AVAILABILITY, SCALABILITY
AND COST-EFFECTIVENESS
OF CLUSTER-BASED INTERNET INFRASTRUCTURES

MINWEN JI

A DISSERTATION

PRESENTED TO THE FACULTY

OF PRINCETON UNIVERSITY

FOR CANDIDACY OF THE DEGREE

OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE

BY THE DEPARTMENT

OF COMPUTER SCIENCE

JUNE 2001

@ Copyright by Minwen Ji, 2001.

All rights reserved.

Abstract

Clusters of commodity computers are a cost-effective hardware platform for large-scale Internet services. Availability and scalability are major concerns in the design of infrastructures for such services. My dissertation examines the opportunities in the data storage systems for improving the availability and scalability of cluster-based Internet infrastructures at a low cost. The goal of availability is to maximize the percentage of client requests that succeed despite the failure of one or more servers in the cluster. The goal of scalability is to efficiently scale the server throughput with the cluster size. My basic approach is to investigate the data and request distribution strategies across nodes in the cluster, i.e. how to partition and replicate data on disk or in memory and how to direct requests to the right partitions in order to achieve high availability and scalability.

Maintaining availability in the face of failures is a critical requirement for Internet services. Existing approaches in cluster-based data storage rely on redundancy to survive a small number of failures, but the system becomes largely unavailable if more failures occur. I study a failure isolation approach in which each server in the cluster can deliver data to clients independently of the failures of other servers. This approach is complementary to existing redundancy-based methods: redundancy can mask the first few failures, and failure isolation can take over and maintain availability for the majority of clients if more failures occur.

The ability to achieve high quality of service with minimal committed resources allows savings in many aspects including equipment cost, power consumption, and administration effort for Internet services. I study how to improve the price-performance ratio of Internet application servers by efficiently managing a cluster of in-memory databases as the cache for dynamic content. I observe that a good management strategy could be found at least for certain applications despite the challenges of dynamic content. It strives to maximize effective cache capacity and minimize synchronization cost. It is light-weighted and adapts dynamically to the changes in loads and access patterns.

// Acknowledgements

Acknowledgements

I would like to thank my dissertation committee for making it at all possible for me to complete this dissertation. I have been receiving continuous support from my advisor, Ed Felten, since I started research at Princeton. I enjoyed the extensive freedom in research directions and academic agenda under his supervision, and always think it is the most important thing to me as a graduate student. I was very much impressed by his sharpness in understanding and commenting on my ideas and results, and learned a lot from him both technically and strategically. I am indebted to Randy Wang, a reader of my dissertation, for his encouragement to me even during my hardest times in graduate school. His amazing photographs and experiences broadened my vision of the great nature. Two other committee members, Doug Clark and Larry Peterson, are unanimously respected in our department as excellent and caring professors. I owe a large debt to Doug for his generous offer of help whenever I needed it.

I was fortunate to have the opportunity to learn from many other professors and researchers in the past few years. JP Singh's interest and support in my work was sometimes the only reason I could move forward. Kai Li has been an unlimited source of advice and help, from career philosophy to paper writing to pet-friendly apartments. Sanjeev Arora shared with me his knowledge and wisdom on hash functions, which later became a fundamental element in my dissertation work. Andrea LaPaugh was always ready to be a referee for my work and shared with me her enthusiasm for new technologies. Bernard Chazelle, my academic advisor during my first year at Princeton, impressed me with his great kindness as well as his incredible sense of humor. John Wilkes, my manager in Hewlett-Packard Labs when I was a summer intern there, will always be a role model in my career.

The work in this dissertation was funded by National Science Foundation (NSF) under grants MIP-9420653 and ANI-9906704.

As a system researcher, I often had to deal with things like equipment hardware and system configuration. I could not have completed my experiments without the help from

Jim Roberts, Chris Tengi, Joe Crouthamel, Steve Elgersma, Tom Knowles, Chris Miller and other technical staff members. Becci Davies, Melissa Lawson, Ginny Hogan, Trish Killian, Michele Brown and others have helped make the computer science building a cozy and refreshing place to stay.

Many fellow graduate students, including Yuanyuan Zhou, Dongming Jiang, Mao Chen, Rob Shillner, Cheng Liao, Sanjeev Kumar, Yaoyun Shi and Amit Chakrabarti, helped me make progress in my study. My friends, colleagues and officemates, such as Zhen Li, Daniel Wang, Aki Nakao, Xiaodong Wen, Hongzhang Shan, John Hainsworth, Jie Chen, Han Chen, Yuqun Chen, George Tzanetakis, Limin Wang, Xiang Yu, Bin Wei, Jessica Fong, Erich Schmidt, David Penry, Kedar Swadi, Yefim Shuf and Jean Gilsing, offered valuable friendship and help that made my life in graduate school a pleasure.

Finally, I owe everything to my family. My parents' pride and hope in me are the persistent driving force for my struggles in career and in life. While they gave me unlimited trust and freedom as I grew up, I gave them gray hairs and sleepless nights. Living thousands of miles away in a completely different world, they might not be able to imagine what I have gone through here. However, I know that their love has always been with me no matter where I am. For the past nine months, I dearly enjoyed the company of a giant but babish, strong but innocent, independent but affectionate boy called Pan Pan. My husband and best friend, Xuefu Wang, helped me survive all those struggles with his care, support, sacrifice, patience and confidence in me. The completion of this dissertation is meant to be a contribution to the new family Xuefu and I are starting to build together.

Table of Contents

Abstract I

Acknowledgements II

Table of Contents IV

1 Introduction 1

1.1 Background and motivation 1

1.1.1 Cluster-based Internet infrastructures 1
1.1.2 Redundancy 2
1.1.3 Isolation 3
1.1.4 Content caching 4
1.1.5 Front-end distributors 5
1.1.6 Back-end distributors 6

1.2 Contributions 8

2 Design of the island-based file system 9

2.1 Analytical model for data loss 10

2.1.1 Non-redundant model 12
2.1.2 Redundancy schemes with grouping 15
2.1.3 Redundancy schemes without grouping 15
2.1.4 Data loss versus storage overhead 16
2.1.5 Partial availability for applications 19

2.2 Island-based design 19

2.2.1 Hash-based data distribution 20
2.2.2 Usage-based metadata replication 22
2.2.3 Reconfiguration and rebalance 24
2.2.4 Other design issues 26

3 Consistency of replicated metadata 29

3.1 Related work 30

3.2 Replication model 32

3.3 Consistency protocol design 33

3.3.1 Atomicity 33
3.3.2 Serialization 36
3.3.3 Recovery 38

3.4 Correctness testing 40

3.5 Summary 43

4 Evaluation of the island-based file system 44

4.1 Statistical analysis 44
4.1.1 Partial availability for applications 44
4.1.2 Replication cost and load distribution 46
4.1.3 Operation breakdown 51
4.2 Implementation 54
4.3 Performance 55
4.3.1 Micro benchmarks 56
4.3.2 Trace-based benchmarks 64
4.4 Related work 67
4.5 Summary 68
5 Affinity-based management of clustered in-memory databases 70
5.1 Assumptions 72
5.2 Challenges of dynamic content illustrated 73
5.3 Observation on query affinity 74
5.4 Exploiting query affinity 75
5.5 Affinity-based management 77
5.5.1 Components 78
5.5.2 Data distribution and consistency 80
5.5.3 Replication of search keys 80
5.5.4 Limitations 83
5.5.5 Implications to other systems 84
5.5.6 Implementation 85
5.6 Simulations of five distribution strategies 85
5.6.1 Experimental setups 86
5.6.2 Case study 1: White pages 88
5.6.3 Case study 2: Auctions 90
5.6.4 Dimensions of query affinity 94
5.6.5 Cooperative caching 95
5.7 Performance measurement on two prototype clusters 97
5.7.1 Experimental setups 97
5.7.2 Single server latencies 98
5.7.3 Measurement on cluster servers 100
5.8 Related work 102
5.9 Summary 104
6 Conclusion 106
6.1 Results 106
6.2 Future work 108

Reference 109

1 Introduction

1.1 Background and motivation

1.1.1 Cluster-based Internet infrastructures

Clusters of commodity computers are a cost-effective hardware platform for running large-scale applications such as file servers, web servers, and other Internet services [47]. The large-scale Internet services studied in my dissertation are those having large data sets and read/write access patterns to the underlying file or database system. Examples of such services include web hosting, web caching, newsgroup, e-mail, e-commerce and search engines. The availability requirement of those services is different from the traditional applications, such as scientific computing. In the traditional environments, we are familiar with the failure mode of all or nothing. But for an Internet server, if there is a partial failure in its infrastructure, the server probably wants to keep running and serve as many clients as possible, rather than go completely offline. The scalability requirement of those Internet infrastructures is also critical because good scalability can translate to low cost for high quality of service. For example, in today's Internet data centers, the ability to guarantee the same Service Level Agreements with less committed resource allows savings in equipment cost, power consumption, environmental control, rack space, and administration effort.

A large-scale infrastructure typically consists of nodes of several distinct roles, such as management nodes, processing nodes and data storage nodes. See Figure 1.1. The processing nodes usually host web servers and application servers and generate results for clients' requests. The storage nodes host file systems or database systems and are responsible for maintaining persistent and consistent data storage and providing highly available data access. In order to process clients' requests, the processing nodes need to access the data stored in the storage nodes, and will probably cache it in their main memories for future requests. The management nodes are responsible for coordinating the tasks of other nodes. For example, an important task of the management is to distribute requests across the processing nodes or to distribute data across the storage nodes.

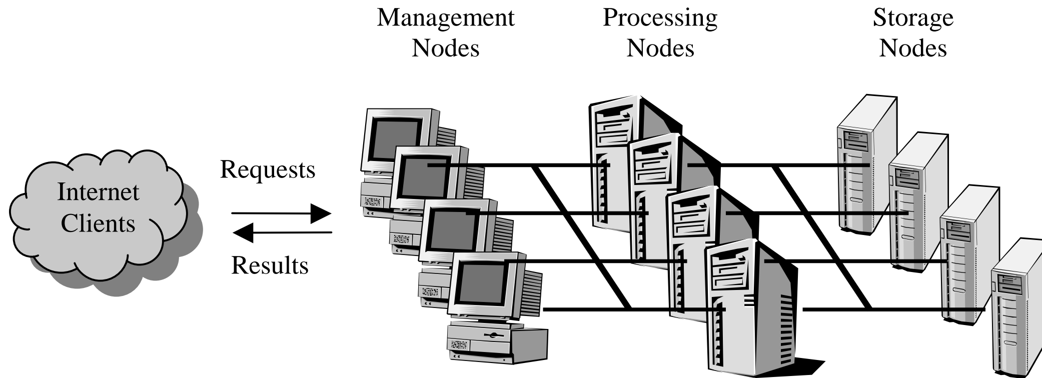

Figure 1.1 A cluster-based Internet infrastructure.

The goal of running such a cluster is to achieve high availability and scalability of the Internet service. A number of content-based request distribution strategies [3] [64] have attempted to scale cluster-based web servers by exploiting locality as well as load balance in the distribution strategy. A new programming model and support has been proposed for highly available and scalable processing nodes [47] [48]. My approach to high availability, scalability and cost-effectiveness is to study the data and request distribution strategies across the storage nodes.

1.1.2 Redundancy

I propose failure isolation as a complementary approach to exiting approaches to improving the availability and scalability of the data storage in cluster-based Internet infrastructures.

A wide variety of research projects on file or storage systems have explored approaches to high reliability and scalability. Redundancy is a standard approach for high reliability and scalability. In file and storage systems, redundancy can be classified in three categories: disk redundancy, server redundancy and client redundancy.

Redundant Arrays of Inexpensive Disks (*RAID*) [45] were designed for high aggregate disk bandwidth and recovery in case of individual disk failures. The basic approach is to break the disk arrays into redundancy groups, with each group having extra "check" disks

containing the redundant information. When a disk fails, the failed disk will be replaced and the information will be reconstructed on to the new disk using the redundant information. Typical organizations of data and check information in a redundancy group include mirroring (first level RAID) and striping (fifth level RAID).

The *servers* that are responsible for managing and delivering data are also extensively replicated, in the forms of wide-area distributed file systems, locally replicated file systems, virtual storage servers, and cluster file systems [5] [1]. Wide-area file systems [51] [53] [54] typically employ *one-copy* availability, in which any data may be updated as long as some copy is available, and trade consistency of the replicated data for availability and performance. Global content distributors [59] [2] [60] offer geographically distributed replication services to read-mostly web content so that the latency in delivering contents to clients can be reduced. They can be viewed as new applications of the traditional wide-area file systems on read-mostly web content. Locally replicated file systems [42][52] typically allow accesses to the *primary copy* only and have the secondary server take over when the primary crashes. Virtual storage servers [7] can be viewed as software RAIDs: a cluster of computers, equipped with disks and interconnected with fast networks, are presented as a single large storage server, while data is mirrored and striped internally. Cluster file systems [5] [1] are typically built on top of a shared virtual storage server: while the storage server takes care of data placement and replication, the objective of the cluster file system is to provide aggregate processing power, memory capacity and network bandwidth for managing and delivering the data.

Client caching is extensively used in distributed file systems [54] [8] [40] to support disconnected operations and to reduce traffics to servers. Similar to server replication, client caching improves availability by data redundancy, i.e. by replicating data in clients. It also improves scalability by reducing server load so that the same number of servers can serve a larger number of clients gracefully.

1.1.3 Isolation

However, reliability comes at a cost. One needs to pay extra storage for redundant data

copies. More importantly, one needs to pay synchronization overhead for keeping the replicas consistent with each other. The more reliability is required, the more cost needs to be paid. In commercial distributed file systems, such as NFS [9] and CIFS [15], individual servers are highly independent of each other and do not share data. Therefore, the failure of an individual server does not affect the functionality of other servers. However, those systems require manual partitioning of data across servers and suffer from system administration overhead at large scales.

Fox et al. [47] [48] suggest that modern Internet applications might prefer to serve as many clients as possible rather than to go completely offline when partial failures are present. In other words, partial availability might be more valuable to those applications than the traditional all-or-nothing failure mode. They also propose a new programming model for Internet applications where application decomposition and orthogonal mechanism are exploited for graceful degradation during partial failures.

Graceful handling of partial failures has been studied in other areas. For example, Chapin et al. Propose isolation of kernels [38] for the operating systems running on large-scale shared-memory multiprocessors. Isolation improves availability because a hardware or software fault damages only an independent component of the system rather than the entire system. Isolation also improves scalability because few resources are shared by processes running on different kernels.

1.1.4 Content caching

Many application servers, especially web portals [81], e-commerce servers [83] and search engines [82], contain a mixture of static and dynamic content. By *static* content, I mean files that HTTP servers directly access from file systems and return to clients, such as HTML pages and images. By *dynamic* content, I mean content that is generated on demand by applications such as CGI or ASP programs. Such content is typically database information about catalogs, inventory, customers, preferences, indexes on web documents, etc. It is often the dynamic content that differentiates a particular service site and the functionality it provides from other sites. However, dynamic content is delivered at a rate often one or two order of magnitudes slower than that of static content [88].

Caching, especially caching with aggregate main memory in a cluster [3] [61], has been established as an effective way to improve the performance and scalability of web servers with static content. Similarly, caching results from queries on dynamic content has been shown to help reduce computation involved in generating results that do not change frequently and do not cause updates to the underlying database [66] [67].

A more general approach to caching dynamic content has been proposed: using an *in-memory database* to cache the frequently accessed data and to provide the same interfaces and functions as an on-disk database, such as indexed search and concurrency control [85]. In-memory databases are optimized in many aspects, such as retrieval and indexing, specifically for memory-resident data, and are usually backed by Uninterrupted Power Supply (UPS) for the durability of transactions [86]. In fact, it has been suggested that it will simplify the construction of scalable applications on a cluster if the application data is stored in a shared, consistent in-memory data store rather than an application-specific data structure [87]. For example, it is non-trivial to port an application server with *session affinity* from a single machine to a cluster of machines if the data is kept inside the application, because the requests in the same session need to access the same data but might be processed by different servers as a result of request distribution for load balance [75]. Therefore, in-memory databases can potentially be useful for Internet servers either as content caches, or as in-memory data stores, or both.

The challenge for using in-memory databases arises in situations where the entire data set cannot fit in a single in-memory database and hence needs to be split into a cluster of in-memory databases. It involves automatically and dynamically partitioning and replicating data across nodes and directing requests to the right partitions. Previous work in the area of data/request distribution falls into two categories: request routers for clustered web servers and data allocators for distributed databases. Based on their positions in the cluster, I call them *front-end distributor* and *back-end distributor*, respectively.

1.1.5 Front-end distributors

State-of-the-art front-end distributors are content-based request distributors. Basically, requests for the same content, usually identified by the *Universal Resource Locator*

(*URL*), are directed to the same node as long as load is balanced across nodes. Because recently accessed files are cached in the main memory of the nodes, the content-based distribution can increase the cache hit ratio by using the aggregate main memories in the cluster as a large global cache. The key idea in content-based distribution is to distribute requests and to have data follow requests. In a content-oblivious request distribution, such as the connection-based distribution by a layer-4 network switch [75], each node will end up caching the same set of most frequently accessed files; therefore, the effective cache size of the cluster remains the same as the cache size of a single node. Content-based distribution has been successful in improving the scalability of web servers with static content [3] [64]. For static content, each file can be uniquely identified by the URL in the HTTP request; therefore, it is straightforward to partition the files and direct the requests to the right partitions based on URL.

However, more and more Internet sites today serve dynamic content. Typical examples are e-commerce servers and search engines, including the most visited sites Yahoo!, Amazon, Ebay, Google, Cnet, etc.. An essential difference in the requests to dynamic content is that the URLs in the requests specify the application and query, but not the data to access. Therefore, data could be accessed through multiple applications or by multiple attributes, and a single query could access multiple data items. Dynamic content can be written as well as read, for example, when the client places an order in an online shop. In front-end distribution, two requests on the same or overlapped data will likely be directed to two different servers if they carry two different queries. Therefore, front-end distribution could not eliminate data sharing across cache servers for dynamic content. If the data is read only, then we waste cache space for redundant copies of the same data. If the data is write-shared, then we pay synchronization cost for the consistency of the data.

1.1.6 Back-end distributors

An orthogonal approach to front-end distributors is to run a distributed database in the back end and have the database decide how to allocate data to individual nodes, i.e. to use a back-end distributor. The allocation is transparent to the web servers; the web servers access the distributed database as if it is a single shared database. A front-end distributor

may or may not be used in conjunction with the back-end distributor.

A lot of work has been done on the data fragmentation and allocation problems for distributed databases. The general problems [61] are stated as follows: Given the queries and updates, the frequencies of their usage, and the sites where the results have to be sent, determine 1) the fragments to be allocated, and 2) allocate these fragments, possibly redundant, and the operations on them to the sites of the computer network such that a certain cost function is minimized. Total data transmission cost is often used as the cost function. The optimal allocation of fragments was shown to be NP-complete [61]. The solutions to this problem are typically off-line, expensive optimization or periodical, heuristic process.

The optimal methods basically search the large solution space for determining data allocations to minimize total transmission cost. The heuristic methods typically start with an initial data allocation and iteratively reallocate fragments to decrease the total transmission cost in a greedy fashion until the cost can no longer be decreased. Due to their complexity, those techniques have to be applied off line or statically. Therefore, they are not readily applicable to those Internet services with dynamic changes in access patterns and loads.

Dynamic data reallocation for databases with changing access patterns and loads is studied by Brunstrom et al. [63]. Rather than complicated and expensive optimization algorithms, a simple heuristic is used that keeps track of accesses to each data block on each site and periodically moves data to the site where it is accessed most without causing load imbalance across sites. The choice of the data movement interval is critical to the performance of this method: excessively large values will prevent the system from responding to workload changes in time while excessively small values will result in oscillation of data between sites and increase the total transmission cost. The right interval value is often application specific and varies for different access patterns.

In general, existing data allocation algorithms were designed for traditional distributed databases; the case for a back-end distributor in a cluster-based Internet infrastructure

differs from the traditional cases in the following ways. The goal of such a distributor is to maximize the cache hit ratio in the main memories of cluster nodes or to minimize disk accesses; therefore, the data to be distributed is in-memory data rather than the entire database. Unlike in a geographically distributed database, queries are not naturally bound to particular nodes because each node is capable of processing any query and a front-end distributor may direct queries to nodes in various ways. Many existing data allocation algorithms take statistical query distribution as input, rather than treat query distribution as a variable in finding the final solution.

1.2 Contributions

The main contributions of my dissertation on high availability and scalability for cluster-based Internet infrastructures are:

1. It addresses the availability issues in the data storage by failure isolation, which is achieved by a combination of novel designs in data distribution and metadata replication.

2. It presents evaluation of the failure isolation approach by statistical analysis on existing systems and performance measurement on a prototype implementation.

The main contributions of my dissertation on high scalability and cost-effectiveness for cluster-based Internet infrastructures are:

1. It helps understand the challenges and possibility of a good management strategy for content cache of Internet application servers, which strives to maximize effective cache capacity and minimize synchronization cost.

2. It presents the design and evaluation of an affinity-based management (ABM) system for clustered in-memory databases as the content cache for Internet application servers; the goal of the management is achieved by a novel combination of two-stage execution, vertical replication and horizontal fragmentation.

2 Design of the island-based file system[1]

Maintaining availability in the face of failures is a critical requirement for Internet services. For example, the downtime at sites like Ebay and Etrade could directly translate to decrease in sales. I study a new approach to maximizing availability of the back-end storage systems for those services.

There are two complementary approaches to maximizing availability. First, I can use redundancy to maintain complete availability in the face of a small number of failures; second, I can try to *isolate* failures in order to serve as many requests as possible even though some cannot be served. These approaches are complementary, since I can use redundancy to mask the first few failures, and then use isolation to cope with any additional failures.

I describe in this chapter an approach to cluster file system design that provides failure isolation. I use the percentage of requests that succeed despite the failure of one or more servers as the availability metric; my goal is to maximize this percentage. I divide the nodes in the system into groups called *islands*. An island might be a single node, or it might be a group of nodes that use redundancy within the island to mask failures. In either case, island-based design strives to serve as many client requests as possible when one or more islands have crashed or are unavailable.

The main idea underlying island-based design is the *one-island principle*: as many file system operations as possible should require the participation of exactly one island. The one-island principle provides good failure isolation because each island can function independently of other islands' failures. In other words, the failure of 1 out of n islands in the island-based file system causes only 1/n of accesses to fail. In addition to its availability advantage, the one-island principle allows island-based systems to scale efficiently with the system and workload sizes because communication and

[1]A subset of the content in this chapter has been included in a paper [93] published in the Proceedings of 4th USENIX Windows Systems Symposium, August 2000.

synchronization across islands are reduced.

My motivation of failure isolation is analogous to the motivation of fault containment in Hive [38]. Hive, an operating system for large-scale shared-memory multiprocessors, attempts to "contain" a failed part so that it does not bring down other parts.

The target application of the island-based file system is the data storage for those Internet services that prefer to serve as many clients as possible rather than to go entirely offline when partial failures are present, that are medium to large scale, e.g. tens to hundreds of PC's connected by commodity local area networks such as Ethernet, and that expect occasional node failures and network partitions. Examples of such services include email, Usenet newsgroup, e-commerce, web caching, and so on.

I evaluated the island-based design by statistical analysis of the access patterns of existing systems. The results show that the partial availability provided by the island-based file system is useful to Internet services because a temporary partial failure can be made unnoticeable to the majority of clients. In one example, if 1 out of 32 islands is down for an hour, I expect that 93.8% clients during that hour will not notice the temporary partial failure. On average 99.8% of operations involve a single island and hence do not require communication or synchronization across islands.

I implemented a prototype of the island-based file system called *Archipelago* on a cluster of PCs running Windows NT 4.0 connected by Ethernet. The measurement of micro benchmarks shows that Archipelago adds little overhead to NTFS and Win32 RPC performance; the measurement of operation mixes based on NTFS traces shows a speedup of 15.7 on 16 islands.

2.1 Analytical model for data loss

In this section, I shall compare the permanent or temporary data loss in the island-based file system in case of partial failures to that of strawman's cluster file systems (CFS). I model the data loss due to independent storage server failures in comparable configurations of the island-based file system and CFS, see Figure 2.1 for an example. At

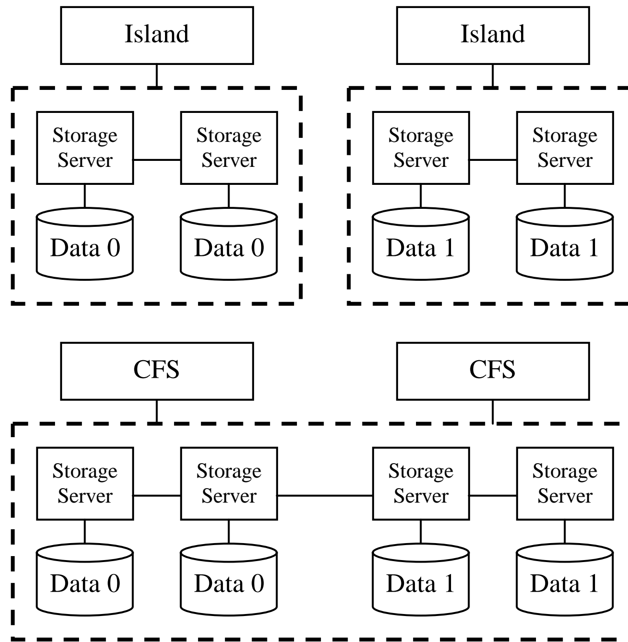

Figure 2.1 Mirrored configurations in data loss models. Storages with the same labels contain the replicas of the same data.

this point I am assuming that the one-island property is achievable. The design of a file system with such a property will be discussed later in this chapter.

