and random access workloads. In summary, the chapter concludes that timestamp ordering based on loosely synchronized clocks has robust performance across low and high contention and in the presence of device-side or host-side caching. At the same time, timestamp ordering requires limited state at the devices and does not suffer from deadlocks.

Chapter 5

Adaptive and automatic function placement

The previous chapter presented an approach based on light-weight transactions which allows storage controllers to be active concurrently. Specifically, multiple controllers can be accessing shared devices while management tasks are ongoing at other controllers. Performance results show that the protocols used to ensure correctness do scale well with system size. The approach described in the previous chapter enables controllers to be actively migrating blocks across devices and reconstructing data on failed devices while access tasks are ongoing. This enables balancing load across disk by migrating storage without disabling access. Balancing load across disks improves the performance of data access for applications using the storage system.

Another issue that affects the performance of storage-intensive applications has to do with properly partitioning their functions between the different nodes in the storage system. Judicious partitioning can reduce the amount of data communicated over bottlenecked links and avoid executing function on overloaded nodes. Rapidly changing technologies cause a single storage system to be composed of multiple storage devices and clients with disparate levels of CPU and memory resources. Moreover, the interconnection network is rarely a simple crossbar and is usually quite heterogeneous. The bandwidth available between pairs of nodes depends on the physical link topology between the two nodes. This chapter demonstrates that performance can be improved significantly for storage management and data-intensive applications by adaptively partitioning function between storage servers and clients. Function can be judiciously partitioned based on the availability and distribution of resources across nodes and based on a few key workload characteristics such as the bytes moved between functional components. This chapter demonstrates that the information necessary to decide on placement can be collected at run-time via continuous monitoring.

This chapter is organized as follows. Section 5.1 highlights the different aspects of heterogeneity in emerging storage systems, and states the assumptions made by the discussions that follow. Sec-

tion 5.2 reviews the function placement decisions of traditional filesystems and how they evolved to adapt to the underlying hardware and to the driving applications. Section 5.3 describes a programming system, called ABACUS, which allows applications to be composed of components that can move between client and server dynamically at run-time. Section 5.4 describes the performance model used by ABACUS to decide on the best placement of function and explains how the information needed by the performance model is transparently collected at run-time. Section 5.5 describes a file system which was designed and implemented on ABACUS. Section 5.6 reports on the performance of this filesystem on ABACUS under different synthetic workloads. Section 5.7 describes how ABACUS can be generalized to enable user-level applications to benefit from adaptive client-server function placement. Section 5.8 evaluates the algorithms used by ABACUS to decide on the optimal function placement under variations in node and network load and in workload characteristics. Section 5.9 discusses the advantages and limitations of the proposed approach. Section 5.10 discusses previous related work. Section 5.11 summarizes the chapter.

5.1 Emerging storage systems: Active and Heterogeneous

Two of the key characteristics of emerging and future storage systems are the general purpose *programmability* of their nodes and the *heterogeneity* of resource richness across them. Heterogeneity mandates intelligent and dynamic partitioning of function to adapt to resource availability, while the programmability of nodes enables it. The increasing availability of excess cycles at on-device controllers is creating an opportunity for devices and low-level storage servers to subsume more host system functions. One question that arises from the increased flexibility enabled by storage device programmability is how filesystem function should be partitioned between storage devices and their clients. Improper function partitioning between active devices and clients can put pressure on overloaded nodes and result in excessive data transfers over bottlenecked or slow network links. The heterogeneity in resource availability among servers, clients and network links, and the variability in workload mixes causes optimal partitioning to change across sites and with time.

5.1.1 Programmability: Active clients and devices

Storage systems consist of storage devices, client machines and the network links that connect them. Traditionally, storage devices have provided a basic block-level storage service while the host executed all of the filesystem code. Moore's law is making devices increasingly intelligent and

trends suggest that soon some of the filesystem or even application function can be subsumed by the device [Cobalt Networks, 1999, Seagate, 1999].

Disk drives have heavily exploited the increasing transistor density in inexpensive ASIC technology to both lower cost and increase performance by developing sophisticated special purpose functional units and integrating them onto a small number of chips. For instance, Siemen's TriCore integrated micro-controller and ASIC chip contained a 100 MHz 3-way issue super-scalar 32-bit data-path with up to 2 megabytes of on-chip dynamic RAM and customer defined logic in 1998 [TriCore News Release, 1997].

Regardless of what technology will prove most cost-effective to bring additional computational power to storage devices (e.g. an embedded PC with multiple attached disks or a programmable on-disk controller), the ability to execute code on relatively resource-poor storage servers creates an opportunity which must be carefully managed. Although some function can benefit from executing closer to storage, storage devices can be easily overwhelmed. Active storage devices present the storage-area-network filesystem designer with the added flexibility of executing function on the client side or the device side. They also present a risk of degraded performance if function is partitioned badly.

5.1.2 Heterogeneity

Storage systems consist of highly heterogeneous components. In particular, there are two important aspects of this heterogeneity, the first is heterogeneity in resource levels across nodes and links and the second is the heterogeneity in node trust levels.

Heterogeneity in node resources

Storage systems are characterized by a wide range in the resources available at the different system components. Storage servers—single disks, storage appliances and servers—have varied processor speeds, memory capacities, and I/O bandwidths. Client systems—SMP servers, desktops, and laptops—also have varied processor speeds, memory capacities, network link speeds and levels of trustworthiness.

Some nodes may have "special" resources or capabilities which can be used to accelerate certain kinds of computations. Data-intensive applications perform different kinds of operations such as XOR, encoding, decoding and compression. These functions can benefit from executing on nodes that have special capabilities to accelerate these operations. Such capabilities can be hardware

accelerators, configurable logic, or specialized processors.

Heterogeneity in node trust levels

In a distributed system, not all nodes are equally trusted to perform a given function: a client may not be trusted to enforce the access control policy that the server dictates, or it may not be trusted to maintain the integrity of certain data structures. For instance, certain filesystem operations, such as directory updates, can only be safely completed by trusted entities, since they can potentially compromise the integrity of the filesystem itself. Where a function executes, therefore, has crucial implications on whether the privacy and integrity of data can be maintained.

Traditionally, many client-server systems [Howard et al., 1988] assume non-trustworthy clients and partition function between client and server such as that all potentially sensitive function is maintained at the server. This assumption does not hold in the case of many emerging clusters, where clients and servers are equally trusted. Such conservative designs under-perform when the resources of trusted clients go underutilized. Other systems, like NFS [Sandberg et al., 1985] and Vesta [Corbett and Feitelson, 1994], have assumed trusted clients. Such systems can suffer serious security breaches when deployed in hostile or compromised environments.

Filesystem designers do not know the trustworthiness of the clients at design time and hence are forced to make either a conservative assumption, presupposing all clients to be untrustworthy, or a liberal one, assuming clients will behave according to policies. In general, it is beneficial to allow trust-sensitive functions to be bound to cluster nodes at run-time according to site-specific security policies defined by system administrators. This way, the filesystem/application designer can avoid hard-coding assumptions about client trustworthiness. Such flexibility enhances filesystem configurability and allows a single filesystem implementation to serve in both paranoid and oblivious environments.

5.1.3 Assumptions and system description

This chapter does not assume a very specific storage system architecture. In fact, its goal is to arrive at a framework and a set of algorithms which enable a filesystem to be automatically optimized at installation-time and at run-time to the particular hardware available in the environment. It follows, therefore, that little should be assumed about the resource distribution or about the workload. Of course, some basic assumptions about the storage model and the kinds of entities in the system are required to enable progress.

The discussion in this chapter is concerned with the partitioning of filesystem functions in *active storage systems*. Active storage systems consist of programmable storage servers, programmable storage clients and a network connecting them. This subsection describes these components in more detail.

Programmable storage servers

A storage server in an active storage system can be resource-poor or resource-rich. It can be a NASD device with a general purpose processor, a disk array with similar capabilities, or a programmable file server machine. A "storage server" refers to a node with a general purpose execution environment that is directly attached to storage. A storage server executes an integrated storage service that allows remote clients and local applications to access the storage devices directly attached to it. The interface to this base storage service can be NFS, HTTP, NASD or any interface allowing logical objects with many blocks to be efficiently named. The implementation used in the experiments reported in this chapter is built on a base storage service which implements the NASD object interface, but it will be clear from the discussion that the approach described in this chapter applies equally well to any other object-like or file-like base storage service.

Because all storage devices and file servers already contain a general purpose processor capable of executing general purpose programs, the specific meaning of "storage server programmability" in this particular context may not be clear. While storage devices are endowed today with general purpose processors, the software executed on these processors is totally written by the device manufacturers. Similarly, while NFS file servers are often general purpose workstations, the function that administrators allow to execute on the server is limited to little beyond file service and the supporting services it requires (e.g. networking, monitoring and administration services).

This chapter assumes that programmable storage servers allow general purpose client extensions to execute on their local processors, possibly subject to certain security and resource allocation policies. These client-provided functions can be downloaded at application run-time and are not known to the storage server or device manufacturer ahead of time. These functions can be part of the filesystem traditionally executed on the client's host system, or alternatively they can be part of user-level applications. All such functions, however, may have to obey certain restrictions to be able to execute on the programmable server. For example, they may be constrained to accessing persistent storage and other resources on the storage server through a specified set of interfaces. The interfaces exported by a programmable storage server to client functions can range from an entire

POSIX-compliant UNIX environment to a limited set of simple interfaces.

This chapter assumes that programmable storage servers export a NASD interface. Client downloaded extensions can be executed on the server and can access local storage through a NASD interface. Client functions can use the server's processor to perform computations and can allocate a bounded amount of memory. They can access memory in their own address space or make invocations to other (local or remote) services through remote procedure calls.

Storage clients

Storage clients represent all nodes in the system that are not storage servers. Alternatively, storage clients are network nodes that have no storage directly attached to them. They access storage by making requests on storage servers. They also have a general purpose execution environment. The clients may or may not be trusted by the storage servers. Filesystem function that does not execute on the storage server must execute on the client. Storage clients include desktops, application servers, NASD file managers, or web servers connected to storage servers through a storage-area network.

5.2 Function placement in traditional filesystems

Currently, distributed filesystems, like most client-server applications, are constructed via remote procedure calls (RPC) [Birrell and Nelson, 1984]. A server exports a set of services defined as remote procedure calls that clients can invoke to build applications. Distributed filesystems therefore have traditionally divided their device-independent functions statically between client and server.

Changes in the relative performance of processors and network links and in the trust levels of nodes across successive hardware generations make it hard for "one-design-fits-all" function partitioning decision to provide robust performance in all customer configurations.

Consequently, distributed filesystems have historically evolved to adapt to changes in the underlying technologies and target workloads. Examples include parallel filesystems, local-area network filesystem, wide-area filesystems, and active disk systems. The following section describes these different systems and how they were specialized to their particular target environment.

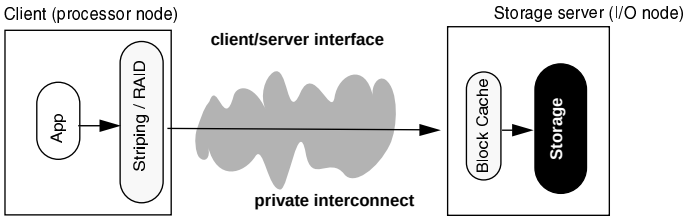


Figure 5.1: Function placement in the Intel concurrent filesystem for the iPSC/2 hypercube multicomputer. Both processor nodes and I/O nodes use identical commodity x86 Intel processors with equal amount of local memory. Up to four disks are connected to each I/O node. The processor and I/O nodes are connected in a hypercube private interconnection network. Given the bandwidth of the private interconnect, CFS caches disk blocks at the I/O nodes. To deliver high-bandwidth to applications running on clients, CFS delegates to the client processor nodes the responsibility of striping and storage mapping.

5.2.1 Parallel filesystems

In order to provide the necessary processing horsepower for scientific applications and on-line transaction processing systems, parallel multicomputers and massively parallel processors were introduced. These systems comprised a large number of processors interconnected via a highly reliable high-bandwidth bus or interconnection network. To provide the processors with scalable input and output to and from secondary storage devices, multicomputer designers developed their own proprietary filesystems, such as the Intel Concurrent Filesystem (CFS) [Pierce, 1989] and the IBM Vesta [Corbett and Feitelson, 1994].

Storage devices in multicomputers are usually attached to processors known as “I/O nodes” (storage servers), while “processor nodes” (clients) execute application code and access storage by making requests to the I/O nodes. Multicomputer file systems are not concerned with security given that all processors are trusted and the interconnect is private to the multicomputer. All processors execute the same operating system and often the same application, and, therefore, are assumed to mutually trust each other. Furthermore, the network is internal to the multicomputer and is safe from malicious attacks. In the Intel CFS filesystem for example, I/O nodes cache disk blocks while processor nodes do not perform any caching. Client processor nodes, on the other hand, perform storage mapping (striping/aggregation) so that they can issue multiple parallel requests to several I/O nodes. Because the latency of a local memory access is comparable to the latency of an access to the memory of an I/O node, server-side caching makes sense since it avoids the complexity and performance overhead of ensuring the consistency of distributed caches. At the same time, client-side storage mapping allows applications executing on the processor nodes to obtain high-bandwidth

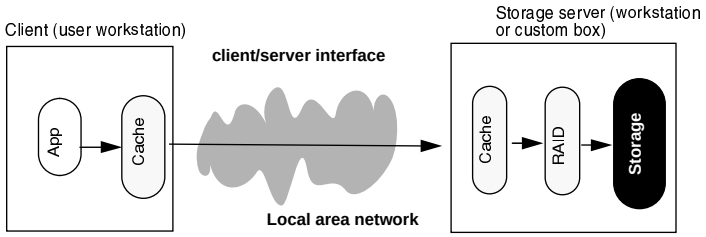


Figure 5.2: Function placement in most network filesystems. Because the network is traditionally a low bandwidth shared bus, striping across devices is only applied at the server. Data is cached as close to the application as possible (at the client) to avoid the slow network.

by striping across links and I/O server nodes. Figure 5.1 depicts a typical partitioning of function in parallel filesystems.

Parallel processor interconnection networks boast high reliability and bandwidth because they span short distances and are installed in controlled environments. Local area networks, such as Ethernet, and wide area internetworks have much lower bandwidth. Filesystems for these networks have therefore chosen a different function placement than parallel filesystems.

5.2.2 Local area network filesystems

The NFS filesystem was designed to allow sharing of files in a local area network. NFS divides machines into clients and servers. Storage devices are attached to the server. Applications execute entirely on clients and access the server to read and write data on the storage devices. Servers implement all the file and storage management functions. For example, file and storage management functions (directory management, reliability, and storage management) execute almost entirely on the server because the server has sufficient resources to manage the limited number of storage devices that are attached to it. Because the network has limited bandwidth, NFS supports the caching of file and attribute information at the client. This caching reduces the need for over-the-network transfers, and also reduces server load. Figure 5.2 depicts the partitioning of function in typical local-area and wide-area network filesystems.

To obtain cost-effective scalable bandwidth on a local area network, data must be striped across the network and across multiple servers. Swift [Cabrera and Long, 1991] is an early array storage system that striped data across multiple storage servers to provide high aggregate I/O rates to its clients. Swift defines a storage mediator machine which reserves resources from communication and storage servers and plans the aggregate transfer as an encrypted coordinated session. Swift

mediators also manage cache coherence using call-backs, and bandwidth reservation by selecting the appropriate stripe unit size. Swift delegates the storage management function to the servers (the storage mediators). Applications execute entirely on clients.

5.2.3 Wide area network filesystems

The Andrew filesystem (AFS) was designed to enable sharing of data among geographically distributed client systems [Howard et al., 1988]. AFS has traditionally served an amalgamation of widely distributed office and engineering environments, where sharing is rare but important and each client is an independent system. Accordingly, the entire application executes at a client and a larger fraction of the client's local disk is reserved for long-term caching of distributed filesystem data.

AFS is designed for wide areas where a local disk access is assumed much faster than an access to the server. Consequently, local disk caching under AFS can improve performance dramatically. Long-term local disk caching is very useful in environments with mostly read-only sharing such as infrequently updated binaries of shared executables. Such caching reduces server load enabling relatively resource-poor AFS servers to support a large number of distant clients.

5.2.4 Storage area network filesystems

Traditionally, the limited connectivity of peripheral storage networks (e.g. SCSI) constrained the number of devices that can be attached to the server. Emerging switched networks are expanding the connectivity and bandwidth of peripheral storage networks and enabling the attachment of thousands of storage devices to the server. As a result, a single file server machine – usually a commodity workstation or PC – cannot handle file and storage management for this large number of devices. Consequently, recent research on network-attached storage has proposed offloading this function to "clients", eliminating the legacy server, enabling striping across multiple servers, effectively replacing the server with a cluster of cooperating clients. Several researchers have proposed scalable storage systems that comprise clusters of commodity storage servers [Anderson et al., 1996, Thekkath et al., 1997, Gibson et al., 1998, Hartman et al., 1999] and clients, which largely offload filesystem functionality from servers to clients to enable the scalability of the distributed filesystem.

Frangipani [Thekkath et al., 1997] is a cluster filesystem which is built on top of closely cooperating network-attached storage servers. The storage servers export the abstraction of a set of virtual

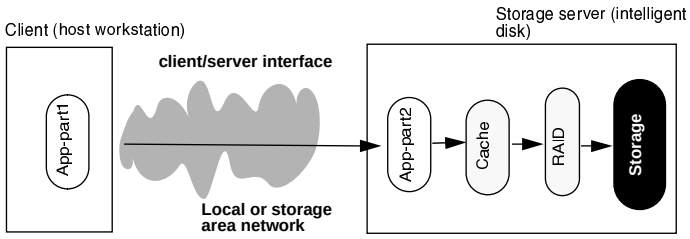


Figure 5.3: Function placement in active disk systems. The server in this case is a smart disk drive capable of executing general purpose application code. The client is a host machine. The data intensive function of the application (part2) is explicitly moved to the disk. A vestigial merge function (part1) is left at the host to coordinate multiple remotely executing functions which can be potentially executing on several active disks.

disks. Each virtual disk is block-addressable, and blocks are replicated across servers for reliability. Frangipani distributes its filesystem function (allocation, namespace management) to a set of cooperating clients, called "file managers." The file managers cooperatively implement a UFS-like filesystem and synchronize via a shared lock manager. Measurements of Frangipani report enhanced scalability from distributed execution.

5.2.5 Active disk systems

A growing number of important applications operate on large data sets, searching, computing summaries, or looking for specific patterns or rules, essentially "filtering" the data. Filter-like applications often make one or more sequential scans of the data [Riedel et al., 1998]. Applications execute on the host, with the storage devices serving as block servers. Proponents of active disk systems claim that the increasing levels of integration of on-disk controllers are creating "excess" computing cycles on the on-disk processor. These cycles can be harnessed by downloading "application-specific" data intensive filters. Currently, data-intensive applications execute entirely on the host, often bottlenecking on transferring data from the storage devices (servers) to the host (client in this case). Figure 5.3 depicts the partitioning of function in an active disk system.

Table 5.1 summarizes the function placement choices for a representative distributed storage system in each of the above categories. For each system, the table shows where caching as well as other filesystem functions such as aggregation/striping and namespace management are placed. The table shows also where application function is executed. The table summarizes this information highlighting the fact that for each function, there is at least one system which chooses to place it at the client and at least one system which chooses to place at the server.

For each filesystem, the partitioning of function was heavily dictated by the topology of the target environment and the characteristics of the important workloads. This makes each filesystem applicable only within the boundaries of the environments that fit its original design assumptions. Function partitioning in distributed filesystems has been specialized in these systems to the underlying technologies and target workloads. This specialization of course came at the expense of considerable development time. Rewriting filesystems to optimize for details in the underlying hardware is not cost-effective in terms of development time. Moreover, rewriting filesystems to optimize for more detailed characterizations of the underlying hardware still cannot adapt to changes in the lifetime of a single workload, or to inter-application competition over shared resources.

5.2.6 A motivating example

Consider the following example demonstrating how adapting function placement between client and server can improve filesystem performance. The previous chapter presented a shared storage array architecture composed of storage devices and storage controllers. In this chapter's terminology, the storage devices are the storage servers and the storage controllers are the clients. Let's assume that the storage device processors are limited compared to that of the storage controllers. Let's further assume that the storage network connecting the devices to the controllers is relatively fast such that the times taken by a local and a remote access between the nodes are relatively indistinguishable. In this case, when the array is in degraded mode, it is advantageous to execute the reconstruct-Read BSTs (XOR intensive operations needed to compute the contents on the failed device) on the storage controller. The storage controller has a fast CPU and transferring the data on the network does not add observable latency.

Now consider the case where the storage devices are upgraded such that executing XORs on the devices is 5X faster. Then, executing the reconstruct-Read BST on the device side will be 5 times faster. If we also assume that the network is of observable latency because of highly active storage controllers, the performance improvement of device-side execution can be even higher.

Traditionally, the server interface defines what function executes at the server, everything else executes at the client. This interface is decided at system design time by server implementors. Client programmers have to abide by this division of labor. Server interface designers factor in assumptions about the relative availability of resources at the client and the server, their relative levels of trust, the performance of the client-server network and the key characteristics of the expected workload. These assumptions do not match the realities of several systems and workloads. The result is subop-

	Intel CFS	**CMU AFS**	**Frangipani**	**Active disks**
Component	**Assumption**			
Client node	Trusted	Not trusted	Not trusted	Trusted
Network	Private interconnect	Wide area network	Scalable switched network	Storage network or LAN
Function	**Placement**			
Aggregation	at client	at server	at server	at server
Namespace	at server	at server	at client	at server
Application	at client	at client	at client	at server

Table 5.1: Function placement in distributed storage systems as a function of machine, workload and network characteristics. The table shows only where striping/aggregation, namespace management, and user applications are placed. CFS is Intel's concurrent file system, AFS is the Andrew filesystem developed at Carnegie Mellon, and Frangipani is the cluster filesystem developed at Compaq SRC. Active disks represent the research prototypes described in the literature [Riedel et al., 1998, Keeton et al., 1998, Acharya et al., 1998].

timal and sometimes disastrous performance. The heterogeneity in workload characteristics, node trust levels, and in node resource availability in actual systems make such assumptions "invalid" for a large number of cases.

Optimal partitioning of function depends on workload characteristics as well as system characteristics. Both must be known before the optimal partitioning is known. For instance, consider a streaming data-intensive application executing on a storage system where the server's CPU is much slower than that of the client and where the network between client and server has a relatively high-bandwidth. In this case, "data shipping" and not "function shipping" is the optimal solution. Transferring the data to the client is inexpensive, and client-side processing will be much faster! Even if the server is powerful, it can be easily overloaded with remotely executed functions causing slow-downs contrary to the desired goals[Spalink et al., 1998].

5.3 Overview of the ABACUS system

To demonstrate the benefits and the feasibility of adaptive function placement, this chapter reports on the design and implementation of a dynamic function placement system and on a distributed filesystem built on it. The prototype is called ABACUS because functions associated with a particular data stream (file) can move back and forth between the client and the server.

5.3.1 Prototype design goals

ABACUS is designed primarily to support filesystems and stream-processing applications. Filesystems and stream-processing applications move, cache and process large amounts of data. They perform several functions on the data-stream as it moves from the base storage server to the end consumer at the client node. Intuitively, the purpose of ABACUS is to discover enough about the resource consumption patterns of the functional components in the stream and about their mutual communication to partition them optimally between the client and the server. Components that communicate heavily should be co-located on the same node. At the same time, the CPU at a node should not be overloaded and load should be balanced across them. Sensitive functions should be executed on nodes marked trusted. ABACUS therefore seeks to automate performance and configuration management and simplify filesystem development by removing from the filesystem and application programmer the burden of load balancing and configuration. Particularly, the ABACUS prototype is designed to meet two principal requirements: to offer an intuitive programming model and to intelligently partition function without user involvement.

Intuitive programming model. The ABACUS-specific efforts expended by the programmer to write a filesystem or application on ABACUS should be limited relative to designing a filesystem or application for a traditional fixed allocation of function. In principle, automating function placement frees the programmer from the burden of optimizing the application for each combination of hardware and environment. ABACUS should make it easy for programmers to write applications so that the effort saved by not developing system-specific optimizations is not replaced by the effort taken to code in ABACUS.

Flexible and intelligent partitioning. The system should partition function so that optimal performance is achieved. In this research, performance is taken to be equivalent to "total execution time." The system, therefore, should partition function so that, in aggregate, applications should take the minimal amount of execution time.

5.3.2 Overview of the programming model

ABACUS consists of a programming model and a run-time system. The ABACUS programming model encourages the programmer to compose applications from small components each performing a functional step on a data stream. The run-time system continuously observes each component's behavior and system resource availability, using these to assign the component to the best network node.

Object-oriented languages such as C++ and java are widely used in building distributed and data-intensive applications. ABACUS chooses to monitor application objects (run-time instantiations of classes) and work to "place" them in the most proper node.

Component objects in an ABACUS application can be declared as either *mobile* or *anchored*. Mobile objects can be adaptively bound to client or to server at application start-time. They can also change placement at run-time. Mobile objects provide methods that explicitly checkpoint to and restore their state from a buffer. The `checkpoint/restore` methods are implemented assuming the object is quiescent, that is, not actively executing any externally exported method. Anchored objects, on the other hand, are fixed to a location determined by the designer at application design time and never migrate.

When the system is running, the application is represented as a graph of communicating mobile objects. Each object embodies state and provides an interface to the external methods which can be invoked to operate on that state. The object graph can be thought of as rooted at the storage servers by anchored (non-migratable) *storage objects* and at the client by an anchored *console* object. The storage objects provide persistent storage, while the console object contains the part of the application which must remain at the node where the application is started. Usually, the console part is not data intensive. Instead, it serves to interface with the user or the rest of the system at the start node. Objects make method invocations to each other, resulting in data moving between them. The data-intensive assumption implies that the application moves a large amount of data among a subset of the component objects.

Component object-based applications

The ABACUS prototype was developed to manage the partitioning of applications written in C++. While java would have been a more appropriate language because of its platform-independence, its limited performance on Linux during the time this research was conducted made it a bad choice. However, the reader will find out that the architecture of ABACUS and its resource management

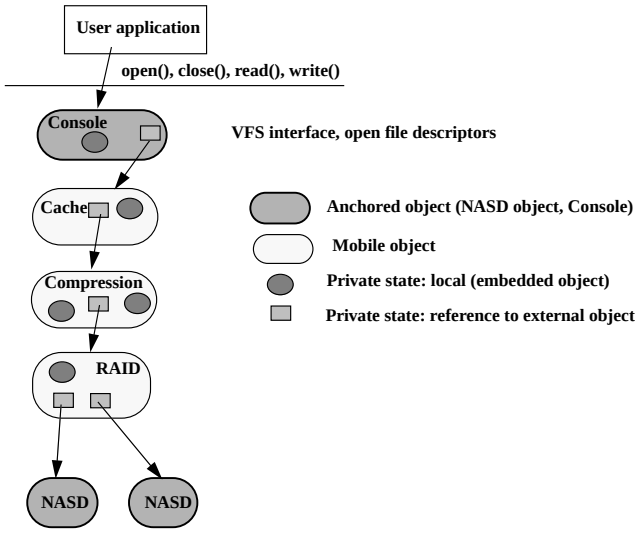


Figure 5.4: A filesystem composed of mobile and anchored objects. The open file table and VFS layer interface code is encapsulated in the console object which does not migrate. Storage is provided by disk-resident functions encapsulated in a base storage service exporting a NASD interface. The intermediate objects shown in the figure perform compression, RAID and caching functions and are mobile. These can migrate back and forth between client and server.

algorithms can be equally applied to a java-based application.

There are two kinds of "applications" that can run on ABACUS, filesystems and user-level applications. In the case of a filesystem, the console object corresponds to the code in the operating system which manages open file tables and implements VFS-layer functions. This layer does not move from the client host. Filesystem objects (e.g. caching, RAID, compression and directory management) do migrate back and forth between the client and the server. Storage access is implemented by a disk-resident function encapsulated in a C++ "NASDObject" class. This class is instantiated on each storage server. The instantiated C++ object is anchored to the server and does not migrate. Figure 5.4 shows a filesystem built on ABACUS. This discussion will sometimes refer to such a filesystem as a *mobile* filesystem.

In the case of a user-level program, the console consists of the `main` function in a C/C++ program. This console part initiates invocations which are propagated by the ABACUS run-time to the rest of the objects in the graph. The application can be composed of multiple mobile objects performing data-intensive processing functions such as decoding, filtering, counting and data mining.

From here on, the discussion will focus on filesystems. Supporting the migration of user-level application objects when such applications are layered atop the filesystem requires making the

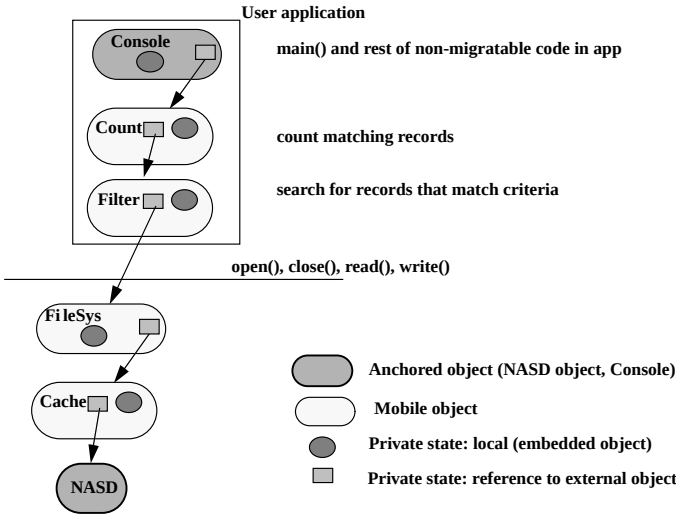


Figure 5.5: A user level application and its filesystem. Both are written such that their functions can migrate. The console is the main function in the application that displays the results on the screen and interacts with the local client. The file system is composed of layers that are encapsulated into migratable objects. The state relevant to the process' open files in the kernel is encapsulated into a migratable object (FileSys) that can migrate to the server.

filesystem console layer (encapsulated as a FileSys object in Figure 5.5) itself migratable. This poses well-known complications because some state, such as open-file descriptors, cannot be easily transferred transparently between nodes [Douglis, 1990]. Figure 5.5 depicts an application written on ABACUS. A discussion of how ABACUS can be extended to support the migration of application objects is deferred to Section 5.7.

As far as the ABACUS run-time is concerned, both filesystems and user applications appear as a graph of self-contained objects that can be monitored as black-boxes and moved back and forth between the client and the server as appropriate. The only property that the run-time cares about is whether the object can be moved or not (mobile or anchored).

ABACUS applications perform data processing by invoking methods that start at the console and propagate through the object graph. In general, an application such as a filesystem decomposes its processing into a set of objects, with each object providing a specific function. These objects can even be adaptively bound on a per file basis to provide different services. For instance, caching, reliability and encryption are functions that may have to be performed on the same file. ABACUS enables filesystems and storage-intensive applications to be composed of explicitly migratable objects, providing storage services such as aggregation, reliability (RAID), cacheability, filters, aggregators,

or any other application-specific processing. Each object embodies state and provides an interface to the external methods which can be invoked to operate on that state.

Block-based processing

Data-intensive applications often naturally process input data one block at a time, for some application-defined block size. Applications are implemented to iteratively process input data blocks because the algorithms they employ usually consume memory resources that grow with the input size. To limit memory usage, applications often allocate a memory buffer that is large enough to process an application-defined block of data, then iterate over the blocks in the input file, reusing the same memory buffer, thereby avoiding excessive memory use. For example, a search application like `grep` searching a file for a specific string works by allocating a fixed memory buffer. File blocks are successively read from the filesystem, scanned for the string, then discarded and replaced by the following block.

Data-intensive C++ objects within an application usually perform their processing by making requests to other objects. The amount of data moved in each invocation is an application-specific, relatively small, block of data (i.e. not the whole file). Most often, objects are organized into a stack, one per application or filesystem layer. Thus, method invocations propagate down and up the stack, processing one block at a time. Block-based processing is an attribute of the programming model that is not mandated for correctness, but for performance. The ABACUS run-time system builds statistics about inter-object communication. These statistics are updated at procedure return from an object. Thus, it is important that the application performs many object invocations during its lifetime to enable ABACUS to collect enough history to guide it in its placement decision.

5.3.3 Object model

ABACUS provides two abstractions to enable efficient implementation of object-based storage-intensive applications: *mobile objects* and *mobile object managers*. Mobile objects are the unit of migration and placement. Mobile object managers group the implementation of multiple mobile objects of the same type on a given node to improve efficiency, share resources or otherwise implement a function or enforce a policy that transcends the boundary of a single object. For instance, a function that requires access to more than one object is a file cache. The file cache implements a global cache block replacement policy and therefore needs to control the cache blocks of all files that it manages. Memory can therefore be reclaimed from a file that is not being accessed and

allocated to another active one.

Anchored objects

Because anchored objects are not mobile, that is, will at all times remain at the node where they are instantiated, ABACUS places little restriction on their implementation. Examples of anchored objects include NASD objects, which provide basic storage services, and atomicity objects which provide atomicity for multi-block writes. Both NASD and atomicity objects are anchored to a storage server. On the other hand, the console object is anchored to the client node.

Storage servers are assumed to implement a "NASD interface". Storage on a storage server is accessed through a C++ object, referred to as a NASD anchored object, instantiated at run-time, and implementing the "NASD interface" described in Chapter 3. ABACUS does not require that storage access follows a NASD interface. The anchored C++ object providing storage access can just as easily export a block-level interface. While C++ objects communicating with storage are required to know the particular interface exported by storage, ABACUS does not. ABACUS treats all invocations between objects as messages that transmit a certain amount of bytes, without attention to semantics.

Mobile objects

A mobile object in ABACUS is explicitly declared by the programmer as such. It consists of a state and the methods that manipulate that state. A mobile object is required to implement a few methods to enable the run-time system to create instantiations of it and migrate it. Mobile objects are usually of large granularity. Rarely are mobile objects simple primitive types such as integer or float. They usually perform functions of the size and complexity of a filesystem layer, or a database relational operator, such as filtering, searching, caching, compression, parity computation, striping, or transcoding between two data formats.

Mobile objects, like all C++ objects, have private state that is not accessible to outside objects, except through the exported methods. Unlike C++ objects, mobile objects in ABACUS do not have public data fields that can be accessed directly by de-referencing a pointer to the objects. Instead, all accesses to the objects state must occur through exported methods. This restriction simplifies the ABACUS run-time system. Since all accesses to a mobile object occur through its exported methods, the run-time's support for location transparency can be focussed on forwarding method invocations to an object to the current location of the object.

The implementation of a mobile object is internal to that object and is opaque to other mobile objects and to the ABACUS run-time system. The private state consists of embedded primitive types and instantiations of embedded classes (i.e. not visible outside the scope of the current object's class) and references to external objects. The ABACUS programming model makes a restriction that all external references must be to other mobile or anchored objects that are known to the ABACUS run-time system. References to other external resources such as sockets and shared memory regions are not legal.

ABACUS maintains information about the locations of mobile and anchored objects that it knows about. It uses this information to forward method invocations to objects as they migrate between client and server.

Of course, a mobile object can have access to its local private state through references that are not redirected or known to the ABACUS run-time system. The mobile object is responsible for saving this private state, however, when it is requested to do so by the system, through the `Checkpoint()` method. It is also responsible for reinstating this state (reinitializing itself) when the run-time system invokes the `Restore()` method. The `Checkpoint()` method saves the state to either an in-memory buffer or to a NASD object. The `Restore()` method can reinstate the state from either place. The signatures for the `Checkpoint()` and `Restore()` methods, which define the base class from which all mobile objects are derived, are illustrated in Figure 5.7. The discussion will differentiate private embedded state from mobile and anchored objects by referring to mobile and anchored objects as "ABACUS objects", since they are the only objects known the ABACUS run-time system.

Mobile object managers

Mobile object managers encapsulate private state for a collection of mobile objects of a given type. Often, a service is better implemented using a single object manager that controls the resources for a group of objects of the same type. Object managers thus aggregate the implementation of multiple objects of the same type. For example, a file system may wish to control the total amount of physical memory devoted to caching, or the total number of threads available to cache requests. Mobile object managers provide an interface that is identical to that of the types they contain except that they take an additional first argument to each method invocation, which represents a reference to the individual object to be invoked from the collection of objects in the aggregated manager object.

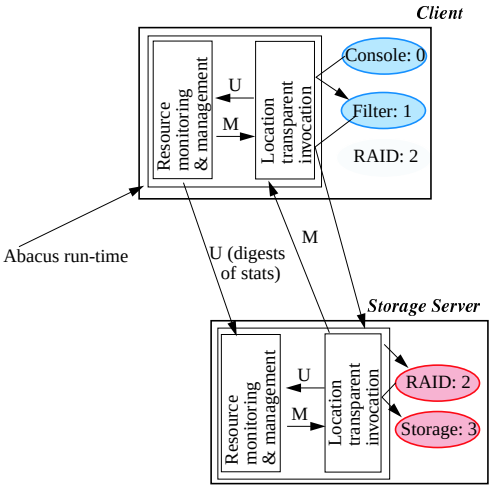


Figure 5.6: An illustration of an ABACUS object graph, the principal ABACUS subsystems, and their interactions. This example shows a filter application accessing a striped file. Functionality is partitioned into objects. Inter-object method invocations are transparently redirected by the location transparent invocation subsystem of the ABACUS run-time. This subsystem also updates the resource monitoring subsystem on each procedure call and return from a mobile object (arrows labeled "U"). Clients periodically send digests of the statistics to the server. Finally, resource managers at the server collect the relevant statistics and initiate migration decisions (arrows labeled "M").

5.3.4 Run-time system

Figure 5.6 represents an overview of the ABACUS run-time system, which consists of (i) a migration and location-transparent invocation subsystem, or *binding manager* for short; and (ii) a resource monitoring and management subsystem, *resource manger* for short. The first subsystem is responsible for the creation of location-transparent references to mobile objects, for the redirection of method invocations in the face of object migrations, and for enacting object migrations. Finally, the first subsystem notifies the second at each procedure call and return from a mobile object.

The resource manager uses the notifications to collect statistics about bytes moved between objects and about the resources used by active objects (e.g., amount of memory allocated, number instructions executed per byte processed). Moreover, this subsystem monitors the availability of resources throughout the cluster (node load, available bandwidth on network links). An analytic model is used to predict the performance benefit of moving to an alternative placement. The model also takes into account the cost of migration — the time wasted to wait until the object is quiescent, checkpoint it, transfer its state to the target node and restore it on that node. Using this analytic model, the subsystem arrives at the placement with the best *net benefit*. If this placement is different

from the current configuration, the subsystem effects object migration(s).

The ABACUS run-time system has mechanisms to find application objects and migrate them. In general, when a function is moved, both the code for the function as well as its execution state (local and external references accessed by the function) must be made accessible at the new node [Jul et al., 1988]. A mechanism to transfer this state from one node to another is therefore necessary to enable adaptive function placement at run-time.

Emerald [Jul et al., 1988] is a seminal implementation of a language and run-time system supporting the mobility of application objects. ABACUS uses similar mechanisms to those provided by Emerald to enact object mobility. The focus of the ABACUS prototype, however, was not on mobility mechanisms but rather on using these mechanisms to improve data-intensive and filesystem performance through judicious and automatic placement of their objects on the proper nodes. The remainder of this section describes the mechanisms used by the ABACUS to effect object migrations and to find objects after they migrate. The following section is devoted to describing the algorithms used by the run-time system to achieve good placement for application objects.

Creating and invoking mobile objects

The creation of mobile objects is done through the ABACUS run-time system. When the creation of a new ABACUS object is requested, the run-time system allocates a network-wide unique run-time identifier (`rid`) for the new object. This identifier is returned to the caller and can be used to invoke the new mobile object from any node, regardless of the current location of the mobile object. After allocating a network wide unique `rid`, the run-time system creates the actual object in memory by invoking the object manager for that type. If no object manager exists, one is created for that type. Object managers must implement a `CreateObj()` method which takes arguments specifying any initialization information and returns a reference that identifies the new object within that object manager. This can be thought of as a virtual memory reference to the created object, although the object manager is free to construct this reference in the way it desires. The object manager creates the "actual" object, *e.g.*, in C++, by invoking the `new` operator, and then return a reference to the object to the run-time. This reference, called a "manager reference", is used to uniquely identify the object within the collection of objects managed by the manager.

The run-time system maintains tables mapping each `rid` to a *(node, object_manager, manager_reference)* triplet. As mobile objects move between nodes, this table is updated to reflect the new node, new object manager, and new manager reference. Mobile objects use the `rid` to invoke

other mobile objects, so that their invocations are redirected to the proper node. The run-time system converts the `rid` to a memory reference within an object manager on a given network node.

Furthermore, types must registered with the ABACUS run-time system, and be associated with an object manager class, so that the run-time system can determine what object manager to create if none exists. Each object type may export a different set of methods. Invocations of ABACUS objects are compiled into statements that first invoke the ABACUS run-time system to map the `rid` onto a *(node, object_manager, manager_reference)* triplet, then forward the invocation to the object manager on the target node where the object currently resides. The object manager is invoked by passing it the *manager_reference* as the first argument, followed by the actual arguments for the method invocation. All ABACUS objects (mobile and anchored) define their interface in IDL (Interface Definition Language [Shirley et al., 1994]), allowing the ABACUS run-time to create stubs and wrappers for their methods.

Locating objects

There are two kinds nodes in an ABACUS cluster: clients and servers. Servers are nodes on which at least one base storage object resides. Clients are nodes that access storage objects on the servers. A server can therefore be a client of another storage server. Top-level invocations originate at the console object, which, like any ABACUS object, may hold `rid` references to other objects in the graph. Inter-object calls are made indirectly via the run-time system. The ABACUS run-time forwards inter-object calls appropriately. For objects in the same address space, procedure calls are used and data blocks are passed without copies. In other cases, remote procedure calls (RPCs) are used. The node where the console object runs is called the “home node” for all the migratable objects in the graphs reachable from it. ABACUS maintains the information necessary to perform inter-object invocations in ABACUS location tables. Location tables are hash tables mapping a `rid` to a node, object manager, manager reference triplet.

Moving objects

Each object must conform to a small set of rules to allow the ABACUS run-time to transparently migrate it. Migration of objects requires the transfer of state of object from source to target node. Consider migrating a object from a client to a storage node. The algorithm proceeds as follows. First, new calls *to* the migrating object are blocked to make it quiescent. Then, the binding manager waits until all invocations that are active in the migrating object have drained (returned). Migration

is canceled if this step takes too long. Once the object is quiescent, it is checkpointed, its state transferred and the checkpoint restored to a newly created object of the same type on the storage node. Then, local and remote location tables are updated to reflect the new object placement. Next, any waiting invocations are unblocked and are redirected to the proper node by virtue of the updated location table. This algorithm extends to migrating whole subgraphs of objects.

ABACUS requires that each mobile object in the graph implement a `Checkpoint()` and `Restore()` method which conclude any background work and then marshall and unmarshall an object's representation into migratable forms. The mobility of code is ensured by having nodes that do not have the code for an object read it from shared storage. The mobility of execution state is enacted through application specific checkpointing. A `Checkpoint()` message is sent to the object on migration. The object marshalls its private state to a buffer and returns it to the runtime system which passes it to the `Restore()` method at the target node. This method is invoked to reinitialize the state of the object before any invocations are allowed.

5.4 Automatic placement

This section describes the performance model and the algorithms used by the ABACUS run-time system to drive placement decisions. ABACUS resource managers gather per-object resource usage and per-node resource availability statistics. The resource usage statistics are organized as graphs of timed data flow among objects. The resource manager on a given server seeks to perform the migrations that will result in the minimal average application completion time across all the applications that are accessing it. This amounts to figuring out what subset of objects executing currently

```
// the abstract mobile object class
// NasdId: a unique identif
class abacusMobileObject {
    public:   ier for a persisent base storage object
    int Checkpoint(void **buffer, NasdID nasdId, int *csize);
    int Restore(void *buffer, NasdId nasdId, int csize);
};
```

Figure 5.7: The interface of a base mobile object in the ABACUS prototype. The interface consists of two methods: `Checkpoint()` and `Restore()`. The type `NasdId` denotes the set of all NASD identifiers. The notation is in C++. Hence, the "*" symbol denotes an argument that is passed by reference. `csize` represents the size of the checkpoint created or passed in the buffer.

on clients can benefit most from computing closer to the data. Migrating an object to the server could potentially reduce the amount of stall time on the network, but it could also extend the time the object spends computing if the server's processor is overloaded.

Resource managers at the servers use an analytic model to determine which objects should be migrated from the clients to the server and which objects, if any, should be migrated back from the server to the clients. The analytic model considers alternative placement configurations and selects the one with the best *net benefit*, which is the difference between the benefit of moving to that placement and the cost of migrating to it.

A migration is actually enacted only if the server-side resource manager finds a new placement whose associated net benefit exceeds a configurable threshold, B_{Thresh}. This threshold value is used to avoid migrations that chase small improvements, and it can be set to reflect the confidence in the accuracy of the measurements and of the predictive performance model used by the run-time system.

While server-side resource managers can communicate with each other to coordinate the best global placement decision, this would cause extra overhead and complexity in configuring the storage servers. Under such a scheme, storage servers have to know about each other and be organized into cooperating groups. The ABACUS implementation foregoes this extra benefit of server-to-server coordination for the sake of robustness and scalability. ABACUS server-side resource managers do not communicate with one another to figure out the globally optimal placement. A server-side resource manager decides on the best alternative placement considering only the application streams that access it.

At any point in time, the object graph for an application is partitioned between client and server. For a server-side resource manager to determine the best placement decision, it must know the communication and resource consumption patterns of the objects that are executing on the client. Given information about the client-side as well as local subgraphs, and given statistics about node load and network performance, the resource manager should be able to arrive at the most proper placement.

This is implemented in ABACUS by having server-side resource managers receive per-object measurements from clients. A server-side resource manager also receives statistics about the client processor speed and current load and collects similar measurements about the local system and locally executing objects. Given the data flow graph between objects, the measured stall time of client-side objects' requests for data, and the round-trip estimated latency of the client-server net-

work link, the model estimates the change in stall time if an object changes location. Given the instructions per byte and the relative load and speed of the client/server processors, it estimates the change in execution time if the object is moved to a new node.

This simple model would suffice if the server resources were not shared by many clients. However, this is very rarely the case in practice. Under a realistic scenario, a migration of an object from a client to a server may slow-down other objects. This effect must be taken into account by the ABACUS performance model.

In addition to the change in execution time for the migrated object, the model also estimates the change in execution time for the other objects executing at the target node (as a result of the increased load on the node's processor). Together, the changes in stall time and execution time amount to the benefit of the new placement. In computing this benefit, the analytic model assumes that history will repeat itself over the next window of observation (the next H seconds). The cost associated with a placement is estimated as the sum of a fixed cost (the time taken to wait until the object is quiescent) plus the time to transfer the object's state between source and destination nodes. This latter value is estimated from the size of the checkpoint buffer and the bandwidth between the nodes.

5.4.1 Goals

There are several different performance goals that the ABACUS run-time system can pursue. One alternative is to allocate server resources fairly among competing client objects. Alternatively, the system can provide applications with performance guarantees and allocate resources to meet the promised guarantees. Yet another goal would be to minimize the utilization of the network. Finally, one goal is maximize a global metric associated with user-perceived performance, such as average completion time of applications.

This chapter describes algorithms that pursue a performance goal which is widely sought in practice, namely that of minimizing the *average* completion time of complete runs of applications. This goal is widely used because it directly maps onto a user-perceived notion of performance. The performance model in ABACUS is self-contained, however, and can be extended or modified to implement different policies. The run-time system makes decisions to adapt the allocation of server resources to minimize average request completion time. We assume here that no explicit information about the future behavior of applications is disclosed to the system. Instead, ABACUS assumes that the future behavior of applications is accurately predicted by their recent past.

Note that the goal of minimizing the number of bytes moved over the network is not always desirable. For example, consider a client-side cache that is aggressively prefetching file blocks from a server to hide network latency from applications. While the cache consumes a lot of network bandwidth and moves a large number of bytes, it does save cache misses and reduces application stall time. Moreover, this goal presumes that moving bytes over the network has uniform cost. In practice, not all communications are equally costly (at least in terms of latency) because the available bandwidth of network links varies across topologies and with time as a function of the applied load.