I make the following assumptions in the analytic models. Data is randomly distributed across storage servers at block, file or other granularity in CFS, and across islands at directory granularity in the island-based file system. The CFS model does not replicate ancestor directories; therefore, the inaccessibility of a directory implies the inaccessibility of all its descendents. I also assume whole file accesses, i.e. the inaccessibility of a part of a file causes the whole file to be counted as lost. In a model with s storage servers, there are a root directory and s sub directories in the root directory. Each sub directory is a complete tree of height h. This is a conservative assumption because as the hierarchy gets more irregular, more files will have longer pathnames and hence have more chances to be inaccessible in CFS. Each directory has d sub directories and f files, and the directory itself has a fixed size equal to the block size bs, hence fits in a single server. Each file also has a fixed size fs. I ignore the impact of lost *inodes* in CFS, i.e. I assume that they are replicated everywhere.

I compare the data loss ratios in the island-based file system and CFS under various redundancy schemes, which are based on the non-redundant model below.

2.1.1 Non-redundant model

In this model, each island in the island-based file system runs on a single storage server. With the failure of 1 out of s servers, the non-redundant island-based file system permanently or temporarily loses $\frac{1}{s}$ data, according to the self-contained property of islands. I compute the data loss ratio with the failure of 1 out of s storage servers in non-redundant CFS as follows.

The amount of data in a tree of height i (a tree with a single node is of the height 0) is

$$T(i) = (\sum_{0 \le k \le i} d^k) \cdot bs + (\sum_{0 \le k < i} d^k) \cdot fs = \frac{d^{i+1} - 1}{d - 1} \cdot bs + \frac{d^i - 1}{d - 1} \cdot f \cdot fs .$$

For a file to be accessible during the partial failure, none of its $\left\lceil \frac{fs}{bs} \right\rceil$ blocks can be on the failed server. If data is randomly distributed across servers at block granularity, the probability for a block to be on the failed server is $\frac{1}{s}$. Therefore, the expected probability of a file being inaccessible is $F_{block} = 1 - (1 - \frac{1}{s})^{\left\lceil \frac{fs}{bs} \right\rceil}$. If data is randomly distributed across servers at file or larger granularity, the probability of a file being inaccessible is $F_{file} = \frac{1}{s}$.

The expected amount of data loss in the tree of height i with the failure of 1 out of s storage servers is

$$L(s,i) = \frac{1}{s} \cdot T(i) + (1 - \frac{1}{s}) \cdot (d \cdot L(s, i-1) + f \cdot F \cdot fs) .$$

That is, if the root of a tree happens to be stored in the failed server (with the probability $\frac{1}{s}$), the whole tree will be inaccessible; otherwise, (with the probability $(1 - \frac{1}{s})$) the

amount of data loss will be the sum of the expected amount $L(s, i\text{-}1)$ of data loss in each of the d sub trees plus the expected number $f \cdot F$ of lost files times the file size fs. By expanding $T(i)$ and F in the equation of $L(s, i)$ above, we get the following iterative equations of $L(s, i)$:

$$L_{block}(s,i) = (1-\frac{1}{s}) \cdot d \cdot L_{block}(s,i-1) + s \cdot (\frac{d^{i+1}-1}{d-1} \cdot bs + \frac{d^{i}-1}{d-1} \cdot f \cdot fs) + (1-\frac{1}{s}) \cdot f \cdot fs \cdot (1-(1-\frac{1}{s})^{\left\lceil \frac{fs}{bs} \right\rceil}) \text{ and}$$

$$L_{file}(s,i) = (1-\frac{1}{s}) \cdot d \cdot L_{file}(s,i-1) + s \cdot (\frac{d^{i+1}-1}{d-1} \cdot bs + \frac{d^{i}-1}{d-1} \cdot f \cdot fs) + (1-\frac{1}{s}) \cdot f \cdot \frac{1}{s} \cdot fs \, .$$

By solving the iterative equations of $L(s, i)$ above, we get the data loss in a directory tree of height h with the loss of 1 out of s servers as follows:

$$\begin{aligned} L_{block}(s,h) = {} & \frac{d^{h} \cdot (d \cdot bs + f \cdot fs) \cdot Q(1-\frac{1}{s}, h+1)}{s \cdot (d-1)} \\ & - \frac{(d + f \cdot fs) \cdot Q(d \cdot (1-\frac{1}{s}), h+1)}{s \cdot (d-1)} \\ & + Q(d \cdot (1-\frac{1}{s}), h) \cdot (1-\frac{1}{s}) \cdot f \cdot fs \cdot (1-(1-\frac{1}{s})^{\left\lceil \frac{fs}{bs} \right\rceil}) \end{aligned} \quad \text{and}$$

$$\begin{aligned} L_{file}(s,h) = {} & \frac{d^{h} \cdot (d \cdot bs + f \cdot fs) \cdot Q(1-\frac{1}{s}, h+1)}{s \cdot (d-1)} \\ & - \frac{(d + f \cdot fs) \cdot Q(d \cdot (1-\frac{1}{s}), h+1)}{s \cdot (d-1)} \\ & + Q(d \cdot (1-\frac{1}{s}), h) \cdot (1-\frac{1}{s}) \cdot \frac{1}{s} \cdot f \cdot fs \end{aligned} \quad ,$$

where $Q(x, y) = \frac{x^{y}-1}{x-1}$.

With the loss of 1 out of s servers, the amount of data loss in a system with u sub trees of height h in the root directory is

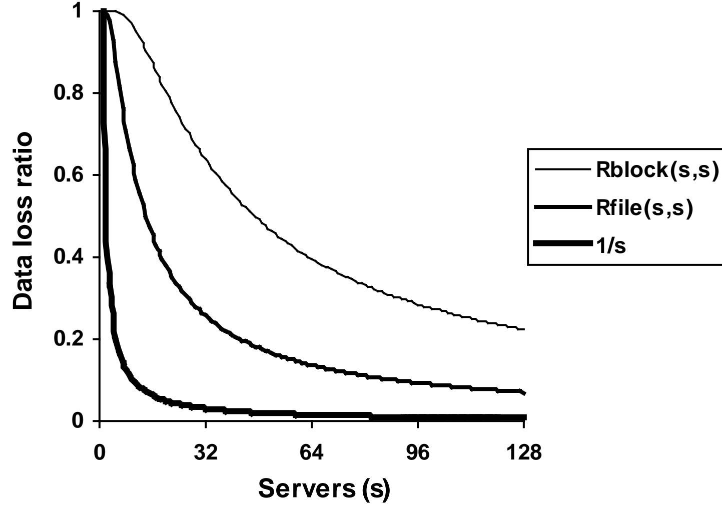

Figure 2.2 Data loss ratios in the non-redundant island-based file system (1/s) and CFS with block or file granularity ($R_{block}(s,s)$ or $R_{file}(s,s)$) with the loss of 1 out of s storage servers.

$$TL(s,u) = \frac{1}{s} \cdot u \cdot T(h) + (1 - \frac{1}{s}) \cdot u \cdot L(s,h) .$$

The data loss ratio is $R(s,u) = \frac{TL(s,u)}{T(h) \cdot u}$. I choose a set of typical parameters based on previous studies of file system contents [46] [43]: *h=8, d=2.5, bs=4096, f=10, fs=98304, u=s*. That is, on average, each server stores 2542 directories and 10166 files, or about 1 GB data.

Figure 2.2 shows the data loss ratios in non-redundant the island-based file system ($\frac{1}{s}$) and CFS ($R_{block}(s,s)$ and $R_{file}(s,s)$) as a function of *s*. CFSs with data distribution granularity larger than file will have the same data loss ratio as $R_{file}(s,s)$ since only accesses to whole files are considered in this model. The figure shows that data loss in CFS can be reduced by using larger granularity. Unfortunately, most existing CFSs are built on top of shared virtual disks [21] [1] [5]; physical block placement in the virtual disks are often made transparent to the file system layer; therefore, CFSs do not have control on the granularity for data distribution. The island-based file system reduces data loss by replicating ancestor directories across servers as well as by using directory

granularity.

I also analyzed the sensitivity of *R(s,u)* to other parameters within practical ranges; the results show that *R(s,u)* increases as *h* (height of the tree), *d* (number of sub directories per directory), *f* (number of files per directory) or *fs* (file size) increases, and decreases as *bs* (block size) increases. With the failures of *x* servers, the island-based file system loses $\frac{x}{s}$ data and CFS loses $x \cdot R(s,u)$ data.

2.1.2 Redundancy schemes with grouping

Many existing redundant storage systems are divided into groups and data redundancy is applied within groups, but not across groups. It results either from the nature of the redundancy scheme, such as mirroring pairs, or from performance optimization, such as RAID-5 striping groups [5] [44]. A CFS running on a shared storage system with *s* redundancy groups can be compared to the island-based file system with *s* islands, each of which runs on a single redundancy group of the same scheme. See Figure 2.1 for a mirrored example. If I treat each group as a single server, I can use the non-redundant model to compute the data loss with the failure of a group in both systems. Since the mean time to failure of a group is reduced by the same factor in both systems, the ratio of data loss in CFS to data loss in the island-based file system is still $R(s,u) \cdot s$.

2.1.3 Redundancy schemes without grouping

In general, the island-based file system can achieve as high reliability as CFS with an arbitrary redundancy scheme by being configured as a single island with a storage system of the same redundancy scheme. The actual gain in reliability needs to be analyzed on a case-by-case basis. Below I compare the data loss in the island-based file system running on mirrored storage with that of CFS running on shared chained-declustering storage [7].

In this model, each system has $2 \cdot s$ storage servers. In the island-based file system, each of *s* islands runs on top of 2 mirrored servers; in CFS, the replica of the data in each server is evenly distributed to the other servers. With the failures of 2 out of $2 \cdot s$ servers,

the island-based file system loses $\frac{1}{s}$ data with the probability $\frac{1}{s-1}$ (if the 2 failed servers happen to be in the same island); CFS loses $\frac{1}{s^2}$ data storage with the probability 1. Interpreting *R(s,u)* as the data loss ratio with the loss of $\frac{1}{s}$ storage, the expected data loss ratios of the island-based file system and CFS are $\frac{1}{s-1} \cdot \frac{1}{s}$ and $R(s^2, s)$, respectively. It can be proven that the data loss ratio of CFS running on mirrored storage is $\frac{1}{s-1} \cdot R(s,s)$ and $R(s^2, s) > \frac{1}{s-1} \cdot R(s,s)$. That is, chained declustering has a higher expected data loss ratio than mirroring.

The models above show that, in comparable redundancy schemes of the island-based file system and CFS, with the failures of the same number of servers, the island-based file system has a significantly lower data loss ratio than CFS, at the cost of replicating ancestor directories. In particular, the models show that mirroring with one-island property achieves higher availability than mirroring alone, which achieves higher availability than chained declustering. If the data is permanently lost, the island-based file system will cause a lower cost for reconstructing the data at application level or manually; if the data loss is only temporary, the island-based file system maintains a higher availability.

2.1.4 Data loss versus storage overhead

In this section, I study the expected data loss ratio in various redundancy schemes as a function of the overhead cost in storage capacity. Let D be the total number of storage servers with data (not including extra check storage), G be the number of data storage servers in a group (not including extra check storage), C be the number of check storage servers in a group, MTTF be the mean time to failure of a single storage server, and MTTR be the mean time to repair of a single storage server. I consider non-redundant storage and single-error repairing RAIDs below.

As derived in [45], the mean time to failure of the entire system, or RAID, is

$$MTTF_{RAID} = \frac{MTTF^2}{(D + \frac{D \cdot C}{G}) \cdot (G + C - 1) \cdot MTTR}.$$

The probability of a failure of the entire system is

$$F_{RAID} = \frac{MTTR}{MTTF_{RAID}} = \frac{MTTR^2 \cdot (D + \frac{D \cdot C}{G}) \cdot (G + C - 1)}{MTTF^2}.$$

The probability of a failure of a non-redundant system is $F_0 = \frac{MTTR \cdot D}{MTTF}$.

In case of failure, the ratio of lost data storage to total data storage is $A = \frac{G}{D}$.

The expected data loss ratio of IFS running on top of a RAID is

$$R_{IFS-RAID} = A \cdot F_{RAID} = \frac{G}{D} \cdot \frac{MTTR^2 \cdot (D + \frac{D \cdot C}{G}) \cdot (G + C - 1)}{MTTF^2} = \frac{MTTR^2 \cdot (G + C) \cdot (G + C - 1)}{MTTF^2}$$

.

The expected data loss ratio of IFS running on top of a non-redundant storage is $R_{IFS-0} = A \cdot F_0 = \frac{MTTR \cdot G}{MTTF}$.

The expected data loss ratio of CFS running on top of a RAID is

$$R_{CFS-RAID} = R(\frac{1}{A}, D) \cdot F_{RAID} = R(\frac{D}{G}, D) \cdot \frac{MTTR^2 \cdot (D + \frac{D \cdot C}{G}) \cdot (G + C - 1)}{MTTF^2}.$$

The expected data loss ratio of CFS running on top of a non-redundant storage is

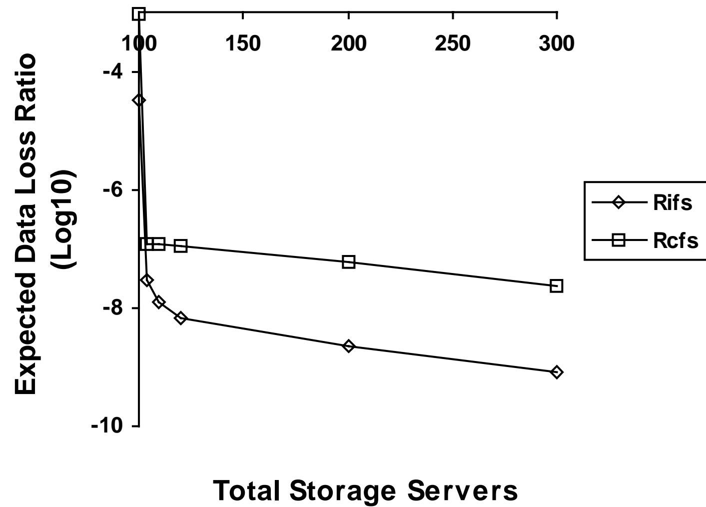

Figure 2.3 Expected data loss ratios in IFS and CFS running on non-redundant storage, RAID-5 storage (with group sizes 25, 10 and 5 respectively), mirrored storage and triple storage.

$$R_{CFS-0} = R(\frac{1}{A}, D) \cdot F_0 = R(D, D) \cdot \frac{MTTR \cdot D}{MTTF}.$$

	Non-redundant	RAID-5 (1)	RAID-5 (2)	RAID-5 (3)	RAID-1 or Mirror	Tripple
D	100	100	100	100	100	100
G	1	25	10	5	1	1
C	0	1	1	1	1	2
Total Storage	100	104	110	120	200	300
MTTF	30,000 hours					
MTTR	1 hour					

Table 2.1 Parameters for expected data loss ratios.

Table 2.1 shows the parameters for expected data loss ratios. The parameters used in R(s,u) are the same as in previous sections. Figure 2.3 shows the expected data loss ratios of IFS and CFS running on various redundancy schemes defined in Table 2.1. The results show that increasing the amount of redundancy reduces data loss ratio for both CFS and IFS and that IFS has lower data loss ratios than CFS in all redundancy schemes. The reduction in data loss ratio attributed to IFS is shown to be more cost-effective than the reduction attributed to increased redundancy. For example, the data loss ratio of IFS on

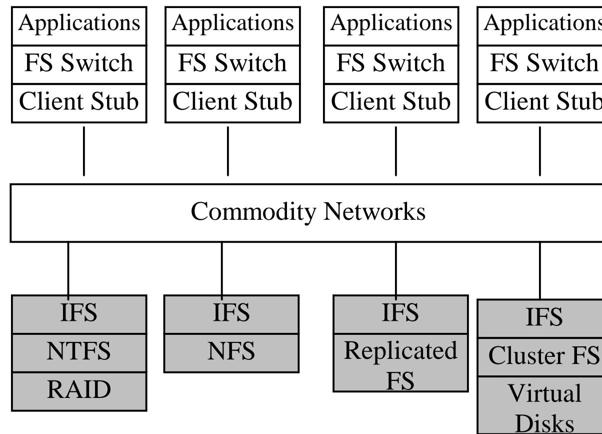

Figure 2.4 Overview of the island-based file system (IFS). Shaded boxes are islands or servers and non-shaded boxes are clients.

RAID-5 with group size 5 is 16 times lower than that of CFS on the same redundancy scheme, while the data loss ratio of CFS on mirrored storage, which requires 1.7 times more total capacity than RAID-5 with group size 5, is only 1.8 times lower than that of CFS on RAID-5 with group size 5.

2.1.5 Partial availability for applications

If a client application needs to access multiple directories or files and any of the directories or files is lost, the application will fail as a whole. The availability of the island-based file system with partial failures depends on the number n of distinct directories applications access. For example, with the failure of 1 out of s islands in non-redundant the island-based file system, the expected probability that an application will *not* be affected is $(1-\frac{1}{s})^n$. The availability of CFS depends on the accessibility of the directories and files applications access and all their ancestor directories; therefore, the partial availability of CFS is always no higher than that of the island-based file system.

The challenge is how to evenly, automatically and dynamically partition a single large file system into a cluster of independent components without causing inconsistency across components in the face of partial failures.

2.2 Island-based design

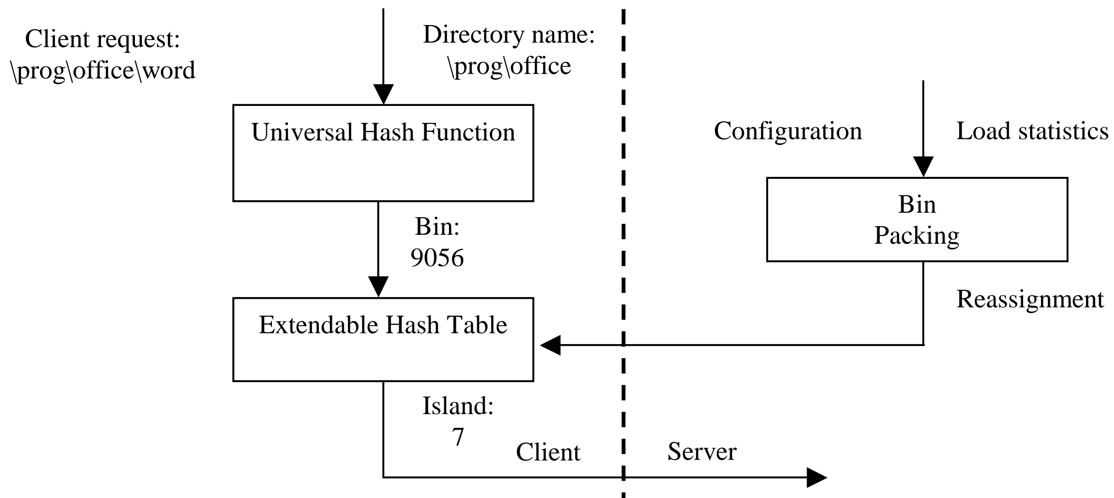

Figure 2.5 Hash-based data distribution in the island-based file system.

Figure 2.4 gives an overview of the island-based file system in a typical configuration. An island consists of a server process running on top of a local file system. Client applications view the island-based file system as a single system and access it through local file system switches and stubs. Islands and clients are connected by commodity local area networks such as Ethernet.

Let me examine two important issues in island-based design, data distribution and metadata replication.

2.2.1 Hash-based data distribution

I designed a new data distribution strategy for the island-based file systems: data is distributed to islands at *directory granularity* by *hashing* the *pathnames* of the directories to island indices.

I choose directory granularity rather than block, file or sub tree granularity because most file system operations involve a single directory and hence satisfy the one-island principle, and directories are finer grained than sub trees so as to allow load balance.

I choose hashing instead of recursive name lookup because hash functions can be computed on the client machines without contacting any servers. I choose to hash

pathnames instead of low-level integer identifiers such as inode numbers because pathnames are the only information that a client can possibly have without contacting any servers, and they are independent of internal representations of file systems.

Clients determine which island to contact for a directory or a file in that directory by hashing the full pathname of the directory to an island index in two steps: first, hashing the pathname to a *bucket* (an integer) with a universal hash function called H_3 [10]; second, hashing the bucket to an island index with an extendible hash table [11]. The universal hash function used in the island-based file system is a consistent mapping from a variable-length character string to a 32-bit integer and has good distribution in the output space independently of the input space. A universal hash function can evenly distribute an arbitrary set of directories to buckets; however, it does not have control on the workload distribution across directories; therefore, an additional level of indirection is necessary to handle the hot spots and dynamic load changes. A subset of the 32 bits is used as the index to the extendible hash table and the table entries are island indices. As load imbalance across islands increases or islands are permanently added or removed during system reconfiguration, the table entries are reassigned to islands to rebalance the load using a bin-packing algorithm. The reassignment is made *monotonic*, i.e. each island either loses data or gains data, but not both. Therefore, only a minimal amount of data needs to be migrated between islands. Section 2.2.3 will give more details about the rebalance procedure, such as the update of hash tables in islands and clients. Figure 2.5 gives an example about the hash-based data distribution in the island-based file system.

Inside each island, I store directories in a *skeleton hierarchy*. I call the file system running inside each island the *internal file system*. An internal file system can be an instance of any existing file system such as a local file system, a replicated file system or a cluster file system. The skeleton hierarchy in an island contains the directories hashed to this island index and their ancestor directories up to the root, and is stored in the unmodified internal file system as a normal tree. This way, islands can function independently of others' failures and I can leverage the functions of the internal file systems. Figure 2.6 gives an example about the skeleton hierarchy. The consequence of

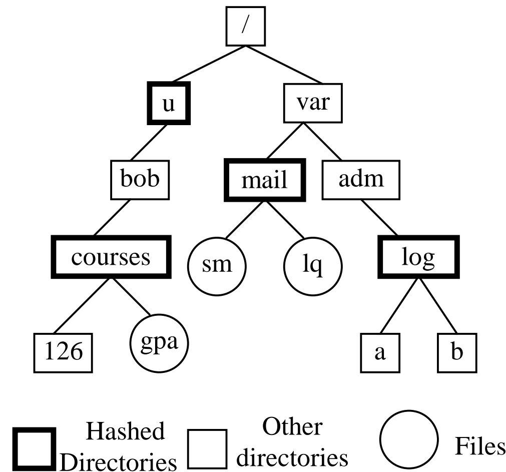

Figure 2.6 Skeleton hierarchy. This is an image of the internal file system in an island that the highlighted directories are hashed to. Other directories are the ancestor or child directories of the hashed directories.

storing data in skeleton hierarchies is the replication of certain metadata or directory attributes.

2.2.2 Usage-based metadata replication

Although it might not take much space to replicate metadata across islands, updates to replicated metadata will have to be done in all replicas and hence violate one-island principle. Therefore, I use a *usage-based* replication scheme in the island-based design, i.e. I replicate metadata that is more frequently used to a higher degree.

To help me explain the usage-based metadata replication, I introduce two terms, *directory owner* and *parent owner*. The *directory owner* of a directory is the island to which the directory is hashed. The *parent owner* of a file or directory is the directory owner of its parent directory. A file resides in exactly one island, its parent owner. A directory will be replicated in its parent owner, in its directory owner and in all the parent owners of its descendent directories. Therefore, the replication scheme can automatically adapt to the usage of the metadata. In particular, files are not replicated across islands; leaf directories are replicated in exactly two islands (the parent owner and the directory owner); intermediate directories are replicated in various numbers of islands; and the root directory is replicated in all islands. Figure 2.7 gives examples about the replication.

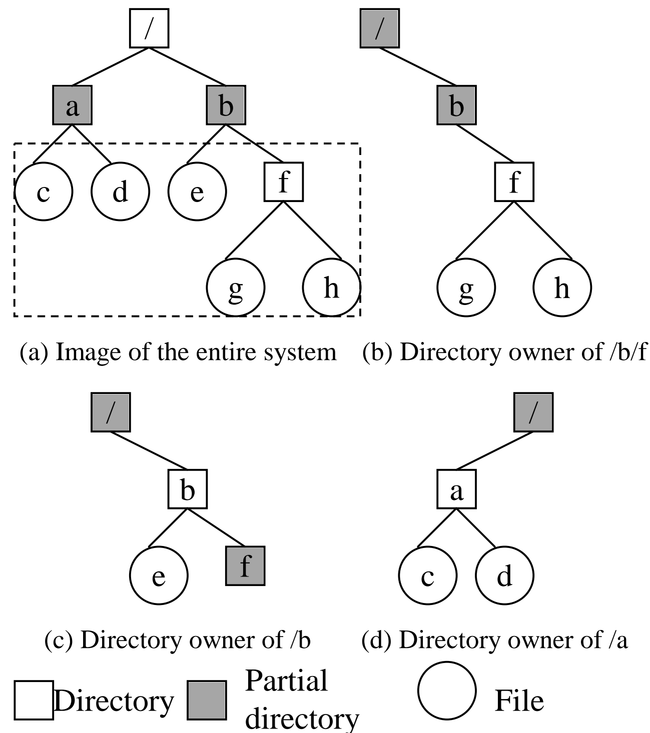

Figure 2.7 Usage-based metadata replication. Figure (a) without the directories and files inside the dashed rectangle is the image of the directory owner of root directory /. (b) (c) and (d) are the images of the internal file systems in three other islands. / is replicated in all islands. /b is replicated in its parent owner (a), directory owner (c) and the directory owner of its sub directory /b/f. /a and /b/f are replicated only in their parent owners and directory owners because they are leaf directories. Shaded directories in the figure represent replicas that contain only attributes and partial contents or no contents.

However, only some directory *attributes*, not the directory *contents*, need to be replicated. Directory contents are the lists of names and addresses of sub directories and files. Only the directory owner keeps a complete copy of the directory contents; other replicas have partial contents or no contents. The partial contents stored in other replicas are determined by the hash functions for data distribution. Changes to directory contents, e.g. adding or removing files, need to be done in the directory owner only. Directory attributes include name, size, security, time stamps, read-only tag, compressed tag, etc.. Changes to directory attributes will, however, affect multiple replicas.

I want to replicate only those attributes that are needed when a descendent of the directory is looked up. I divide directory attributes into two categories, *static* attributes

and *dynamic* attributes, based on their access patterns. A static attribute is more frequently read than written, and a dynamic attribute is more frequently written than read. Attributes such as name, security labels, read-only tag and compressed tag are static. Attributes such as size and time stamps are dynamic. I replicate the static attributes and do not replicate the dynamic attributes. Static attributes can be read in any one of the replicas; updates to static attributes are broadcast to all replicas; the overhead of updates is acceptable since static attributes rarely change. Chapter 3 discusses the consistency protocol in more detail. Dynamic attributes are read and written in a single island, the directory owner.

2.2.3 Reconfiguration and rebalance

When load imbalance across islands exceeds a threshold as the system ages or when islands are permanently added to or removed from the system, hash table entries need to be reassigned to islands and data needs to be migrated between islands to rebalance the load. (Note that rebalance will not be invoked when islands leave or join the system due to failures and recoveries.) I describe the protocol in details below.

One island is designated as the coordinator in each rebalance. Each island has a unique identifier ranging from 0 to n-1, where n is the number of islands in the current configuration. In order to reduce administrative complexity in identifying the islands to be added or removed, the coordinator is chosen with the following rules. If no islands are added or removed during a rebalance, island 0 is the coordinator. Only the highest numbered islands can be removed from or added to the system during a reconfiguration. (If a system administrator wants to remove an island other than the highest numbered island, the islands need to be re-numbered by changing the hash tables.) If k islands (numbered n-k through n-1) are to be removed, island n-k will be the coordinator; if k islands (numbered n through n+k-1) are to be added, island n+k-1 will be the coordinator. Therefore, given the current configuration and its own identifier, a coordinator automatically knows which other islands are to be added or removed without input from system administrators.

The rebalance is committed in two phases. Each configuration is associated with a

version number, and each committed rebalance increases the version number by 1. First, the coordinator attempts to collect workload statistics from all islands, each island logs a "preparing rebalance" message in permanent storage. If any island is inaccessible, the coordinator aborts and notifies the system administrator; otherwise, the coordinator constructs a new hash table that rebalances the workload across the islands in the new configuration, and publishes the new configuration file, including the new hash table and increased version number, at a well-known location. Second, the coordinator sends a "committing rebalance" message to all islands including the added or removed ones, and then all islands load the new configuration file from the well-known location.

Once the rebalance is committed, each island checks whether it is the source or destination of the monotonic data migration by comparing the old and new hash tables. The destination islands simply log a "rebalance completed" message and return to normal state. Each source island forks a thread, called the *migrator*, to migrate the directories that are no longer hashed to its own index to their new owners. Migration can be done in parallel in all islands since I need not worry about an island becoming full during the migration. The migration will be resumed as necessary with the information recorded in the log, should an island crash during the rebalance. When it finishes, the migrator logs the "rebalance completed" message and exits.

There are two forms of migration during the rebalance: background migration and on-demand migration. The migrators move data in the background. If a new owner receives a request for a file that has not been migrated yet, it issues a request to the old owner to move the file immediately. I call this *on-demand migration*. This is a better approach than waiting for the migrator in the old owner to initiate the movement because waiting could lead to deadlock. However, on-demand migration can cause three types of race conditions: (1) the migrator could not find a file because that file had already been migrated on demand; (2) the migrator tries to move a file but the file has already been created in the destination island by on-demand migration; (3) a file could not be moved because it was in use by another thread. To cope with the on-demand migration, the migrator repeatedly scans the internal file system, detects the race conditions and

temporarily skips the suspect directories and files. The same error detection scheme applies to situations where destination islands crash during the rebalance. Client accesses during migration will *not* directly cause race conditions because clients are never allowed to access files or directories in their old owners once the rebalance is committed. They can only access files or directories in their new owners *after* the files or directories have been migrated either in background or on demand.

The hash table is replicated on all clients' machines as well as in all islands, along with the version number. The table size is proportional to the number of islands. Clients' copies of the hash table are updated *lazily*: each request from a client carries the client's current version number, and a client will be asked to load the new configuration file from the well-know location when its version number is found to be out of date. To avoid single point of failure, the well-know location can be mirrored inside or outside the island-based file system, or both. Islands act as clients when they communicate with each other; therefore, the same scheme applies to islands that crash or disconnect from the coordinator before they receive the "committing rebalance" messages: upon first contact to any updated island, the out-of-date islands are forced to load the new configuration file.

The rebalance procedure will be invoked when the load imbalance exceeds a threshold so that no island could become full during normal operations. In fact, the rebalance procedure does what the system administrators manually do today, e.g. adding or removing file servers appliances and remounting partitions. Most administrators can manage to do it with tolerable or unnoticeable impact on the clients. Therefore, I expect that a reasonable threshold can be set in the island-based file system so that the rebalance occurs at a non-disruptive frequency, e.g. once every month.

2.2.4 Other design issues

The island-based file system inherits most functions from its internal file systems, such as metadata structures, disk allocation, I/O scheduling, caching, locking, security, recovery, etc.; therefore, I are not concerned about all the low-level details in file system design and implementation. However, certain functions in internal file systems need to be

extended to adapt to a distributed environment. The consistency protocol used in the island-based file system is discussed Chapter 3.

2.2.4.1 Symbolic links and renaming directories

Symbolic links in the island-based file system are implemented as files containing a pathname to a file or directory. Symbolic links to files are easy to manage because they cause at most a redirection from the owner of the symbolic link to the owner of the target file. However, a pathname with symbolic links to directories will not be hashed to the proper island. To solve this problem, I replicate all symbolic links to directories in all islands. Upon receiving a request for a file or directory that is not found locally, an island checks whether any components of the pathname are symbolic links to directories, without contacting other islands. If none of them is, it returns an error; otherwise, it redirects the request to the real owner after resolving the symbolic links. This is done by communications between islands and is transparent to the clients. Similar to the replication of static directory attributes, the replication of symbolic links to directories does not require much space, and the creation, modification and deletion of symbolic links, which will involve all islands, are rare operations (Section 4).

Renaming a directory in the island-based file system is an expensive operation because all the subdirectories below the renamed directory are likely to be hashed to different islands. I try to hide the latency of such an operation by using a symbolic link and a thread similar to the migrator described in Section 2.2.3. A special symbolic link is created with the new directory name, pointing to the old directory, a migrator thread is forked, and then the rename operation returns as if it is completed. The migrator recursively moves subdirectories and files from their old owners to their new owners in the background. If a request arrives for a file that has not been moved yet, the special symbolic link in the pathname will be resolved and the file will be migrated on demand. If a directory is renamed again before the migration completes, accesses to this directory will require multiple special symbolic link resolutions. The special symbolic links will be removed after the migration completes.

2.2.4.2 Security and caching

I designed and implemented a security model in the island-based file system, using the security facilities available in existing file systems and communication protocols, namely access control lists, permission bits, authentication and impersonation. A client is authenticated with its credentials when a connection to an island is established. A thread is forked in an island upon each request from the client. The thread extracts the client's credentials from the authenticated connection and impersonates the client when it processes the request. In this way, file accesses in the request are checked with the client's credentials against the access control in the internal file systems.

Server-side caching is done in the internal file systems automatically. The island-based file system inherently provides locality by hashing, i.e. client requests will always be sent to the server that might have cached the requested data in memory, as long as rebalancing is not in progress. Most of the client-side caching protocols in previous work [8] [25] can be adopted in the island-based file system. I have not implemented a client-side caching protocol, but I do not expect the island-based design to add any difficulty to the implementation.

3 Consistency of replicated metadata[2]

As discussed in the previous chapter, in the island-based file system, a small portion of read-mostly metadata, e.g. directory attributes, is replicated across nodes to allow independent accesses to data in each node. As in any other systems where data replication and updates to replicated data are present, these systems face the challenge of keeping its replicated metadata consistent across nodes. Hazards could occur if the replication is not handled with care, because the structural integrity of metadata is critical for the system to function correctly and to recover successfully from possible failures, and because metadata tends to be shared by multiple clients more frequently than user data.

The following are two examples of possible hazards in a general distributed file system where directories are replicated across servers and clients are allowed to access any replicas:

1. An empty directory *a* is replicated in cluster servers 1 and 2; client *B* deletes directory *a* in server 1 and server 1 propagates the deletion request to server 2; simultaneously, client *C* creates a sub directory *d* in *a* in server 2 and server 2 propagates the creation request to server 1; the deletion is aborted in server 2 because *a* is not empty and the creation is aborted in server 1 because *a* no longer exists; in a consistent system, only one, not both, of the operations would abort.

2. A directory *a* with a file *b* is replicated in servers 1 and 2; client *C*, the owner of *a*, changes *a*'s permission from 700 to 755 (world-readable) in server 1 and server 1 propagates the change to server 2; client *D* successfully reads file *b* in server 1 but, shortly after, it gets a "permission denied" error when it tries to list the content of directory *a* in server 2. In a single system, client *D* is expected to have access to directory *a* as well after it successfully reads file *b*.

The hazards occur because the file system operations are not *serialized*, or clients observe

[2] An abbreviated version of the content in this chapter has been published as an extended abstract [94] in the Proceedings of 1st IEEE International Conference on Cluster Computing, November 2000.

the results of the operations in conflicting orders; the consequence is that the system no longer behaves in the same way as its single-system counterparts and start generating confusing or incorrect answers to clients' requests. Furthermore, the chance for such hazards is highly magnified when failures, such as server crashes and network partitions, are present.

My goal in maintaining the consistency of the replicated metadata is to minimize the efforts for porting applications from single, tightly-coupled and/or small-scale systems to large cluster-based environments. In particular, I want to eliminate as many hazards as possible that a cluster environment might introduce. Meanwhile, I do not want the consistency protocol to have an intolerable impact on the performance and scalability otherwise achievable in these two systems.

Both the island-based design and affinity-driven distribution offer an opportunity for strong consistency without sacrificing performance in common cases. Since they strive to reduce data sharing across nodes, the cost for maintaining consistency of shared data can potentially be reduced as well. Therefore, it is possible to achieve strong consistency for the small set of shared data while maintaining the overall performance and scalability of the system.

I discuss in this section how to design a robust and efficient protocol for the synchronization of operations on replicated data in the face of node failures and network partitions. Before discussing my contributions in detail, I present a brief overview of prior work on replication and consistency issues in file systems, databases and Internet applications.

3.1 Related work

File system replication and consistency issues have been studied in a wide variety of contexts. Consistency guarantees vary largely from system to system due to the differences in their system structures and replication models.

Wide-area distributed file systems such as Ficus [51], Coda [54] and Locus [53] employ

optimistic *one-copy* availability, in which any data may be updated as long as some copy, including the client cache, is available. Strong semantics such as serialization of operations on replicated data are traded for availability and performance in those systems. The systems choose to guarantee “eventually” consistent data instead, i.e. they allow temporary inconsistency and try to detect it, which must then be resolved by applications or users. (The exception is that Ficus can automatically reconcile conflicting updates to its directories.)

Harp [50] and Echo [52] use primary-copy scheme with logging for replication, where clients can access only the primary copy. Harp is able to guarantee the atomicity and serialization of updates with write-behind logging. Since it handles updates to data and metadata in the same way, it relies on in-memory logging and uninterruptible power supply (UPS) to reduce the overhead of the consistency protocol. In a recent distributed file system [55], the overhead is reduced by distributing load across servers and amortizing the costs of individual operations with file sessions.

In recent cluster file systems like Frangipani [1] and xFS [5], data redundancy is provided in the virtual block device layer, not in the file system layer. Locking scheme is used in the block device layer for consistency of data replicas. Since updates to data and metadata are handled in the same way in the block device layer, those systems typically use fast system area network such as ATM for aggressive communications across data replicas [7].

The replication model generalized from my systems is similar to the models in typical replicated databases [49] [50]. However, my consistency requirement differs, primarily because I do not require transactional semantics for file accesses or persistence for in-memory data. Therefore, I believe that a light-weight protocol can be designed for the model in my systems.

The Cluster-Based Scalable Network Services (SNS) [47] provide an architecture and programming model for building Internet services that are willing to trade consistency for availability. My approach is complementary to theirs in that, while their system can be

used for creating new, scalable Internet services on loosely-coupled clusters, I strive to make it easy to run existing applications, such as the well-adopted web servers [57], database servers [58] and database applications in a cluster as well as on a single machine.

The context of system structure and replication model in which my consistency protocol is considered differs from the contexts in previous studies. In the island-based file system, only certain directory attributes, but not the directory contents (the lists of names and addresses of sub directories and files) or files, are replicated across islands. The degree of replication varies by directories based on their usage, and changes dynamically as the usage changes. The contributions of this part of my work are the following:

1. I design a consistency protocol that offers stronger semantics than "eventual" consistency, and hence increases the likelihood that applications can be ported from single systems to cluster-based systems with few modifications.

2. The overhead of the consistency protocol is reduced by taking advantage of the data distribution strategies in the target systems and using a light-weight non-locking algorithm, rather than using additional hardware or fast network.

3. I take arbitrary sequences of failures into consideration and use a recovery procedure based on a finite state machine model to handle the failures. I check the correctness of the protocol by randomized failure injection into the implemented prototype.

3.2 Replication model

I define a few terms below to assist in generalizing the replication model in the island-based file system. For each replicated *object* (a set of attributes of a directory or the entire set of search keys), a particular node (the directory owner) is chosen as the *coordinator* of the global operations or updates on this object, or simply called the coordinator of the object. The copy of an object in its coordinator is called the *primary* copy and the other copies are called *secondary* copies. Any node (an island or a pre-executor) that has a copy of the object is called a *replica* of the object. Each operation *originates* from a

single replica. Each update must originate from the coordinator and be propagated to other replicas. All objects in a node are readable by operations originated from or propagated to this node.

3.3 Consistency protocol design

A strawman's approach to the consistency of replicated directories across replicas is to lock a directory before operating on it. Locking schemes, especially ones with multi-reader-single-writer locks, are a typical approach to the consistency on replicated data in general. To avoid deadlocks and to handle partial failures and network partitions, a locking scheme often needs to be used in combination with other mechanisms such as timeout [29], majority consensus [1] and/or versioning [15].

Unfortunately, such a scheme can seriously weaken the availability and scalability. In the island-based file system, since each operation implicitly involves recursive lookup and permission checking with the ancestor directories, the ancestor directories need to be locked for the operation as well. A lock on each directory requires at least two round-trip messages, acquiring the lock and releasing/revoking the lock, to and from the coordinator of the directory. Consequently, there will no longer be "one-island" operations in the island-based file system since almost every operation needs to contact multiple islands for locking involved directories. If I use a global lock for the entire system rather than a lock per replicated object, I can reduce the communication cost for locking, but I also reduce the parallelism offered by the cluster structure.

I use a novel combination of logical clock synchronization [23], two-phase commit [24], logging [14] and finite-state-machine-based recovery to serialize the updates while keeping the synchronization for one-island operations or read-only queries local. My methodology takes three steps. First, I guarantee that each update is atomic; second, I serialize updates and other operations in common cases; third, I ensure the serialization of updates during a recovery from failures.

3.3.1 Atomicity

The basic consistency guarantee my protocol offers is the *atomicity* of the updates, i.e.

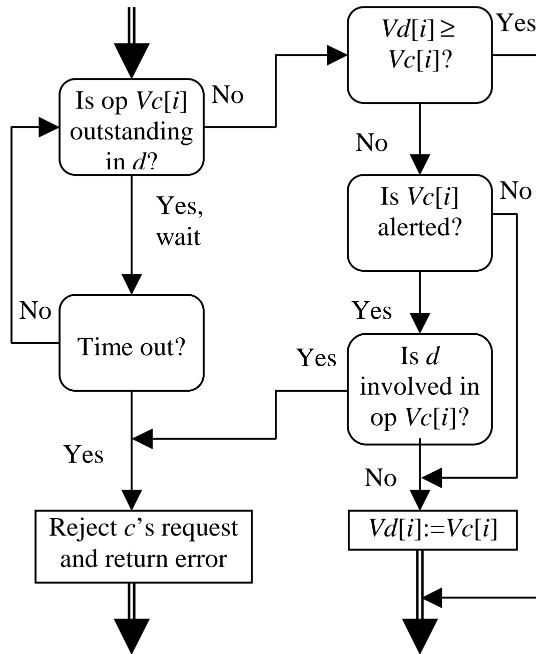

Figure 3.1 Synchronization of a client *c*'s clock *Vc*[*i*] with a coordinator *d*'s clock *Vd*[*i*], where *i* is a replica in the system. Op *Vc*[*i*] is the update that was originated from *i* and caused the clock in *i* to increase to *Vc*[*i*].

clients would never observe the intermediate state of any update. In other words, once a client observes the result of an update in a node, it would always observe the result of that operation in other replicas afterwards.

I use a vector of logical clocks for the atomicity of updates. Each coordinator has its *local* logical clock and each update coordinated by this node increases the clock by 1, or *generates* a new clock value. Each replica or client maintains a vector of all coordinators' clocks. Each request to a replica carries the sender's current clock vector for synchronization with the receiver's vector before the request is processed, and returns the receiver's vector to the sender after the request is completed. I say vector $V2$ is equally or more up-to-date than vector $V1$, or $V2 \geq V1$, if and only if $V2[i] \geq V1[i]$, for all $0 \leq i < n$, where n is the number of coordinators.

I maintain the following invariants:

1. The local commit of an update and the increase of the local clock are atomic in each coordinator, which is guaranteed with a local lock in that coordinator.

2. A coordinator does not release the new clock value to a client until it has notified all replicas of the operation, i.e. until the operation is either *outstanding* or *committed* in all replicas. This is guaranteed with a two-phase commit [24]: the coordinator notifies all replicas of the operation in phase 1, then locally commits the operation and updates the clock, and asks replicas to commit the operation in phase 2.

3. A request cannot be processed in a replica if the request carries a clock that is generated by an outstanding operation in that replica. Based on invariants 1 and 2, this invariant means that once a client observes the result of an operation in at least one replica, it will always observe the result of that operation in other replicas afterwards. This is guaranteed by the clock synchronization algorithm in Figure 3.1, which is an extension to Lamport's algorithm [23].

The three invariants above guarantee that a replica will never expose the intermediate state of any operation to clients. Invariant 2 ensures that synchronization in a replica for reads does not need communication with the coordinator, if no network partition is present.

I make an exception to invariant 2 to handle network partitions. If any replica is inaccessible due to either a node crash or network partition during phase 1 of the commit, the coordinator updates its clock with an *alerted* bit (part of the coordinator's clock) set. The alerted bit will be propagated together with the clock till it is reset. During the clock synchronization with a client, a replica must ask for a confirmation from the coordinator about its involvement in an alerted operation that it has not seen but the client has. If the coordinator crashed or disconnected from a replica after phase 1, the operation will be outstanding in the replica till the coordinator reconnects. This type of failure will be detected by a timeout in the clock synchronization (Figure 3.1). The alerted bit will be cleared once the nodes reconnect and all outstanding operations are either committed or aborted.

The pseudo C code for a global update is following:

```
//A global update op in coordinator d requested by client c
GlobalUpdate(c, d, op){
```

```
    //Synchronize c's clock vector Vc with d's clock vector Vd
    for each replica i do
          //Figure 3.1
          SynchronizeClock(Vc, Vd, i);
    //One global update at a time per coordinator
    EnterCriticalSection(GlobalUpdateSection);
    //Take a snapshot of the current clock vector in d
    EnterCriticalSection(ClockSection);
    Vd := CurrentClockVector;
    LeaveCriticalSection(ClockSection);
    //Clear the alert bit for this op
    Vd &= ~ALERT;
    //Increase d's clock by 1 for this op
    Vd[d] := Vd[d] + 1;
    //Have all involved replicas log the op in memory or
    //mark the op as outstanding in all involved replicas
    for each involved replica j do
          SecondaryLog(j, d, Vd[d], op);
    if logging successful in all involved replicas then
          state := COMMIT;
    else
          //Set the alerted bit in d's clock
          Vd[d] |= ALERT;
          state = PARTIAL_COMMIT;
    //Log the op on disk in coordinator d
    PrimaryLog(Vd[d], state, op);
    //Make the local commit and update of clock atomic
    EnterCriticalSection(ClockSection);
    //Locally execute the op
    PrimaryExecute(op);
    if local execution successful then
          state := COMMIT;
    else
          state = ABORT;
    //Update d's clock while keeping the alerted bit if it is set
    CurrentClockVector[d] := Vd[d] | (CurrentClockVector[d] & ALERT);
    LeaveCriticalSection(ClockSection);
    //Globally commit or abort the op, depending on the state
    for each involved replica j do
          SecondaryExecute(j, d, Vd[d], op, state);
    //Leave the critical section
    LeaveCriticalSection(GlobalUpdateSection);
}
```

3.3.2 Serialization

The higher-level consistency guarantee my protocol offers is the *serialization* of the updates, i.e. clients observe the results of all operations in the same order in all replicas. All the updates on the same object are coordinated by the same node, hence can be serialized by a local mutex in that node, unless a replica failed.

The serialization in case of failures is guaranteed by *write-ahead logging* [14]. The

coordinator always writes a record with its clock vector to stable storage before it locally commits an update. Only after the operation is committed in all replicas, the record can be removed from the log.

When a replica b is reconnected, the coordinator a sends to b a list of operations that involved b but have not been committed on b. The operations will be committed in b in ascending order of their clocks ($V[a]$'s), i.e. in the same order as if b had not been disconnected from a. Note that b needs not know about the local operations on the same objects that were done while it was disconnected from a because it would not have known those operations even if it had not been disconnected.

If a client *thread* issues at most one request at a time, all the operations by the same thread are serializable even if a replica failed. Consecutive operations by the same thread are guaranteed to have ascending clock vectors because, with the logical clock synchronization (Figure 3.1), the clock vectors in all replicas and clients never decrease and always increase upon updates, even with network partitions. Therefore, recovering replicas are able to commit the operations by the same client thread in the same order as if it had not failed, by sorting the operations from all coordinators in the ascending order of their clock vectors.

If two clients interact with each other by accessing the same objects, then the operations by the two clients are serializable in the face of failures. For example, if two clients, $c1$ and $c2$, access the same object at time $t1$ and $t2$ ($t1<t2$) and receive the clock vectors $V1$ and $V2$ respectively, then $V1 \leq V2$ because the vectors are issued by the same replica; therefore, $c1$'s operations before $t1$ (with vectors $<V1$) and $c2$'s operations after $t2$ (with vectors$>V2$) are serializable.