5.4.2 Overview of the performance model

The ABACUS run-time must identify the assignment of mobile objects to network nodes that is best in reducing the average completion time. This discussion considers a single server case but with multiple clients accessing it. A cost-benefit performance model is derived for this case. Servers in ABACUS act independently and do not coordinate resource allocation decisions. This design requirement was made to limit complexity and improve robustness.

An ABACUS cluster is composed of clients and servers. The mobile objects associated with an open file start executing by default on their "home node", the node where the file was open. They can migrate to one of the storage servers, where the NASD objects storing that file reside. At any point in time, the graph of mobile objects associated with a given file is partitioned between the home node and those storage servers, referred to as the "base storage servers."

The server is shared by many clients and hosts a number of non-migratable objects that provide basic storage services to end clients. Because non-migratable objects cannot be executed at a client, while "mobile" objects can, the ABACUS run-time system is concerned with allocating server resources beyond what is consumed by non-migratable objects to "mobile" application objects. The ABACUS server-side resource manager is responsible for allocating its resources to the proper mobile objects such that the performance goal is maximized. To estimate the average application completion time given an object placement, an analytic model that estimates average application execution time in terms of inherent or easily measured system and application parameters is developed. The discussion first considers the case of a single application executing by itself. Then, it generalizes the model to handle the case of concurrent applications.

Figure 5.8 shows a sketch of an applications executing on a client and accessing a storage server. Filesystem and application objects are organized into layers. The application's console makes iter-

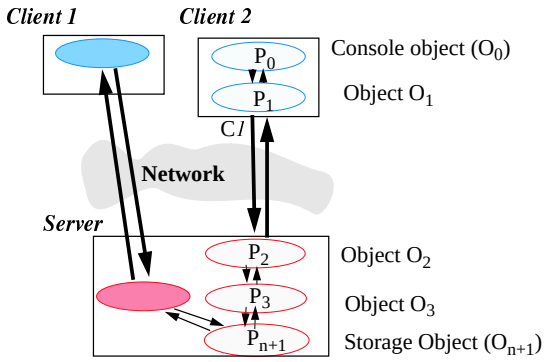


Figure 5.8: This figure shows how mobile objects interact in a layered system. Most of the invocations start at the topmost console object, trickling down the layers. An object servicing a request may invoke one or more lower level mobile or NASD objects.

ative requests that trickle down the layers, causing an amount of data to be read from storage and processed, and passed up. At each layer some processing occurs, reducing or (seldom) expanding the amount of data returned back up to the upper layer. Similarly, the application can be writing data to stable storage, where data flows downward through the layers, each layer performing some processing and passing the data down. This layered model of processing simplifies the analysis, yet it is general enough to capture a large class of filesystem and data-intensive applications. Mobile objects perform two kinds of activities: computing (executing instruction on the local CPU) or communicating (making a method invocations to another mobile or NASD object). One object may invoke more than one object in a lower layer.

To describe the analytic model in detail, a few definitions and notations are required. An object O_j is characterized by an inherent processing ratio, expressed in instructions/byte, denoted by h_j. This ratio captures the inherent compute-intensiveness of the processing performed by the object on each byte of data and is independent of the processor speed or the bandwidth available to the object. It can therefore be used in estimating the execution time (to process one block) of the object when moved between nodes.

In this discussion, the *raw* processing rate of node k's processor is denoted by R_k, expressed in instructions per second. The effective processing rate of node k as observed by an executing object O_j on that node is denoted by r_j. This is equal to the effective processing rate available from the processor's node, denoted as r_k, and is less than the raw processing rate of the processor because multiple threads may be contending on the processor. Of course, the obvious inequality holds at all

times, for all objects O_j executing on node k:

$$r_j = r_k \leq R_k$$

5.4.3 Single stack

Consider the application of Figure 5.8. Assume that each object is associated with a single thread. The thread awaits invocations, performs some processing then invokes one mobile object at the lower layer. The thread blocks until the invocation returns. A single NASD object on a single server is accessed by the application. The bandwidth of read or write requests to a NASD object is denoted by D. This is assumed to be independent of server load, because the server is assumed to serve NASD requests at a higher priority than remotely executing objects.

Invocations start at the console object and trickle down objects O_1 through O_n. The application is observed over a window of time H during which N console requests are initiated and completed. Let's denote by b_j the total number of bytes processed by object O_j during the observation window. Let p_j denote the time spent processing by object O_j. Assume, without loss of generality, that under the original placement, objects O_1 through O_k execute on the client and objects O_{k+1} through O_n execute on the server. The console is referred to as O_0 and the NASD non-migratable object is denoted by O_{n+1}. The elapsed time for the application in Figure 5.8 can be written as:

$$T_{app} = \sum_{j=0}^{n} p_j + \sum_{j=0}^{n} C_j \tag{5.1}$$

where C_i denotes the communication "blocking" or "stall" time between O_i and O_{i+1}. Communication time is the time during which the call thread blocks waiting for data to be sent or received to the invoked object. This does not include processing time at the invoked object, but the time truly spent blocking while the data is being transferred. That is, after the processing at the invoked object has completed. Equation 5.1 can be rewritten in terms of inherent application and system parameters, as follows:

$$T_{app} = \sum_{j=0}^{n} \frac{h_j b_j}{r_j} + \sum_{j=0}^{n} C_j \tag{5.2}$$

Equation 5.2. expresses the processing time p_j in terms of the instructions per byte executed by the object, the number of bytes processed by the object during the observation window, and the effective processing rate r_j. The numerator $h_j b_j$ represents the number of instructions executed by the object during the observation window, and the denominator is the virtual processor rate

as observed by the object. Let's further assume that local communication within one machine is instantaneous, i.e. an object does not block when communicating with another object co-located on the same node any longer than the completion time of that lower lever object. Then, Equation 5.2. can be written as:

$$T_{app,old} = (\sum_{j=0}^{k} \frac{h_j b_j}{r_{client}}) + C_k + (\sum_{j=k+1}^{n} \frac{h_j b_j}{r_{server}}) \tag{5.3}$$

If the object placement changes so that objects O_1 through O_{k-1} execute on the client and objects O_k through O_n execute on the server, then the new application execution time can be expressed as:

$$T_{app,new} = (\sum_{j=0}^{k-1} \frac{h_j b_j}{r'_{client}}) + C_{k-1} + (\sum_{j=k}^{n} \frac{h_j b_j}{r'_{server}}) \tag{5.4}$$

Let's further assume, that $r'_{j,client} = r_{j,client} = R_{client}$ and $r'_{j,server} = r_{j,server} = R_{server}$ because the application is not sharing the client or server processors with any other concurrent applications, and because different objects in the application stack process the data serially and therefore do not contend for the processor at the same time. Then, Equation 5.4 can be rewritten as:

$$T_{app,new} = (\sum_{j=0}^{k-1} \frac{h_j b_j}{r_{client}}) + C_{k-1} + (\sum_{j=k}^{n} \frac{h_j b_j}{r_{server}}) \tag{5.5}$$

In this simplistic case, the optimal placement can be determined by finding the k for which $T_{app,new}$ is minimized. For instance, let's further assume that the server and client processor rates are the same ($r_{client} = r_{server}$), then the ideal k would be the one which minimizes the stall time, C_{k-1}. In this simple case, the equation implies that the stack of objects should be partitioned at the level that would minimize the number of bytes transferred across the network (synchronous communication), or the point of minimal communication between two successive layers in the stack.

5.4.4 Concurrent stacks

The effective processing rate of object O_j at the server before and after migration is denoted in Equations 5.3 and 5.4 by r_{server} and r'_{server} respectively. These processing rates can be related to the raw processing rate at the server and the current load on that server. The effective server processing rate can be estimated as the raw processing rate divided by the node load $L(server)$. This is simply the average number of processes in the ready queue at the server. Here, the "processor

sharing" assumption is made so that object O_j receives a processing rate from the processor that is the ratio of the raw rate to the current load, or more precisely:

$$r_{server} = \frac{R_{server}}{L(server)} \tag{5.6}$$

In general, any priority or processor scheduling scheme can be supported as long as ABACUS has an adequate description of it. However, a processor sharing policy at equal priority was chosen for the implementation. $L(server)$ at the server can be measured. The change in $L(server)$ as a result of a relocation must be estimated. Each active graph can be thought of as contributing a load between 0 and 1. Intuitively, the load is interpreted as follows. If the object graph is computing all the time, never blocking for I/O, then its associated load is 1. If the object graph is computing only $\frac{1}{10}$th of the time, then the load it contributes to the system is $\frac{1}{10}$. The node load is is simply the sum over all the active sessions at the server of their contributed load, or equivalently:

$$L'(server) = \sum_{a \in ActiveSessions} CL(a) \tag{5.7}$$

The load contributed by a session, $CL(a)$ can be computed from the stall and processing time of the entire stack, or:

$$CL(a) = \frac{\sum_{j=k+1}^{n} p_j}{C_k + \sum_{j=0}^{n} p_j} \tag{5.8}$$

Equation 5.8 computes the load of active session a as the ratio of the processing time at the server to the total (processing and stall time). The portion of the stack at the server is passive whenever the client part of the stack is active, or whenever data is being communicated over the network. These times are summed over all the objects in the session. The processing time for an object is simply the ratio of the bytes processed by the effective processing rate. Expanding the processing time in terms of these parameters, Equation 5.8 becomes

$$CL(a) = \frac{\sum_{j=k+1}^{n} \frac{b_j h_j}{r_{server}}}{C_k + \sum_{j=0}^{k} \frac{b_j h_j}{r_{client}} + \sum_{j=k+1}^{n} \frac{b_j h_j}{r_{server}}} \tag{5.9}$$

After object O_k migrates to the server, the new load can be computed by rewriting Equation 5.9:

$$CL'(a) = \frac{\sum_{j=k}^{n} \frac{b_j h_j}{r'_{server}}}{C_{k-1} + \sum_{j=0}^{k-1} \frac{b_j h_j}{r_{client}} + \sum_{j=k}^{n} \frac{b_j h_j}{r_{server}}} \tag{5.10}$$

Equation 5.10 requires the new effective processing rate of the server's processor. The effective processor rate is simply the raw rate divided by the new node load $L'(serer)$.

5.4.5 Cost-benefit model

If the server has foreknowledge of all the application that will start, then it can determine the best "schedule" for allocating its resources to minimize average completion time. In the absence of this knowledge, the server has to operate based only on statistics about the past behavior of applications that have already started. The approach implemented in the ABACUS prototype is to allocate the server's resources greedily to the currently known applications, and then reconsider the decision when other applications start. The implementation of this approach is described in this section.

Because of the absence of future knowledge, the greedy algorithm may migrate an application to the server, and then shortly after that, a more data-intensive application may start. Because the cost of relocating objects is not negligible, ABACUS has to consider whether reallocating the server's resources to the more "deserving" application is worth it.

A cost-benefit model can be used to drive such decisions. Each object relocation induces an overhead and therefore adds to the application execution time. It, of course, could potentially result in substantial savings in execution (stall) time. Cost-benefit analysis states that a relocation decision should be taken if the cost that it induces is offset by the benefit that it will bring about. Cost-benefit analysis requires first defining a common currency in which cost and benefit can be expressed, and devising algorithms to estimate the benefit and the cost in terms of this common currency. ABACUS uses *application elapsed time* measured in seconds as its common currency. The net benefit of a migration R, $B_{net}(R)$, is computed as the potential application benefit, ΔT_{apps}, minus the cost of migration, $\mathcal{C}(R)$:

$$B_{net}(R) = \Delta T_{apps} - \mathcal{C}(R) \tag{5.11}$$

The first term in the equation above accounts for the change in the execution time for the affected applications. For example, if a filter application object is moved from client to server, the affected applications include the filter application and the applications currently executing on the server. The first term of Equation 5.11 is a sum over all the affected applications. It may include positive terms, in the case when a relocation speeds up an application, and negative terms when an application is slowed down as a result of the new placement, either because of increased network stall time, or due to an increase in the node load on a shared processor.

The second term of equation 5.11 accounts for the migration penalty of waiting until the object is quiescent, checkpointing it, transferring its state, and restarting it on the new node. The cost of migration, $\mathcal{C}(R)$, is a sum over all the applications that need to be migrated. The expected time waiting until the object is quiescent is predicted by a per-node history of the time needed to make an object quiescent in a particular graph. Checkpoint and restore are modeled with a fixed cost, and the expected cost of the transfer is the expected size of the marshalled state divided by the recently observed bandwidth of the transfer medium. The expected size of the marshalled state can be inferred from the program's data size, since that is likely to be a conservative predictor. [1]

ABACUS treats server memory as a hard constraint. Applications that migrate to the server are not to exhaust server memory. If server memory is exhausted, mobile objects are evicted. Mobile objects may be evicted back to their home nodes when memory is short. This may be necessary to free enough memory to allow a new object currently running on a client to migrate to the server. This will happen, of course, only if ABACUS estimates that the new relocation is likely to maximize the net benefit.

Recall that Equations 5.3 and 5.4 estimate the execution time of an application before and after migration. Taking the difference results in the change of execution of one application. Summing that difference over all applications produces an estimate of ΔT_{apps}. This and Equation 5.11 can be used, therefore, by the ABACUS resource managers to estimate net benefit.

Stall time over the network before and after migration, C_k and C_{k-1} in the example of Equations 5.3 and 5.4, must be estimated by the ABACUS run-time system. Because some communication is asynchronous, and because messages can be issued in parallel, estimating stall time — even before migration — is not straightforward. So far, this section has assumed that the object that needs to be migrated from a client is invoked by a single application. In general, a filesystem object may be shared by several active applications so that its migration affects the performance of all of them. In latter general case, some application sharing an object may benefit from the migration while others may suffer. This, however, is naturally accounted for because ΔT_{apps} is estimated by aggregating the difference of equation 5.3 and 5.4 over all affected applications, including all applications sharing access to the migrated object. Similarly, Equation 5.11 can be very easily extended to handle the multiple object migration case.

[1] In the ABACUS prototype, interested object managers can assist the system in estimating the amount of data that needs to be checkpointed by implementing an optional `GetCheckpointSize()` method. For instance, a filesystem cache may allocate a large number of pages, but the amount of data that needs to be checkpointed on migration is proportional to the dirty pages which must be written back or transferred to the target node, which is usually much smaller.

Client-side algorithm

Each ABACUS client-side resource manager periodically computes local statistics about the inherent processing ratios, bytes moved and stall times. For each candidate graph, it identifies source and sink storage nodes and iteratively considers "pushing up or down" objects to these nodes, by sending the relevant statistics to the target server asking it to compute the net benefit of each relocation using the above described model.

Server-side algorithm

Server-side resource managers collect statistics from the clients that are actively accessing them. They estimate the net benefit for each alternative placement, B_{net}, and then initiate a migration to the placement that generates the largest net benefit. The server can choose a different value of k, the point at which to split an object stack between client and server, for each active file. It has to select the combination of k's that generates the smallest average remaining completion time. For each combination of k's across the active stacks, the server-side resource manager computes the net benefit from moving to this alternative placement.

5.4.6 Monitoring and measurement

The previous section presented an analytic model that predicts the net benefit from a given relocation. The performance model requires inputs about object processing rate, stall time, and number of bytes processed over the observation window. Estimating the net benefit aggregates the difference of equations 5.3 and 5.4, which require values for h_i, b_i. These values are independent of the particular object placement, and depend only on algorithmic and input characteristics of the application. These two values must be measured or estimated from observed measurements. In addition to these two values, these equation requires knowledge of the change in stall time between different placements C_i, and the node load at the nodes under each possible placement, namely L_{client} and L_{server}. Furthermore, because ABACUS "bin-packs" objects in the server subject to the amount of memory available at the server, the memory consumption of mobile objects must be monitored and recorded.

On a single node, threads can cross the boundaries of multiple mobile objects by making method invocations that propagate down the stack. The resource manager must charge the time a thread spends computing or blocked to the appropriate object. Similarly, it must charge any allocated memory to the proper object.

The benefit of judicious object placement is especially important for active applications, those applications that actively access storage and process data. Discovering and correcting an improper partitioning for such applications translates into substantial performance gains. The observation that this research makes is that it is exactly those applications that a monitoring system should know the most about. Applications that actively move and process large amounts of data expose to ABACUS valuable and recent information about their communication and resource consumption patterns. The ABACUS run-time collects the required statistics over a moving history window, capturing those active applications that would benefit from a potential better placement. Precisely, ABACUS maintains statistics about the previous H seconds of execution, a window referred to as the observation window. This subsection describe how some of these statistics are collected.

Memory consumption

Tracking the memory consumption of mobile objects is problematic for two reasons. First, mobile objects can dynamically allocate and free memory. This requires tracking their dynamic memory allocation to know their true memory consumption. Dynamic memory allocation can be monitored by providing wrappers around the `new` and `delete` operators for applications to use.

A more difficult problem is caused by object managers. Object managers manage resources on behalf of multiple mobile objects of the same type. The implementation of mobile objects is therefore opaque to the run-time system. For example, the memory consumed by a mobile "cache" object depends on the number of pages owned by that object within the "cache object manager." This information is only known to the object manager.

The approach taken by the run-time system to monitor memory consumption is to require each object manager to implement a `GetMemCons()` method. This method takes a *manager_reference* as a first argument, and returns the number of bytes consumed by the object. ABACUS assumes that object managers on different nodes use similar implementations. Thus, the memory consumption of an object in one manager is a good predictor of its consumption on a remote object manager.

The ABACUS run-time does not reserve memory with the operating system. Instead, it assumes that it is allocated an amount of memory by the operating system. ABACUS manages the use of this memory by allocating it to the proper application objects. ABACUS monitors the memory consumption of application objects and is able to detect memory shortage by keeping track of the total amount of unallocated memory. This is updated every time a memory allocation or de-allocation is performed by an application object.

Bytes moved: b_i

The bandwidth consumption of mobile objects is monitored by observing the number of bytes moved between mobile objects in inter-object invocations. Mobile objects invoke each other through the ABACUS run-time, which in turns sends a message to the local ABACUS resource manager, specifying the network-wide unique object identifier (`rid`) of the source object and of the target, as well the number of bytes moved between them. Resource managers therefore accumulate a timed data flow graph whose nodes represent mobile objects and edges represent bytes moved along inter-object invocations. These data flow graphs are of tractable size because most data-intensive applications and filesystems have a relatively limited number of layers or objects.

Inherent processing ratio: h_i

Estimating CPU consumption for a mobile object on a given node is more problematic than estimating memory consumption or the bytes moved between objects. Not only do object managers hide the implementation of objects, they cannot be asked for assistance in estimating CPU time consumed by each object. Estimates of the memory consumed by an object's implementation is relatively easy to provide by an object manager. This is not the case for CPU consumption. The object manager can insert timestamps whenever processing starts on behalf of a given object and whenever it finishes. But this is not sufficient since the execution of some statements can cause the whole process to block, which results in inflated estimates.

The operating system maintains CPU consumption information on behalf of operating system units of execution such as processes or threads. Such OS-level entities may contain many object managers, each with several mobile objects. ABACUS estimates the instructions per byte as follows. Recall that ABACUS monitors the number of bytes moved between objects by inspecting the arguments on procedure call and return from a mobile object. The number of bytes transferred between two objects is then recorded in a timed data flow graph. Given the number of bytes processed by an object, computing the instructions/byte amounts to monitoring the number of instructions executed by the object during the observation window. Given the processing rate on a node, this amounts to measuring the time spent computing within an object. Because an OS scheduler allocates the CPU to the different execution entities transparently, accurately accounting for the time spent executing within an object requires the operating system to notify ABACUS when scheduling decisions are made.

In the prototype implementation, ABACUS is implemented on a Pentium cluster running the

Linux operating system. In this environment, ABACUS uses a combination of the Linux interval timers and the Pentium cycle counter to keep track of the time spent processing within a mobile object with a limited level of operating system support (albeit at the cost of some inaccuracy in the measurements). ABACUS uses two mechanisms to measure this time, interval timers and the Pentium cycle counter. Linux provides three interval timers for each thread. The `ITIMER_REAL` timer decrements in real time, `ITIMER_VIRT` decrements whenever the thread is active in user-mode, and `ITIMER_PROF` decrements whenever the thread is active in user or system-mode. ABACUS uses the `ITIMER_VIRT` and `ITIMER_PROF` timers to keep track of the time spent computing in user/system mode and then charge that time to the currently executing object of a thread.

The only complication is that interval timers have a 10 ms resolution and many method invocations complete in a shorter period of time. To measure short intervals accurately, ABACUS uses the Pentium cycle counter which is read by invoking the `rdtsc` instruction (using the `asm("rdtsc")` directive within a C/C++ program). Using the cycle counter to time intervals is accurate as long as no context switch has occurred within the measured interval. Hence, ABACUS uses the cycle counter to measure intervals of computation during which no context switches occur, otherwise, ABACUS relies on the less accurate interval timers. We detect that a context switch has occurred by seeing if the time reported by `ITIMER_PROF`/`ITIMER_REAL` and the cycle cycle counter for the candidate interval differ significantly.

While this scheme requires less operating system support and complexity, it is less accurate then one in which the operating system scheduler notifies the ABACUS run-time system whenever it makes a processor scheduling decision.

Stall time

Measuring stall time at current node. To estimate the amount of time a thread spends stalled in an object, one needs more information than is currently provided by the POSIX system timers. We extend the `getitimer`/`setitimer` system calls to support a new type of timer, which is denoted by `ITIMER_BLOCKING`. This timer decrements whenever a thread is blocked and is implemented as follows: When the kernel updates the system, user, and real timers for the active thread, it also updates the blocking timers of any threads in the queue that are marked as blocked (`TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`).

Estimating new stall time at new node. When an object has multiple threads, it can potentially overlap outstanding messages with each other or with computation. Thus, the network time

spent by messages over the network does not translate into "stall time". To account for parallelism, ABACUS must differentiate among two types of inter-module communication, *synchronous* and *asynchronous*. Synchronous calls block the application until they complete. Asynchronous calls do not block the application, generally performing some form of background task like prefetching, write-back or call-back.

ABACUS resource managers must ignore "asynchronous communication" because it does not add to stall time and therefore should not be accounted for in calculating estimated benefit. Asynchronous communication can be explicitly declared by an object as such. Otherwise, it can inferred whenever possible; any invocation that starts or completes (returns to the object) when no synchronous application requests are in progress is considered asynchronous. The effects of asynchronous messages are indirectly accounted for, however, because network bandwidths and processor speeds are observed, not predicted.

Synchronous communications can also occur in parallel with one another rather than serially. In this case, although the number of bytes moved by a pair of objects is the same, the stall time would be lower for the object making parallel transfers. In practice, an object performing serial messaging would benefit more from avoiding the network because it is blocking on the network more often. Fortunately, the resource manager has information in its data flow graph about the timings of when communications were performed, so it knows what groups of messages are sent "simultaneously." The resource managers coalesce messages leaving an object within a short window of time into a single "round of messaging." The "stall time" can then be estimated from the number of rounds and the measured bandwidth of the network links used.

Processor load and available network bandwidth

ABACUS measures the load on a given node, defined as the average number of threads in the ready queue over the observation window, H. This value is required to estimate the processing time for an object after migration to a new node given the object's instruction per byte and number of bytes processed. Linux reports load averages for 1 min., 5 min., and 15 min. via the `/proc/loadavg` pseudo-file. Linux was augmented with an ABACUS specific load average which decays over the past H seconds and report this value as a fourth value in `/proc/loadavg`.

ABACUS resource managers monitor bandwidth availability on the network periodically by "pulling" a specified number of bytes from remote storage servers that are actively being accessed, deriving the fixed and per-byte cost of communication over a given link. These storage servers

represent potential candidates where mobile objects can migrate to.