Clients that do not interact through accesses to the same objects might have *concurrent* clock vectors. I say two vectors $V1$ and $V2$ are concurrent if and only if there exist i and j, $i \neq j$ and $0 \leq i,j<n$, such that $V1[i]<V2[i]$ and $V1[j]>V2[j]$, where n is the number of coordinators. During a failure recovery, concurrent vectors will be sorted with a simple tie resolution rule consistent across all replicas, which does not necessarily reflect the

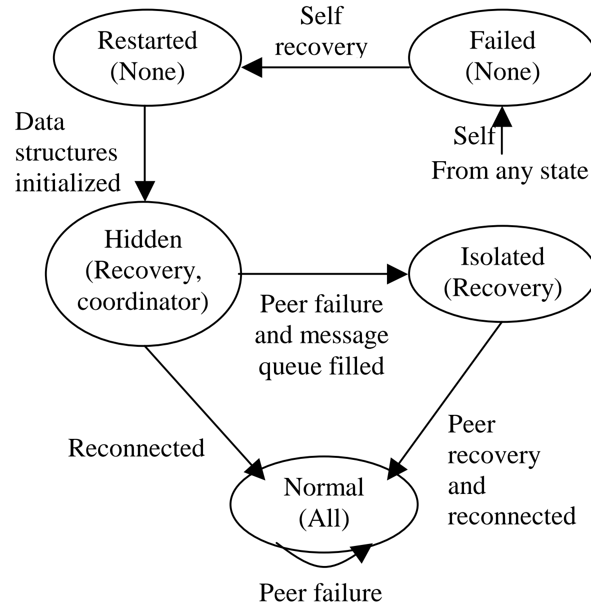

Figure 3.2 State transitions of an island in response to various failures and recoveries. The types of requests accepted in each state are listed in parenthesis. Each transition is labeled with the event that triggers the transition. "Reconnected" is the event that the recovering island has reconnected to and resynchronized with all other islands.

real-time ordering. The reordering of concurrent operations would not be observable and could not cause problems as long as the replicated objects were concerned [23].

3.3.3 Recovery

I have designed a recovery procedure for replicas to recover from arbitrary sequences of failures back to consistent states. Table 3.1 shows the possible failures and remedies for an individual replica. The recovery procedure will be invoked after those (physical) remedies are applied.

Failures	Definitions	Examples	Remedies
Self Failures	Any failures that stop the replica itself from functioning	Software failures, machine crashes, disk failures, power failures	Rerun software, reboot machines, repair disks, restore power
Peer Failures	Any failures that make other replicas inaccessible from this replica	Self failures of other replicas, network partitions	Recover other replicas, repair networks

Table 3.1 Possible failures and remedies for an individual replica.

Given the finite set of possible failures and the infinite set of possible sequences of the failures, I find it a good practice to model the recovering replica as a finite state machine, in which each state corresponds to a set of behaviors that are allowed in the recovering replica, and each state transition is triggered by a failure or recovery event. Figure 3.2 shows the state transitions of a replica in response to the possible failures and recoveries. A replica can be in one of the 5 states, *normal*, *failed*, *restarted*, *hidden* and *isolated*. Each state is distinguished from others by the types of requests the replica is allowed to process in that state. The types of requests a replica receives include *client* requests (from the clients), *coordinator* requests (from the coordinators of updates), *recovery* requests (from the recovering or reconnecting replicas), etc.

In the *normal* state, a replica processes all requests. A self failure in any state causes the replica to transit to the *failed* state, in which no requests, of course, are processed. When it is recovered, a replica transits from the failed state to the transient *restarted* state, in which it initializes necessary data structures while rejecting all requests. It automatically transits to the *hidden* state after all data structures are initialized. In the hidden state, it attempts to reconnect to other nodes and to synchronize replicated state with other nodes by log exchanges. In the hidden state, the replica rejects all client requests so that inconsistency, if it is present in the replica, is not visible to clients. The replica accepts requests from other recovering or reconnecting nodes so that both can make progress. It also accepts requests from the coordinators of new updates and stores them in a *message queue* for sorting with other operations when all have arrived. If the queue becomes full, the replica transits from the hidden state to the *isolated* state, in which it accepts no more coordinator requests. (Note that the buffer for keeping outstanding operations in the normal state will never be filled because there is at most one outstanding operation per coordinator in the buffer.)

When all nodes have reconnected and exchanged logs with it, the replica commits all the operations stored in the message queue in the ascending order of their clock vectors. If it is in the isolated state, it asks for new operations from coordinators that it has rejected. After it commits all pending operations, it transits to the normal state.

3.4 Correctness testing

As discussed in the previous sections, the combination of logical clock synchronization, two-phase commit and write-ahead logging maintains the following invariants in the face of failures:

1. All updates on the replicated metadata are atomic.

2. All updates on the replicated metadata are serialized.

3. In most cases, read-only operations can be processed locally, i.e. without contacting other replicas for synchronization purpose.

However, the correctness of the systems that use this consistency protocol largely relies on the details in implementation, which are hard to model or check using existing tools [31] [37]. Therefore, I use a randomized test engine to test the correctness of the protocol in the face of failures. The test engine is extended from a model checker originally developed in Hewlett-Packard Labs [36]; the model checker is based on the input/output automata (IOA) [30]. I extended the tool so that it checks the implementation of a system, rather than a simulation written in IOA style. Unlike the tools that exhaustively search the state space [31] [37], the randomized testing tools cannot prove that a system is correct. Instead, it helps identifying incorrect parts of a system by injecting various sequences of events to the system and analyzing the results. Such events typically could not possibly be experienced in real workloads or manual tests during a short period of time.

Archipelago, the prototype of the island-based file system, is tested with the randomized test engine.

The test engine consists of three components, *terminators*, *network partitioner* and *clients*. The terminators are independent threads or processes, one for each replica. Each terminator injects *crash* or *reboot* events to its associated replica at intervals randomly chosen within given ranges. It simulates a crash of the replica by killing the server process of that replica, and the reboot of the replica by forking a new server process for that replica. The network partitioner is an independent thread that simulates network

partitions between replicas. At random intervals, it randomly chooses a pair of replicas and sends a message to both replicas to tear down or to reestablish the connections between them. Since multiple pairs can be disconnected this way, a sequence of such events can generate complicated partitions. The clients are multiple threads that share the same set of objects (files, directories and symbolic links) in Archipelago. Each client generates workloads on the file system by repeatedly issuing a randomly chosen request with given frequencies on a randomly chosen object.

The IOA formal language has an interface for defining models for *safety* and *liveness* checking [30]. A safety model specifies a property that must hold at any time, while a liveness model specifies an event that must eventually occur. A prototype of the interface was implemented in the original tool, but I have not ported it to the test engine yet. Instead, I check the safety of the protocol by manually inserting assertions to key parts of the code. A few examples of the assertions are: there is at most one outstanding operation coordinated by each node at any given time; there is no gap and no overlap in the clocks of the operations coordinated by the same node; the coordinator i always has a more or equally up-to-date clock $V[i]$ than any other replicas or clients; etc.. These assertions have been surprisingly helpful in my preliminary experiments. Liveness assertions such as that a replica will eventually transit from the failed state to the normal state in the recovery procedure will be added once the system has passed the simpler tests.

Events	Parameters (% or seconds)	Numbers of Events
CreateDir	3.2279 %	1565
CreateFile	2.8244 %	1369
DeleteFile	1.9206 %	974
DeleteLinkDir	0.8070 %	221
ReadDir	11.2169 %	5273
ReadFile	13.1536 %	8162
RemoveDir	2.4209 %	1469
ResolveLinkDir	7.3434 %	530
SetDirAttr	5.6488 %	2609
SetFileAttr	21.9819 %	14970
SymLinkDir	0.8070 %	227
WriteFile	28.6475 %	16394
Crash	60 to 120 sec	28

Reboot	8 to 16 sec	24
Partition	15 to 30 sec	7
Reconnection	2 to 4 sec	4

Table 3.2 Parameters and results in testing Archipelago in the randomized test engine. The parameters are the given frequencies for normal operations and the given interval ranges for failure/recovery events. For example, each time a client randomly chooses an operation, the probability that CreateDir is chosen is 3.2279%; the terminator waits for an interval randomly chosen from 60 to 120 seconds each time before it kills the server process. The results are the actual numbers of successful operations or events in the test. The actual numbers are different from the specified values due to randomization, race conditions and simulated failures. The operations *SymLinkDir*, *ResolveLinkDir* and *DeleteLinkDir* are creating a symbolic link to a directory, reading the directory entries in a symbolic link to a directory and deleting a symbolic link to a directory, respectively.

The test engine takes parameters such as the interval ranges of failure/recovery events, and the relative frequencies of operations. I selected the intervals in such a way that they both allow a sufficient workload in each state of the system, and allow the overlap of failure/recovery events to exercise the recovery procedure. I exaggerated the frequencies of updates from real workloads by two orders of magnitude to stress the consistency protocol. I tested Archipelago with 4 islands in the randomized test engine. Table 3.2 shows the parameters and results in my latest test. After surviving through 28 node crashes and 7 network partitions, Archipelago failed one of the assertions and caused the test engine to halt.

I found 14 non-obvious bugs in the protocol during two days of testing Archipelago. The bugs are all at implementation detail level and do not invalidate the overall protocol design. An example of the bugs I found is following. The coordinator of an update crashed after it notified the replicas of the operation, but before it logged the operation on disk. Therefore, the operation was aborted in the coordinator, but outstanding in the replicas. When the replicas received the next operation from the same coordinator later, the assertion of at most one outstanding operation per coordinator failed. The fix to this bug is to clear the relevant buffers of outstanding operations upon reconnection of two nodes.

The preliminary results in the correctness testing are encouraging, and I believe that

randomized failure injection is a promising approach to checking the implementation correctness of a complicated system.

3.5 Summary

I design and implement a protocol for the atomicity, serialization and recovery of updates on replicated metadata in the island-based file system. I build a randomized test engine to check the correctness of the protocol in the face of arbitrary sequences of failures.

In summary, the consistency protocol guarantees the following serializations for updates in the face of arbitrary sequences of node failures and network partitions:

1. All operations on the same object are serializable.

2. All operations by the same client thread are serializable.

3. Operations by different clients are serializable if the clients interact with each other by accessing the same object(s).

In addition, the ordering relations of operations are *transitive*, i.e. if operation 1 is observed to happen before 2 and 2 before 3 then 1 is observed to happen before 3, because the ordering relations of clock vectors are transitive, i.e. if $V1<V2$ and $V2<V3$ then $V1<V3$.

Under this protocol, the replicas never expose the intermediate state of updates to clients and clients never observe the results of updates in conflicting orders; therefore, the chance for hazards introduced by the cluster environment is largely reduced, and it is possible to port applications from single systems to the cluster-based systems with few modifications.

4 Evaluation of the island-based file system[3]

4.1 Statistical analysis

In this section, I study the partial availability, load balance and consistency cost in the island-based file system by collecting statistics from existing file systems in use. Although the island-based file system design was motivated by the access patterns of Internet services, I evaluated it in a more generic context.

4.1.1 Partial availability for applications

The effective availability of an island-based file system with partial failures depends on the number of distinct directories that clients access because a partial failure in the system causes a random set of directories to be inaccessible.

With the access logs from the web site of computer science department in Princeton University [92], I divided requests and clients into "bins", where requests and clients in the same bin accessed the same number of distinct directories, and computed the histograms of clients and requests across different bins. I assume that the island-based file system acts only as a content provider to the web server, i.e. accesses to control information or executables of the web server itself do not count in my statistics. I grouped the HTTP requests into clients by the hostnames or IP addresses in the requests, and within each client, I grouped requests into directories by the URLs in the requests and maintained a counter for the total number of requests. I computed the histograms from two months' traces, July 1998 (137248 clients and 1304975 requests in total) and January 1999 (166804 clients and 1297428 requests in total). I kept the distinct directories and total number of requests for each client up to an hour, updated the histograms and cleared all clients' records in the end of each hour, and restarted recording for the next hour. The histograms were cumulated for the two months. See Figure 4.1. The results show that the largest portion (48.3%) of clients accessed only 1 distinct directory in an hour and the largest portion (17.9%) of requests were issued by clients who accessed 2 distinct

[3] A subset of the content in this chapter has been included in a paper [93] published in the Proceedings of 4th USENIX Windows Systems Symposium, August 2000.

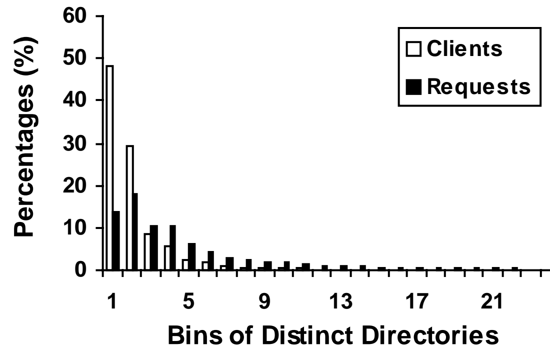

Figure 4.1 Histograms of clients and requests across bins of distinct directories accessed during every hour in the web traces. The numbers read as "48.3% of clients accessed 1 distinct directory during every hour" or "17.9% of requests were issued by clients who accessed 2 distinct directories during every hour". Accesses to more than 24 directories account for 0.4% of clients and 19.3% of requests in total, and are omitted in the graph for readability.

directories in an hour. Requests are more scattered across categories because larger categories tend to have more accesses and hence weigh more. I computed the histograms for other time windows ranging from 30 minutes to 8 hours, but there was no significant difference across time windows, which implies that client sessions are short, i.e. less than 30 minutes.

Given the statistics of distinct directories, I computed the expected availability of the island-based file system for data, clients and requests, respectively, shown in Figure 4.2. Since the majority of web clients access a small number of distinct directories, the expected availability for this class of clients is high in spite of the fact that a partial failure in the system causes a random set of directories to be inaccessible. For example, if 1 out of 32 islands is down for an hour, I expect that 93.8% clients of the web server during that hour will not notice the temporary partial failure.

Traditional file access

I also computed the histograms of application groups and file system calls across the bins of distinct directories they accessed, using the file system traces taken on a file server in Hewlett-Packard Labs for the week starting September 24, 1999, which consisted of 5,995,712 pathname-based low-level file system calls such as open(). The users of that

file server are 5 to 10 researchers who access files through applications like emacs, g++, netscape and shells on UNIX workstations. I grouped file system calls by process ids and divided process ids into “application groups” by using the logged fork() system calls. Each application group is associated with a window or session manager, but some are finer-grained because I do not know about the fork() events that happened before the tracing program started. In the traces I used, 183,915 fork() events were recorded and 5,170 groups were identified. I computed the histograms for the time windows ranging from 1 minute to 1 hour. I use the overall histogram of application groups below since there was no significant difference across time windows. Similar to the web traces, the largest portions, 26.2% and 14.8%, of application groups accessed 1 and 2 distinct directories, respectively; different from the web traces, more groups accessed a larger number of distinct directories, e.g. 17.3% groups accessed more than 24 directories. As time window increases, more file system calls were counted in larger categories of distinct directories. For example, in 5 to 10 minute windows, the largest portion (17.6%) of calls were in the category of 1 distinct directory; in the 1-hour windows, the largest portion (44.4%) of calls were in the category of 7801 distinct directories. The users of those application groups will be affected by a lasting partial failure in the island-based file system, for the island-based file system was not designed for that class of users.

4.1.2 Replication cost and load distribution

In this section, I analyze storage cost for the usage-based metadata replication and load balance in the hash-based data distribution in the island-based file system, using the contents of five existing systems in use.

I took snapshots of five UNIX and Linux file systems, using the shell command "ls -l -A -R /". The first two rows of Table 4.1 show the names and clients of the five systems. Most of the systems consist of multiple partitions that are mounted together via NFS. For my statistical studies, I pretended that each system is a single file tree stored in the island-based file system.

4.1.2.1 Replication cost

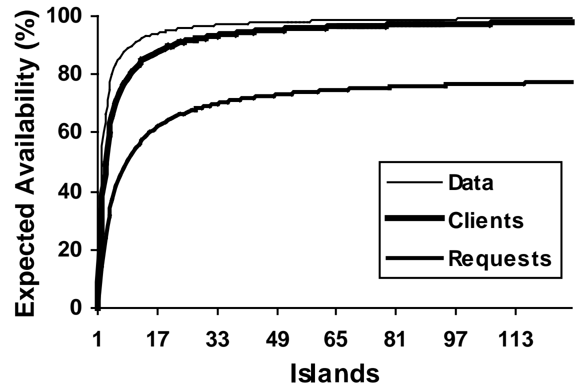

Figure 4.2 Expected availability for data, clients and requests in the web traces with the failure of 1 out of *n* islands. The x axis is the number *n* of islands. The y axis is the expected percentages of successful accesses, i.e. (1-1/*n*) for data and $\sum p(i) \cdot (1-1/n)^i$ for clients and requests, where *p(i)* is the percentage of clients or requests that accessed *i* distinct directories.

I computed the *upper bound of the replication storage*, i.e. storage for replicating all non-leaf directory attributes and all symbolic links to directories in all islands. Let D be the number of directories, F be the number of files, I be the inode size in bytes, and T be the total number of bytes for directory and file contents. Then the storage required for the entire system without replication, the *net storage*, is $I \cdot (D+F)+T$ bytes. Let S be the number of islands, N be the number of non-leaf directories, Q be the number of symbolic links to directories, and L be the size of a symbolic link. Then the upper bound of the replication storage is $I \cdot N \cdot (S-1) + (I+L) \cdot Q \cdot (S-1)$.

File Systems	Web	Project	Cdroms	Department	University
Clients of file systems	Web server of CS department	"SHRIMP" project	CD server of CS department	People in CS department	People in Princeton University
Directories (*D*)	5938	16233	25195	178662	178974
Files (*F*)	104186	222958	228326	3377478	1653946
Contents (*T*) (GB)	4.74	11.01	14.55	105.9	51.27
DirLinks (*Q*)	183	450	1010	3339	35698

Non-leaves (N)	1863	4189	10102	46639	45427
Islands (S)	1	3	4	31	15
Rep. (GB)	0.000	0.036	0.130	5.815	4.809
Rep. (percent)	0.0%	0.3%	0.8%	4.7%	7.7%

Table 4.1 Replication cost. Each column is an existing file system in use. Row "Rep. (GB)" shows the upper bound of the replication storage, computed as $I \cdot N \cdot (S-1) + (I+L) \cdot Q \cdot (S-1)$. Row "Rep. (percent)" shows the percentage of the upper bound of replication storage to the total storage. The net storage is computed as $I \cdot (D+F) + T$. The replication storage in the web system is zero because there is only one island for the web file system.

Based on existing system configurations, I assumed that the capacity of each island was roughly 4 GB, hence the number of islands for each file system is the total number of bytes in the system divided by 4 GB. I computed for each system the upper bound of replication storage with I=4KB and L=1KB. See Table 4.1. The percentage of replication storage to total storage ranges from 0.3% to 7.7%. Given the decreasing costs for storage devices nowadays, the replication cost is insignificant.

4.1.2.2 Expected load imbalances

Assuming that objects O are to be distributed to units U, we define the *imbalance* I_{OU} as the standard deviation of objects O in units U divided by the average objects in each unit. I_{OU} is zero if the distribution is perfectly even. Let B be the number of buckets, D be the number of directories, W be the workload, and S be the number of islands. I define the variables x_{ij} as

$$\forall i = 1..D,\ j = 1..B,\ x_{ij} = \begin{cases} 1, \text{ if directory } i \text{ is hashed to} \\ \text{bucket } j; \\ 0, \text{ otherwise.} \end{cases}$$

The bucket size is $Y_j = \sum_{i=1}^{D} x_{ij}$, the expected bucket size is $E[Y_j] = \frac{D}{B}$, and the variance (square of standard deviation) of bucket size is

$$Var[Y_j] = Var[\sum_{i=1}^{D} x_{ij}] = \sum_{i=1}^{D} Var[x_{ij}]$$
$$= \sum_{i=1}^{D} E[(x_{ij} - E[x_{ij}])^2] = \sum_{i=1}^{D} \frac{(1-\frac{1}{B})^2 + (\frac{1}{B})^2 \cdot (B-1)}{B}$$
$$= \frac{D}{B}(1-\frac{1}{B}).$$

The second step of derivation above is based on the property of universal hash functions that x_{ij}'s are pairwise independent. The others are by definitions. The imbalance in directory distribution across buckets is

$$I_{DB} = \frac{\sqrt{Var[Y_j]}}{E[Y_j]} = \sqrt{\frac{B-1}{D}}.$$

I define w_i as the workload in directory i and $W = \sum_{i=1}^{D} w_i$. The workload in a directory does not include the loads in its sub directories, hence I can assume that w_i's are pairwise independent. The expected directory workload is $E[w_i] = \frac{W}{D}$, the variance of directory workload is $Var[w_i] = \frac{\sum_{i=1}^{D} w_i^2}{D} - E^2[w_i]$, and the imbalance in workload distribution across directories is $I_{WD} = \sqrt{D \cdot \frac{\sum_{i=1}^{D} w_i^2}{W^2} - 1}$. The workload in bucket j is $Z_j = \sum_{i=1}^{D} x_{ij} \cdot w_i$, the expected bucket workload is $E[Z_j] = \frac{W}{B}$, the variance of bucket workload is

$$Var[Z_j] = Var[\sum_{i=1}^{D} x_{ij} \cdot w_i] = \sum_{i=1}^{D} Var[x_{ij} \cdot w_i]$$
$$= \frac{\sum_{i=1}^{D} w_i^2}{B} - \frac{\sum_{i=1}^{D} w_i^2}{B^2},$$

and the load imbalance across buckets is

$$I_{WB} = \sqrt{(B-1) \cdot \frac{\sum_{i=1}^{D} w_i^2}{W^2}}$$. I substitute $\frac{\sum_{i=1}^{D} w_i^2}{W^2}$ with $\frac{I_{WD}^2 + 1}{D}$ and get $I_{WB} = I_{DB} \cdot \sqrt{I_{WD}^2 + 1}$.

4.1.2.3 Measured load imbalances

With the same snapshots of the five file systems, I compute the load imbalances described in the previous section, and compare them with their theoretical expectations computed in the previous section. Since the access logs of the five systems are not all available to me, the number of bytes, instead of accessed bytes, was used as the measure of workload in my study. See Table 4.2.

To compare I_{DB} with its theoretical expectation $\sqrt{\frac{B-1}{D}}$ and across different systems, I fixed the number B of buckets or extendible hash table entries to be 256. The directory distribution across buckets is very close to its theoretical expectation. The byte distribution across buckets is less close to its expectation due to the assumption of pairwise independency between directory workloads. The byte distribution across directories is determined by the usage of the systems and is considerably uneven. The second step of hashing, i.e. the extendible hash table, was designed to balance the workload, i.e. the number of bytes in my study, across islands. The number of islands for each system is the same as in Table 4.1. Table 4.2 shows that bytes are evenly distributed across islands. The extendible hashing algorithm is independent of the inputs; therefore, it can also evenly distribute actual workload across islands if the input is the actual workload recorded in real systems.

	Web	Project	Cdroms	Department	University
I_{DB} (predicted)	0.21	0.13	0.10	0.04	0.04
I_{DB} (measured)	0.19	0.13	0.10	0.04	0.04
I_{WD}	5.93	15.56	17.58	11.95	17.70
I_{WB} (predicted)	1.14	2.03	1.76	0.48	0.71

I_{WB} (measured)	1.23	1.94	1.81	0.68	0.71
I_{WS}	0	0.0004	0.0001	0.0279	0.0087

Table 4.2 Load imbalances in five file systems. I_{DB} **(measured) is the measured imbalance in directory distribution across buckets.** I_{DB} **(predicted) is the theoretical expectation** $\sqrt{\frac{B-1}{D}}$. I_{WD} **is the imbalance in byte distribution across directories.** I_{WB} **(measured) is the imbalance in byte distribution across buckets.** I_{WB} **(predicted) is the theoretical expectation** $I_{DB} \cdot \sqrt{I_{WD}^2 + 1}$. I_{WS} **is the imbalance in byte distributions across islands. The imbalance value is 0 if the distribution is perfectly even.** I_{WS} **is 0 in the web system because there is only one island for the web system.**

4.1.2.4 Hot spots

Table 4.3 shows the largest/average sizes in various distributions. I observed the following properties in all five systems: the largest directory, one that contains the most bytes, has 81.30% to 99.99% of its bytes stored in a single file, which in turn is the largest file in the entire system; the largest file is small compared to the entire system, hence it does not prevent a good overall load balance across islands. It is worth noting that the relatively high imbalance in the departmental file system is due to the fixed number 256 of hash table entries: the largest table entry accounts for more than $\frac{1}{256}$ of total bytes. In my implementation, the table size grows with the number of islands.

	Web	Project	Cdroms	Department	University
H_{DB}	1.68	1.47	1.30	1.11	1.12
H_{WD}	243.1	1843.6	939.1	5703.3	3077.1
H_{WB}	10.76	29.56	13.47	9.54	6.50
H_{WS}	1	1.0003	1.0002	1.0385	1.007

Table 4.3 Hot spots. H_{DB} **is the largest/average bucket size in directories;** H_{WD} **is the largest/average directory size in bytes;** H_{WB} **is the largest/average bucket size in bytes;** H_{WS} **is the largest/average island size in bytes. The value is 1 if the distribution is perfectly even.** H_{WS} **is 1 in the web system because there is only one island for the web system.**

4.1.3 Operation breakdown

In this section, I present statistical results to show how many operations in existing access

patterns require cross-island communication and synchronization. In the island-based file systems, the following operations are cross-island operations as a result of the metadata replication: *CreateDir, RemoveDir, SetDirAttr, ResolveLinkFile* (resolving a symbolic link to a file)*, SymLinkDir* (creating a symbolic link to a directory)*, ResolveLinkDir* (resolving a symbolic link to a directory)*,* and *DeleteLinkDir* (deleting symbolic link to a directory).