5.5 A *mobile* filesystem

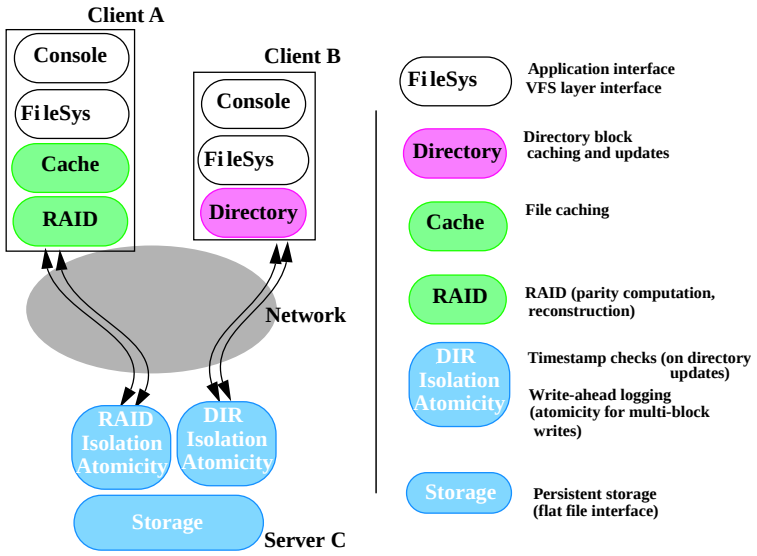
To help validate that the utility of the ABACUS programming model and to demonstrate the effectiveness of the ABACUS run-time, a prototype distributed filesystem, ABACUSFS, was designed and implemented on it. This section presents an overview of this mobile filesystem built on ABACUS. The next section presents an evaluation of the ability of ABACUSFS to adapt. It also includes a more detailed description of the specific filesystem components being evaluated.

5.5.1 Overview

Stackable and object-based filesystems, such as Spring and Hurricane, already demonstrate that specializing filesystem functions on a per-file basis are possible, and can be implemented elegantly with an object-like model [Khalidi and Nelson, 1993]. The following section describes in more detail how a distributed filesystem was developed on ABACUS. The ABACUSFS filesystem serves two purposes. First, it is a serious complex application that tests the proposed programming model and run-time system. Indeed, when developing the filesystem, several shortcomings in ABACUS had to be fixed. Second, the filesystem is a prime example of a distributed application that can benefit from the adaptive placement of its functions, in particular, the cache, directory management and RAID functions. The ABACUSFS is described in detail, since it will be used to drive the evaluation.

The ABACUSFS filesystem can be accessed in two ways. First, applications that include mobile objects can directly append per-file mobile object graphs onto their application object graphs for each file opened. The run-time system will convert method invocations from application objects to filesystem objects into local or remote RPC calls, as appropriate.

Second, the ABACUSFS filesystem can be mounted as a standard filesystem, via VFS-layer redirection to a user-level process implementing the extended (ABACUSFS) filesystem. Unmodified applications using the standard system calls can thus interact with the ABACUSFS filesystem via standard POSIX system calls. The filesystem process's VFS interaction code will interface with per-file/directory object graphs via a console object (in the first approach, the operating system is bypassed.) Although it does not allow legacy application objects to be migrated, this second mechanism does allow legacy applications to benefit from the filesystem objects adaptively migrating beneath them. Figure 5.9 represents a sketch of the ABACUSFS prototype distributed filesystem.



(a) Typical object placement (b) Example filesystem objects

Figure 5.9: The prototype filesystem ABACUSFS and how it is decomposed into migratable component objects. The figure shows only the filesystem and no user applications. The console object for the filesystem represents the code that is a part of the operating system and that interfaces system calls to the VFS layer. This code is not an object and is not a mobile object; it currently always executes on the node on which the parent operating system executes. The FileSys object implements VFS layer functions to interface to the operating system as well as functions allowing applications to link in the ABACUS filesystem directly into user-space. When an application is written explicitly for ABACUS, it can bypass the the operating system and directly access the FileSys object, which provides a system-call like interface for file access that can be directly invoked by the application. In this case, the FileSys object is a mobile object.

The filesystem is decomposed into component objects. Some objects are static and are always bound to the storage servers. These include the NASD objects, the isolation and atomicity objects and the cache coherence objects. The other objects, RAID, caching, and directory management are migratable and can be located at any node in the network.

Per-file object stacks

The ABACUSFS filesystem provides coherent file and directory abstractions atop a flat object space exported by base storage servers. The filesystem function is decomposed into different objects performing different services such as: caching, RAID, cache coherence, and NASD basic storage. Often, the same file is associated with more than one service or function. For instance, a file may be cacheable, striped and reliable. Filesystems can be composed by constructing objects from other

objects, adding layers of service, as demonstrated by the stackable [Heidemann and Popek, 1994] and composable filesystems [Krieger and Stumm, 1997] work, and by the Spring object-oriented operating system [Khalidi and Nelson, 1993]. At each layer, a new object is constructed from one or more object from lower layers. For example, the constructor of a cacheable object requires a backing object to read from and write back to. The backing object can be a NASD object, a mirrored object or even another cacheable object. An object may have references to more than one object from a lower layer. For example, to construct a mirrored object, two base objects are used.

In order to enable files to have different performance and reliability attributes, the prototype filesystem enables each file to be associated with a potentially different stack of layers of service. This flexibility is useful because the creators of specific files and directories may mandate different reliability and performance requirements: infrequently written and frequently written files, important and temporary files may require different striping and reliability guarantees [Wilkes et al., 1996]. For example, using RAID storage makes writes more expensive. It is usually a good tradeoff to use non-redundant storage for temporary files used by compilers and other utilities, because performance is more important than reliability. This section discusses object stacking, deferring the discussion of the details of each object to the following sections.

Each file or directory in ABACUS is associated with an *inode* which contains the file or directory's metadata. These inodes are initialized when the file is created and are used to refer to the file by the objects that make up the filesystem. When a file or directory is created, it is associated with a stack of types. This stack represents a template, which is used to instantiate the requisite objects when the file is accessed. When a file is created, the constructors for the objects are invoked to allocate and initialize the storage and metadata needed to create the file. Precisely, the constructor of the topmost type is invoked passing it the template. This constructor invokes lower-level constructors to allocate objects that are lower in the stack. For example, a default file is associated with a stack consisting of a cache, a RAID, and a NASD layer. The cache object keeps an index of a particular object's blocks in the shared cache. The RAID level 5 object stripes and maintains parity for individual files across sets of storage servers. The constructor of a cache object expects a backing object. It creates a backing object of the type specified in the layer below it in the stack descriptor. In this case, a RAID level 5 object is created, which in turn creates possibly several NASD objects.

Once a file is created, it can be be accessed by opening it and issuing read and write calls. When a file is opened, an object of the type of the topmost layer is *instantiated* [2]. As part of this

[2] Instantiation refers to the creation of a run-time C++ object of the proper type. Creation, as used in the previous paragraph, however, refers to the actions taken when a file is created, and which often require the allocation and initialization

instantiation, a reference to the inode for the file is passed as an argument. The file's inode stores persistent metadata on behalf of each layer of the stack which describes information required to initialize the objects in a file's stack. For example, if a file is bound to a RAID layer, the RAID level 5 object needs to know how the file is striped, i.e. what base NASD objects it is mapped onto, or whether storage must be allocated for the new file. This information is maintained in the inode, which contains metadata on behalf of each layer. The RAID level 5 object inspects the inode's section for the RAID layer to determine the identity of the lower-level NASD objects that the file is mapped onto. This information is written into the inode by the RAID level 5 object constructor when it allocates storage for the file during file creation.

Access to a file always starts at the top-level layer. A file is usually associated with one object of the top layer's type. That top-level object hold references to other objects, and propagates the access down after performing some processing. For example, a file is usually associated with a cache object, which may hold a reference to a backing RAID object, which in turn may hold references to multiple base NASD objects. During an open, the top-level object is instantiated, and in turn instantiates all the lower level objects in the object graph.

5.5.2 NASD object service

The design of the prototype filesystem must accommodate the underlying NASD architecture. In a NASD cluster, storage severs export a flat-file like interface, as shown in Table 3.1. A NASD object manager on each storage server manages the persistent NASD object space. It provides read/write accesses to arbitrary ranges within a NASD object. In particular, it implements the following methods: `CreateObj()`, `RemoveObj()`, `WriteObj()`, and `ReadObj()`. Further, each manager that is always resident on a storage device can access a per-manager well-known object via the `GetWellKnownObj()` method. Object managers use the well-known object to store a reference to root objects, write-ahead logs or other objects that are needed at startup.

The details of the important part of the NASD interface are shown in Table 3.1. The table shows the input parameters, results parameters, and return values for each method. `NasdId` is the type of the identifiers that are associated with a persistent NASD object.

of persistent state backing the run-time object.

5.6 Evaluation of filesystem adaptation

The experiments of this chapter show that performance depends on the appropriate placement of function. They include several benchmarks where ABACUS discovers the best placement automatically at run-time even in cases where it is hard or impossible to anticipate at design-time. This includes scenarios in which the best location for a function is based on hardware characteristics, application run-time parameters, application data access patterns, and inter-application contention over shared data. This also includes scenarios that stress adaptation under dynamic conditions: phases of application behavior and contention by multiple applications.

This section contains an evaluation of the benefits of filesystem adaptation over ABACUS, while Section 5.8 reports on further evaluation of the dynamic behavior of the ABACUS run-time system. The evaluation approach considers several filesystem objects, and shows through synthetic workloads that the best object placement (client or server) varies with workload and system parameters. In each case, the performance of the workload under a fixed allocation of function is compared to performance under ABACUS. The experiments show whether ABACUS can discover the best placement when the object starts on the wrong node and the overhead it induces.

5.6.1 Evaluation environment

The evaluation environment used consists of eight clients and four storage servers. All twelve nodes are standard PCs running RedHat Linux 5.2 and are equipped with 300 MHz Pentium II processors and 128 MB of main memory. Each storage server contains a single Maxtor 84320D4 IDE disk drive (4 GB, 10 ms average seek, 5200 RPM, up to 14 MB/s media transfer rate). There is no heterogeneity in the hardware resources across the storage servers or clients. Such heterogeneity will be simulated by creating a base workload that consume resources at certain nodes.

The network, on the other hand, is heterogeneous. Particularly, the evaluation used two networks, a 100 Mb/s Ethernet, which is referred to as the *SAN* (storage-area network) and a shared 10 Mb/s segment, which is referred to as the *LAN* (local-area network). All four storage servers are directly connected to the SAN, whereas four of the eight clients are connected to the SAN (called SAN clients), and the other four clients reside on the LAN (the LAN clients). The LAN is bridged to the SAN via a 10 Mb/s link. Figure 5.10 graphically sketches of the evaluation environment. While these networks are of low performance by today's standards, their relative speeds are similar to those seen in emerging high-performance SAN and LAN environments (Gb/s in the SAN and 100 Mb/s in the LAN).

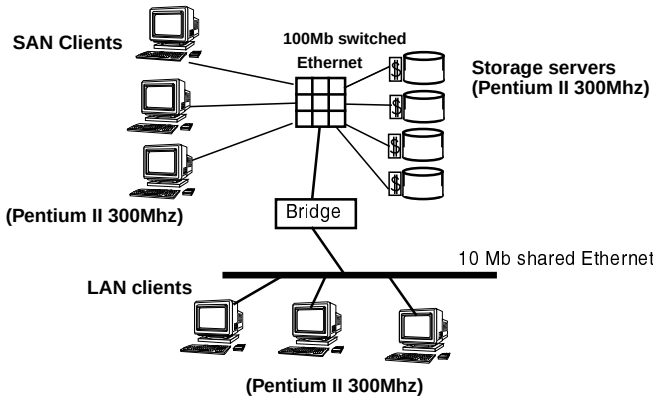


Figure 5.10: Evaluation environment. The system consists of storage servers containing Pentium II 300 Mhz processors with 128 MB of RAM and a single Maxtor IDE disk drive. Each disk has a capacity of 4GB, an average seek time of 10 ms and a sustained transfer rate of up to 14 MB/s. The clients have the same processor and memory capacity. The network is heterogeneous. It consists of a switched 100 Mb/s Ethernet bridged to a 10 Mb/s shared Ethernet segment.

5.6.2 File caching

Caching is an important function of a distributed filesystem. There are two kinds of caches, client-side and server-side caches. Client-side caches usually yield dramatic reduction in storage access latencies because they avoid slow client networks, increase the total amount of memory available for caching relative to server-side caching only, and reduce the load on the server by not needing to forward the read to the server at all. A server-side cache can better capture reuse characteristics across clients, simplifies and avoids the cost of maintaining client cache consistency, and also effectively lowers disk latencies especially with a fast network.

The ABACUSFS prototype filesystem contains a cache object that starts on the client by default, and is moved to the server if higher performance mandates this migration. While client-side caching is usually effective, it can sometimes cause opposite performance effects even with a slow network. Consider an application that inserts small records into files stored on a storage server. These inserts require a read of the much larger enclosing block from the server (an *installation read*), the insertions, and then a write back of the enclosing block. Even when the original block is cached, writing a small record in a block requires transferring the entire contents of the enclosing block to the server. Under such a workload, it is more advantageous to send a description of the update to the server rather than update the block locally at the client [O'Toole and Shrira, 1994].

Caching in ABACUSFS is provided by a cache object manager. The cache manager on a node

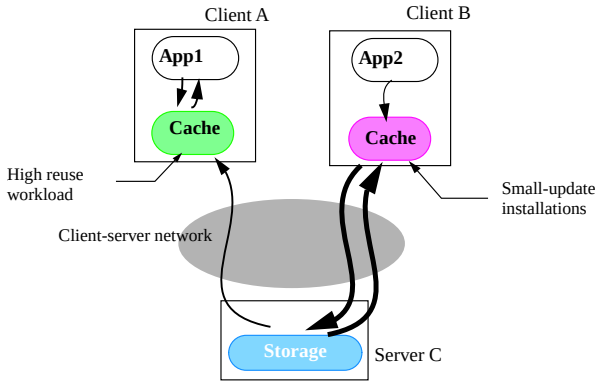


Figure 5.11: Function placement for objects in a file cache. Client A has a high reuse workload, such that the amount of data moved between the application and the cache is higher than that moved between the storage server and the cache. In this case, client caching is effective. In the case of client B, the workload includes small update installations causing the cache to fetch a much larger block from the server, install the small update, then write the larger block back to the server. Assuming no reuse, the amount of data moved between the server and client B's cache is larger than that moved between its cache and the application (App 2), favoring server-side cache placement.

manages cache objects for all the files accessed on that node. In addition to the `ReadObj()` and `WriteObj()` methods, the cache provides methods for cache coherence. In particular, the `BreakCallback()` method is invoked by a server to notify the cache that a file has been updated and the cached version is no longer valid.

Cache Placement: Adapting to data access patterns

Client-side caches in distributed file and database systems often yield dramatic reduction in storage access latencies because they avoid slow client networks, increase the total amount of memory available for caching, and reduce the load on the server. However, enabling client-side caching can yield the opposite effect under certain access patterns. This section shows experimentally that ABACUS can appropriately migrate the per-file cache object in response to data access patterns via generic monitoring without knowledge of object semantics.

Experiment. The following experiment was carried out to evaluate the impact of adaptive cache placement on application performance and to test the ability of ABACUS to discover the best placement for the cache under different application access patterns. Using the evaluation environment described above, the history window of ABACUS, H, was set to one second, and the threshold benefit was set to 30%. In the first benchmark, *table insert*, the application inserts 1,500 128 byte

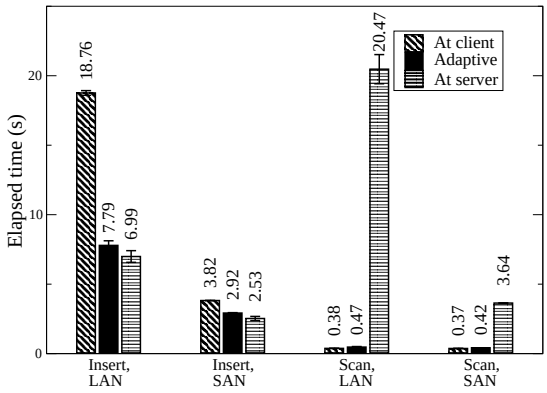


Figure 5.12: This figure shows that client-side caching is essential for workloads exhibiting reuse (Scan), but causes pathological performance when inserting small records (Insert). ABACUS automatically enables and disables client caching in ABACUSFS by placing the cache object at the client or at the server.

records into a 192 KB file. An insert writes a 128 byte record to a random location in the file. In the second benchmark, *table scan*, the application reads the 1,500 records back, again in random order. The cache, which uses a block size of 8 KB, is large enough for the working set of the application. Before recording numbers, the experiment was run once to warm the cache.

Results. As shown in Figure 5.12, fixing the location of the cache at the server for the insert benchmark is 2.7X faster than at a client on the LAN, and 1.5X faster than at a client on the SAN. ABACUS comes within 10% of the better for the LAN case, and within 15% for the SAN case. The difference is due to the relative length of the experiments, causing the cache to migrate relatively late in the SAN case (which runs for only a few multiples of the observation window). The table scan benchmark highlights the benefit of client-side caching when the application workload exhibits reuse. In this case, ABACUS leaves the ABACUSFS cache at the client, cutting execution time over fixing the cache at the server by over 40X and 8X for the LAN and SAN tests respectively.

Cache coherence

The cache coherence object manager is responsible for ensuring data blocks of a lower layer's stored object are cached coherently in each of the multiple client caches. Files are mapped onto one or more underlying objects. When file data is cached on a client, data from these underlying objects is cached. A cache coherence object is associated with each underlying object. The cache coherence object is anchored to the storage server which hosts the underlying object.

The cache coherence object performs its function by intercepting `ReadObj()` requests and

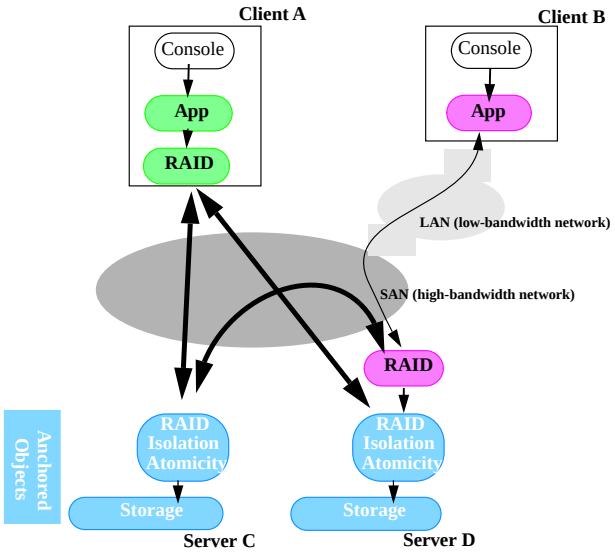


Figure 5.13: This figure illustrates how different ABACUS clients may place function associated with the same file in different locations. Clients A and B access the same file bound to a simple object stack. The file is partitioned across devices C and D. The console object initiates requests that trickles down to an application object (App), a RAID object, and finally a storage object.

installing a callback for the read block [Howard et al., 1988]. The object also intercepts `Write-Obj()` requests and breaks any matching installed callbacks. Cache coherence functions on a storage server are implemented by a cache coherence object manager.

5.6.3 Striping and RAID

When the network consists of switched high bandwidth links and files are completely stored on a single storage server, the storage access bandwidth can be severely limited by the bandwidth of the server machine. Striping data across multiple storage servers eliminates the single server bottleneck from the data transfer path, enabling higher bandwidth to a single client, as well as substantially larger aggregate bandwidth to multiple clients. Several filesystems were proposed to exploit the potential of network-striping. Examples include Zebra [Hartman and Ousterhout, 1993], Swift [Long et al., 1994], and the Cheops system of Chapter 3.

Large collections of storage also commonly employ redundancy codes such RAID levels 1 through 5 transparently to applications, so that simple and common storage server failures or outages can be tolerated without invoking expensive higher-level failure and disaster recovery mechanisms. The prototype filesystem implements striping and RAID across storage servers, through the RAID

class. Figure 5.13 shows a typical stack that includes a RAID object. RAID objects can be configured to initially start on the client or on the server. The choice depends on the network bandwidth and the trustworthiness of the client. The RAID object is layered atop low level storage objects. Underlying storage objects are accessible on the storage servers who may act independently of each other. Each RAID object is invoked by the objects higher in its stack to perform reads and writes on behalf of the application.

RAID objects perform exactly four operations, divided into access tasks and management tasks. The access tasks are reads and writes (*hostread* and *hostwrite* operations as described in Chapter 4). These tasks provide semantics essentially identical to reading and writing a base storage object. The management tasks are reconstruction and data migration (*reconstruct* and *migrate* operations respectively). Each high-level task is mapped onto one or more low-level read and write requests to (contiguous) physical blocks on a single storage object (*devread* and *devwrite* described in Chapter 4). Depending on the striping and redundancy policy, and whether a storage device has failed, a *hostread* or *hostwrite* may map onto different base storage transactions (BSTs).

Blocks within a RAID object are mapped onto one or more physical storage objects. RAID access operations (read and write) as well as management operations (reconstruction and migration) invoke one or more basic BSTs. Following the designs of Chapter 3 and 4, the representation of a RAID object is described by a stripe map which specifies how the object is mapped, what redundancy scheme is used, and what BSTs to use to read and write the object. Stripe maps are cached by RAID object managers at a node to allow direct access storage from that node. The RAID layer performs no caching of data or parity blocks, leaving the function of caching to the other objects in the stack, such as the cache object. RAID object managers in ABACUSFS use the timestamp ordering protocol described in Chapter 4 to ensure that parity codes are updated correctly and that migration and reconstruction tasks are correctly synchronized with access tasks. Timestamp checks at the storage servers are performed using a RAID Isolation and Atomicity (RIA) Object. This is implemented as one object manager on each storage server. This manager groups the implementation of all local RIA objects on a given device and implements the timestamp ordering protocol.

RAID Placement: Adapting to system resource distribution

The proper placement of the RAID object largely depends on the performance of the network connecting the client to the storage servers. Recall that a RAID level 5 small write, as described in

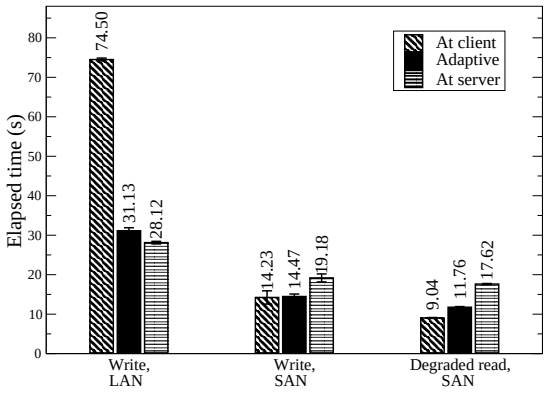


Figure 5.14: This figure shows the results of the RAID benchmarks. Contention for the server's CPU resources make client-based RAID more appropriate, except in the LAN case, where the network is the bottleneck.

Section 2.3, invokes four I/Os, two to pre-read the old data and parity, and two to write the new data and parity. Similarly, when a disk failure occurs, a block read requires reading all the blocks in a stripe and XORing them together to reconstruct failed data. This can result in substantial network traffic between the RAID object and the storage servers.

Two workloads were constructed to evaluate ABACUSFS RAID performance with ABACUS adaptivity. The first consists of two clients writing two separate 4 MB files randomly. Two clients were used to attempt to see if ABACUS makes the proper trade-off between client-side execution of parity computations on the less loaded client processors and between the network efficiency of server-side parity computation (which saves messaging). The stripe size is 5 (4 data + parity) and the stripe unit is 32 KB. The second workload consists of the two clients reading the files back in degraded mode (with one disk marked failed).

Results. As shown in Figure 5.14, executing the RAID object at the server improves RAID small write performance in the LAN case by a factor of 2.6X over executing the object at the host. The performance of the experiment when ABACUS adaptively places the object is within 10% of the fastest. Conversely, in the SAN case, executing the RAID object locally at the client is 1.3X faster because the client is less loaded and able to perform the RAID functionality more quickly. Here, ABACUS comes within 1% of this fastest value. The advantage of client-based RAID is slightly more pronounced in the more CPU-intensive degraded read case, in which the optimal location is almost twice as fast as at the server. Here, ABACUS comes within 30% of the better. In every instance, ABACUS automatically selects the best location for the RAID object.

	LAN	SAN
At client	65.47	4.60
Adaptive	4.33	3.33
At server	3.02	2.83

Table 5.2: Migrating bulk data movement. This table shows the time in seconds taken to copy a 16 MB file from one storage server to another on both our LAN and SAN configurations. The table shows the copy function statically placed at the client, adaptively located by ABACUS, and statically placed at the storage node.

RAID: Copy BST placement

Experiment. One typical operation in managing large storage systems is data reconfiguration, that is, migrating data blocks between devices to re-balance load or to effectively use the capacity of newly added devices. This can be done by a UNIX user with `rdist`, `rcp`, or `tar` if the system does not provide automatic support for load re-balancing. Copy applications are ideal candidates for migration from client to storage nodes, because they often overwhelm the client's cache and move a lot more data than necessary across the network. A migratable version of the copy task, called `abacus_copy`, was implemented on ABACUSFS.