Previous studies of file system traces indicated that the cross-island operations are rare. Traces taken on the Sprite system [40] show that *setattr*, *rmdir* and *mkdir* account for only 0.7%, 0.03%, and 0.02% of total operations, respectively. The SPEC SFS or LADDIS benchmark [12] generates an operation mix based on NFS client workload studies, which consists of 1% *setattr* operations, 1% *remove* operations and 2% *create* operations. Recent traces taken on NFS clients [39] consist of 0.092% *chmod*, 0.015% *chown*, 0.003% *symlink*, 0.015% *readlink*, 0.013% *rename*, 0.013% *mkdir*, and 0.012% *rmdir*. The majority of the operations in all those studies are reading attributes, reading files, writing files and reading directories, which account for 84% to 96% of total operations. Some of the operations in those studies, e.g. *setattr* and *chmod*, were not recorded for files and directories separately; therefore, the percentages of those operations on directories will be even lower than reported.

It is well known that file access patterns are always specific to the operating systems where the traces were taken. Since I implemented the island-based file system on Windows NT as opposed to UNIX, in which the Sprite and NFS traces were taken, I felt it important to study the file access patterns in NTFS. I chose 7 workstations in Princeton and Montreal running Windows NT 4.0 and collected statistics on operations by running a trace program on each workstation in March 1999. The users of the workstations include three graduate students, a software engineer, a home user and several lab users. The trace programs were run for 2 to 7 days and collected 30,391 to 480,385 total events.

The trace program forks a thread to wait on each file system related event such as FileAdded through the NTFS event notification interface *ReadDirectoryChangesW* [41]. The events are not necessarily one-to-one mapped to file system operations, and there is

no detailed documentation on the mapping. Hence I present the raw events in Table 4.4 and infer the operation breakdown with the empirical rules: reads to files and directories are not detected if the reads hit in cache; writes to files and directories are not detected until the cache is flushed; an attribute-change event comes with a name-change, size-change, or security-change event; reading attributes as well as reading contents changes last access time if it does not hit in cache.

Table 4.4 shows that, on average, one-island operations account for 99.8% of total operations. The slow operations in the island-based file system, e.g. setting directory attributes, renaming directories, creating symbolic links to directories, are rare. Therefore, the amortized cost for keeping replicated state consistent across islands can potentially be made low in the island-based file system. Section 4.3.1.3 shows the results in measuring the impact of the consistency protocol on the overall scalability.

No.	Events	Average	Standard Deviation
1	Total Events	244408	140571
2	*FileAdded*	3.34%	1.70%
3	*FileRemoved*	2.38%	1.70%
4	*FileRenamed*	0.41%	0.31%
5	*DirAdded*	0.04%	0.07%
6	*DirRemoved*	0.03%	0.07%
7	*DirRenamed*	0.00%	0.00%
8	*FileAttrModified*	26.8%	10.8%
9	*FileWritten*	35.5%	11.3%
10	*FileAccessed*	16.3%	8.60%
11	*FileSecurityModified*	0.03%	0.04%
12	*DirAttrModified*	0.07%	0.07%
13	*DirWritten*	1.23%	1.59%
14	*DirAccessed*	13.9%	17.8%
15	*DirSecurityModified*	0.00%	0.00%
16	*FileLinkModified*	0.16%	0.08%
17	*FileLinkRead*	0.09%	0.10%
18	*DirLinkModified*	0.00%	0.00%
19	*DirLinkRead*	0.001%	0.002%

Table 4.4 Percentages of file system events in NTFS traces. Row 1 (Total events) shows the total number of events in each trace. Rows 2 through 19 show the

percentage of each event. Shaded events correspond to cross-island operations in the island-based file system. The *FileLinkModified* (row 16) and *DirLinkModified* (row 19) events include creating, removing, writing and setting attributes on symbolic links to files and directories, respectively. The *FileLinkRead* (row 17) and *DirLinkRead* (row 19) events are resolving symbolic links to files and directories, respectively. The column "Average" shows the percentage of each event averaged over all traces. The column "Standard Deviation" shows the standard deviation of the percentages of each event in each trace. Events not shown in the table have zero percentages.

4.2 Implementation

I have implemented a prototype of the island-based file system, called *Archipelago*, on a cluster of Pentium II PCs running Windows NT 4.0. NTFS [15] is used as the internal file system. NTFS uses extensive caching and name indexing for better performance and logs metadata changes for local recoverability. An access control list is associated with each file or directory to check access rights. NTFS can be configured to run on a group of disks with parity striping for high reliability.

An Archipelago server runs on each machine and forms an island. Each client accesses files through a local stub, which forwards the request to a server through Windows remote procedure call (Win32 RPC). The tasks of the server include authenticating clients, validating clients' versions of the hash table, synchronizing clients' clock vectors, and processing clients' requests in the internal file system. The functions of the stub include hashing a pathname to an island, updating local copies of the hash table, synchronizing the clock vectors with servers, maintaining secure RPC connections to servers, tolerating network failures and making file locations transparent to clients.

The server is implemented as a user-level process. For expediency, my prototype client is implemented as a stub .dll that redirects requests for Archipelago files directly to servers, bypassing the in-kernel file system drivers. This solution is adequate for experimental purposes, although it does not provide total seamless integration with existing applications. A more complete solution would implement a full installable file system driver [15]. I believe the performance difference in these two solutions to be negligible compared with the time to service file system requests in a distributed file system.

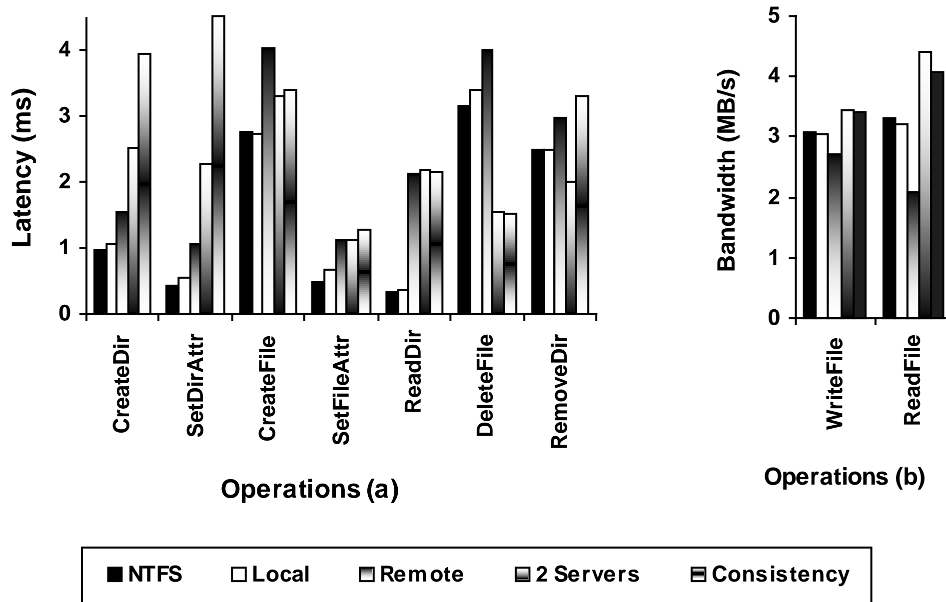

Figure 4.3 Single client performance. A single client runs the micro benchmarks in five cases: directly on NTFS (NTFS), on the local machine of an Archipelago server (Local), on a remote machine from the server (Remote), with two servers (2 Servers), and with the consistency protocol turned on with two servers (Consistency), respectively. The y-axis in (a) is the latency in milliseconds measured at the client side. Lower columns represent better performance. The y-axis in (b) is the bandwidth in MB/s in the *WriteFile* and *ReadFile* operations measured at the client side. Higher columns represent better performance.

The server and stub are implemented in C++, and consist of 3088 and 5415 lines of code, respectively. The server program is linked with the stub library for code reuse purpose. In addition, there are 24042 lines of automatically generated C code for RPC and system call interception.

4.3 Performance

In this section, I present the results of running micro benchmarks and operation mixes on Archipelago in various configurations. The 23 machines used in my experiments have Pentium II 300 MHz processors, 128 MB main memories and 6.4 GB Quantum Fireball IDE hard disks for use by Archipelago. The PCs are connected by an Intel Express 510T Ethernet 100Mbps 24-port switch and run in full-duplex mode. The PCs run Windows NT Workstation 4.0 and the hard disks for Archipelago are formatted in NTFS.

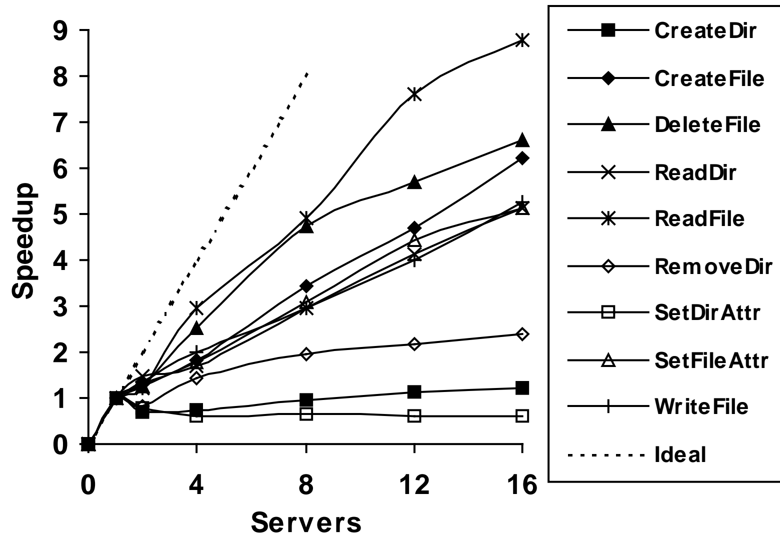

Figure 4.4 Speedup of throughputs on private data as a function of the number of servers. Number of servers = number of clients = number of private data sets. The speedup is calculated as the absolute throughput divided by the throughput with 1 server. The dotted line represents the speedup with 100% efficiency.

4.3.1 Micro benchmarks

The set of micro benchmarks consists of 9 phases and each phase exercises one of the file system calls: CreateDir, SetDirAttr, CreateFile, SetFileAttr, ReadDir, WriteFile, ReadFile, DeleteFile and RemoveDir. The basic data set for the micro benchmarks is a project directory that consists of 90 directories, 646 files and 77.2 MB of data in files. I duplicated the directories 40 times, the files 6 times and the contents 2 times, respectively. The 9 resulting phases are: create 3600 directories, set 3600 directory attributes, create 3876 files with pre-allocated space in 540 directories, set 3876 file attributes, read 6634 directory entries, write 154.4 MB data in 1292 files or 180 directories, read 154.4 MB data in 1292 files, delete 3876 files, and remove 3876 directories. The transferred block size in the WriteFile and ReadFile phases is 64 KB or the file size, whichever is smaller. With the data set inflated, the results of the micro benchmarks are reasonably stable. Each test was run more than 3 times and the results shown in this section are the averages.

Other operations, such as moving a file and reading a symbolic link, were implemented

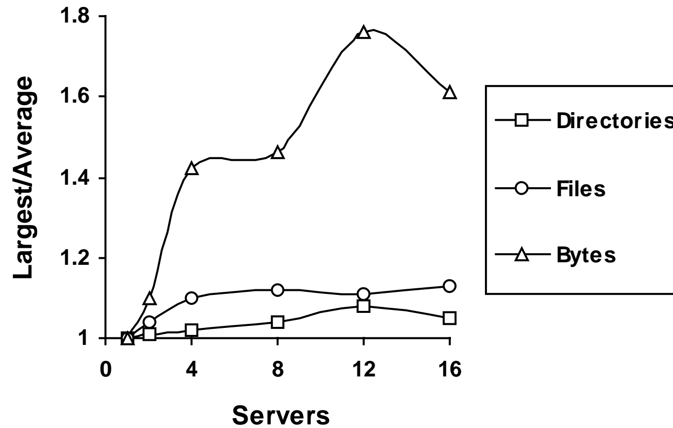

Figure 4.5 Load imbalances. The y-axis is the largest number of directories, files or bytes stored in a server divided by the average number over all servers. The y-value is 1 if load is perfectly balanced.

with the operations in these micro benchmarks; hence, I did not include them in the tests. I did not intentionally flush the file cache in NTFS during the tests because I would like to treat NTFS, the internal file system, as a functional black box. However, the amounts of data in the WriteFile and ReadFile phases were large enough to overflow the cache.

4.3.1.1 Single client performance

I ran the micro benchmarks with a single client in 5 cases: directly on NTFS (1), in the same address space as an Archipelago server (2), on a separate machine from an Archipelago server (3), with two Archipelago servers, all on separate machines (4), and with the consistency protocol turned on in case 4 (5). Figure 4.3 shows the bandwidth in *WriteFile* and *ReadFile* and the response times in other operations, all measured at the client side.

The difference between case 1 and 2 is the overhead of computing hash functions. This overhead, roughly 0.1 ms, is low compared to the operation time itself. The difference between case 2 and 3 is the communication (RPC) time between the client and the server. I used Win32 RPC on top of TCP/IP on 100 Mbps switched Ethernet. In my experiments, the average round-trip RPC latency for small messages (~256 bytes) is 0.48 ms and the average one-way large data (64 KB) transfer rate in RPC is 8.67 MB/s. The performance decreased from case 2 to case 3 by an amount comparable to the RPC overhead. The

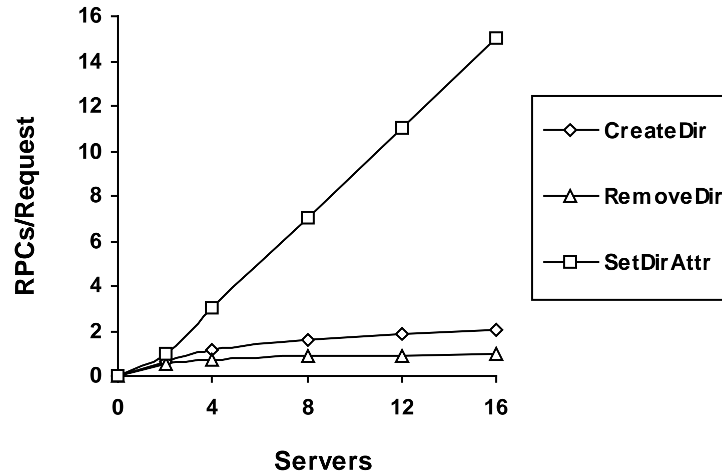

Figure 4.6 Number of server-to-server RPCs per request as a function of the number of servers. Operations that do not show up in this figure have zero server-to-server RPCs.

difference between case 3 and case 4 is that the cross-island operations *CreateDir*, *RemoveDir* and *SetDirAttr* involve 1 island in case 3 and 2 islands in case 4. Therefore, the response times of *CreateDir* and *SetDirAttr* were increased by roughly 1 ms from case 3 to case 4. Operations such as *ReadFile* and *RemoveDir* were faster in case 4 because there was more total cache space in case 4. The difference between case 4 and case 5 is the overhead of the consistency protocol. The consistency protocol slows down the cross-island operations by a factor of 1.6 to 2, depending on the number of islands involved in the operation. It adds little overhead, roughly 0.1 ms, to one-island operations. The response times of *CreateFile* are larger than those of *CreateDir* in all cases because the client pre-allocated space for each file in the *CreateFile* phase.

4.3.1.2 Scalability of individual operations

In this new set of tests, there are multiple clients, each running an instance of the micro benchmarks on its own private data set. Before each phase, all clients are synchronized at a barrier. Each server ran on a separate machine and 1 to 3 clients ran on the same machine. The number of clients was configured to be the same as the number of servers. Given the 23 machines connected by the 24-port Ethernet switch, I scaled the number of servers and clients up to 16 each. I have tested Archipelago with 25 servers on an Ethernet hub and expect the system to be able to scale to larger configurations. Here I

present only the results of scaling from 1 to 16 servers.

I measured throughput at the server side, i.e. the total number of bytes requested divided by the time for all servers to complete, for *WriteFile* and *ReadFile*, and the total number of requests divided by the time for all servers to complete for other operations. To compare the scalability across operations, I calculated the speedup as the absolute throughput divided by the throughput with 1 server. The 1 server case is the same as case 3 in Figure 4.3. Figure 4.4 shows the speedup of throughputs on private data as a function of the number of servers. Most operations scale linearly with the number of servers, but at a less than ideal slope. The inefficiency in scalability of the one-island operations results from load imbalance, while the inefficiency in scalability of the cross-island operations results from both load imbalance and communications.

The directory, file and byte operations are distributed across 3600, 540 and 180 directories, respectively. The load distribution is expected to be less than ideal due to the small size of the data sets (compared to the size of an entire file system). I calculated the load imbalance as the largest load divided by the average load of servers. See Figure 4.5. The load imbalance ranges from 1.0 to 1.8. I expect the operations to scale better in real systems with the rebalance protocol, which will be studied in Section 4.3.1.6.

Figure 4.6 shows the average number of server-to-server RPCs per request measured in the tests. The two-phase commit protocol was turned off in this set of tests; therefore, the actual numbers of RPCs will be doubled with the protocol turned on. The impact of the consistency protocol on performance is discussed separately in Section 4.3.1.3. One-island operations do not show up in Figure 4.6 because they require no server-to-server RPCs. The number of RPCs for SetDirAttr grows linearly with the number of servers; therefore, the speedup curve for SetDirAttr in Figure 4.4 is nearly flat. The numbers of RPCs for CreateDir and RemoveDir are nearly constants; therefore, these two operations do scale with the number of servers, but slower than the one-island operations.

4.3.1.3 Impact of the consistency protocol

I turned on the consistency protocol, i.e. clock synchronization, two-phase commit and

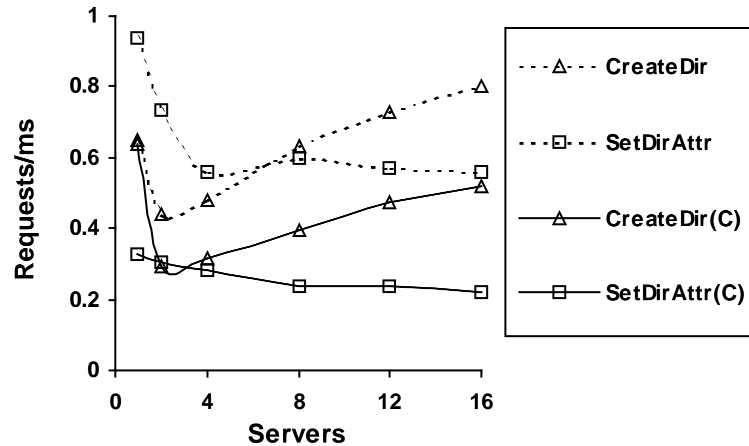

Figure 4.7 Impact of the consistency protocol on cross-island operations. The curves CreateDir and SetDirAttr are the throughputs (requests/ms) in the experiment shown in Figure 4.4. The curves CreateDir(C) and SetDirAttr(C) are the throughputs in the same benchmark but with the consistency protocol turned on. The curves Delays-C and Delays-S are the average delays per request for CreateDir and SetDirAttr with the protocol turned on, respectively.

logging, and reran the micro benchmarks. As expected, the protocol adds little overhead, roughly 0.1 ms, to one-island operations. Figure 4.7 shows the throughputs of two cross-island operations, *CreateDir* and *SetDirAttr*. (*RemoveDir* is similar to *CreateDir*.) The protocol increases the RPCs between servers for cross-island operations by a factor of 2 and requires a log write per successful cross-island operation. As expected, the consistency protocol brings considerable overhead to cross-island operations. The throughput of *SetDirAttr* does not scale with or without the consistency protocol. The throughput of *CreateDir* scales at roughly the same rate with or without the protocol.

4.3.1.4 Scalability of operation mixes

I ran a new benchmark of randomized operation mixes to measure the overall scalability of Archipelago. The new benchmark is similar to the SPEC SFS or LADDIS benchmark [12], but with the following extensions:

- Clients access shared objects as well as private objects.
- Files and directories are stored in a hierarchical structure rather than in a flat structure.

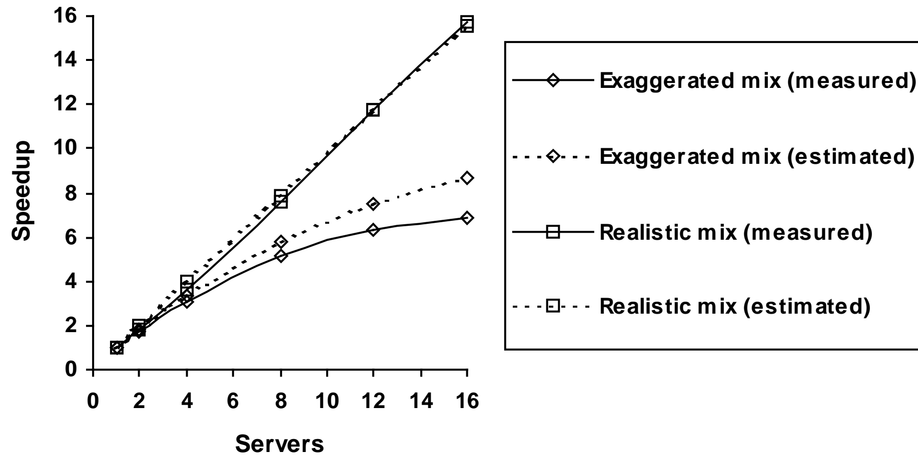

Figure 4.8 Speedup of throughputs of randomized operation mix. The speedup is calculated as the absolute throughput (requests/sec) divided by the throughput of 1 server. The throughput of 1 server is 75.6 requests/sec in operation mix 1 and 80.1 requests/sec in operation mix 2, respectively.

- The pathnames are generated with randomization, rather than pre-defined manually.
- The file sizes are variable rather than fixed.
- The workloads scale with the system size.

Since Archipelago is implemented on top of NTFS, the operation mix in my benchmark uses NTFS API and is based on the operation breakdown I measured on Windows NT workstations (Section 4.1.3).

I ran the benchmark with 1 to 16 clients and servers on 1 to 16 machines. Each client runs on the same machine as a server, but accesses random files, directories and symbolic links across the entire system. The pre-created data set includes 2000 shared directories, 2000 shared files, 100 shared symbolic links, and the same numbers of private objects per client. The client randomly chooses an operation at specified frequencies. For each operation, the client randomly chooses an object, either from the existing shared or private objects, or by generating a new name in an existing directory, depending on the operation. The *WriteFile* operation writes a random number (chosen from 0 to 1 MB) of bytes to the file; both *WriteFile* and *ReadFile* operations transfer up to 8KB per request

so that the operation time is comparable to those of other operations. Each client maintains lists of the shared objects and its private objects, but does not synchronize with other clients on the creation and deletion of the objects in the shared directories. Therefore, an operation on a shared object might fail if it conflicts with a previous operation on the same object from another client. After the data set is pre-created, all clients run the randomized operation mix for 10 minutes. The throughput is calculated as the total number of successful operations by all clients divided by 10 minutes.

I ran the benchmark with two different operation mixes. The "exaggerated mix" exaggerates the cross-island operations in the measured breakdown and the "realistic mix" is closer to the measured breakdown. The mixes cover a number of typical operations from each category, i.e. one-island, two-island and all-island. Uncovered operations in the measured breakdown are replaced by operations in the same category, e.g. the operation of reading a symbolic link to a file counts for 0.09% in my measured breakdown and is replaced in the mix with the same number of operations that read a symbolic link to a directory. I recorded the actual client operations and server-to-server RPCs in the benchmarks, and estimated the speedups of the overall operation mix accordingly. Table 4.5 shows the recorded operation mixes and Figure 4.8 shows both the measured speedups and estimated speedups. Assuming that each local operation and RPC takes the same amount of time, the estimated speedup with *n* servers is *n/(1+overhead-per-operation)*, where the *overhead-per-operation* is the total number of server-to-server RPCs divided by the total number of successful client operations.