Results. Table 5.2 shows the time taken to copy a 16 MB file from one storage node to another using `abacus_copy`. Running the copy object at the storage node is most beneficial when the client is connected to the low-speed LAN. In this case, ABACUS migrates the object to the storage nodes and achieves within 43% of the optimal case in which the copy object begins at the storage node. This optimal case is over 20X better than the case in which the object executes at the client. When the copy task is started on a SAN client, ABACUS does not initiate migration. The experiment on this fast network runs so quickly that the cost of migration would be comparatively high. Naturally, when moving enough more data, ABACUS will also perform the migration even in the SAN configuration. Further, even more benefit is observed from migrating the copy when the source and destination storages nodes are the same (*i.e.,* only one storage node is involved).

5.6.4 Directory management

The directory object manager is multi-threaded and supports replication of directory data across multiple hosts. The directory module implements a hierarchical namespace as that implemented by UNIX filesystems. It enables directories to be replicated at several nodes providing trusted hosts

with the ability to locally cache and parse directory blocks. It also supports scalable directory management by using an optimistic concurrency control protocol based on timestamp ordering, using timestamps derived from loosely synchronized real-time clocks guided by the algorithms of Chapter 4.

Directory concurrency control

Races can occur between clients concurrently operating on shared directories. As an example, consider a directory update operation such as `MkDir()`, which proceeds by scanning the parent directory for the name to make sure it is not already there, then updating one of the directory blocks to insert the new name and the associated metadata (inode number). Since blocks are cached at each host, two hosts trying to insert two directories in the same parent directory can both scan the locally directory cached block, pick the same free slot to insert the new directory and write the parent directory back to storage, resulting only one directory being inserted.

Since directories can be potentially cached and concurrently accessed by multiple hosts, they are bound to the following stack: directory (implementing directory parsing and update operations such as `CreateFile()`, `RemoveFile()`, *etc*.), and a directory isolation and atomicity object (DIA) object. This discussion will describe directory objects that are not bound to a RAID object (are not mirrored or parity protected). The DIA object ensures that concurrent directory operations are isolated from one another. It uses a write-ahead log to ensure consistency in the event of failures during operations. The DIA object also maintains callbacks so that all cached directory blocks are coherent. The cache coherence object used for data files is not used in this case because combining timestamp checking with coherence allows several performance optimizations without complicating the reasoning about correctness.

The directory object manager provides POSIX-like directory calls, using the shared cache discussed above and the underlying object calls. The DIA object manager provides support for both cache coherence and optimistic concurrency control. The former is provided by interposing on `ReadObj()` and `WriteObj()` calls, installing call-backs on cached blocks during `ReadObj()` calls, and breaking relevant call-backs during `WriteObj()` calls. The latter is provided by timestamping cache blocks [Bernstein and Goodman, 1980] and exporting a special `CommitAction()` method that checks specified readSets and writeSets. The readSet (writeSet) consists of the list of blocks read (written) by the client.

To illustrate how the directory manager interacts with the DIA object manager, let's take a

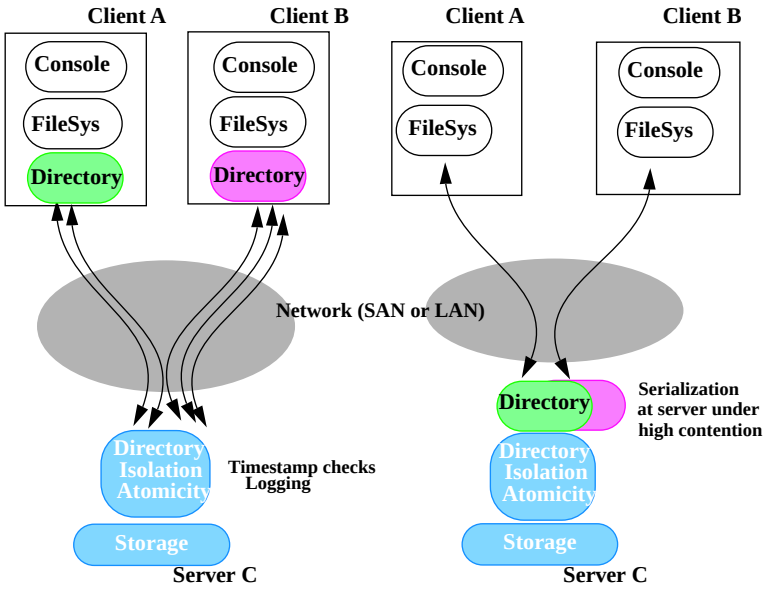


Figure 5.15: Directory management in ABACUSFS. The directory object receives requests to operate on directories, for example to insert a name in a directory, or to list directory contents. While performing directory management at the client is more scalable in general under low contention (left), it can lead to degraded performance in the case of high contention. Under high contention, the distributed concurrency control and cache coherence traffic among clients can induce enough overhead that a centralized server-side implementation becomes favorable (right).

simple concrete example of a directory operation: an `MkDir()`. When the operation starts at the directory manager, a new timestamp `opts` is acquired and an action is initialized. An action is a data structure which includes a *readSet*, a *writeSet* and a timestamp. The *readSet* (respectively *writeSet*) contain a list of names of the blocks read (written) by the action and their timestamps. As the blocks of the parent directory are scanned for the name to be inserted, their identifiers [3] and their timestamps are appended to the *readSet*. Assuming the name does not already exists, it is inserted in a free slot. The block where the name is inserted is added to the *writeSet*. As soon as the operation is ready to complete locally, a `CommitAction()` request is sent down the stack, with the action and the new block contents as arguments. During the commit, the block is locked locally so it is not accessed. The lock is released once the operation completes.

The DIA object manager performs timestamp checks against recently committed updates in a manner very similar to the algorithms of Chapter 4. Precisely, the checks establish that the blocks in the *readSet* are the most recent versions, and that `opts` exceeds the `rts` and `wts` for the blocks

[3] A block is represented in a read set or write set by the parent storage object it belongs to and its offset into that object.

in the read and write sets. Note that because clocks are loosely synchronized, a block's `wts` and `rts` need to be maintained only for a short time window (T), after which they can be discarded and logically replaced with the current time minus T. This may result in rejecting some operations that would otherwise be accepted but it will not result in incorrect operation. If the checks succeed, the new block contents are committed to the log and callbacks are broken to nodes caching that blocks. Otherwise, a rejection is returned to the host, who refreshes its cache and retries the insert from the beginning.

Directory placement: Adapting to contention

The function placement problem involves a fundamental choice between the scalability of client-side execution where CPU and memory resources are more abundant and the potential network-efficiency of server-side execution. In particular, it involves a choice between replicating function across clients (which comes at the cost of higher synchronization overhead but which allows client resources to be exploited) and centralizing it on a few nodes. The placement of directory management function exemplifies this trade-off and demonstrates that transparent monitoring can effectively make the proper trade-off in each case.

Filesystem functionality such as caching or pathname lookup, for example, is often distributed to improve scalability [Howard et al., 1988], as are functions of other large applications. Synchronization among parallel clients is an overhead which varies depending on the amount of inter-client contention over shared data. Consider the example of a file cache which is distributed across clients. Files are stored on the server and cached on clients. Furthermore, assume that files are kept coherent by the server. When a client reads a file, the server records the name of the file and the client that read it, promising the client to notify it when the file is written. When another client writes a file, the write is immediately propagated to the server, which notifies the clients that have the file cached that their version is now stale (via a "callback") The clients then access the server to fetch the new copy. This is how coherence is achieved in AFS [Howard et al., 1988] for example.

Now consider the case where clients are accessing independent files, each write is propagated to the server and does not generate any "callbacks" because the file being written is cached only at the client that is writing it. In this case, except for the initial invalidation message, there is no further coherence-induced communication from the server to the clients. Thus, placing the cache at the client does not induce any more synchronization overhead than if it was placed at the server.

On the other hand, consider the case where the clients are all actively accessing, reading and/or

writing, the same file. In this case, each write by a client results in "callbacks" to the active clients, who in turn contact the server to fetch the recently written version of the block. Soon after that, the same client or another client writes the block again, causing a callback to be propagated by the server to the rest of the clients. The clients then re-fetch the new copy of the block that was updated. Under such a workload, placing the cache at the client causes excessive synchronization overhead in the form of coherence traffic (callbacks) and data re-fetching. To sum up, the placement of function under certain workloads can have a dramatic impact on the amount of synchronization overhead, and consequently the amount of network messaging. The effect of this overhead must be weighed against the benefit of wider scale replication (parallelization) of function.

Experiment. To validate this hypothesis, a few experiments were conducted. A workload that performs directory inserts in a shared namespace was chosen as the contention benchmark. This benchmark is more complicated than in the distributed file caching case and therefore more challenging to ABACUS. Directories in ABACUS present a hierarchical namespace like all UNIX filesystems and are implemented using the object graph shown in Figure 5.15. When clients access disjoint parts of the directory namespace (*i.e.*, there are no concurrent conflicting accesses), the optimistic scheme in which concurrency control checks are performed after the fact by the isolation (DIA) object works well. Each directory object at a client maintains a cache of the directories accessed frequently by that client, making directory reads fast. Moreover, directory updates are cheap because no metadata pre-reads are required, and no lock messaging is performed. Further, offloading from the server the bulk of the work results in better scalability and frees storage devices to execute demanding workloads from competing clients. When contention is high, however, the number of retries and cache invalidations seen by the directory object increases, potentially causing several round-trip latencies per operation. When contention increases, the directory object should migrate to the storage device. This would serialize client updates through one object, thereby eliminating retries.

Two benchmarks were constructed to evaluate how ABACUS responds to different levels of directory contention. The first is a high contention workload, where four clients insert 200 files each in a shared directory. The second is a low contention workload where four clients insert 200 files each in private (unique) directories.

Results. As shown in Figure 5.16, ABACUS cuts execution time for the high contention workload by migrating the directory object to the server. In the LAN case, ABACUS comes within 10% of the best, which is 8X better than locating the directory object at the host. ABACUS comes within

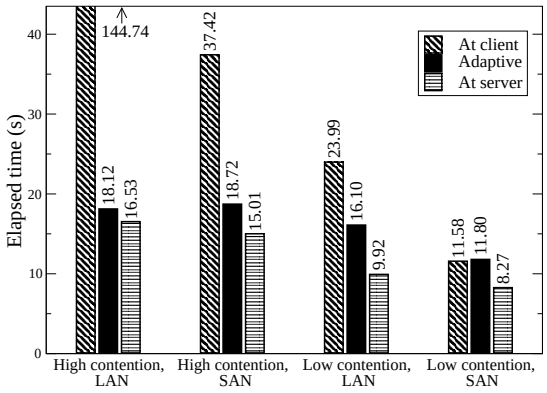


Figure 5.16: This figure shows the time to execute our directory insert benchmark under different levels of directory contention. ABACUS migrates the directory object in all but the fourth case.

	LAN	SAN
At client	125	86
Adaptive	7	27
At server	0	0

Table 5.3: This table shows the number of retries (averaged over 5 runs) incurred by the optimistic concurrency control scheme when inserting entries into a highly contended directory. Results are shown for the case where the directory object is statically placed at the client, adaptively located by ABACUS, and statically placed at the storage server.

25% of the best for the high contention, SAN case (which is 2.5X better than the worst case). Note the retry results summarized in Table 5.3. There are lower retries under ABACUS for the high contention, LAN case than for the high contention, SAN configuration. In both cases, ABACUS observed relatively high traffic between the directory object and storage. ABACUS estimates that moving it closer to the isolation object would make retries cheaper (local to the storage server). It adapts more quickly in the LAN case because the estimated benefit is greater. ABACUS had to observe far more retries and revalidation traffic on the SAN case before deciding to migrate the object.

Under low contention, ABACUS makes different decisions in the LAN and SAN cases, migrating the directory object to the server in the former and not migrating it in the latter. For these tests, he benchmark was started from a cold cache, causing many installation reads. Hence, in the low contention, LAN case, ABACUS estimates that migrating the directory object to the storage server, avoiding the network, is worth it. However, in the SAN case, the network is fast enough that the

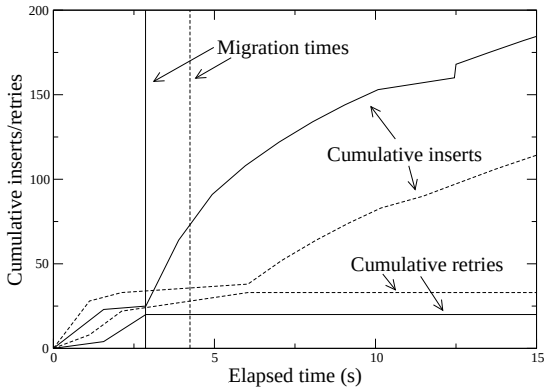


Figure 5.17: This figure shows the cumulative inserts and retries of two clients operating on a highly contended directory over the SAN. Client 1's curves are solid, while client 2's are dotted.

ABACUS cost-benefit model estimates the installation read network cost to be limited. Indeed, the results show that the static client and storage server configurations for the SAN case differ by less than 30%, the threshold benefit for triggering migration.

Note that clients need not agree to migrate the directory objects to the storage device at the same time. They can decide independently, based on their migration benefit estimation. Correctness is ensured even if only *some* of the clients decide to move the object to the storage device because all operations are verified to have occurred in timestamp order by the isolation object, which is always present on the storage servers. Figure 5.17 shows a time-line of two clients from the high contention, SAN benchmark. The graph shows the cumulative number of inserted files and the cumulative number of retries for two clients. One client experiences a sharp increase in retries and its object is moved to the server first. The second happens to suffer from a relatively low, but steady retry rate, which triggers its adaptation a little later. The first client experiences a sharp increase in the rate of progress soon after it migrates. The second experiences a substantial, but *lower*, increase in its rate of progress after it migrates, which is expected as storage server load increases.

5.7 Supporting user applications

While the ABACUS system focussed on adaptive function placement in distributed filesystems, the approach that it embodies can in fact be readily generalized to support adaptive function placement for all stream-processing kinds of applications. This section describes some example applications that can benefit from adaptive function placement over ABACUS. In particular, it describes the case of a data filtering application which was ported to ABACUS. This section reports on its performance

and implementation on the ABACUS run-time. This discussion clarifies how function placement can be generalized beyond the ABACUS filesystem to user-level applications, assuming the underlying filesystem is itself mobile.

5.7.1 Example application scenarios

There is a growing trend for businesses to access a variety of services via the web. There is a trend even to host traditional desktop applications on remote servers, at an "application service provider" (ASP). This moves the tasks of upgrade, maintenance and storage management to the ASP, reducing the costs to the client. On the other hand, streaming of real-time multimedia content customized to the user's interests is becoming commonplace. Such applications benefit from adaptive placement of their components due to the wide variability in client-server bandwidths on the internet and due to the great disparity in client resources (PDAs to resourceful workstations) and in server load. As investments in the Internet's infrastructure continue, bandwidth to the server improves, and ASPs and content-rich applications become increasingly attractive and widespread. However, heterogeneity will remain a major challenge for application performance management.

Distributed web applications

ASPs can provide more robust performance across workloads and network topologies by partitioning function between the client and the server dynamically at run-time. Such a dynamic function placement can still maintain the ease of management of server-side software maintenance, while opportunistically shipping function to the client when possible.

Customized multimedia reports

An increasing number of applications on the internet today compile multiple multimedia streams of information, and customize these streams to the needs of end users, their language, interests and background. Such applications aggregate content from different sites, merge and filter this information together and deliver it to the end client. The optimal place to execute the different functions on the data set or stream depends on the kind of client used, e.g. a PDA or high-end workstation, the current load on the server, and on the performance of the network between the client and the server. Dynamic partitioning of function based on black box monitoring can simplify the configuration of such applications over wide area networks and heterogeneous client and server pairs.

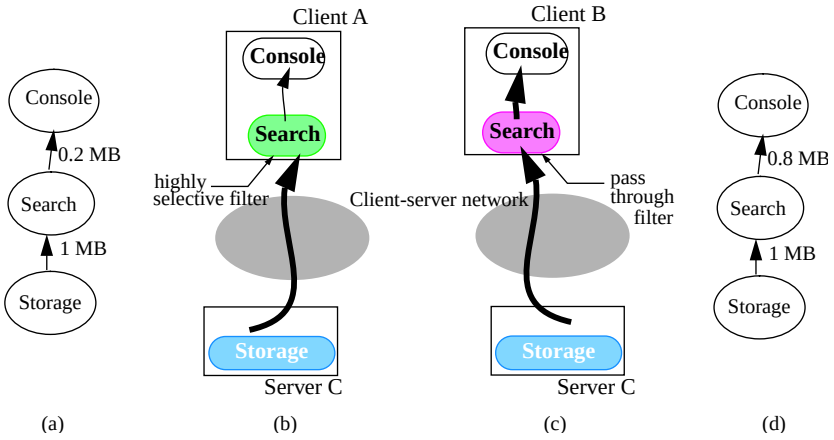


Figure 5.18: The alternative placements of a filter object. Thicker arrows denote more larger data transfers. If the filter is highly selective, returning a small portion of the data it reads, as in the case of client A, then it can potentially benefit from executing at the server. This reduces the amount of data transferred over the network. If the filter has low selectivity, as in the case of client B, passing through most of the data that it reads, then it would not benefit much from server-side execution.

5.7.2 Case study: Filtering

Consider the example of a filtering application running on ABACUS. The application consists of a synthetic benchmark that simulates searching. It filters an input data set, returning a percentage of the input data and discarding the rest. This percentage can be specified to the program as an argument. The application program is composed of a console part (or a "main" program) that performs initialization and input/out, and a filter object.

The filter object accesses the filesystem to read the input data set. In this simple example, the filesystem is accessed via remote procedure calls to a "storage object" anchored to the server. For simplicity, the file accessed by the filter was not bound to a realistic ABACUSFS stack (containing caching and striping). This makes the experiment simple and allows us to focus on the placement of the filter object. Data is not cached on the client side. The filter exports one important method, namely `FilterObject()`, which takes two arguments, the size of the block to filter. The percentage of the data to filter out is specified to the filter when it is first instantiated. The filter object records its position in the input file. When it receives a `FilterObject` invocation, it processes a block of data from its current position, and returns data to the console in a result buffer.

The *selectivity* of a filter is defined as the ratio of the data discarded to the total amount of data read from the input file. Thus, a filter that throws away 80% of the input data, and returns a fifth of

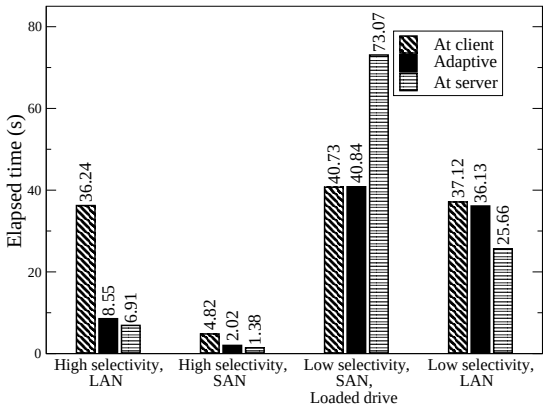


Figure 5.19: The performance of the filter benchmark is shown in this figure. Executing the filter at the storage server is advantageous in all but the third configuration, in which the filter is computationally expensive and runs faster on the more resource-rich client.

it, has a selectivity of 0.8. Filters with high selectivity fit the intuition of being "highly selective", choosing only a few from a large set. Precisely: $\text{selectivity} = (1 - \text{output}/\text{input})$. Applications can exhibit drastically different behavior based on run-time parameters. This section shows that the selectivity of the filter (which depends on the data set and the pattern being searched — a program input) determines the appropriate location for the filter to run. For example, there's a drastic difference between `grep kernel Bible.txt` and `grep kernel LinuxBible.txt`.

Experiment. As data sets in large-scale businesses continue to grow, an increasingly important user application is high-performance search, or data filtering. Filtering is generally a highly selective operation, consuming a large amount of data and producing a smaller fraction. A synthetic filter object was constructed that returns a configurable percentage of the input data to the object above it. Highly selective filters represent ideal candidate for execution close to the data, so long as computation resources are available.

In this experiment, both the filter's selectivity and CPU consumption were varied from low to high. A filter labeled low selectivity outputs 80% of the data that it reads, while a filter with high selectivity outputs only 20% of its input data. A filter with low CPU consumption does the minimal amount of work to achieve this function, while a filter with high CPU consumption simulates traversing large data structures (e.g., the finite state machines of a text search program like `grep`).

Results. The filtering application starts executing with the console invoking the method `FilterObject()`, exported by the filter object. As the application executes, data is transferred from the storage object (the storage server) to the filter (the client node), and from the filter to the con-

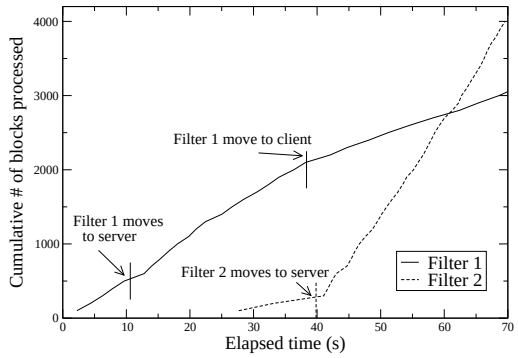


Figure 5.20: This figure plots the cumulative number of blocks searched by two filters versus elapsed time. ABACUS's competition resolving algorithm successfully chooses the more selective Filter 2 over the Filter 1 for execution at the storage server.

sole. The ABACUS run-time system quickly accumulates a history of the amount of data moved between objects by recording the amount of data moved in and out of an object. These statistics are updated on procedure return from each object. Figure 5.18 (a) and (c) illustrates the data flow graphs constructed by ABACUS at run-time in the case of two filters with different selectivities.

Figure 5.19 shows the elapsed time to read and filter a 16 MB file in a number of configurations. In the first set of numbers, ABACUS migrates the filter from client to storage server, coming within 25% of the best case, which is over 5X better than filtering at the client. Similarly, ABACUS migrates the filter in the second set. While achieving better performance than statically locating the filter at the client, ABACUS reaches only within 50% of the best because the time required for ABACUS to migrate the object is a bigger fraction of total runtime. In the third set, a computationally expensive filter was started. We simulate a loaded or slower storage server by making the filter twice as expensive to run on the storage server. Here, the filter executes 1.8X faster on the client. ABACUS correctly detects this case and keeps the filter on the client. Finally, in the fourth set of numbers, the value of moving is too low for ABACUS to deem it worthy of migration.

5.8 Dynamic evaluation of ABACUS

The previous section demonstrated the benefits of adaptive placement and showed through several microbenchmarks that ABACUS can discover the best placement automatically under relative static

workload and network conditions. This section evaluates the ability of ABACUS to adapt under more dynamically varying conditions.

5.8.1 Adapting to competition over resources

Shared storage server resources are rarely dedicated to serving one workload. An additional complexity addressed by ABACUS is provisioning storage server resources between competing clients. Toward reducing global application execution time, ABACUS resolves competition among objects that would execute more quickly at the server by favoring those objects that would derive a greater benefit from doing so.

Experiment. In this experiment, two filter objects are started on two 32 MB files on our LAN. The filters have different selectivities, and hence derive different benefits from executing at the storage server. In detail, Filter 1 produces 60% of the data that it consumes, while Filter 2, being the more selective filter, outputs only 30% of the data it consumes. The storage server's memory resources are restricted so that it can only support one filter at a time.

Results. Figure 5.20 shows the cumulative progress of the filters over their execution, and the migration decisions made by ABACUS. The less selective Filter 1 was started first. ABACUS shortly migrated it to the storage server. Soon after, the more selective Filter 2 was started. Shortly after the second filter started, ABACUS migrated the highly selective Filter 2 to the server, kicking back the other to its original node. The slopes of the curves show that the filter currently on the server runs faster than when not, but that Filter 2 derives more benefit since it is more selective. Filters are migrated to the server after a noticeable delay because in our current implementation, clients do not periodically update the server with resource statistics.

5.8.2 Adapting to changes in the workload

Applications rarely exhibit the same behavior or consume resources at the same rate throughout their lifetimes. Instead, an application may change phases at a number of points during its execution in response to input from a user or a file or as a result of algorithmic properties. Such multiphasic applications make a particularly compelling case for the function relocation that ABACUS provides.