	Exaggerated mix (%)	Realistic Mix (%)
CreateDir	0.9297	0.0522
CreateFile	4.0314	3.5661
DeleteFile	2.7731	2.4353
DeleteLinkDir	0.9850	0.0128
ReadDir	14.4505	15.6528
ReadFile	14.1343	15.2778
RemoveDir	0.7543	0.0162
ResolveLinkDir	1.7205	0.1014
SetDirAttr	1.0383	0.0713
SetFileAttr	26.6085	29.2835
SymLinkDir	1.0089	0.0109

WriteFile	31.5656	33.5194
Successful	45360 to 309960	48042 to 756120
Total	48042 to 325534	48043 to 780260

Table 4.5 Operation mixes. The actual numbers of operations generated in the benchmarks are slightly different from the specified frequencies due to randomization and failed requests. Each percentage in this table is the number of successful requests on each operation divided by the total number of successful requests, averaged over 1 to 16 clients and servers. The total numbers of requests grow with the numbers of clients and servers for the fixed 10 minutes period; the ranges are shown in the last two rows in the table. See Table 4.2 for explanations for certain operations.

The exaggerated mix scales at a less than ideal slope due to the relatively large number of cross-island operations. For example, with 16 servers, the average overhead-per-operation is 0.8. The difference between the estimated speedup and measured speedup is due to the assumption of equal RPC processing times and local operation times. Load is well balanced across servers in both operation mixes; the largest/average requests per server are below 1.1 in all cases. The realistic mix is closer to the measured breakdown, i.e. contains a smaller number of cross-island operations; it scales nearly ideally in both estimated and measured throughputs. It is worth noting that the realistic mix scales better than the pure one-island operations in Section 4.3.1 because considerable load imbalance is present in that benchmark due to the small number of working directories.

4.3.1.5 Implications for larger scales

Given the probabilities of one-island ($P1$), two-island ($P2$) and all-island (Pa) operations, where $P1+P2+Pa=1$, I can predict the speedup efficiency at large scale with a simple model. Assuming that each local operation and RPC takes the same amount of time, the estimated speedup efficiency with n servers is 1/(1+*overhead-per-operation)*, where *overhead-per-operation* is the average number of server-to-server RPCs per operation and equals $(2-1)*2*P2+(n-1)*2*Pa$. (The factor 2 results from the two-phase commit protocol. See Section **3.3**.) Two-island operations include CreateDir, RemoveDir, ReadFileLink and ReadDirLink; all-island operations include SetDirAttr, SetDirSecurity, SymLinkDir and RenameDir. Some operations, e.g. SetDirSecurity and SymLinkDir, did not show up in my statistical experiments; I inferred their percentages from other

statistics [39]. The resulting percentages are $P1$=99.768%, $P2$=0.161% and Pa=0.071%. From the speedup efficiency model above, I predict that, with the efficiency higher than 50%, the system can scale up to 702 islands.

4.3.1.6 Discussion

Although the target applications of the island-based file system are Internet services, I use a more generic benchmark in the scalability measurements. My purpose of those measurements is to learn the impact of cross-island operations on the overall scalability of the island-based file system, but web access logs only give file-reading operations. I do not model in my benchmark the self similarity or hot spots in web accesses because it is not clear whether the same patterns will necessarily show up in disk accesses if web requests can be processed with data in the main memory cache of web servers or file system clients.

4.3.2 Trace-based benchmarks

I simulated on top of Archipelago the web server of computer science department in Princeton University [92] and measured the performance during online reconfiguration. The file system that the web server originally runs on consists of 5934 directories, 103,426 files and 4.74 GB of contents. It was first copied to an Archipelago with two islands. I added and then removed two islands to the system and studied the performance of data migration and its impact on the performance of web accesses. The hardware used in this set of tests is the same as in previous tests. Table 4.6 shows the statistics in the addition and removal of two islands without client accesses.

The web server was a Netscape Enterprise Server 3.5.1 running on Solaris 2.6. The hardware for the web server was a Sun Ultraserver-2 with 256 MB of memory and 1 Gbps fiber network connection. The web server kept access logs, which include pathnames of accessed pages, time stamps, client IP addresses, etc. The traces used in the tests were recorded from 00:01:34 through 18:01:48 on March 1, 1999.

Reconfiguration	Addition	Removal
Time (minutes)	26.04	26.03
Migrated (GB)	2.58	2.58
Migrated (files)	52152	52134
Migrated (dirs)	2964	2954
Bytes Before(GB)	2.52, 2.64, 0, 0	1.29, 1.29, 1.29, 1.29
Bytes After (GB)	1.29, 1.29, 1.29, 1.29	2.58, 2.58, 0, 0

Table 4.6 Statistics in the addition and removal of two islands without client accesses. The row "Time" shows the elapsed time in minutes since the reconfiguration started till the migration of data was completed in all islands. The next three rows show the migrated bytes, files and directories during the reconfiguration, respectively. The last two rows show the byte distribution across four islands before and after the reconfiguration, respectively. (I use the number of bytes as the measure for server loads for simplicity in these experiments.)

I simulated the web server with 16 threads on separate machines, reading the access log and issuing requests to Archipelago as clients. The absolute time stamps in the log were ignored and the traces in the log were consumed as fast as possible. Each thread issued 3000 requests per test. 699 MB of data in 48000 files were accessed in each test, of which 86 MB of data and 7218 files were distinct.

I ran the simulation in five different cases relevant to the addition of two islands and measured the impact of data migration on client performance. The migration was expected to affect client accesses in three ways. 1) The background migrators compete with the clients for resources like disk and network bandwidths. 2) When the clients try to access files in the new islands, some files have to be migrated on demand from the old islands. 3) On-demand migration causes race conditions, which are detected and removed by the migrators and are made transparent to the clients.

In addition to running the simulation before and after the reconfiguration, I ran the simulation in three cases during the reconfiguration to separate the impacts of different sources. 1) I ran the simulation in the beginning of the reconfiguration to see the impacts of both background migration and on-demand migration. 2) I ran the simulation again, later in the same reconfiguration; since all the requested files had been migrated on demand in the first simulation, the slowdown in this case came solely from the migrators' competition. 3) I reran the reconfiguration and simulation with the migrators disabled to

see the slowdown solely from on-demand migration. Table 4.7 shows the results of the simulated web accesses in the five cases.

The difference between the "background" and "after" cases indicates that the background migrators had a minor impact on the client performance, consuming only 7% of the overall disk bandwidth and imposing a performance penalty of only 4.5%. The percentages are dependent on the relative numbers of migrators to clients, i.e. 2 to 16 in this case. The difference between the "on-demand" and "after" cases shows that client bandwidth was nearly halved by on-demand migration because the amount of data transferred to satisfy a request was doubled. The disadvantage of disabling migrators is that the first accesses to files in the new islands will always require on-demand migration and will see a significant performance drop. Additionally, without migrators, a system administrator cannot tell when exactly the migration is completed. Therefore, enabling migrators is a good idea.

I also recorded the number of files migrated on demand and the number of race conditions caused by on-demand migration. (The race conditions in the "background" case occurred when the migrators initiated on-demand migration for directory attributes replication.) The numbers of race conditions were relatively small compared to the number of files migrated on demand, i.e. a race condition was detected and removed for every 86 to 172 migrated files. With on-demand migration, the system reconfiguration was made transparent to the clients.

Cases	Total Client Bandwidth (MB/s)	Total Migrator Bandwidth (MB/s)	Migrated files	Race conditions
Before	3.94	0	0	0
Both	4.52	0.36	7191	84
On-demand	5.68	0	7218	42
Background	9.05	0.68	0	8
After	9.48	0	0	0

Table 4.7 Results in the simulated web accesses. The five cases are before the addition of two new islands (before), with both background migrators and on-demand migration (both), with on-demand migration only (on-demand), with background migrators only (background) and after the addition of two new islands (after). The total client bandwidth is the total number of bytes accessed by 16

threads during the simulation divided by the simulation time. The total migrator bandwidth is the total number of bytes read and written by the 2 migrators during the simulation divided by the simulation time. The column "Migrated files" shows the number of files migrated *on demand* during the simulation. The column "Race conditions" shows the number of race conditions during the simulation due to on-demand migration.

The measured impacts of background migrators and on-demand migration in the reconfiguration tests also apply to the cases of renaming directories (Section 2.2.4.1) because these two procedures share most of the code.

4.4 Related work

Existing file systems designed for high availability, such as Coda [54] and Ficus [51], replicate data across servers. Our approach in island-based file system, i.e. failure isolation, is complementary to the data redundancy approach for high availability. Client caching is extensively used in distributed file systems like Coda [54], Andrew [8] and Sprite [40] to support disconnected operations and to reduce traffic to servers. Similar to server replication, client caching improves availability by data redundancy, i.e. by replicating data in clients. It also improves scalability by reducing server load so that the same number of servers can serve a larger number of clients gracefully. Our scalability goal in island-based file system is to achieve efficient speedup when servers are added to the cluster, which is orthogonal to the goal of client caching. We have not implemented client caching in Archipelago, but we do not expect the island-based design to add any difficulty to such implementation.

State-of-the-art cluster file systems like Frangipani [1] and xFS [5] achieve high reliability and scalability by data redundancy. A fast system area network such as ATM is typically used in those cluster file systems for aggressive communications across data replicas. The majority of operations in island-based file systems do not require communication or synchronization across islands; therefore, an island-based file system can scale efficiently with commodity networks such as Ethernet. The ideal configuration for maximal reliability, availability and scalability is to run an island-based file system with a file system like Frangipani or xFS inside each island.

In terms of failure isolation, cross-node communications, locality and leveraging functions in local file systems, island-based file systems are comparable to distributed file systems like NFS [9], JetFS [15] and CIFS [15]. However, those systems do not share with island-based file systems scalability, load balance, and/or automatic data partitioning and reconfiguration.

In Teradata [56], two orthogonal hash functions are used to map data items to two nodes. In an island-based file system, each data item is mapped to a single island but redundancy might be used inside the island. The Teradata approach offers better load balance when a single node fails, but the failures of two nodes always render a portion of data inaccessible. Our approach makes most operations involve a single island, isolates failures across islands, and does not lose data unless all replicas in the same island fail.

The Cluster-Based Scalable Network Services (SNS) [47] [48]provides a programming model for Internet applications. In particular, the authors proposed application decomposition and orthogonal mechanism for graceful degradation during partial failures. Island-based design addresses failure isolation in the storage system. While the programming-model approaches suffice for read-mostly access patterns and weak consistency requirements, a robust and scalable storage system is necessary for services with read-write access patterns and strong consistency requirements, such as shared calendar services and online shopping sites. The combination of the approaches can potentially achieve high availability and scalability for Internet services with a wide range of access patterns and consistency requirements.

4.5 Summary

I have designed the island-based file system as the data storage for highly available and scalable Internet services. I evaluated the design by statistical analysis of the access patterns in existing systems. I implemented Archipelago, a prototype of the island-based file system, and studied the performance of Archipelago in micro benchmarks and operation mixes.

I draw the following conclusions on this part of my work:

1. The failure isolation provided by the island-based file systems is useful to Internet services because a temporary partial failure can be made unnoticeable to the majority of clients.

The island-based file system can scale well with the system and workload sizes because the majority of operations do not require communication or synchronization across islands.

5 Affinity-based management of clustered in-memory databases

I shall start the discussion in this chapter by considering how to improve the performance of the new generation of Internet servers (or "application servers") in a cost-effective way, i.e. how to achieve high quality of service with minimal committed resources. In today's Internet data centers [89], the ability to guarantee the same service level agreement with less committed resource allows savings in equipment cost, power consumption, environmental control, rack space, and administration effort.

While the old generation of Internet sites present mostly static content, many popular sites today, especially e-businesses, web portals and search engines, provide value-added services as well as information to their clients in the format of dynamic content. *Dynamic content* is generated on demand by applications that access and perhaps update back-end databases and communicate with the web servers using protocols such as Common Gateway Interface (CGI) and Server Application Programming Interface (SAPI). S*tatic content* refers to files that web servers directly read from file systems, such as HTML pages and images. The growing popularity and the order-of-magnitudes slower delivery rate of dynamic content [88] led to the development of *application servers* [90]. Application servers are typically persistent processes that perform state and session management in addition to the roles of traditional CGI programs.

I discuss in this chapter how to improve the performance of application server infrastructures in a cost-effective way by caching dynamic content. Although caching and buffer management inside database systems have been extensively studied [95] [97], traditional databases are optimized for disk I/O or client/server communications rather than for memory-resident operations. Therefore, light-weighted caching techniques for Internet infrastructures are being investigated.

Caching query results has shown to reduce CPU cost and disk access involved in generating pages that do not change frequently and do not cause updates to the underlying database [66]. Application-specific annotation is usually needed for managing

cached results due to the diversity in contents and operations [67]. Result caching has the following limitations: 1) Certain queries modify data and hence cannot be cached. 2) Certain applications need to be rewritten to explicitly keep cached results up to date. 3) There are often overlaps across different query results and cache space is wasted on duplicate information.

An alternative to result caching has been proposed: interposing between the application servers and the backend, on-disk databases an *in-memory database* that caches the frequently accessed data from the on-disk databases [85]. In-memory databases [86] are optimized in many aspects, such as retrieval and indexing, specifically for memory-resident data, and can be backed up with Uninterrupted Power Supply (UPS) for the durability of transactions. Unlike result caching, which is often done in a proprietary or application-specific fashion, in-memory databases are available as commodity software and provide the same, standard functions as traditional, on-disk databases, such as indexed search, concurrency control and consistency guarantee. Therefore, use of in-memory databases can potentially help reduce the time to market in building scalable and cost-effective services. Since an in-memory database caches raw data rather than query results, there will be no duplicates of cached data across queries.

However, it is not unusual that the size of an on-disk database, e.g. tens of gigabytes to terabytes, exceed the physical memory capacity of a single commodity machine, e.g. hundreds of megabytes to a few gigabytes. A single in-memory database might limit the scalability of the entire infrastructure due to thrashing or CPU saturation. Intuitively, a scalable and cost-effective cache might be composed of a cluster of in-memory databases, each of which runs on a commodity computer and contributes to the aggregate memory capacity and computing power of the cluster as a whole.

A well-designed management system for such a clustered in-memory database is needed in order to achieve the projected scalability and cost effectiveness. The task of management is to automatically and dynamically partition and replicate data across individual databases and to direct queries to the right databases, while maintaining the consistency across databases at a low cost. The goal of this task is to maximize effective

cache capacity and minimize synchronization cost.

Conflicts across queries for dynamic content, i.e. the facts that dynamic content can be accessed through multiple applications or by multiple attributes and that a single query can access multiple data items, raise challenges for the management of clustered in-memory databases. The conflicts result in data sharing across individual databases. If the data is read only, sharing causes data replication or transmission across databases. If the data is written, e.g. when a query implicitly causes an update to a customer preference database, one needs to pay synchronization cost for the data sharing. Therefore, data sharing across individual databases needs to be reduced for both read-only data and write-shared data in order to improve the performance-to-cost ratio of clustered in-memory databases.

In the rest of the chapter, I investigate scalable and cost-effective management strategies for clustered in-memory databases. First, I will show that a good partitioning of dynamic content does exist in certain applications despite the conflicts across queries. Second, given that a good partitioning exists, I propose a distribution strategy in the management system that implements such a partitioning by handling the conflicts properly. I evaluate the distribution strategy in comparison to alternative strategies that handle the conflicts to different degrees. Finally, I present the performance comparison of two prototype clusters, a clustered in-memory database backed by an on-disk database, and a replicated on-disk database with large buffer cache.

5.1 Assumptions

The discussion in the rest of this chapter is based on the following assumptions about the cluster-based infrastructures. A number of *processing nodes* run web servers and application servers, and each web server or application server is capable of processing any HTTP or application-specific requests. The application servers store all their persistent data in a shared, on-disk database, which I call the *master* database. A clustered in-memory database is *transparently* situated between the application servers and the master database, caching partial or all data from the master database. An individual node in the clustered in-memory database is called a *cache server*. Data accessed by a query

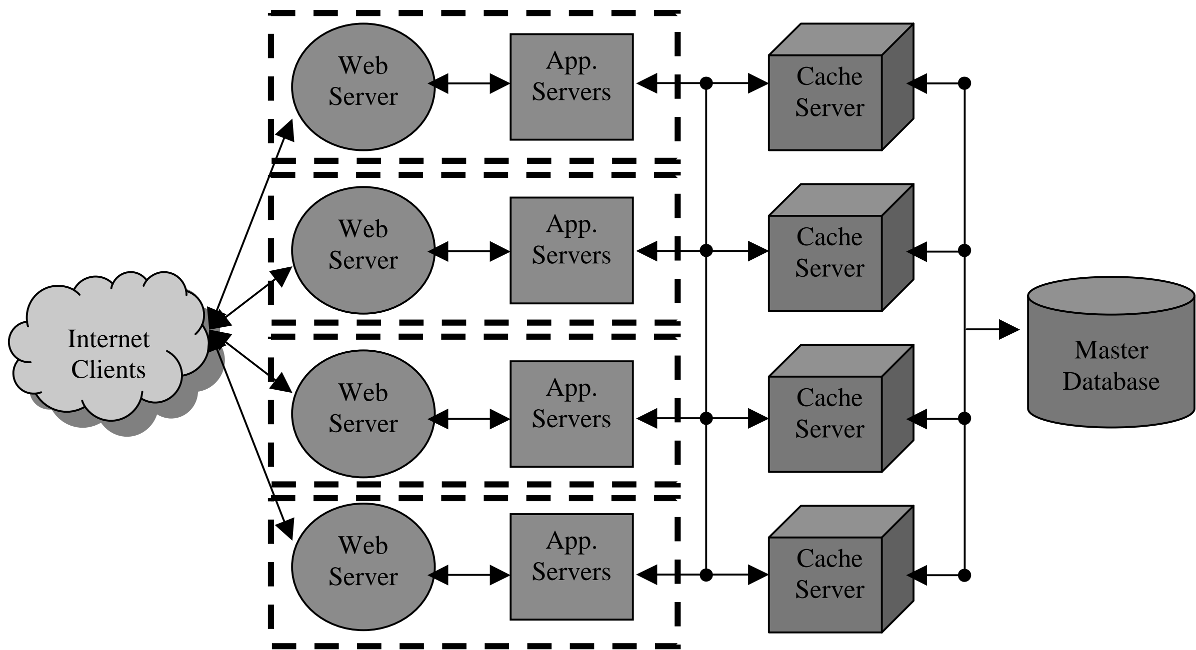

Figure 5.1 Clustered in-memory databases in Internet infrastructures. A cache server is an individual node in the clustered in-memory database.

will be *loaded* from the master database into a cache server before the query is executed in that server. Data will stay in the cache server until it is *evicted* by a cache replacement policy or *invalidated* by a synchronization protocol. The cache servers may or may not be physically co-located with the application servers. See Figure 5.1.

5.2 Challenges of dynamic content illustrated

As discussed in the previous section, the challenges in the management strategies are mainly raised by the conflicts across queries for dynamic content, which result in data sharing across nodes in the cluster. It can be best explained by examples.

In the first example, I consider an online bookstore like Amazon.com, where books can be queried by subject, by author or by ISBN. In an ideal case, the books are cached in a minimal number of in-memory databases, i.e. no data is stored redundantly; each query can be executed in one of the individual databases without data transmission from any others. Unfortunately, queries by different attributes require the books to be partitioned in different, possibly conflicting ways. For instance, if the books are assigned to individual databases by a random hash function on their ISBNs, none of the individual databases

could guarantee to contain the complete results of any query by author or subject. Even if the books are partitioned in the ideal way, it is still hard to direct the queries to the right databases for execution without knowing the query results first.

In the second example, I consider two queries to a general search engine like Google, one containing the keyword "herbs" and the other "vitamins", and a highly ranked article on "healthy food". The search engine will typically return this article to both queries. Similarly to the first example, it is hard to direct the two queries to the database that caches this article. In fact, it is even hard to predict that the two queries should be executed in the same database without executing them first.

Both examples above are about read-only data, in which case the conflicts across queries result in inefficient use of memory space. If the data is written as well, e.g. when a query implicitly causes an update to a customer preference database, one needs to pay the cost of communication and synchronization across nodes for maintaining the consistency of write-shared data. Therefore, data sharing needs to be reduced for both read-only data and write-shared data in order to improve the performance-to-cost ratio of clustered in-memory databases.

5.3 Observation on query affinity

Despite the variety of the contents and services that the Internet provides, I observe query affinity in a wide range of applications, including e-commerce, search engines, maps, directories, news, and digital libraries. By *query affinity* I mean the fact that there exists a way of dividing queries into groups where queries in the same group access the same or overlapped data sets (I call them *affined* queries) while queries in different groups access separate data sets. In addition, query affinity is a natural result of the content structures or access patterns of the application, rather than the result of the physical data storage.

I hereby introduce two important sources of query affinity:

- **Containment**: Data accessed by certain queries tends to *contain* data accessed by certain other queries. In the previous bookstore example, the books written by an

author, e.g. Biran W. Kernighan, often belong to a few particular subjects, e.g. computer programming, which the author has expertise knowledge in. In another example, such as MapQuest, the map of California contains the map of Palo Alto because California geographically contains Palo Alto. The containment relationship is transitive; therefore, there might exist chains of queries in an application where queries in the front contain queries in the back. In this case, data sets accessed by different containment chains are separate from each other.

- **Ranging**: In range searches, data items "close" to each other in a domain-specific sense are often accessed together in the same query while data items "far" apart are rarely accessed together. Examples include searches for restaurants by distance to a given location, searches for articles by a range of publishing dates, and searches for people by similar names.

Table 5.1 shows a summary of instances of query affinity in a set of popular Internet services.

Services	Containment	Ranging
Book stores	Subject ⊃ author ⊃ ISBN	Books with close publishing dates
Auctions	Category ⊃ seller ⊃ item	Items with similar prices or locations
Maps	Country ⊃ state ⊃ city ⊃ zip code	Geographically close places
News	Category ⊃ sub category ⊃ article	Articles of related topics
Yellow Pages	Category ⊃ brand name ⊃ retailer	Geographically close businesses
White Pages	State ⊃ city ⊃ phone	People with similar names
Digital Libraries	Subject ⊃ journal	Papers in related areas
Search Engines	General keyword ⊃ specific keyword	Documents with similar keywords

Table 5.1 Query affinity. The "containment" column shows the containment chains in the databases. The "ranging" column shows the "close" items in the databases.

5.4 Exploiting query affinity

The presence of query affinity indicates that a good management strategy could be found for clustered in-memory databases. If affined queries are directed to the same node, the data sharing across nodes can potentially be reduced. Queries on different nodes are

likely non-affined queries, and hence will access separate data sets by definition. Furthermore, if I can direct each query to the same node as its affined queries as it comes in, I can make online, localized decisions on query distribution, rather than search the large solution space offline for a globally optimal solution. Intuitively, query affinity may be exploited to find a management strategy that is as effective as the back-end data allocation approaches, and as light-weighted and dynamic as the front-end request distribution approaches.

In order to direct a query to the same node as its affined queries as it comes in, I need the following inputs:

1. To which queries in the past this query is affined.

2. To which nodes the affined queries have been directed in the past.

Based on the facts that affined queries access the same or overlapped data and that data accessed by recent queries is kept in the cache servers where the queries were executed, the two required inputs to the management can be reduced to the following two inputs respectively:

1. What data the query will likely access.

2. In which nodes some or all of the data is currently cached.

By selecting the node that currently has in its cache the most data for the query, I can effectively partition data across nodes with reduced sharing.

This strategy works only for applications that exhibit query affinity, and the quantitative reduction in data sharing is a function of the *dimensions* of query affinity, which I define as the average number of separate groups a query is affined to. The ideal number of dimensions is 1. Larger dimensions result in more conflicts across queries and hence less reduction in data sharing. In range searches, the conflicts exist in the situations where a data item is "close" to more than one separate set of data items. In the containment case, the conflicts may arise if the containment relation is not strict, i.e. the data accessed by a

query spreads across the data sets of more than one containing query. For example, the books written by Isaac Asimov belong in many different subjects.

5.5 Affinity-based management

I design an affinity-based management (ABM) strategy that strives to maximize effective cache capacity and minimize synchronization cost in clustered in-memory databases. The basic approach in ABM is to divide the execution of each query into two stages. The first stage is computation-intensive; the function and data required for the first-stage execution are replicated with every database client. The second stage is data-intensive; data accessed during the second-stage execution is partitioned into individual in-memory databases. The first stage determines the set of data that the query will likely access, which is then used to determine the destination of the query in its second stage. The second stage completes the rest of the query execution and generates results. The intention of the two-stage execution is to determine the destination of each query with the knowledge of the data to be accessed.

In the discussion below, I assume that readers are familiar with the following database terms: *table* (a collection of data items of the same structure or having the same *attributes*), *row* (a data item with the complete set of attributes in a table), *row id* (the unique id of a row in a table), *column* (an attribute of all data items in a table), *search key* (a sequence of one or more attributes that is used to identify rows, uniquely or not, in selections), *index* (an auxiliary structure on a search key that is designed to efficiently locate all rows with a given value of the search key), *selection condition* (a Boolean combination of the comparisons of attributes to constants or other attributes), etc.

In order to build the ABM system without modifying existing in-memory databases, I decompose each original query into two sub queries, called *selector* and *imposter*, which can be executed as regular queries in the unmodified local databases. The two stages of execution are then implemented as the executions of the two sub queries. The selector consists of all or part of the selection conditions and operations on their results, e.g. set operations or join operations; in other words, it consists of the components in the original query that identify the set of data to be accessed. The selector returns the unique ids of

the matching rows in database tables. The imposter is transformed from the original query by replacing the components that appear in the selector with the result of the selector. The imposter returns the same results as the original query but requires less execution time because it incorporates the results of the selector rather than repeats executing the selector.

5.5.1 Components

A clustered in-memory database with ABM consists of the following components:

Database clerk: It is installed on each processing node and intercepts function calls to the master database client library, made by the application servers. Its major functionality is to decompose original queries into selectors and imposters and to execute the selectors locally. The search keys used in queries are replicated in each database clerk for local execution of the selectors. The replicated keys are kept consistent with the master database in the face of updates. This will be discussed in more detail in Section 5.5.3.

Database manager: It maintains information on data locations, server loads, and read/write locks. It consists of the following functional components, which share a fair amount of information in common:

- *Location manager*: It keeps track of the current location, i.e. cache server, of data at row granularity. With the results of the selector as input, it outputs a list of candidate cache servers sorted in descending order of cache hits.

- *Load manager*: It keeps track of the current load, e.g. number of outstanding queries, of each cache server. With the list of sorted candidate servers from the location manager as input, it selects the server that has the most cache hits and is not *overloaded*, as defined by a set of adjustable thresholds. If there is no candidate server, i.e. the required data is not cached anywhere yet, or if all candidate servers are overloaded, the load manager selects the currently least loaded server. With this load balance mechanism, a small amount of frequently accessed data will be replicated or migrated across servers on demand.

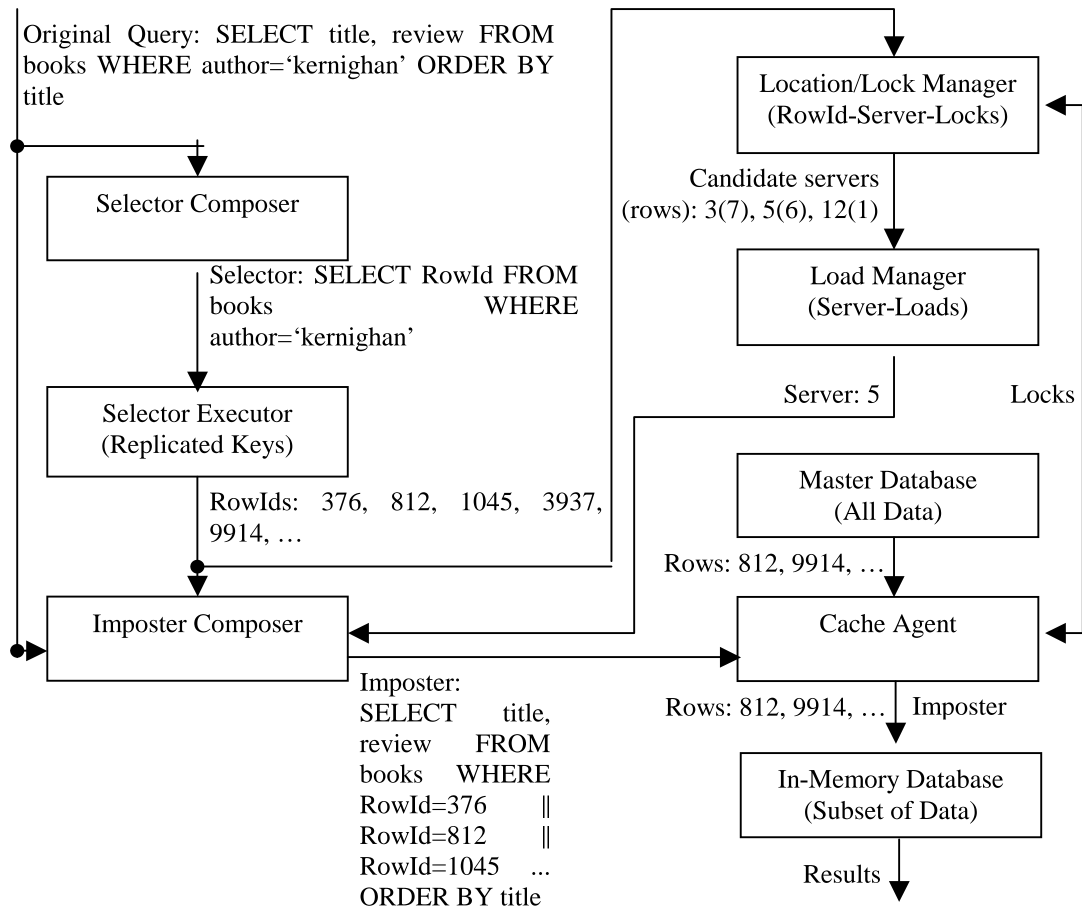

Figure 5.2 A clustered in-memory database with affinity-based management. The data noted in parenthesis in some components of the infrastructure is the data stored with those components.

- *Lock manager*: It maintains a consistency protocol across cache servers at row granularity. Multiple servers can hold a read-only copy of the same data simultaneously. When a server asks for a write lock on the data, the read-only copies will be invalidated from any other servers; otherwise, data will stay in cache till it is evicted as a result of cache replacement. Upon eviction of a row, the server contacts the lock manager for releasing the lock.

Cache agent: It is installed on each cache server and is responsible for loading data into and unloading data from the in-memory database on that server in response to events like cache miss, cache eviction and cache invalidation. Each imposter is sent to the selected

server agent for processing. The agent first makes sure that all the data the query needs is cached locally. If any data is missing, it loads it from the master database. This is an important step because, without this step, the affined queries sent to the same server in the future will *not* be guaranteed cache hits. Finally, the co-located in-memory database server executes the imposter as a normal query. (However, there is an upper limit on the number of rows that can be cached for a single query; if a query exceeds the limit, it will be sent to the master database rather than to a cache server.)

Figure 5.2 illustrates the data and control flows in a clustered in-memory database with ABM.

5.5.2 Data distribution and consistency

The data distribution and consistency maintenance in the affinity-based design can be summarized as follows: A large table in database, which does not fit in a single in-memory database, will both be *horizontally* fragmented and *vertically* replicated.

The horizontal fragmentation refers the distribution of data across cache servers at row granularity, as discussed in the previous section. A fairly standard algorithm is used in the lock manager to maintain the consistency of horizontally fragmented data: namely a synchronous, avoidance-based algorithm with invalidation for remote updates [71].

The vertical replication refers to the replication of the columns used in search keys in each database clerk for the local execution of selectors. I will discuss the vertical replication in more detail in the next section.

5.5.3 Replication of search keys

The columns used in search keys are typically of short or "narrow" data types, such as integers, timestamps, short character strings, etc. Columns of long or "wide" data types, such as texts, images, and streaming data, are either stored inside the database as regular columns or stored as files in external file systems and represented in the database as references to the external files [74]. In either case, it would be overwhelming to use the "wide" columns directly in search keys. For example, *inverted files* (containing

<*keyword*, *document-id-list*> pairs) are generated on the *text databases* of search engines for fast query evaluation and document retrieval [89]. In this case, the *keyword* columns in the inverted files are "narrow" columns and are used as a search key, while the *document-id-list* columns in the inverted files and the full documents in the text databases are "wide" columns and are not used in search keys.

Based on this classification of columns, I vertically divide a large table into two parts, one with "narrow" columns (or search keys) and the other with "wide" or non-searchable columns. I replicate the part of search keys together with the row ids in database clerks. The replication enables the database clerks to locally process the selector queries that involve the replicated keys only. And yet the space required for replicating search keys is strictly less than the space required for replicating the entire table.

In the master database, indexes are typically built on the searchable columns to speed up queries. For the same reason, indexes are built for the replicated keys in the database clerks. However, indexes on the replicated keys are not needed in the cache servers because, in the cache servers, data is accessed by row ids only. (See the description on the imposter queries in Section 5.5.) Therefore, the horizontal and vertical data distribution in this design does not require any more space for indexes than a fully replicated database.

Any update to the database will be executed in the master database before the update operation finishes. The semantic guarantees of the master database, such as the ACID (Atomicity, Consistency, Isolation, Durability) properties of a transactional database, will remain intact because, with or without the caching and replication, the master database will receive the same sequence of updates and execute them in the same way. With the caching and replication, the master database will receive a smaller number of read-only queries, which do not alter the state of the database.

Any update that results in modifications to the replicated keys, such as insertions, deletions or updates on the replicated columns, will be broadcast to all replicas. However, only the replicated attributes are broadcast, but not the entire update operation. For

example, assume that the data item on a new book is inserted to a table of books, which consists of the search keys title and author, and the non-searchable columns review text and cover page image. Only the title and author of the new book are broadcast to the replicas for insertions, but not the review text or the cover page image. Therefore, the network bandwidth required for broadcasting the updates on replicated keys is strictly less than the bandwidth required for broadcasting the updates in a fully replicated database.

However, the replication of search keys raises a challenge for consistency, as in any other situations where data replication and updates to replicated data are present. The replication of search keys can be generalized with the same replication model as the metadata replication in the island-based file system (Chapter 3). In this particular case, the search keys are the *replicated objects*, the master database has the *primary* copy, the database clerks and the cache servers have the *secondary* copies, the master database is the *coordinator* of all updates on the search keys, and each update on the search keys *involves* all the database clerks and the cache servers that have a copy of the updated search keys. A single logical clock is maintained for all the replicated keys. The master database, the database manager, the database clerks and the cache agents keep a local copy of the clock each. The broadcast of updates and synchronization of clocks use the same protocol as described in Chapter 3. The cache servers are a special case of the model: they do not have a copy of the replicated keys unless they are co-located with the database clerks, but they participate in the consistency protocol as replicas because it is implicitly assumed that the cache servers have a consistent view of the database state with the database clerks. To determine the set of cache servers involved in an update, the location manager is consulted before the commit of the update starts. Another special case in the replication of search keys is that the secondary copies will be completely lost during node crashes. The write-ahead logging is no longer necessary because, during the recovery of a database clerk after a crash, a complete snapshot of the replicated keys, rather than updates that occurred after the crash, will need be copied from the master database.

The customized consistency protocol maintains the following invariants in a clustered in-memory database with ABM:

4. All updates on the replicated keys are atomic, e.g. if a query is pre-executed with a certain update in effect, the same update is guaranteed to be in effect in the selected cache server when the query is executed there unless the cache server is not involved in that update.

5. All updates on the replicated keys are serialized, e.g. queries are pre-executed and executed as if all relevant updates have been committed in the same order both in the database clerk and in the selected cache server.

6. In most cases, read-only queries can be executed locally, i.e. without contacting other database clerks or cache servers for synchronization purpose.

5.5.4 Limitations

Due to the replication of search keys, a table in the database needs to satisfy at least one of the following conditions in order to benefit from ABM:

- The ratio of the space required for search keys to the space required for non-searchable data is low; or

- The ratio of the updates on search keys to the updates on non-searchable data is low.

Applications with massive, non-searchable data, such as images of maps, description texts of merchandises, and PDF files in digital libraries, can potentially benefit from ABM. Applications with all or mostly searchable attributes in their data, such as a simple phone directory, might perform just as well with a fully replicated database.

Another limitation of ABM is the potential bottleneck and single point of failure in the database manager. In general, clustering techniques [76], such as fail-over and mirroring, can improve the scalability and availability of the manager at the cost of additional hardware and fast interconnection. An alternative is to divide the data or functionality of the manager into finer grains. For example, the tables in a large database can be divided

into smaller groups, as long as no query will access data across groups, and a manager can be assigned to each group, rather than the entire database. The centralized manager can also run as three separate managers, the location manager, the load manager and the lock manager, at the cost of storing redundant information and passing synchronization messages across different managers.

As a bottom line, the design of ABM guarantees correctness in all situations and offers lower cost and better performance than a fully replicated database.

5.5.5 Implications to other systems

Containment and ranging are two prevalent sources of query affinity in today's Internet applications. As new types of services evolve, I expect that they will share these properties and may also expose new sources of query affinity or other insights that can be generalized and exploited in the construction of scalable infrastructures. I believe that the general approach to managing clustered in-memory databases by exploiting query affinity is valuable.

The approach to exploiting query affinity presented in this chapter is a dynamic one. The key insight of exploiting query affinity, however, can be exploited by other means as well. For example, application writers can redesign the URLs or cookies in the HTTP requests to carry enough information for a front-end request distribution. Such an approach is a static one and has the advantage of being able to leverage a front-end distributor. However, it requires the applications to be modified substantially, requires the requests to carry information that is not necessary for processing the requests alone, and requires additional intelligence and human assistance in the front end.

In the design of the clustered in-memory database with ABM, I choose relational database vs. object-oriented database or file system as the type of master database in the infrastructure, because relational databases are widely available as commercial or open source products and are widely used in the back end of Internet infrastructures [84] [58]. However, the insights on query affinity are independent of physical data storage and can be exploited for other types of storage systems as well.

5.5.6 Implementation

I have implemented a prototype of the clustered in-memory database with ABM on a cluster of Linux servers connected by switched Ethernet. The database manager runs as an RPC server. I implement most of the database-related functionality on top of the MySQL database [58], an open source, relational database. The database clerk in my implementation intercepts the function calls to the C library of MySQL clients. (MySQL drivers for other languages, such as PERL, were built on top of the C library.) The in-memory tables of MySQL, called *heap tables*, are used to construct the in-memory databases in the cache servers. The replicated columns for the database clerks are stored in heap tables as well. In this way, both imposter queries and selector queries can be processed as regular queries on the heap tables in the unmodified MySQL servers. The heap tables in the cache servers have the same attributes as the on-disk tables in the master database, while the heap tables for the database clerks have the search keys only. If a database clerk is co-located with a cache server, both can share the same local MySQL server. In theory, such an infrastructure can work with any master database that supports the standard query language. In my experiments, I run a MySQL as the master database.

5.6 Simulations of five distribution strategies

As discussed earlier in this chapter, handling conflicts across queries for dynamic content is the main challenge for the management of clustered in-memory databases. The task of the management system can be separated to two parts: consistency maintenance and data/query distribution. While the goal of consistency maintenance is correctness, the goal of data/query distribution is to maximize effective cache capacity and minimize synchronization cost by handling conflicts across queries properly. In this section, I present the evaluation of the distribution strategy in ABM in comparison to alternative distribution strategies that handle the conflicts to different degrees. With a set of simulations, I study the impact of the following factors on the performance of various distribution strategies: applications, access patterns, human assistance, dimensions of affinity, memory sizes and cooperative caching. This study is analogous to the study on

request distribution strategies for clustered web servers [3].

5.6.1 Experimental setups

I use a modified version of the web server cluster simulator previously used for distribution strategies [3]. The original simulator models the scheduling of CPU queues, disk queues and incoming request queues as well as activities on the main memory cache in the server machines. It assumes that the entire data set is replicated on the local disks of all server machines and a subset of data is cached in the main memory cache of server machines where it is frequently accessed. A request is processed in the following steps: connection setup, disk reads (if needed), target data transmission, and connection teardown. Parameters such as memory size, CPU speed, disk speed, network speed and caching protocol are configurable. A detailed description of the original simulator can be found in [3].

The major modifications I make to the original simulator in order for the simulator to work for application servers and database servers rather than web servers and file systems are following. For write-shared data, I model a multi-reader-single-writer locking protocol for cache consistency at row granularity. (See the lock manager described in Section 5.5.1.) Each lock-related operation is charged for a round-trip network latency. In addition to the steps of connection setup and teardown for each query, each accessed row in the query is processed in the following steps: lock acquisition (if needed), disk reads (if needed), data processing or transmission, and writes (if needed). For written data, I assume an asynchronous cache write-through policy; that is, I charge the CPU overhead for sending data to disk, but not disk write time. Therefore, written data is immediately visible to successive reads. In fact, I do not charge the delay caused by synchronization in any lock-related operations. These assumptions are conservative with regard to the benefits of ABM because they lower the performance penalty of the events that ABM is designed to reduce, i.e. shared writes.

Table 5.2 shows the parameter settings common in all simulations reported in this chapter.

Parameters	Values
Connection setup and teardown time per HTTP request	750 μs
Initialization overhead per request to an application server	1 ms
Data processing or transmission time per 512 bytes	85 μs
Disk transfer time per 512 bytes	50 μs
Average disk seek time	10 ms
CPU overhead per 512 bytes accessed on disk	4 μs
Memory page size	1 KB
Replication of hot data allowed	Yes
Cache replacement policy	LRU

Table 5.2 Common parameter settings. Data is cached at row or page granularity, whichever is larger.

I compare the following five distribution strategies in each experiment:

1. Weighted round-robin distribution (WRRD): Queries are distributed to cache servers in a round-robin fashion, weighted by the servers' loads. This is analogous to a front-end, connection-based distribution strategy for clustered web servers [75].

2. Query-based distribution (QBD): Identical queries are directed to the same server, but distinct queries that return the same or overlapped data will not necessarily be directed to the same server. In practice, this strategy cannot make a good decision for queries that consist operations unrelated to selection conditions, such as an "ordered by" operation. However, such operations are not present in the simulations, which benefits this strategy. This strategy is analogous to a content-based distribution strategy for clustered web servers [3].

3. Human-assisted QBD or enhanced QBD (EQBD): Domain-specific information is added to QBD by application programmers or system administrators. The additional information helps QBD select the most relevant attributes in queries for decision making. Therefore, queries on the same data by the same selected attributes will be directed to the same server. However, queries on the same data by different selected attributes will not necessarily be directed to the same server.

4. Affinity-based distribution (ABD): This is the distribution strategy used in ABM. Queries on the same or overlapped data, i.e. affined queries, will be directed to the

same server regardless of the attributes or operations in the queries.

5. An ideal case (Ideal): This is a multi-processor machine with hardware shared memory of the same capacity as its cluster counterpart. No data replication or transmission is needed since all processors have access to all data in memory. This is viewed as an ideal case for comparison purpose.

I choose two example applications for case studies, white pages [78] and auctions [79]. These two examples exhibit the two sources of query affinity I observe, i.e. ranging and containment, respectively.

Note that all five distribution strategies achieve comparable and good load balance in the simulations, which confirm the previous results [3], and hence load balance is not discussed in detail below.

5.6.2 Case study 1: White pages

I extract the traces on a university white page service [78] from the access logs of the university web server in the year 1999, which include 809194 queries by names and 181719 queries by phone numbers, email addresses and/or departments. 23442 distinct names were queried in total. Since the results of the experiments depend on the actual data accessed by the queries as well as the queries themselves, and since the actual data is not contained in the access logs, I resend the queries on the distinct names to the web server that generated the traces, and store the returned data from the web server for use in the simulations. For queries that contain multiple names, I use the intersection of the results for each name as the results for the query, which is what I observe the original CGI program does. To avoid resending an excessively large number of requests to the web server, I exclude the queries by phone numbers, email addresses and/or departments. Due to the small size of the university, only 19534 valid, distinct people's data is accessed in total. Since I am interested in studying a large data set that does not fit in the memory of a single node, I augment the actual data set by a factor of 8. Assuming each person's data takes 1024 bytes, the augmented data set is less than 153 MB. I set the memory size of individual nodes to 16 MB to 32 MB in my experiments.

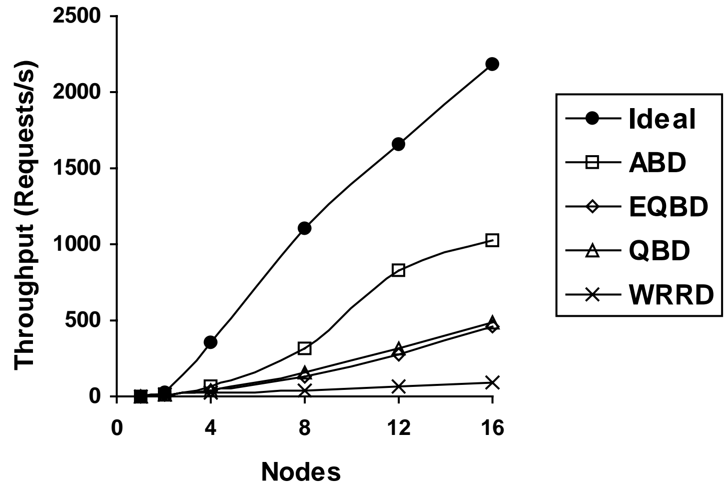

Figure 5.3 Throughput of the white page traces.

I run the five simulated distribution strategies with the white page traces. QBD and EQBD direct each query to a node based on the whole query and the first queried name, respectively. Figure 5.3 shows the throughput of the five distribution strategies on 1 through 16 nodes with 24 MB memory per node. Figure 5.4 shows the cache miss ratio, which explains the differences in throughput. (All ratios shown in this chapter are the absolute number of events, e.g. cache misses, divided by the total number of data accesses in the simulation.) The results of WRRD match those for static content in clustered web servers: the cache miss ratio does not decrease as the cluster size increases, because the most accessed data tends to be replicated in all nodes. In QBD and EQBD, queries on the same names are directed to the same node; therefore, QBD and EQBD achieve locality to a certain degree, reduce cache miss ratio and improve throughput by a factor of 5 over WRRD with 16 nodes. QBD performs slightly better than EQBD because EQBD, using a single name in each query for distribution, experiences a small degree of load imbalance. ABD reduces cache miss ratio and improves throughput by a factor of 2 over QBD and EQBD and 9 over WRRD.

I also run the simulations with 16 MB and 32 MB memory per node, respectively. The results show that, as the memory to data ratio increases, the performance difference among the different distribution strategies decreases, which matches the results observed for static content in clustered web servers [3].

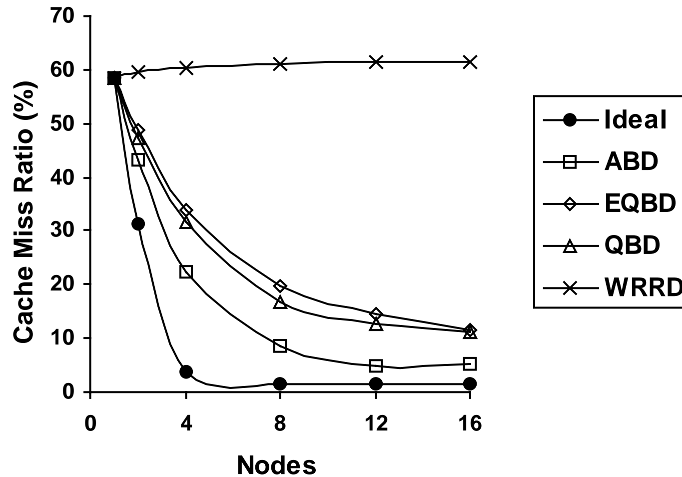

Figure 5.4 Cache miss ratio of the white page traces.

To summarize, read-only dynamic content behave largely in the same way as static content in clustered web servers under various distribution strategies except for the fact that there tends to be more data sharing for dynamic content; ABD reduces data sharing and improves throughput and scalability by directing queries on similar names, i.e. affined queries, to the same node.

5.6.3 Case study 2: Auctions

I downloaded the bid history of the first 50 completed items in 3212 categories from the well-known auction site Ebay [79]. I extracted the following events with time stamps and relevant parameters from the bid histories: 117623 *SellItem* events (with the *seller*, *category* and *item* parameters), 231521 *BidItem* events (with the *bidder* and *item* parameters) and 117623 *CompleteItem* events (with the *item* parameter).

Since these events are only a subset of the actual events at Ebay and represent only write accesses to the database, I synthetically add other events to the traces based on expected user behaviors. For each *SellItem(seller, category, item)* event, I generate a configurable number of *BrowseCategor(category)* events within half an hour before, *ViewItemsBySeller(seller)* events within half an hour after, *ViewItem(item)* events within half an hour after, and *ReviseItem(item)* events within two days after. For each *BidItem(bidder, item)* event, I generate a configurable number of

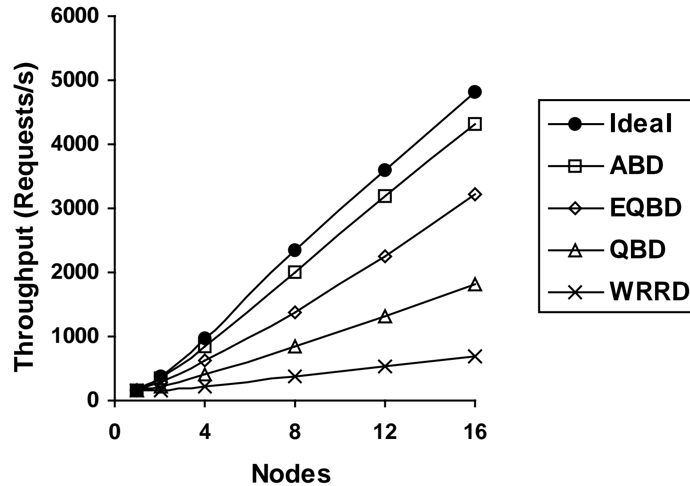

Figure 5.5 Throughput of the auction traces.