Experiment. Let's now revisit our file caching example but make it multiphasic this time. This cache benchmark does an insert phase, followed by a scanning phase, then an inserting phase, and finally another scan phase. The goal is to determine whether the benefit estimates at the server will eject an application that changed its behavior after being moved to the server. Further, we wish to

	Insert	Scan	Insert	Scan	Total
At client	26.03	0.41	28.33	0.39	55.16
Adaptive	11.69	7.22	12.15	3.46	34.52
At server	7.76	29.20	7.74	26.03	70.73
MIN	7.76	0.41	7.74	0.39	16.30

Table 5.4: This table shows the performance of a multiphasic application in the static placement cases and under ABACUS. The application goes through an insert phase, followed by a scan phase, back to an insert phase, and concludes with a final scan phase. The table shows the completion time in seconds of each phase when the application is fixed to the server for its entire lifetime (all phases), when it is fixed to the client, and when it executes under ABACUS.

see whether ABACUS recovers from bad history quickly enough to achieve adaptation that is useful to an application that exhibits multiple contrasting phases.

Results. Table 5.4 shows that ABACUS migrates the cache to the appropriate location based on the behavior of the application over time. First, ABACUS migrates the cache to the server for the insert phase. Then, ABACUS ejects the cache object from the server server when the server detects that the cache is being reused by the client. Both static choices lead to bad performance with these alternating phases. Consequently, ABACUS outperforms both static cases — by 1.6X compared to fixing function at the client, and by 2X compared to fixing function at the server. The "MIN" row refers to the minimum execution time picked alternatively from the client and server cases. Note that ABACUS is approximately twice as slow as MIN, if it were achieved. This is to be expected, as this extreme scenario changes phases fairly rapidly. Figure 5.21 represents a sketch of the timeline of the caching application. The application changes phases at 5, 10, and 15 seconds.

5.8.3 System overhead

ABACUS induces direct overhead on the system in two ways. First, it allocates space to store the application and filesystem object graphs and the associated statistics. Second, it consumes CPU cycles to crunch these statistics and decide on the next best placement.

In a typical open file session, when the file is bound to three to five layers of mobile objects, ABACUS requires 20 KB to store the graph and the statistics for that open file session. A good fraction of this overhead can probably be optimized away through a more careful implementation. Furthermore, ABACUS can limit the amount of space it consumes by carefully monitoring only a

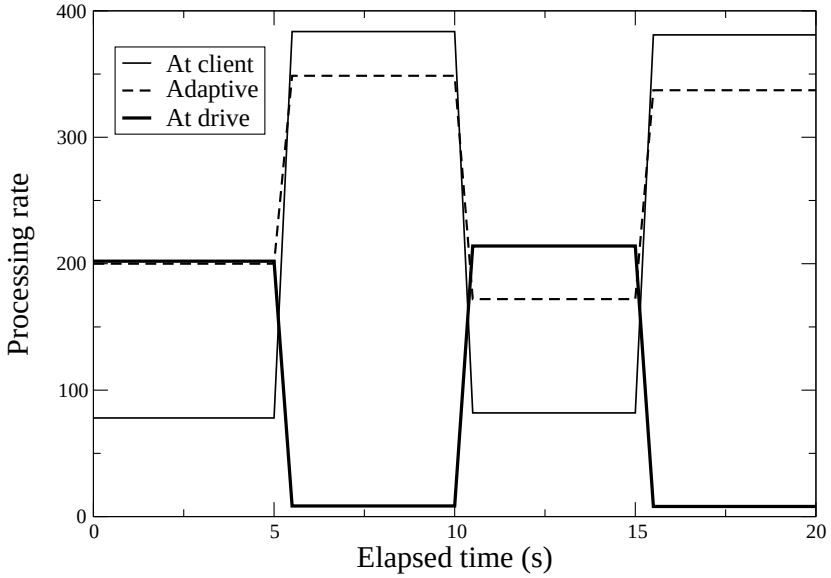


Figure 5.21: This figure plots the processing rate in number of records per second for three configurations. In the first configuration, the cache is anchored to the client, in the second, it is anchored to the server, and in the third it is allowed to migrate under ABACUS. Because progress is measured at discrete time intervals as the cumulative number of records processed, this graph can not be used to infer the exact times at which ABACUS performed a migration from one node to the other.

subset of the open file sessions, those that move a lot of data, for example. For the other sessions, the system can simply maintain summary information such as the total amount of data read from the base storage objects. This summary information is necessary to discover open file sessions that become data-intensive and promote them into a state where they are fully monitored.

The ABACUS run-time system also consumes CPU cycles when the resource manager analyzes the collected graphs to find out if a better placement exists. In the above experiments, the ABACUS resource manager was configured to wake up once every 200 milliseconds and inspect the graphs. The amount of overhead can be configured by limiting the frequency of inspections. The observable overhead while executing applications in the above experiments was mostly within 10%. At worst, it was as large as 25% in the case of short-lived programs for which ABACUS-related initializations and statistics collections were not amortized over a long enough window of execution.

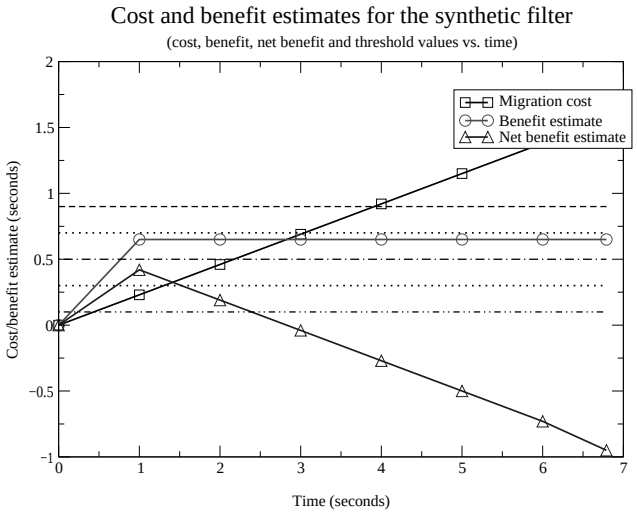


Figure 5.22: Cost and benefit estimates versus time for the synthetic filter application.

5.8.4 Threshold benefit and history size

This section attempts to gain some insight into the dynamics of the ABACUS cost-benefit estimations. Consider the example of a highly selective filter application processing a 4 MB file, and returning 20% of the data it reads. The next experiment starts the filter at the client and collects and logs the cost and benefit estimates computed by ABACUS. ABACUS was directed not to invoke any migrations although it continued to compute the required estimates. The client-server 10 Mb Ethernet network was measured to deliver an approximate end-to-end application read bandwidth of 0.5 MB/s. Since the filtering application's execution time was dominated by server to client network transfers, filtering a 4 MB file on the client required approximately 8 seconds. Performing the filtering on the server would have required approximately only 2 seconds (only 1 MB would be transferred to the client). Thus, the benefit of server-side execution over client-side execution for the entire duration of the application can be approximated as 6 seconds, or .75 seconds per each second of execution.

Figure 5.22 shows the estimates of cost and benefit computed by the ABACUS run-time system versus time as the application executed on the client. Notice that the benefit after an initial increase flattened at about 0.65 seconds per second of execution. ABACUS approximated the benefit of server-side execution to be a reduction of execution time by .65 seconds over the observation history window (of one second). This number corresponds to the value of ΔT_{app}, computed by taking the difference of Equations 5.4 and 5.3. ABACUS observed that the output of the filter was only 20% of

the size of the data currently being communicated over the network. Using its estimates of network bandwidth, ABACUS computed the benefit to be $2/3$ of the history window. This benefit value is relatively close to the value expected from the analysis of the previous paragraph, or 0.7.

The figure also plots the cost of migration over time. The cost of migration depends on the size of the state that must be transferred from the client to the server. As the filter executed, its state increased and the cost accordingly increased. At some point (around $t = 3$), the cost of migration began to exceed the benefit from migration. This is because ABACUS estimated the benefit assuming that the application will execute only for another H seconds. Longer history windows would have allowed for the benefit to be higher, overcoming the one-time migration cost and resulting in more migrations. Longer history windows make the system slow to react to changes in the application, however.

The figure also shows the net benefit from migration. The net benefit initially increased and then started decreasing as the cost of migration rose and the benefit remained flat. Note also how the threshold benefit values, marked by the horizontal dashed lines in the figure, are key to deciding whether or not a migration occurs. In a set of migration experiments, the threshold value was set to: $0.1, 0.3, 0.5, 0.7$ and 0.9. With threshold values of 0.5 and higher, no migrations occurred. This is explained by the plot of the net benefit estimate in the figure, which does not exceed 0.5, at any time during execution.

5.9 Discussion

This section discusses some specific aspects and limitations of the ABACUS prototype which were not already sufficiently addressed.

5.9.1 Programming model and run-time system

The mere existence of mechanisms to change the placement of components, and the development of an application according a given programming model does not always imply that run-time mobility will improve application performance. The application should still be designed with the goal of better performance through mobility. Just like the use of a modular programming language does not imply a modular application, the availability of mobility mechanisms does not imply performance gains. For instance, the run-time system can be overwhelmed with huge object graphs if the programmer chooses to make every base type a migratable object. In this case, the run-time system

must move large subgraphs to generate any benefit and the overhead of the monitoring and placement algorithms can become excessive. Tools that assist the programmers in properly decomposing applications at the proper granularity would be helpful. For filesystems, the decomposition is relatively straight-forward with each layer in a stackable filesystem being encapsulated in an object. For user applications, it is not as easy.

Providing universal guidelines for the design of applications which can benefit from the adaptive placement of their components is challenging. More work on this question is needed. However, it is clear that developing a tool that assists programmers in understanding resource consumption and data flow through their programs can prove helpful in properly decomposing an application into ABACUS objects.

ABACUS separates resource management from monitoring and from method redirection mechanisms. It is therefore relatively simple to implement a different resource management policy. The ABACUS benefit estimates are based on the artificial assumption that the set of applications executing during the observation window (of length H seconds) will execute exactly for another H seconds. Equivalently, ABACUS assumes that history will repeat itself only for another H seconds-long window. ABACUS discards history information beyond H seconds ago. It also does not attempt to estimate the application remaining execution time from the size of the data set, for example. The algorithms used by ABACUS can be improved by more accurately estimating the remaining application execution time and by using old history information rather than discarding it.

The threshold benefit test employed by ABACUS migration algorithms is important because it dampens oscillations and helps mask short fluctuations in resource availability. A low threshold benefit will make ABACUS chase small benefits that may not materialize. This can be because ABACUS adapted too quickly to a short perturbation in network performance for instance. In general, the threshold benefit should be set such that it does not react to measurement or modeling error. If the threshold benefit exceeds the tolerance allowed for measurement and modeling errors, migration will most often be a good decision.

5.9.2 Security

The ability of distributed applications to adaptively place their components at the server opens the potential for security threats. Mechanisms to protect against these threats are necessary to make application and filesystem reconfiguration possible. The threats that adaptive component placement creates can be organized into four categories:

- Application compromising the server. The application can maliciously compromise the server by modifying its code, state or other resources that the applications should not access. The resources that can be accessed by an application are: memory, filesystem, and the network. The safety of memory accesses can be controlled through Java, address spaces, or interpretation, for example. Rights in the filesystem can be be enforced by ABACUSFS according to its access control policy, as the server would if the application was making requests from its home client. Access to the network is made through the operating system, and so in principle does not pose any threats to the server (except for denial of service which is discussed below).

- Server compromising the application. The server can also be malicious and compromise the application, by changing its code or state. This is a problem in the general case where the server is some untrusted remote node. In the context of this thesis, the server machine is trusted, and it should be since it is storing and serving the data to clients.

  The ABACUS prototype avoids the safety threats associated with migrating malicious or buggy application components to a server by running them in a separate address space. Filesystem components are linked in with the filesystem process running at the server. The code is guaranteed not to be tampered with because it is directly read from the filesystem and not accepted from a client node.

- Applications compromising each other. One application that is remotely executing on the server can access state of other applications and may modify it. The same mechanisms used to protect the server's resources from the application can be used to isolate applications from each other.

- Denying server resources. An application can consume all the server's resources, rendering it useless to other applications. In ABACUS, this does not compromise the availability of the storage server itself since the basic storage service runs at a higher priority then any client-shipped code. However, an application might be designed to convince the run-time system to always select it for server-side execution over other competing applications. This is conceivable in ABACUS because the resource management policy works towards global reduction of execution time across clients and not towards fairness.

  Although any mobile object can migrate to the server and server consume resources, ABACUS can, in principle, restrict to which nodes a mobile object can migrate through the use of underlying storage server access control such as NASD capabilities. NASD storage servers may

accept the migration of mobile objects only if they are authorized by some well-known and trusted entity. This NASD manager entity can hand out unforgeable capabilities to the mobile objects authorizing them to use resources on a given server. If the capability verification fails in any specific migration, migration is refused.

5.10 Related work

There exists a large base of excellent research and practical experiences related to code mobility and and function placement in clusters. The idea of function migration was introduced over two decades ago as a way to balance load across multiple CPUs in a system [Stone and Bokhari, 1978, Bokhari, 1979], and as a way to ensure continuous service availability in the presence of system faults [Rennels, 1980]. This section briefly reviews this related work.

5.10.1 Process migration

Systems such as DEMOS/MP [Powell and Miller, 1983], Sprite [Douglis and Ousterhout, 1991], System V [Theimer et al., 1985] and Condor [Bricker et al., 1991] developed mechanisms to migrate entire processes. Process migration is complex because the entire state of a process, which can be scattered throughout operating system data structures and the process' own address space, must be made accessible to the process on the new node and the semantics of all operations should be the same before, during, and after the migrations. The process state includes the contents of its address space (virtual memory), open files (open file entries and cached blocks), communication channels, and the processor's state.

Process migration can be enacted transparently to the process or can occur through a process-visible checkpoint/restore mechanism. Transparent process migration has been built using a complete kernel-supported migration mechanism, or using only a user-level package. User-level implementations tend to have limited transparency and applicability, because they cannot achieve full transparency. As well, they cannot make all the process' state that is embedded in the operating system on the source node available on the target node after migration.

Transparent kernel-supported migration across *heterogeneous* platforms is even more complicated than between *homogeneous* machines. The contents of the process virtual memory cannot be simply transferred to target node because the two machines may represent programs, numbers or characters differently. Even if an executable version of the code to be migrated is available for both

architectures, accurately reconstructing the contents of the process' address space in the absence of data type information is a complex and error prone task [Chanchio and Sun, 1998]. Most prior work and all the systems described here assume homogeneous clusters.

DEMOS/MP

The DEMOS/MP distributed operating system is one of the earliest operating systems to provide kernel-supported transparent process migration [Powell and Miller, 1983]. The DEMOS operating system uses the abstraction of *links* to implement communication between processes and between a process and the operating system.

This nice abstraction enables the operating system to enact migration elegantly. All communication with a process occurs through links. Thus, adding support for location transparency is focussed on making link behavior independent of the node. A link is attached to a process and not to a node. Thus, after the process migrates, the communication subsystem correctly forwards the messages to the new process. Because all system calls use links, this mechanism provides the required transparency.

Sprite

Location transparency is an original design goal of Sprite [Douglis and Ousterhout, 1991]. For example, Sprite includes a highly location transparent network filesystem. Yet, transparent migration still proved difficult even in the case of open file state in this location-transparent network filesystem.

Open files have three kinds of state associated with them: cached data, a file position, and a file reference. Because the state associated with an open file is copied from source to target, it is shared across nodes. This sharing is the source of the problem. The Sprite file server has a policy where it disables file caching when a file is open for writing by more than one node. In such cases, writes are propagated immediately and less efficiently to the server. When a process is migrated, its files all appear to be open by source and destination, causing caching to be disabled. Alternatively, if the file is first closed at the source node, it can be incorrectly deleted, if it is a temporary file, which is to be immediately removed on close. These issues were overcome but required Sprite to develop complex modifications to migration and to the operating system.

Sprite associates a home node with each process which is the machine where the process was started. System calls that depend on the location of a process are forwarded to the process's home node. As long as the number of these system calls is small, the impact of this forwarding on

performance may be acceptable. `fork` and `exec` are examples of expensive system calls that must be forwarded to the home node.

Systems like the V kernel, DEMOS, and Accent [Richard Rashid, 1986], where all interactions with a process, including OS system calls, occurred through a uniform communication abstraction, can elegantly enact migration by making the endpoints of their communication channels location transparent. Messages were forwarded to the new location after migration. Sprite, on the other hand, allowed processes to interact with the operating system through traditional system calls which required less elegant, albeit often more efficient, operating system support. However, both approaches require some form of rebinding after migration. For example, if a process wants direct access to a hardware resource on the node it is running on, its request cannot be forwarded to its home node. It is sometimes hard to know which node the programmer wants a system call to effect.

Condor

While kernel-supported process migration can be transparent to user processes, it has not achieved widespread acceptance in commercial operating systems. I believe the complexity of kernel-supported migration and the lack of strong demand for it so far has discouraged its commercial inclusion. While there is significant demands for process migration for the purpose of exploiting idle workstation resources in clusters, this demand is satisfied by simpler user-level migration implementations such as Condor and the Load Sharing Facility [Bricker et al., 1991, Zhou et al., 1992], or through application-specific mechanisms [Noble et al., 1997].

The approach to process migration that gained commercial success is the less transparent user-level implementation, of which Condor is a good example [Bricker et al., 1991]. Condor employs three principle mechanisms to provide the load balancing in a cluster: Classified Advertisements, Remote System Calls, and Job Checkpointing. Classified Ads are the mechanism that Condor uses to pair up Resource Requests and Resource Offers. Remote system calls redirect the system calls of the new copy of the process to a shadow process running on the user's local workstation. The system calls are executed on the local workstation by the shadow, and the results are sent back to the application on the remote workstation. This enables Condor to migrate a process to a remote workstation that does not have all the capabilities of the original workstation. For example, the remote workstation may have ample CPU cycles but no access to a network filesystem that the application uses to store files. In this case, after migration, the file access system calls are sent back to be serviced at the original node by the shadow (vestigial) process.

Checkpointing in Condor is performed through checkpointing and restart libraries linked to with the process. When a process is to be migrated, Condor sends a signal which is captured by the checkpointing library. This library contains code that saves the virtual memory contents of the process to be used to reinitialize the new process on the target node. Because Condor is a totally user-level implementation, it is much less transparent than kernel supported approaches. For instance, forking and process groups are not supported.

The research described in this chapter addresses the problem of migrating application objects that are usually of finer granularity than entire processes. Rather than focusing on the migration mechanism, ABACUS asks for the help of the programmer in abiding to a programming discipline that simplifies migration. The focus in ABACUS is instead on where to place components and how to dynamically adjust this placement.

5.10.2 Programming systems supporting object mobility

Mobile object systems are programming systems where objects can move between nodes transparently to the application programmer. A seminal implementation of object mobility is demonstrated by Emerald [Jul et al., 1988].

Emerald is a distributed programming language and system which supports fine-grained object mobility [Jul et al., 1988]. Emerald provides a uniform object model, where everything in the application is an object. Built-in types (integers, floats), as well as arrays and structs are objects. Object invocation is location transparent and has the same semantics in both local and remote cases. An Emerald object consists of a name, which uniquely identifies the object, a representation, which – except for the case of primitive types – consists of references to other objects, a set of operations (an interface), and an optional process, which is started after the object is created and executes in parallel to invocations on the object. Emerald does not support inheritance. Objects in Emerald are derived from abstract data types, can be either passive or active and can be passed as arguments to method invocations. Active objects are associated with a process that is started after the object is created and executes concurrently to the invocations performed on the object. To ensure proper synchronization between concurrent invocations and the internal, active process, Emerald offers its programmers monitors and condition variables.

Emerald implements data migration through a combination of by-copy, by-move and by-network-reference mechanisms. Local references are changed to network references when an object migrates. For immutable objects, however, Emerald uses data migration by copy. The objects can be declared

as "immutable" by the programmer (i.e. the values of the fields in the object do not change over time), in which case they are freely copied across the network simplifying sharing. An object of a built-in type that is passed as an argument to a remote object method is also copied.

Other objects are sent as network references. Since an invocation to a remote object can pass several other local objects as arguments, performance can degrade if the argument objects are not moved to the target node. Emerald supports call-by-move semantics where argument objects are migrated to the target node hosting the invoked object. This can be specified by the programmer using special keywords. When an object is moved, its associated methods and state must be moved with it.

ABACUS uses similar mechanisms to those proposed in Emerald to find mobile objects at run-time. While Emerald enables object mobility between nodes, ABACUS focuses on the complimentary problem of deciding where to locate objects within a cluster.

5.10.3 Mobile agent systems

Recent work in mobile agents has proposed a different programming model to support explicit application-supported migration [Dale, 1997, Gray et al., 1996, Chess et al., 1997, Knabe, 1995]. The growth of the internet has recently catalyzed research on languages and run-time systems supporting "mobile agent" applications where an agent "roams around" the network, moving from one site to another performing some computation at each site. An example of a mobile agent is a program that searches for the cheapest airfare between two cities by crawling from one airline's site to the next. Mobile agents are attractive in that they can support "spontaneous electronic commerce" [Chess et al., 1997], electronic commerce transactions that do not require the prior agreement or cooperation of the sites involved. Computations can roam the network choosing their path dynamically and freely. Mobile agents raise security issues since a server is usually nervous about accepting an agent without knowing its intentions.

The elegance and wide range of the potential applications of mobile agents have resulted in several mobile agent programming systems [Dale, 1997, Gray, 1996, Hylton et al., 1996, Straer et al., 1996, Acharya et al., 1997, Bharat and Cardelli, 1997]. Most systems are object-based although some are scripting languages with support for migration.

Mobile agents, like ABACUS, use explicit `checkpoint/restore` methods to save and restore their state when they migrate. However, while mobile agents are responsible for deciding where they should execute and when they should move from one node to another, mobile objects in

ABACUS delegate that responsibility to the ABACUS run-time system.

5.10.4 Remote evaluation

Traditional distributed applications statically partition function between client and server. Application programmers decide on this function partitioning at design-time by fixing the "programming interface" to the server. The list of remote procedure calls (RPC) implemented by the server is therefore fixed at design-time. Clients build applications by building on the services provided by the servers. Applications that are not well-matched with this division of labor between client and server suffer from inefficient performance.

Consider the example of a distributed filesystem. The server provides procedures to insert a new file and delete an existing file from a hierarchical tree-like namespace. It also provides a lookup procedure which returns the contents of a directory (all the files that exist in that directory). The new file name and the parent directory are specified as arguments to the insert procedure. The file name of the existing file and the parent directory are specified as arguments of the delete procedure. A lookup procedure call takes the parent directory as an argument and returns its contents as a result. Now consider a client that desires to delete all files with a ".tmp" extension in the namespace. This client is required to issue a large number of successive RPCs, to lookup all the directories in the namespace and then to delete, one RPC at a time, the matching files. In such a scenario, it would be more efficient to send the "delete subtree" program to the server and execute it there and avoid this excessive client-server communication.

Remote evaluation [Stamos and Gifford, 1990] is a more general mechanism to program distributed systems. It allows a node to send a request to another node in the form of a "program". The destination node executes the program and returns the results to the source node. While with remote procedure calls, server computers are designed to offer a fixed set of services, remote evaluation allows servers to be dynamically extended. Remote evaluation can use the same argument passing semantics as RPCs, and masks computer and communication failures in the same way. It can also provide for a static checking framework to identify "programs" that can not be sent to a given node for execution, although in general this is very hard to do without significant restrictions on the programming model.

Stamos' Remote Evaluation allows flexibility in the placement of function (execution of services) in a fashion similar to ABACUS. However, it does not provide an algorithm or suggest a framework which allows this placement to be automatically decided. The programmer decides

when to invoke a service locally and when to ship it to a remote node. A filesystem built on Remote Evaluation would require the filesystem programmer to think about when to do local versus remote execution.

5.10.5 Active disks

A growing number of important applications operate on large data sets, searching, computing summaries, or looking for specific patterns or rules, essentially "filtering" the data. Filter-like applications often make one or more sequential scans of the data [Riedel et al., 1998]. Applications execute on the host, with the storage device serving as block servers. Active disk systems claim that the increasing levels of integration of on-disk controllers are creating "excess" computing cycles on the on-disk processor. These cycles can be harnessed by downloading "application-specific" data intensive filters. Currently, data-intensive applications execute entirely on the host, often bottlenecking on transferring data from the storage devices (servers) to the host (client in this case).

Recently, considerable interest has been devoted to the "remote execution" of application-specific code on on-disk processors. Several systems have been proposed such as active and intelligent disks [Riedel et al., 1998, Keeton et al., 1998, Acharya et al., 1998]. Remote execution is especially appealing for data-intensive applications that selectively filter, mine, or sort large data sets. Active disks thus propose executing the data intensive function of an application on the on-disk processor.

Acharya et al. [Acharya et al., 1998] propose a stream-based programming model, where user-downloaded functions operate on data blocks as they "stream by" from the disk. One problem with this stream-based model is coherence of data cached by applications executing on the host. Since data can potentially be replicated in the host and in the on-disk memory, consistency problems can arise. Moreover, this programming model is quite restrictive. For instance, to limit the amount of resources consumed by downloaded functions, user-downloaded functions are disallowed from dynamically allocating memory.

Active disks delegate to the programmer the task of partitioning the application. In the best possible case, the query optimizer-like engine is used to partition functions between host and active disk [Riedel, 1999]. While query optimizers use a-priori knowledge about the function being implemented to estimate what partitioning is best, ABACUS uses black-box monitoring which is more generally applicable albeit at the cost of higher run-time overhead.

5.10.6 Application object partitioning

Coign [Hunt and Scott, 1999] is a system which optimizes the partitioning of objects in a multi-object application executing over a cluster of nodes. It traces the communications of the application during some initial executions and uses this to decide where each object should live. Coign does not allow objects to move after the application starts. When the application starts, objects are anchored to their locations. Coign relieves the programmer from allocating resources and deciding object placement at design time. Coign employs scenario-based profiling and graph-cutting algorithms to partition application objects in a distributed component application between nodes of a cluster. Coign focuses on finding the proper initial placement of objects at installation time. It uses binary rewriting techniques to collect statistics about inter-object communication when a typical workload is applied to the system which are then used to decide on the proper placement of component objects given the availability of CPU and network resources.

Coign does not perform run-time monitoring or preemptive migration of component objects. Its placement algorithms are executed off-line and therefore have relatively forgiving response time requirements. On the other hand, Coign enables optimization of function placement at the granularity of an object, and not the granularity of entire processes. In the presence of a "typical scenario", Coign can be very effective in improving performance. However, when the proper placement depends on invocation-time parameters or on dynamic changes in resource availability, this approach can be suboptimal.

River [Arpaci-Dusseau et al., 1999] is a data-flow programming environment and I/O substrate for clusters of computers. It is designed with the goal of providing maximum performance in the common case despite underlying heterogeneity in node resources and despite other glitches and non-uniformities that might affect node performance. River balances load across consumers of a data set using a distributed-queue. River effectively balances load by allocating work to consumers to match their current data consumption rates. This ensures load balancing across multiple consumers performing the same task. This research complements River by addressing the case of multiple concurrent tasks.

The closest previous system to the approach taken by this dissertation is Equanimity. Equanimity dynamically rebalances service between a client and its server [Herrin, II and Finkel, 1993], using heuristics based on the amount of data communicated between function. Equanimity did not consider the impact of function placement on load imbalance and used only simple communication-based heuristics to partitioning the graph of function between a client and its server.

This research builds on Equanimity by considering service rebalancing in more realistic environments which exhibit client competition, data-intensive applications layered atop filesystems, heterogeneous resource distributions and shared data computing.

5.10.7 Database systems

Database management systems must often provide stringent guarantees on transaction throughput and maximum latency. Database management systems include a query optimizer which compiles a query in a structured high-level language onto an execution plan which is carefully selected to maximize a given performance goal. Query optimizers decide what part of the query to execute on which node by consulting a rule-based system or a predictive performance model. These approaches apply effectively to relational queries because there is a limited number of query operators and the operators are known to the optimizer ahead of time.

Traditional relational database system are based on a "function shipping" approach. Clients submit entire queries to the servers which execute the query and return the results. Object-oriented database system are often based on a "data shipping" approach that makes them similar to distributed filesystems. Data is transferred from the servers to the client where it is cached. Queries are executed on this data locally at the client. While data shipping is more scalable in principle because it uses client resources, network efficiency often mandates a function shipping approach.

Hybrid shipping [Franklin et al., 1996] is a technique proposed to dynamically distribute query processing load between clients and servers of a database management system. This technique uses *a priori* knowledge of the algorithms implemented by the query operators to estimate the best partitioning of work between clients and servers. Instead, ABACUS applies to a wider class of applications by relying only on black-box monitoring to make placement decisions, without knowledge of the semantics or algorithms implemented by the application components.

One way to view ABACUS research is that it attempts to bridge the gap between database systems and filesystems by bringing the benefits automatic resource management capabilities of database query optimizers to the applications that use filesystems and other object stores. Unlike database query optimizers, ABACUS uses a generic mechanism based on monitoring inter-object communication and object resource consumption to help it predict the optimal placement.

5.10.8 Parallel programming systems

Many programming languages and systems have recently investigated ways to improve the locality of data accesses for parallel applications including [Amarasinghe and Lam, 1993, Hsieh et al., 1993, Chandra et al., 1993, Carlisle and Rogers, 1995]. For example, Olden [Carlisle and Rogers, 1995] and Prelude [Hsieh et al., 1993] attempt to improve locality by migrating computations to the data. A computation accessing data on a remote node may be moved to that node. COOL [Chandra et al., 1993] is a parallel language with a scheduling algorithm that attempts to enhance the locality of the computation while balancing the load. COOL provides an affinity construct that programmers use to provide hints that drive the task scheduling algorithm.

The research in this chapter builds on this previous work by applying such techniques to the case of storage-intensive applications. Storage-intensive applications can be effectively modeled by a timed data flow graph which can be used to make effective placement decisions. Moreover, such applications move a large amount of data allowing a run-time system to learn valuable information about inter-object communication and object resource consumption quickly, and judiciously use it to select the best placement possible. Furthermore, this research is the first, to the best of our knowledge, to apply these techniques to the case of a particular and important system application, namely a distributed filesystem.

5.11 Summary

Emerging active storage systems promise dramatic heterogeneity. Active storage servers—single disks, storage appliances and servers—have varied processor speeds, memory capacities, and I/O bandwidths. Client systems—SMP servers, desktops, and laptops—also have varied processor speeds, memory capacities, network link speeds and levels of trustworthiness. Application tasks vary their load over time because of algorithmic or run-time parameters. Most importantly, dynamically varying application mixes result from independent and stochastic processes at different clients. These disparities make it hard for any design-time "one-system-fits-all" function placement decision to provide robust performance. In contrast, a dynamic function placement scheme can achieve better performance by adapting to application behavior and resource availability.

Previous systems demonstrated different function placement decisions, accentuating the fundamental trade-off between the scalability of client-side execution and the network efficiency of source/sink-side computing. However, due to the variability in application resource consumption,

in application mixes and in cluster resource availability, the tension between scalability and source-sink computing cannot be easily resolved until run-time. This chapter presents an overview and evaluation of ABACUS, an experimental prototype system used to demonstrate the feasibility of adaptive run-time function placement between clients and servers for filesystem functions as well as stream-processing type of applications. ABACUS uses an algorithm that continuously monitors resource availability as well as function resource consumption and inter-function communication and uses this knowledge to intelligently partition function between client and server.

This chapter describes a distributed filesystem, ABACUSFS, ported to the ABACUS system and reports on its ability to adapt. Microbenchmarks demonstrate that ABACUS and ABACUSFS can effectively adapt to variations in network topology, application cache access pattern, application data reduction (filter selectivity), contention over shared data, significant changes in application behavior at run-time, as well as dynamic competition from concurrent applications over shared server resources. Microbenchmark results are quite promising; ABACUS often improved application execution time by a factor of 6 or more. Under all experiments in this chapter, ABACUS selects the best placement for each function, "correcting" placement if function was initially started on the "wrong" node. Under more complex scenarios, ABACUS outperforms experiments in which function was statically placed at invocation time, converging to within 70% of the maximum achievable performance. Furthermore, ABACUS adapts placement without knowledge of the semantics implemented by the objects. The adaptation is based only on black-box monitoring of the object and the number of bytes moved between objects.

Chapter 6

Conclusions and future work

This chapter concludes this dissertation by summarizing the main contributions and describing directions for future work. It is organized as follows: Section 6.1 summarizes the research reported on in this dissertation. Section 6.2 highlights the main contributions. Section 6.3 discusses directions for future work.

6.1 Dissertation summary

6.1.1 Network-attached storage

Storage bandwidth requirements continue to grow due to rapidly increasing client performance, new, richer content data types such as video, and data intensive applications such as data mining. This problem has been recognized for at least a decade [Long et al., 1994, Patterson et al., 1988, Hartman and Ousterhout, 1993]. All storage system solutions to date incur a high overhead cost for providing bandwidth due to existing storage architectures' reliance on file servers as a bridge between storage and client networks. Such storage systems do not scale because they rely on a central controller to manage and mediate access to the physical storage devices. Requests from the application all pass through the storage controller, which then forwards them to the storage devices, storing and copying data through it on every access. Storage systems administrators expand storage capacity by using multiple underlying disk arrays, and partitioning the data set manually between the arrays. Unfortunately, even if load balancing was a good use of time, the system administrator is rarely well equipped with the dynamic information to perform balancing in a timely and satisfactory manner.

Storage architectures are ready to change as a result of the synergy from four overriding factors: increasing object sizes and data rates in many applications, new attachment technology, the

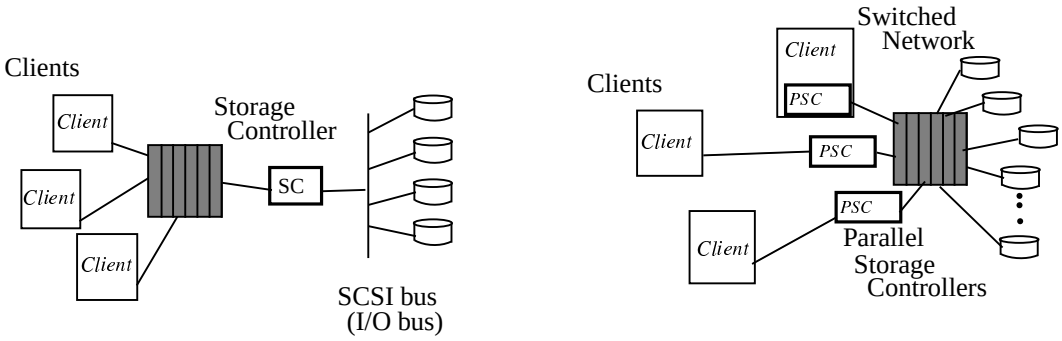
convergence of peripheral and interprocessor switched networks, and an excess of on-drive transistors. Network-Attached Secure Disks (NASD) [Gibson et al., 1997b, Gibson et al., 1998] is an architecture that enables cost-effective bandwidth scaling. NASD eliminates the server bottleneck by modifying storage devices so they can transfer data directly to clients. Further, NASD repartitions traditional file server functionality between the NASD drive, client and server.

NASD does not advocate that all functions of the traditional file server need to be or should be migrated into storage devices. NASD devices do not perform the highest levels of distributed file system function — global naming, access control, concurrency control, and cache coherency — which define semantics that vary significantly across distributed file systems and to which client applications and operating systems tightly bind. Instead, the residual file system, which is called the file manager, continues to define and manage these high level policies while NASD devices implement simple storage primitives efficiently and operate as independently of the file manager as these policies allow. The low cost of storage is due to the large market for mass-produced disks. This mass production requires a standard interface that must be simple, efficient, and flexible to support a wide range of file system semantics across multiple technology generations.

NASD enables clients to perform parallel data transfers to and from the storage devices. This dissertation describes a storage service, Cheops, which implements such function. Real applications running on top of a Cheops/NASD prototype receive scalable data access bandwidths that increase linearly with system size. For a Cheops client to conduct parallel transfers directly to and from the NASD devices, it must cache the stripe maps and capabilities required to resolve a file-level access and map it onto accesses to the physical NASD objects. The Cheops approach is to virtualize storage layout in order to make storage look more manageable to higher-level filesystems. Cheops avoids reinlisting servers to synchronously resolve the virtual to physical mapping by decomposing and distributing its access functions and management functions such that access function is executed at the client where the request is initiated. Cheops managers are responsible for authorization and oversight operations so that the participating clients always do the right thing.

6.1.2 Shared storage arrays

For the sake of scalability, Cheops allows clients to access shared devices directly. This fundamentally makes each storage client a storage controller on behalf of the applications running on it. Each storage controller can serve clients and manage storage. Unfortunately, such shared storage arrays lack a central point to effect coordination. Because data is striped across several devices and often



(a) Traditional storage systems (b) Scalable storage arrays

Figure 6.1: Traditional storage systems (a) use a single controller. Shared arrays (b) use parallel cooperating controllers to access and manage storage.

stored redundantly, a single logical I/O operation initiated by an application may involve sending requests to several devices. Unless proper concurrency control provisions are taken, these I/Os can become interleaved so that hosts see inconsistent data or corrupt the redundancy codes.

This dissertation proposes and evaluates an architecture that enables the controllers to concurrently access shared devices, migrate data between devices, and reconstruct data on failed devices while ensuring correctness and recovering properly from failures. The difficult aspect of this task is to ensure that the solution is scalable, thereby delivering the scalability of the NASD architecture. The proposed approach is to avoid central global entities and opt for distributing control overhead instead. Both concurrency control and recovery protocols are distributed. Specifically, the approach proposes breaking storage access and management tasks into two-phased light-weight transactions, called base storage transactions (BSTs). Distributed protocols are used to ensure concurrency control and recovery. These protocols do not suffer from a central bottleneck. Moreover, they exploit the two-phased nature of BSTs to piggy back control messages over data I/Os, hiding control messaging latency in the common case.

The base protocols assume that within the shared storage array, data blocks are cached at the NASD devices and not at the controllers. When controllers are allowed to cache data and parity blocks, the distributed protocols can be extended to guarantee serializability for reads and writes. This dissertation demonstrates that timestamp ordering with validation performs better than device-served leasing in the presence of contention, false sharing and random access workloads, all typical of clustered storage systems. In summary, it concludes that timestamp ordering based on loosely

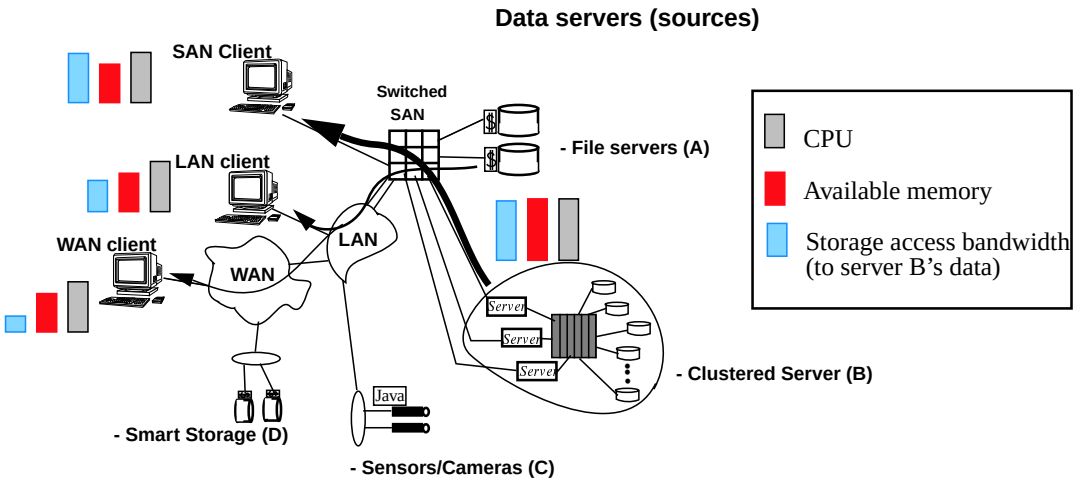


Figure 6.2: Emerging systems include programmable clients and servers. However, the clients and the servers vary in their computational and memory resources. The optimal partitioning of function between client and server varies based on run-time characteristics such as node load, network bandwidth, and amount of communication between application components. This figure shows the available CPU and memory resources on some of the nodes. The available access bandwidth between a node and the storage on server B is also graphed. Assume a generic computation over a data set streaming out of server B, where should the computation be performed (client or server?).

synchronized clocks has robust performance across low and high contention levels, in the presence of device-side or host-side caching. At the same time, timestamp ordering requires limited state at the devices and does not require the devices to perform any extra messaging on behalf of hosts (such as lease revocation).

6.1.3 Dynamic and automatic function placement

Another challenging aspect of storage management concerns the proper partitioning of function between the different nodes in the storage system. The partitioning of function between client and server has a direct impact on how load is balanced among a server and its clients and on how much data is transferred between client and server. Naive placement can cause resources to go underutilized and load to be imbalanced or large amounts of data to be transferred (unnecessarily) over bottlenecked links.

Currently, the partitioning of filesystem function between client and server is decided by the application programmer at design time. Filesystems designers decide on function partitioning after careful consideration of a multitude of factors including the relative amounts of resources assumed

to be at the client and the storage device, the performance of the network connecting them, the trustworthiness of the clients, and the characteristics of the target workloads. New hardware generations change the performance ratios among the system components, invalidating the design assumptions predicating the original placement. To cope with this, applications are often tuned for each new environment and for each hardware generation.

In general, dynamic variations in resource distribution and in workload characteristics during the lifetime of an application's execution often mandate a change in function placement. Even for applications that have a relatively constant behavior during their execution, concurrency and contention on resources and data among applications often induce dynamic changes that can not be anticipated beforehand.

This dissertation research observes that the proper partitioning of function is crucial to performance. It investigates algorithms that optimize application performance by intelligently and adaptively partitioning the application's processing between the client and the server. The findings suggest an automatic and transparent technique that enables the "effective bandwidth" seen by data-intensive applications to be increased by moving data-intensive functions closer to the data sources (storage servers) or sinks (clients) based on the availability of processing power and the amount time spent communicating between nodes.

In particular, the findings establish that dynamic placement of functions at run-time is superior to static one-time placement. Further, it shows that dynamic placement can be effectively performed based only on black-box monitoring of application components and of resource availability throughout the cluster. It proposes a programming model to compose applications from explicitly migratable *mobile objects*. A run-time system observes the resources consumed by mobile objects, and their intercommunication and employs on-line analytic models to evaluate alternative placement configurations and adapt accordingly.

6.2 Contributions

This dissertation makes several contributions; some in the form of fundamental scientific results, and others in the form of artifacts and prototypes which support further investigations.

6.2.1 Results

- *An approach based on specialized transactions to structuring storage access and management in a RAID array with multiple concurrent controllers.*
- *Distributed device-based protocols to ensure correctness in a shared RAID array with multiple concurrent controllers. The protocols offer good scalability and limited load and state at the nodes.*
- *Experimental data showing the potential importance of dynamic function placement for data-intensive applications and the feasibility of deciding best placement based on black-box monitoring.*

6.2.2 Artifacts

- Cheops prototype. Cheops is a striping library for network-attached secure disks. Cheops provides a "virtual" NASD interface atop physical NASD objects. It allows clients to cache virtual to physical mappings and therefore have direct parallel access to the storage devices.
- ABACUS and ABACUSFS. The ABACUS prototype can be used to experiment with dynamic function placement in clusters. ABACUSFS is a composable object-based filesystem enabling adaptive function placement between client and server.

6.3 Future work

More experience with the ABACUS programming model would be valuable. Applying the techniques of continuous monitoring and adaptive placement to streaming applications over the web is promising. Stream-processing application functions can be automatically distributed between client, server and proxy. The algorithms used by ABACUS should be extended to handle the placement over multiple intermediate nodes. Also, to work well in geographically wide areas, the measurement and statistics collection technology must be made more robust to wild perturbations and fluctuations in performance, a typical characteristic of wide area networks. ABACUS can benefit from using Java instead of C++ as its base programming language. Java is platform-independent and therefore can enable migration across heterogeneous architectures.

The intelligence of the ABACUS run-time system can be extended in several directions to improve its performance. Currently, it reacts only to recent cluster activity. One approach is to augment

the system with the ability to maintain long-term past profiles (on the granularity of a day, week, or month) to make more intelligent migration decisions. Similarly, application hints about their future accesses can be integrated to improve function placement decisions. A further use of history can improve the benefit estimation for the cost/benefit analysis. It is not worth migrating an object that will terminate shortly. Remaining time for applications can be estimated using heuristics [Harchol-Balter and Downey, 1995] or by consulting a database of past profiles.

Bibliography

[Acharya et al., 1997] Acharya, A., Ranganathan, M., and Saltz, J. (1997). Sumatra: A language for resource-aware mobile programs. In *Mobile Object Systems: Towards the Programmable Internet*, pages 111–130. Springer-Verlag. Lecture Notes in Computer Science No. 1222.

[Acharya et al., 1998] Acharya, A., Uysal, M., and Saltz, J. (1998). Active disks: Programming model, algorithms and evaluation. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 81–91, San Jose, CA.

[Adya et al., 1995] Adya, A., Gruber, R., Liskov, B., and Maheshwari, U. (1995). Efficient optimistic concurrency control using loosely synchronized clocks. In Carey, M. J. and Schneider, D. A., editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 23–34, San Jose, CA.

[Agrawal and Schafer, 1996] Agrawal, R. and Schafer, J. (1996). Parallel mining of association rules. In *IEEE Transactions on Knowledge and Data Engineering*, volume 8, pages 962–969.

[Agrawal and Srikant, 1994] Agrawal, R. and Srikant, R. (1994). Fast algorithms for mining association rules. In Bocca, J. B., Jarke, M., and Zaniolo, C., editors, *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 487–499, Santiago, Chile. Morgan Kaufmann.

[Amarasinghe and Lam, 1993] Amarasinghe, S. and Lam, M. (1993). Communication optimization and code generation for distributed memory machines. In *Proceedings of SIGPLAN Conference on Program Language Design and Implementation*, Albuquerque, NM.

[Amiri et al., 2000] Amiri, K., Gibson, G., and Golding, R. (2000). Highly concurrent shared storage. In *Proceedings of the 20th International Conference on Distributed Computing Systems*, Taipei, Taiwan, Republic of China.

[Anderson et al., 1996] Anderson, T. E., Dahlin, M. D., Neefe, J. M., Patterson, D. A., Roselli, D. S., and Wang, R. Y. (1996). Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79.

[ANSI, 1986] ANSI (1986). Small Computer System Interface (SCSI) Specification. ANSI X3.131.

[ANSI, 1993] ANSI (1993). Small Computer System Interface (SCSI) Specification 2. ANSI X3T9.2/375D.

[Arpaci-Dusseau et al., 1999] Arpaci-Dusseau, R. H., Anderson, E., Treuhaft, N., Culler, D. E., Hellerstein, J. M., Patterson, D., and Yelick, K. (1999). Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 10–22, Atlanta, GA.

[Baker et al., 1991] Baker, M., Hartman, J., Kupfer, M., Shirriff, K., and Ousterhout, J. (1991). Measurements of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 198–212, Pacific Grove, CA, USA.

[Benner, 1996] Benner, A. F. (1996). *Fibre Channel: Gigabit Communications and I/O for Computer Networks*. McGraw Hill, New York.

[Bernstein and Goodman, 1980] Bernstein, P. A. and Goodman, N. (1980). Timestamp-based algorithms for concurrency control in distributed database systems. In *Proceedings of the 6th Conference on Very Large Databases (VLDB)*, pages 285–300, Montreal, Canada.

[Bharat and Cardelli, 1997] Bharat, K. and Cardelli, L. (1997). Migratory applications. In *Mobile Object Systems: Towards the Programmable Internet*, pages 131–149. Springer-Verlag. Lecture Notes in Computer Science No. 1222.

[Birrel and Needham, 1980] Birrel, A. D. and Needham, R. M. (1980). A universal file server. In *IEEE transactions on software engineering*, volume 6, pages 450–453.

[Birrell and Nelson, 1984] Birrell, A. D. and Nelson, B. J. (1984). Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59.

[Blaum et al., 1994] Blaum, M., Brady, J., Bruck, J., and Menon, J. (1994). EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 245–254, Chicago, IL.

[Boden et al., 1995] Boden, N. J., Cohen, D., Felderman, R. E., Kulawik, A. E., Seitz, C. L., Seizovic, J. N., and Su, W.-K. (1995). Myrinet—a Gigabet-per-second local-area network. *IEEE Micro*, 15(1):29–36.

[Bokhari, 1979] Bokhari, S. H. (1979). Dual processor scheduling with dynamic reassignemnt. *IEEE Transactions on Software Engineering*, 5(4):341–349.

[Bolosky et al., 1996] Bolosky, W. J., III, J. S. B., Draves, R. P., Fitzgerald, R. P., Gibson, G. A., and Jones, M. B. (1996). The tiger video fileserver. In *Proceedings of the Sixth International Workshop on Network and Operating System Support for Digital Audio and Video*, Zushi, Japan.

[Bricker et al., 1991] Bricker, A., Litzkow, M., and Livny, M. (1991). Condor technical summary. Technical Report 1069, University of Wisconsin—Madison, Computer Science Department.

[Buzzard et al., 1996] Buzzard, G., Jacobson, D., Mackey, M., Marovich, S., and Wilkes, J. (1996). An implementation of the Hamlyn sender-managed interface architecture. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 245–259.

[Cabrera and Long, 1991] Cabrera, L.-F. and Long, D. D. E. (1991). Swift: Using distributed disk striping to provide high I/O data rates. In *Computing Systems, Fall, 1991*, pages 405–436.

[Cao et al., 1994] Cao, P., Lim, S. B., Venkataraman, S., and Wilkes, J. (1994). The TickerTAIP parallel RAID architecture. *ACM Transactions on Computer Systems*, 12(3):236–269.

[Carey et al., 1994] Carey, M. J., DeWitt, D. J., Franklin, M. J., Hall, N. E., McAuliffe, M. L., Naughton, J. F., Schuh, D. T., Solomon, M. H., Tan, C. K., Tsatalos, O. G., White, S. J., and Zwilling, M. J. (1994). Shoring up persistent applications. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 383–394, Minneapolis MN (USA).

[Carlisle and Rogers, 1995] Carlisle, M. and Rogers, A. (1995). Software caching and computation migration in olden. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA.

[Chanchio and Sun, 1998] Chanchio, K. and Sun, X. H. (1998). Memory space representation for heterogeneous network process migration. In *Proceedings of the 12th International Parallel Processing Symposium*, pages 801–805. IEEE Computer Society Press, Los Alamitos, CA.

[Chandra et al., 1993] Chandra, R., Gupta, A., and Hennessy, J. (1993). Data locality and load balancing in cool. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA.

[Chao et al., 1992] Chao, C., English, R., Jacobson, D., Stepanov, A., and Wilkes, J. (1992). Mime: A high performance parallel storage device with strong recovery guarantees. Hewlett-Packard Laboratories Technical Report HPL-SSP-92-9.

[Chen and Bershad, 1993] Chen, J. B. and Bershad, B. N. (1993). The impact of operating system structure on memory system performance. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 120–133.

[Chess et al., 1997] Chess, D., Harrison, C., and Kershenbaum, A. (1997). Mobile agents: Are they a good idea? – update. In *Mobile Object Systems: Towards the Programmable Internet*, pages 46–48. Springer-Verlag. Lecture Notes in Computer Science No. 1222.

[Cobalt Networks, 1999] Cobalt Networks (1999). Cobalt networks delivers network-attached storage solution with the new NASRaQ. Press release, http://www.cobaltnet.com/company/press/-press32.html.

[Corbett and Feitelson, 1994] Corbett, P. and Feitelson, D. (1994). Design and performance of the Vesta parallel file system. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 63–70, Knoxville, TN, USA.

[Corbett et al., 1996] Corbett, P., Prost, J.-P., Demetriou, C., Gibson, G., Riedel, E., Zelenka, J., Chen, Y., Felten, E., Li, K., Hartman, J., Peterson, L., Bershad, B., Wolman, A., and Aydt, R. (1996). Proposal for a common parallel file system programming interface. Technical Report CMU-CS-96-193, School of Computer Science, Carnegie Mellon University, Pittsbrugh, PA.

[Courtright, 1997] Courtright, W. (1997). *A transactional approach to redundant disk array implementation*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA.

[Dahlin, 1995] Dahlin, M. (1995). *Serverless Network File Systems*. PhD thesis, Computer Science Division, Department of Electrical Enginnering and Computer Science, University of California, Berkeley, CA.

[Dale, 1997] Dale, J. (1997). *A Mobile Agent Architecture for Distributed Information Management*. PhD thesis, University of Southampton.

[de Jonge et al., 1993] de Jonge, W., Kaashoek, M., and Hsieh, W. (1993). The logical disk: A new approach to improving file systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 15–28, Asheville, NC, USA.

[Dewitt and Hawthorn, 1981] Dewitt, D. and Hawthorn, P. (1981). A performance evaluation of database machine architectures. In *Proceedings of the 7th International Conference on Very Large Data Bases (VLDB)*.

[Douglis, 1990] Douglis, F. (1990). *Transparent Process Migration in the Sprite Operating System*. PhD thesis, University of California, Berkeley, CA. Available as Technical Report UCB/CSD 90/598.

[Douglis and Ousterhout, 1991] Douglis, F. and Ousterhout, J. (1991). Transparent process migration: Design alternatives and the Sprite implementation. *Software—Practice & Experience*, 21(8):757–785.

[Drapeau et al., 1994] Drapeau, A. L., Shirrif, K. W., Hartman, J. H., Miller, E. L., Seshan, S., Katz, R. H., Lutz, K., Patterson, D. A., Lee, E. K., Chen, P. H., and Gibson, G. A. (1994). RAID-II: A high-bandwidth network file server. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 234–244.

[Emanuel, 1997] Emanuel, S. (1997). Storage growth fuels admin woes. http://www.idg.net/crd_data_87572.html.

[English and Stephanov, 1992] English, R. and Stephanov, A. (1992). Loge: a self-organizing disk controller. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 237–251, San Fransisco, CA, USA.

[EnStor, 1997] EnStor (1997). The storage challenge. http://www.enstor.com.au/challenge.htm, Feb 2000.

[Eswaran et al., 1976] Eswaran, K., Gray, J., Lorie, R., and Traiger, L. (1976). The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 9(11):624–623.

[Faloutsos, 1996] Faloutsos, C. (1996). *Searching Multimedia Databases by Content*. Kluwer Aacademic Publishers Inc.

[Fayyad, 1998] Fayyad, U. (1998). Taming the giants and the monsters: Mining large databases for nuggets of knowledge. In *Database Programming and Design*.

[Forum, 1995] Forum, T. M. (1995). The MPI message-passing interface standard. http://www.mcs.anl.gov/mli/standard.htm.

[Franklin et al., 1996] Franklin, M. J., Jónsson, B. T., and Kossmann, D. (1996). Performance tradeoffs for client-server query processing. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, volume 25 of *ACM SIGMOD Record*, pages 149–160, New York, NY.

[Gibson et al., 1997a] Gibson, G., Nagle, D. F., Amiri, K., Chang, F. W., Gobioff, H., Riedel, E., Rochberg, D., and Zelenka, J. (1997a). Filesystems for network-attached secure disks. Technical Report CMU-CS-97-118, School of Computer Science, Carnegie Mellon University, Pittsbrugh, PA.

[Gibson, 1992] Gibson, G. A. (1992). *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. MIT Press.

[Gibson et al., 1998] Gibson, G. A., Nagle, D. F., Amiri, K., Butler, J., Chang, F. W., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D., and Zelenka, J. (1998). A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 98)*, pages 92–103, San Jose, CA.

[Gibson et al., 1997b] Gibson, G. A., Nagle, D. F., Amiri, K., Chang, F. W., Feinberg, E. M., Gobioff, H., Lee, C., Ozceri, B., Riedel, E., Rochberg, D., and Zelenka, J. (1997b). File server scaling with network-attached secure disks. In *Proceedings of the 1997 ACM International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS)*, pages 272–284, Seattle, WA.

[Gibson et al., 1999] Gibson, G. A., Nagle, D. F., Courtright, W., Lanza, N., Mazaitis, P., Unangst, M., and Zelenka, J. (1999). NASD scalable storage systems. In *Proceedings of the USENIX '99 Extreme Linux Workshop*, Monterey, CA.

[Gibson and Wilkes, 1996] Gibson, G. A. and Wilkes, J. (1996). Self-managing network-attached storage. In *ACM Computing Surveys*, volume 28,4es, page 209.

[GigabitEthernet, 1999] GigabitEthernet (1999). Gigabit Ethernet overview - updated May 1999. http://www.gigabitethernet.org/technology/whitepapers/gige_0698/papers98_toc.html.

[Gobioff, 1999] Gobioff, H. (1999). *Security for a High Peformance Commodity Storage Subsystem*. PhD thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[Golding et al., 1995] Golding, R., Shriver, E., Sullivan, T., and Wilkes, J. (1995). Attribute-managed storage. In *Workshop on Modeling and Specification of I/O*, San Antonio, TX.

[Golding and Borowsky, 1999] Golding, R. A. and Borowsky, E. (1999). Fault-tolerant replication management in large-scale distributed storage systems. In *Proceedings of the 18th Symposium on Reliable Distributed Systems*, pages 144–155, Lausanne, Switzerland.

[Gong, 1989] Gong, L. (1989). A secure indentity-based capability system. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 56–63, Oakland, CA.

[Gray and Cheriton, 1989] Gray, C. and Cheriton, D. (1989). Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 202–210, Litchfield Park AZ USA. ACM.

[Gray, 1997] Gray, J. (1997). What happens when processors are infinitely fast and storage is free? Keynote Speech at the Fifth Workshop on I/O in Parallel and Distributed Systems, San Jose, CA.

[Gray et al., 1975] Gray, J., Lorie, R., Putzulo, G., and Traiger, I. (1975). Granularity of locks and degrees of consistency in a shared database. IBM Research Report RJ 1654.

[Gray and Reuter, 1993] Gray, J. and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

[Gray et al., 1996] Gray, R., Kotz, D., Nog, S., Rus, D., and Cybenko, G. (1996). Mobile agents for mobile computing. Technical Report PCS-TR96-285, Dept. of Computer Science, Dartmouth College.

[Gray, 1996] Gray, R. S. (1996). Agent Tcl: A flexible and secure mobile agent system. In *Proceedings of the 4th Annual Tcl/Tk Workshop*, pages 9–23, Monterey, CA.

[Grochowski, 2000] Grochowski, E. (2000). Storage price projections. http://www.storage.ibm.com/technolo/grochows/g05.htm.

[Grochowski and Hoyt, 1996] Grochowski, E. G. and Hoyt, R. F. (1996). Future trends in hard disk drives. In *IEEE Transactions on Magnetics*, volume 32.

[Haerder and Reuter, 1983] Haerder, T. and Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317.

[Harchol-Balter and Downey, 1995] Harchol-Balter, M. and Downey, A. B. (1995). Exploiting life-time distributions for dynamic load balancing. *Operating Systems Review*, 29(5):236.

[Hartman and Ousterhout, 1993] Hartman, J. and Ousterhout, J. (1993). The Zebra striped network file system. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 29–43, Asheville, NC, USA.

[Hartman et al., 1999] Hartman, J. H., Murdock, I., and Spalink, T. (1999). The Swarm scalable storage system. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*.

[Heidemann and Popek, 1994] Heidemann, J. S. and Popek, G. J. (1994). File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89.

[Herrin, II and Finkel, 1993] Herrin, II, E. H. and Finkel, R. A. (1993). Service rebalancing. Technical Report CS-235-93, Department of Computer Science, University of Kentucky.

[Hitz et al., 1990] Hitz, D., Harris, G., Lau, J. K., and Schwartz, A. M. (1990). Using UNIX as one component of a lightweight distributed kernel for multiprocessor file servers. In *Proceedings of the 1990 Winter USENIX Conference*, pages 285–295.

[Hitz et al., 1994] Hitz, D., Lau, J., and Malcolm, M. (1994). File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, San Fransisco, CA, USA.

[Holland et al., 1994] Holland, M., Gibson, G. A., and Siewiorek, D. P. (1994). Architectures and algorithms for on-line failure recovery in redundant disk arrays. *Journal of Distributed and Parallel Databases*, 2(3):295–335.

[Hollebeek, 1997] Hollebeek, R. (1997). Spurring economic development with large scale information infrastructure. In *Proceedings of the 4th Inernational Conference on Computational Physics*, Singapore.

[Horst, 1995] Horst, R. (1995). TNet: A reliable system area network. *IEEE Micro*, 15(1):37–45.

[Howard et al., 1988] Howard, J., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. (1988). Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81.

[Hsieh et al., 1993] Hsieh, W., Wang, P., and Weihl, W. (1993). Computation migration: Enhancing locality for distributed-memory parallel systems. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA.

[Hunt and Scott, 1999] Hunt, G. C. and Scott, M. L. (1999). The Coign automatic distributed partitioning system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 187–200, New Orleans, Louisiana.

[Hylton et al., 1996] Hylton, J., Manheimer, K., Drake, Jr., F. L., Warsaw, B., Masse, R., and van Rossum, G. (1996). Knowbot programming: System support for mobile agents. In *Proceedings of the Fifth International Workshop on Object Orientation in Operating Systems*, pages 8–13, Seattle, WA.

[Intel, 1995] Intel (1995). Virtual interface (VI) architecture. http://www.viarch.org/.

[Jones, 1998] Jones, A. (1998). Creating and sustaining competitive advantage. http://www.one-events.com/slides/stratus/.

[Jul et al., 1988] Jul, E., Levy, H., Hutchinson, N., and Black, A. (1988). Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133.

[Keeton et al., 1998] Keeton, K., Patterson, D., and Hellerstein, J. (1998). A case for intelligent disks (IDISKs). *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 27(3):42–52.

[Khalidi and Nelson, 1993] Khalidi, Y. and Nelson, M. (1993). Extensible file systems in Spring. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 1–14.

[Kim, 1986] Kim, M. Y. (1986). Synchronized disk interleaving. In *IEEE Transactions on computers*, number 11, pages 978–988.

[Knabe, 1995] Knabe, F. C. (1995). *Language Support for Mobile Agents*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA. Also available as Carngie Mellon School of Computer Science Technical Report CMU-CS-95-223 and European Computer Industry Centre Technical Report ECRC-95-36.

[Krieger and Stumm, 1997] Krieger, O. and Stumm, M. (1997). HFS: A performance-oriented flexible file system based on building-block compositions. *ACM Transactions on Computer Systems*, 15(3):286–321.

[Kung and Robinson, 1981] Kung, H. T. and Robinson, J. T. (1981). On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226.

[Lamb et al., 1991] Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. (1991). The ObjectStore database system. *Communications of the ACM*, 34(10):50–63.

[Lee and Thekkath, 1996] Lee, E. K. and Thekkath, C. A. (1996). Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, Massachusetts.

[Long et al., 1994] Long, D. D. E., Montague, B. R., and Cabrera, L.-F. (1994). Swift/RAID: A distributed RAID system. *Computing Systems*, 7(3):333–359.

[Lycos, 1999] Lycos (1999). Squeezing it all into top-shelf storage:Rising storage demands drive firms to capacity planning and management. http://www.zdnet.-org/eweek/stories/general/0 11011 389633 00.html.

[M. Livny and Boral, 1987] M. Livny, S. K. and Boral, H. (1987). Multi-disk management algorithms. In *Proceedings of the 1987 ACM Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS)*, pages 69–77.

[Maeda and Bershad, 1993] Maeda, C. and Bershad, B. (1993). Protocol service decomposition for high-performance networking. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 244–255.

[McKusick et al., 1984] McKusick, M., Joy, W., Leffler, S., and Fabry, R. (1984). A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197.

[McKusick et al., 1996] McKusick, M. K., Bostic, K., Karels, M. J., and Quarterman, J. S. (1996). *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, Inc.

[McVoy and Kleiman, 1991] McVoy, L. and Kleiman, S. (1991). Extent-like performance from a UNIX file system. In *Proceedings of the USENIX Winter 1991 Technical Conference*, pages 33–43, Dallas, TX, USA.

[Metcalfe and Boggs, 1976] Metcalfe, R. M. and Boggs, D. R. (1976). Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404.

[Mills, 1988] Mills, D. L. (1988). Network time protocol: specification and implementation. DARPA-internet RFC 1059.

[Mullender et al., 1990] Mullender, S. J., van Rossum, G., Tanenbaum, A. S., van Renesse, R., and van Staveren, H. (1990). Amoeba: A distributed operating system for the 1990s. *IEEE Computer Magazine*, 23(5):44–54.

[Muntz and Lui, 1990] Muntz, R. and Lui, J. (1990). Performance analysis of disk arrays under failure. In *Proceedings of the 16th Conference on Very Large Data Bases (VLDB)*, pages 162–173, Brisbane, Queensland, Australia.

[Neuman and Ts'o, 1994] Neuman, B. C. and Ts'o, T. (1994). Kerberos: An authentication service for computer networks. In *IEEE Communications*, volume 32, pages 33–38.

[News, 1998] News, I. (1998). Price Waterhouse predicts explosive e-commerce growth. http://www.internetnews.com/ecnews/article/0,1087,4_26681,00.html.

[NIST, 1994] NIST (1994). Digital signature standard. NIST FIPS Pub 186.

[Noble et al., 1997] Noble, B. D., Satyanarayanan, M., Narayanan, D., Tilton, J. E., Flinn, J., and Walker, K. R. (1997). Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 276–287, Saint Malo, France.

[OSI Standard, 1986] OSI Standard (1986). OSI Transport Protocol Specification. Technical Report ISO-8073, ISO.

[O'Toole and Shrira, 1994] O'Toole, J. and Shrira, L. (1994). Opportunistic log: Efficient installation reads in a reliable storage server. In *Proceedings of the 1st USENIX Symposium on Operating*

Systems Design and Implementation (OSDI): November 14–17, 1994, Monterey, CA, USA, pages 39–48.

[Ousterhout, 1990] Ousterhout, J. K. (1990). Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer 1990 USENIX Conf.*, pages 247–256, Anaheim, CA (USA).

[Ousterhout et al., 1985] Ousterhout, J. K., Costa, H. D., Harrison, D., Kunze, J. A., Kupfer, M., and Thompson, J. G. (1985). A trace-driven analysis of the Unix 4.2 BSD file system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 15–24, Orcas Island, WA.

[Papadimitriou, 1979] Papadimitriou, C. H. (1979). The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653.

[Patterson et al., 1988] Patterson, D., Gibson, G., and Katz, R. (1988). A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, Washington, DC, USA.

[Patterson et al., 1995] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. (1995). Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 79–95, Copper Mountain Resort, CO.

[Pierce, 1989] Pierce, P. (1989). A concurrent file system for a highly parallel mass storage system. In *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, pages 155–160, Monterey, CA.

[Powell and Miller, 1983] Powell, M. L. and Miller, B. P. (1983). Process migration in DEMOS/MP. In *Proceedings of the 9th Symposium on Operating System Principles*, pages 110–119.

[Rambus, 2000] Rambus (2000). Rambus for small, high performance memory systems. http://-www.rambus.com/developer/downloads/Value_Proposition_Networking.html.

[Rennels, 1980] Rennels, D. A. (1980). Distributed fault-tolerant computer systems. *IEEE Computer*, 13(3):39–46.

[Richard Rashid, 1986] Richard Rashid (1986). From RIG to Accent to Mach: The evolution of a network operating system. In *Proceedings of the ACM/IEEE Computer Society Fall Joint Computer Conference*, pages 1128–37.

[Riedel, 1999] Riedel, E. (1999). *Active Disks - Remote Execution for Network-Attached Storage*. PhD thesis, Department of Electrical Enginnering, Carnegie Mellon University, Pittsburgh, PA.

[Riedel et al., 1998] Riedel, E., Gibson, G., and Faloutsos, C. (1998). Active storage for large-scale data mining and multimedia. In *Proceedings of the 24th International Conference on Very Large Databases (VLDB)*, pages 62–73, New York, NY.

[Rosenblum, 1995] Rosenblum, M. (1995). *The Design and Implementation of a Log-structured File System*. Kluwer Academic Publishers, Norwell, MA, USA.

[Sandberg et al., 1985] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. (1985). The design and implementation of the Sun network filesystem. In *Proceedings of the USENIX Summer 1985 Technical Conference*, pages 119–130.

[Satyanarayanan, 1990] Satyanarayanan, M. (1990). Scalable, secure and highly available distrbuted file access. *IEEE Computer*, 23(5):9–21.

[Schulze et al., 1989] Schulze, M., Gibson, G., Katz, R., and Patterson, D. (1989). How reliable is a RAID? In *Intellectual leverage, COMPCON Spring 89*, pages 118–123, San Francisco (USA). IEEE.

[Seagate, 1999] Seagate (1999). Jini: A pathway for intelligent network storage. Press release, http://www.seagate.com:80/newsinfo/newsroom/papers/D2c1.html.

[Shirley et al., 1994] Shirley, J., Hu, W., and Magid, D. (1994). *Guide to Writing DCE Applications*. O'Reilly & Associates, Inc., Sebastopol, CA 95472, second edition.

[Spalink et al., 1998] Spalink, T., Hartman, J., and Gibson, G. (1998). The effect of mobile code on file service. Technical Report TR98-12, The Department of Computer Science, University of Arizona.

[Stamos and Gifford, 1990] Stamos, J. W. and Gifford, D. K. (1990). Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565.

[Stone and Bokhari, 1978] Stone, H. S. and Bokhari, S. H. (1978). Control of distributed processes. *IEEE Computer*, (7):97–106.

[Straer et al., 1996] Straer, M., Baumann, J., and Hohl, F. (1996). Mole – a Java based mobile agent system. In *Proceedings of the 2nd ECOOP Workshop on Mobile Object Systems*, pages 28–35, Linz, Austria.

[Tanenbaum, 1992] Tanenbaum, A. S. (1992). *Modern Operating Systems*. Prentice Hall, New Jersey.

[Technology, 1998] Technology, S. (1998). Cheetah: IndustryLeading Performance for the Most Demanding Application. World Wide Web, http://www.seagate.com/.

[Theimer et al., 1985] Theimer, M., Lantz, K., and Cheriton, D. (1985). Preemptable remote execution facilities for the V-System. In *Proceedings of the 10th Symposium on Operating System Principles*, pages 2–12.

[Thekkath et al., 1997] Thekkath, C. A., Mann, T., and Lee, E. K. (1997). Frangipani: A scalable distributed file system. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, pages 224–237, Saint Malo, France.

[TPC, 1998] TPC (1998). TPC-C and TPC-D executive summaries. http://www.tpc.org/.

[TriCore News Release, 1997] TriCore News Release (1997). Siemens' new 32-bit embedded chip architecture enables next level of performance in real-time electronics design. http://www.tricore.com/.

[University of Antwerp Information Service, 2000] University of Antwerp Information Service (2000). Campus connectivity status map. http://www.uia.ac.be/cc/internet.html.

[Van Meter et al., 1996] Van Meter, R., Finn, G., and Hotz, S. (1996). Derived virtual devices: A secure distributed file system mechanism. In *Proc. Fifth NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 95–97, College Park, MD.

[Vetter, 1995] Vetter, R. J. (1995). ATM concepts, architectures, and protocols. *Communications of the ACM*, 38(2):30–38.

[von Eicken et al., 1995] von Eicken, T., Basu, A., Buch, V., and Vogels, W. (1995). U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th*

Symposium on Operating Systems Principles (SOSP-95), pages 40–53, Copper Mountain Resort, Colorado.

[Watson and Coyne, 1995] Watson, R. and Coyne, R. (1995). The parallel I/O architecture of the high-performance storage system (HPSS). In *Proceedings of the 14th IEEE Symposium on Mass Storage Systems*, pages 27–44.

[Wilkes, 1995] Wilkes, J. (1995). The Pantheon storage-system simulator. Hewlett-Packard Laboratories Technical Report HPL-SSP-95-114.

[Wilkes et al., 1996] Wilkes, J., Golding, R., Staelin, C., and Sullivan, T. (1996). The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136.

[Wong and Wilkes, 2000] Wong, T. and Wilkes, J. (2000). My cache or yours? In *preparation*.

[Yee and Tygar, 1995] Yee, B. S. and Tygar, J. D. (1995). Secure coprocessors in electronic commerce applications. In *Proceedings of the 1995 USENIX Electronic Commerce Workshop*, pages 166–170, New York, NY (USA).

[Zhou et al., 1992] Zhou, S., Wang, J., Zheng, X., and Delisle, P. (1992). Utopia: A load sharing facility for large, heterogeneous distributed computing systems. Technical Report CSRI-257, University of Toronto.