ViewItemsByCategory(item.category) events within half an hour before, *ViewItem(item)* events within 15 minutes before, *ViewBidsByItem(item)* events within 5 minutes before and after, and *ViewBidsByBidder(bidder)* events within 5 minutes after. To each generated event, I assign a time stamp that is randomly chosen within the given time range. I sort all generated events together with the original events in ascending order of time stamps and use them as the input to the simulator.

In the traces used in the simulations, there are 698288 *ViewItemsByCategory* events, 698288 *ViewItem* events, 117624 *CompleteItem* events, 231521 *BidItem* events, 115769 *ViewBidsByBidder* events, 463042 *ViewBidsByItem* events, 117623 *SellItem* events, 58729 *ViewItemsBySeller* events, and 11917 *ReviseItem* events. There are 2512801 events in total and 81% of them are reads.

The data set includes 117623 items in 3212 categories, offered by 42536 distinct sellers, and 231521 bids, made by 167752 distinct bidders. I assume that the row size of item is 4 KB and the row size of bid is 128 bytes. For simplicity, rows smaller than a page are padded to a page in cache (1 KB in the simulations). The memory footprint of all items and bids is roughly 686 MB. The memory space needed for the replicated keys, e.g. item ids and bidders, is roughly 21 MB per node. I choose 64 MB as the memory size per node so that the entire data set can fit in the memory in the best case in the simulations, i.e. a

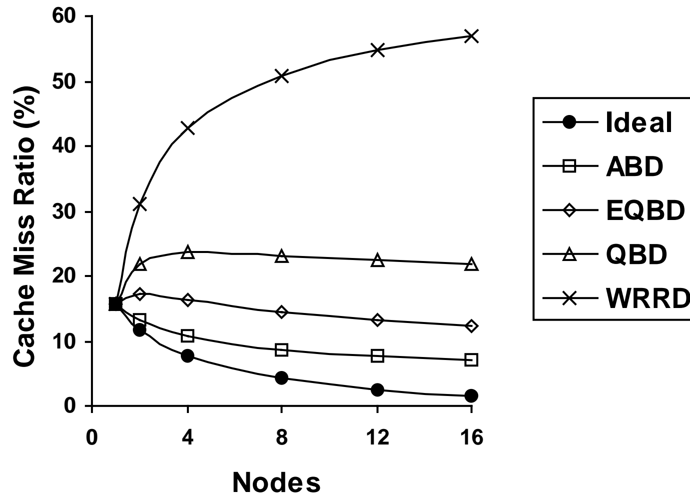

Figure 5.6 Cache miss ratio of the auction traces.

machine with 16 processors, 1 GB hardware shared memory and no replication.

I expect that the actual memory and data sizes at Ebay are much larger than the sizes in the simulations, but the relative data to memory ratio in the simulations is reasonably realistic. It is reported that there are roughly 4 million items on sale at Ebay everyday, meaning that it will take 16 GB to cache the items.

Figure 5.5 shows the throughput of the five distribution strategies on 1 through 16 nodes. Like in the white page case, the throughput is determined by the cache miss ratio, shown in Figure 5.6. Unlike the white page case, which has a read-only access pattern, the cache misses are caused both by memory pressure and by synchronization in this case. Figure 5.7 shows the cache eviction ratio as a measure of the memory pressure. Figure 5.8 shows the cache invalidation ratio as a measure of synchronization cost. The cache eviction ratio decreases as the cluster size increases except in WRRD, because the effective cache size increases with the cluster size in the other four cases. The cache invalidation ratio slightly increases with the cluster size in all but the ideal case because the number of nodes that write-share data increases. As a result, the overall cache miss ratio decreases at a slower speed than the cache eviction ratio except in the ideal case. ABD, EQBD and QBD achieve 90%, 67% and 38% of the ideal throughput with 16 nodes, respectively.

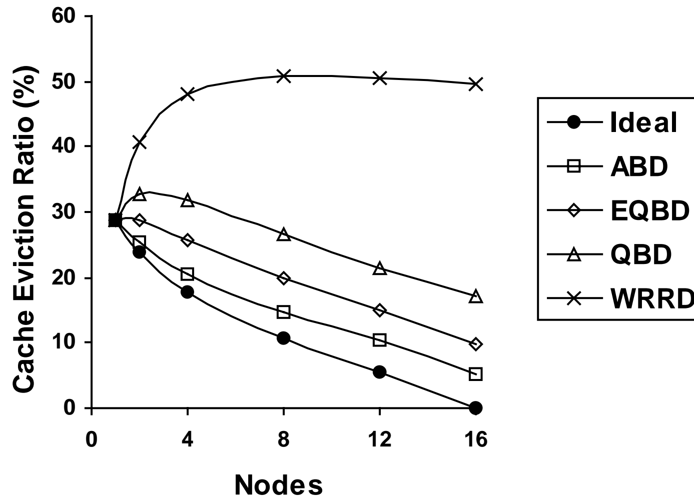

Figure 5.7 Cache eviction ratio (the number of old rows evicted from cache to make room for new rows, divided by total accesses) of the auction traces.

The throughput curves also suggest that, in order to achieve the same throughputs as WRRD, QBD and EQBD with 16 nodes, ABD requires only 4, 10 and 12 nodes, respectively.

It is known for static or read-only dynamic content that increasing memory size reduces the difference in distribution strategies (Section 5.6.2). I rerun the auction simulations with the memory size increased to 128 MB per node. The results show that there are no longer cache evictions in ABD, EQBD and QBD with 16 nodes, but the increased memory size does not help reduce cache invalidations. From 12 to 16 nodes, the cache eviction ratio is reduced from 5%, 10% and 17% to 0% while the overall throughput is improved by only 1%, 5% and 2% for ABD, EQBD and QBD, respectively.

To summarize, ABD improves the throughput and scalability of write-shared dynamic content because it reduces synchronization cost as well as memory pressure for write-shared data. The difference across distribution strategies for write-shared data is not as sensitive to memory size as that for static or read-only data because synchronization cost does not decrease as memory size increases.

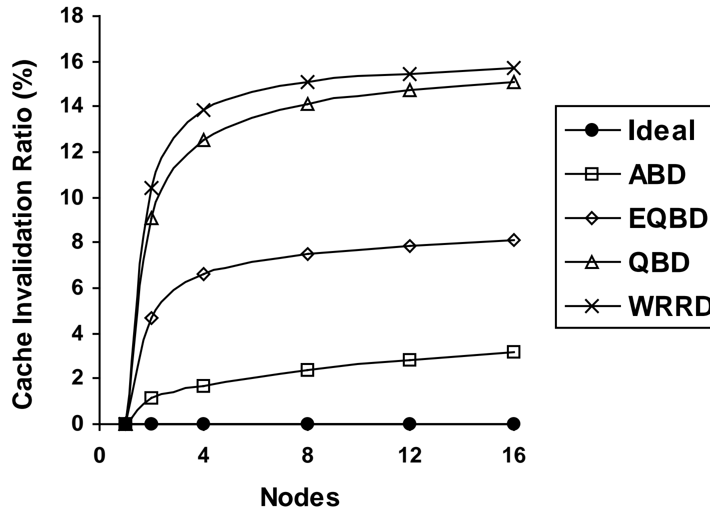

Figure 5.8 Cache invalidation ratio (the number of rows removed from cache because the node no longer holds the read or write lock, divided by total accesses) of the auction traces.

5.6.4 Dimensions of query affinity

The statistics on the white page traces show that on average each person's record is accessed by 6.8 distinct queries in EQBD and by 2.4 distinct queries in ABD. As a result, ABD reduces the average number of replicas per record from 6.8 to 2.4. The number 2.4 is in fact the dimensions of query affinity in this application. It results from the fact that each person has 2 to 3 names, i.e. the first name, the last name and probably the middle name, and queries with any of the names will access the person's record.

In the auction application, there also exists dimensions of query affinity that are larger than 1. For example, the items offered by the same seller could fall into more than one category. I examine the cache miss ratio for three queries, *ViewItem*, *ViewItemsBySeller* and *ViewBidsByBidder*, separately in ABD to study the impact of the multi-dimensional affinity on cache miss ratio. Table 5.3 shows the average and maximum numbers of containing queries and the cache miss ratio of the three queries with 16 nodes. The numbers of containing queries shown in the table are statistical results from the traces. The table shows that larger numbers of containing queries result in higher cache miss ratio.

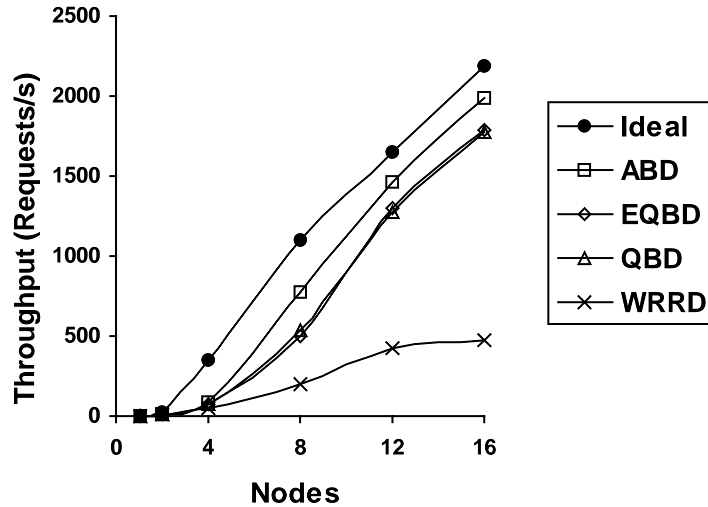

Figure 5.9 Throughput of the white pages traces with cooperative caching.

Queries	Containing Queries	Ave. Number of Containing Queries	Max. Number of Containing Queries	Cache Miss Ratio
ViewItem	*ViewItemsBy-Category*	1	1	11.1%
ViewItems-BySeller	*ViewItemsBy-Category*	2.77	197	35.6%
ViewBids-ByBidder	*ViewBids-ByItem*	1.38	32	21.9%

Table 5.3 Impact of dimensions of query affinity on cache miss ratio.

5.6.5 Cooperative caching

I study the impact of cooperative caching [72] [73] in both the white page case and the auction case. With "pull-based" cooperative caching, data can be transferred from a server's cache to another rather than be loaded from disks. The lock manager in the consistency protocol provides the locations of cached rows for cooperative caching. In the simulator, each cache-to-cache data transfer is charged 0.5 ms network latency and 6 MB/s network transfer time. Without cooperative caching, each cache miss is charged 10 ms disk seek time and 9.8 MB/s disk transfer time.

Figure 5.9 and Figure 5.10 show the throughput of the white page traces and auction traces with cooperative caching, respectively. In the white page traces, the throughputs of QBD, EQBD and ABD are all significantly improved. In the auction traces, ABD is not

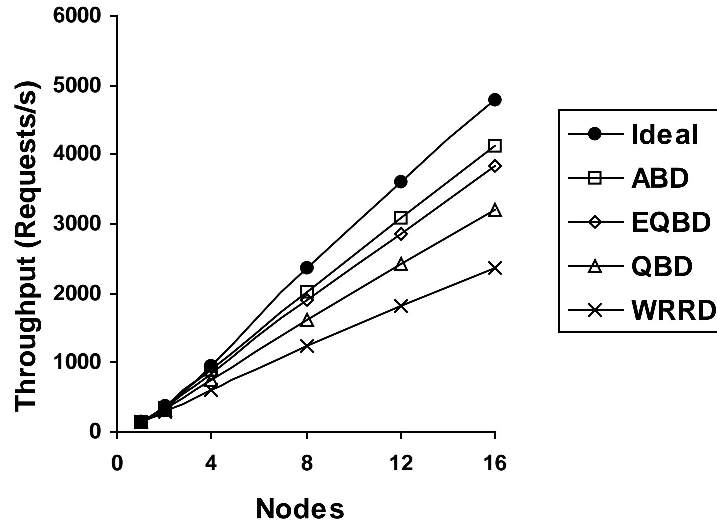

Figure 5.10 Throughput of the auction traces with cooperative caching.

improved much because it is already close to the ideal case without cooperative caching. Cooperative caching improves performance for two reasons: it replaces disk accesses with network accesses, and it helps make better cache replacement decisions due to the availability of global reference information. However, it is complementary to ABD in that it does not reduce data replication or write sharing. In other words, it does not reduce the number of costly events, but lowers their cost.

QBD benefits the most from cooperative caching and achieves 86% and 78% of the throughput of ABD with 16 nodes in the white page and auction cases, respectively. However, the results are based on conservative assumptions about the cost of cooperative caching and on the use of simplified queries in requests. In the simulator, I assume infinite network capacity, no synchronization delay and no bottleneck at the centralized lock manager for cooperative caching.

I have also studied another cooperative caching protocol, in which each row is cached and accessed in a fixed node. It makes efficient use of cache space and avoids synchronization across nodes for write-shared data. However, it outperforms pull-based cooperative caching only under high memory pressure, e.g. with 1 to 4 nodes in the simulations, and suffers from lower local cache hit ratio than pull-based protocol as

memory size increases.

Cooperative caching improves QBD and EQBD relative to ABD. However, in reality, the closing in the gap would likely be much smaller. Besides, many systems do not incorporate cooperative caching, while ABD is a general and easy-to-deploy strategy whether or not cooperative caching is present.

5.7 Performance measurement on two prototype clusters

In this section, I present the results of a set of experiments on two prototype clusters: a clustered in-memory database with ABM backed by an on-disk database (called the ABM cluster hereafter), and a replicated on-disk database with large buffer cache (called the REP cluster hereafter). It is known that various forms of replicated on-disk databases, often custom-built, are extensively used in real-world web sites [81] [82] [83]. The purpose of running the experiments is to compare the proposed cluster (ABM) with the existing cluster (REP) in terms of latency, throughput, memory usage, etc.

5.7.1 Experimental setups

The machines used in the experiments are Pentium III 500 MHz PCs with 256 MB main memories, running Linux 2.2.14. The machines are interconnected with 100 Mbps switched Ethernet. In the experiments, 1 machine is used to run the master database, 1 machine is used to run the database manager, 6 machines are used to run the clients, and 1 through 8 machines are used to run the web servers and application servers.

Apache 1.3.12 [57] is used as the web server and MySQL 3.23.18 [58] is used as the database clerks, the cache servers and the master database servers. I implement a prototype of the auction server for the experiments, using FastCGI 2.2.4 [69]. FastCGI applications are persistent CGI processes; therefore, they do not have the overhead of invoking a new process or establishing a new connection to the database upon each client request. Roughly 256 parallel client processes are running on the 6 client machines; each of them makes synchronous and continuous HTTP requests to the cluster server.

The inputs to the experiments are benchmarks based on the same Ebay traces and

additional events as in the simulations (Section 5.6.3). I generate three sets of traces with different read-to-write ratios or access patterns, namely write-intensive, read-mostly and read-only access patterns. The traces contain 67.4%, 85.7% and 100% read events, respectively.

I run the experiments on two prototype clusters, REP and ABM. In the REP cluster, client requests are distributed to the Apache web servers in a round-robin fashion. The database is replicated on each machine that Apache runs. Replication is a built-in functionality of MySQL. A separate MySQL server is configured as the master server, while the others are slave servers. Updates to the master database are automatically propagated to the slaves. Apache forwards the requests to a FastCGI process, i.e. the auction server, on the same machine. In processing the requests, the auction server sends read-only queries to the local database and updates to the master database. Recently accessed data is automatically cached in the Linux kernel buffers [77]. All the components in this configuration are off-the-shelf components. In the ABM cluster, the local MySQL servers serve as database clerks and cache servers, rather than slaves in a replicated database. The built-in replication of MySQL is turned off, and the master database server is only responsible for managing data on its local disk, but not responsible for propagating updates to other MySQL servers. The auction servers send all queries and updates to the local database database clerks. The database clerks, the cache servers and the database manager together manage the clustered in-memory database.

Because the data set in the experiments is small compared to the physical memory size (256 MB) per node, I artificially limit the amount of memory available (90 MB) to the cache servers in ABM. In REP, the amount of memory available to the buffer cache is only limited by the system load. The built-in replication of MySQL does not guarantee the atomicity or serialization of updates; to measure comparable performance, the consistency protocol for the replicated keys in ABM is turned off during the experiments, i.e. updates to the replicated keys are broadcast to all replicas in a naïve, synchronous manner. This has not caused any data corruption in the ABM cluster during experiments.

5.7.2 Single server latencies

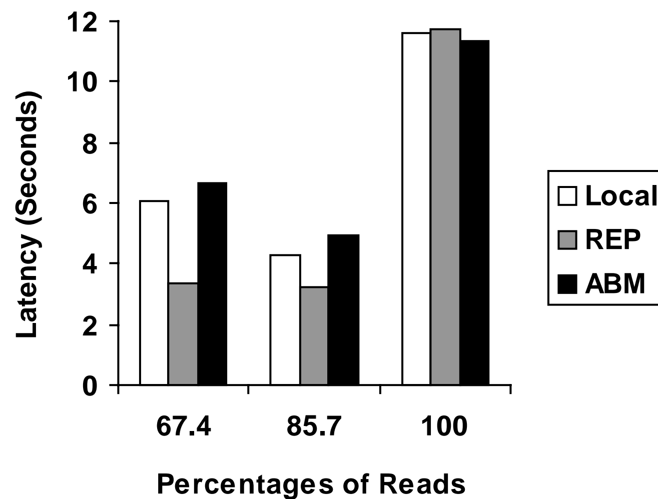

Figure 5.11 Average latency per request with a local, standalone MySQL server, a single slave server in replicated MySQL and a single cache server with ABM, respectively. The x-axis is the access pattern of traces, i.e. write-intensive, read-mostly, and read-only.

I measure the average request latencies of single servers in three different configurations: a local, standalone MySQL server, a single slave server in replicated MySQL, and a single cache server with ABM. The latter two correspond to the simple cases of the REP and ABM clusters, respectively. The measurement of single server latencies is intended to show the overhead of replicated MySQL and ABM added to local MySQL. In replicated MySQL, the expected overhead is the remote updates to the master server and the propagation of updates from the master server to the slave server. In ABM, the expected overhead is the round-trip message to the database manager, the remote updates to the master server, and the data transferring from the master server to the cache server in case of cache misses.

Figure 5.11 shows the latencies of single servers. ABM has slightly longer latencies than local MySQL in the traces with write events. However, ABM is slightly faster than local MySQL in the read-only traces, because the overhead for updates is not present in the read-only traces and the speedup due to the caching in ABM dominates the overhead of ABM. It was not expected that, in the traces with write events, replicated MySQL performs better than local MySQL and ABM. This is an artificial result of the weak

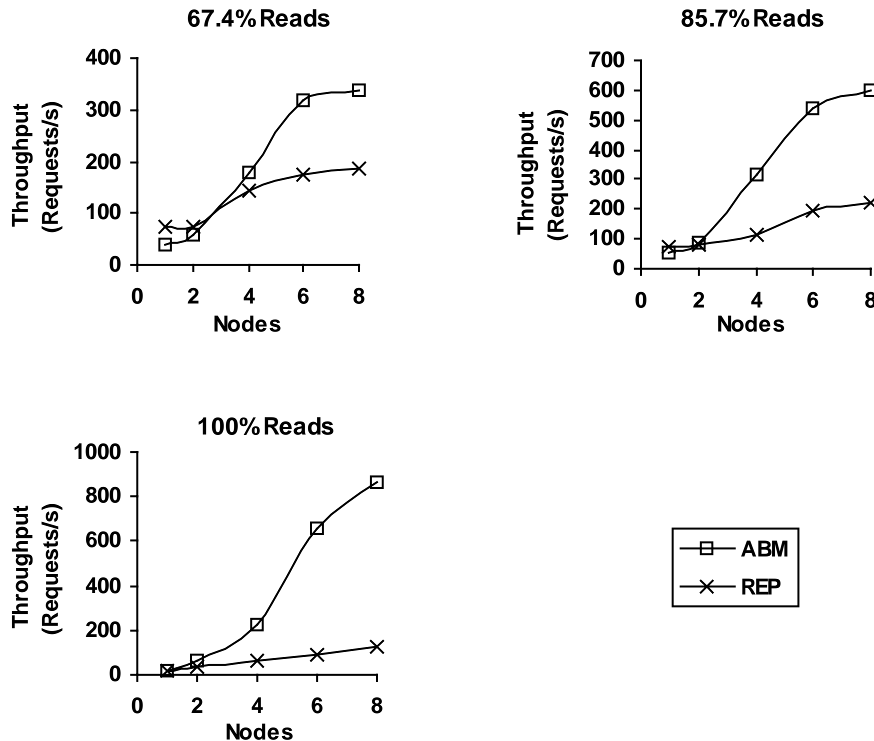

Figure 5.12 Throughputs of cluster servers.

consistency in MySQL replication. I observe from the experiments that the slaves in the replication are disconnected from the master when the system load is high and updates to the master database are not propagated to the slaves synchronously. Therefore, subsequent queries return smaller (and incorrect) sets of results and appear to be faster. In fact, many updates are still missing in the slaves after the experiments are stopped. (Note that this is not documented in MySQL manual [80].) In ABM, all updates are guaranteed to reach all replicas before the operations complete.

5.7.3 Measurement on cluster servers

Figure 5.12 shows the throughputs of the cluster servers in requests per second for three traces with different access patterns. ABM performs better than REP except with small numbers of servers, when the artifact of the weak consistency in MySQL replication dominates, as discussed in the previous section. With 8 nodes, ABM outperforms REP by a factor of 2 to 7. The throughput curves also suggest that, in order to achieve the same

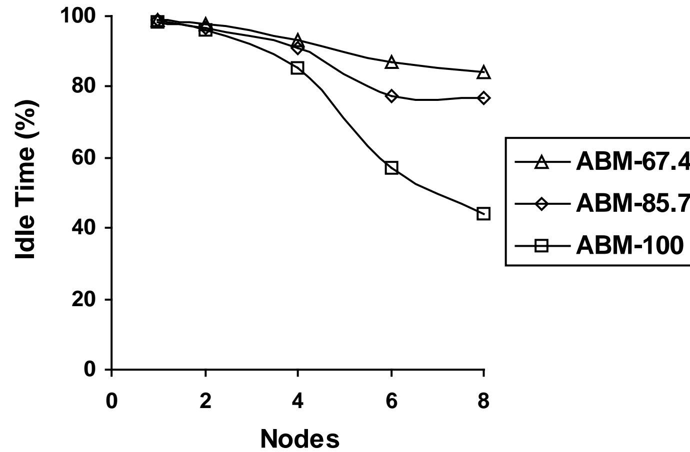

Figure 5.13 Percentages of the time when the CPU of the distributor in ABM is idle.

throughputs as REP with 8 nodes, ABM requires only 2, 3 and 3 nodes for the write-intensive, read-mostly and read-only patterns, respectively.

Figure 5.13 shows the percentages of the time when the CPU of the database manager in ABM is idle, which can be used to predict the potential that the manager be saturated or become a bottleneck in the cluster. With 8 server nodes and the fastest traces, i.e. the read-only traces, the manager has roughly 40% idle time. Assuming that addition of server nodes causes linear increase of workload in the manager, I expect that a single manager can potentially support more than 20 server nodes of the same hardware configuration. The assumption is conservative in terms of the scalability of the manager because, as the experimental results show, decrease in the manager idle times slows down as the number of server nodes increases.

Figure 5.14 shows the amounts of free memory space in the server machines during the experiments. In ABM, there is less free memory with 8 nodes than with 6 nodes. This is due to a simplification in the prototype implementation: each application has its own connections to all cache servers, rather than share the connections with other applications on the same machine. Therefore, when the number of server machines increases, the number of database connections increases rapidly and a larger amount of memory is consumed by the connections. The same effect was observed in related work [87] as well. The solution to this problem is to have the database clerk in ABM multiplex the

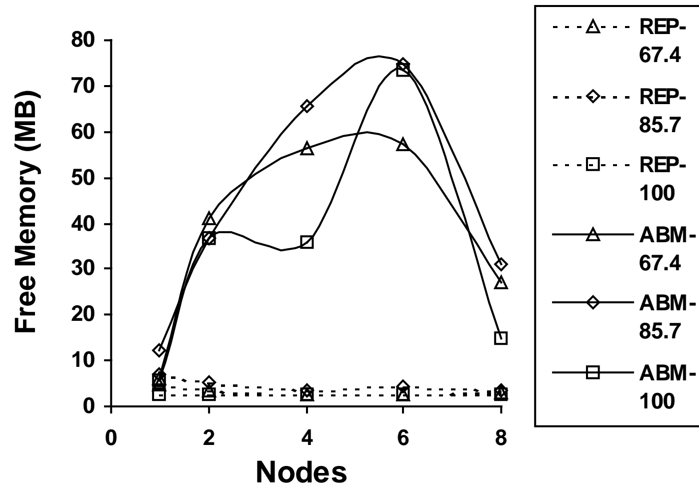

Figure 5.14 Average free memory space in the server machines during the experiments.

connections to the cache servers for the applications on the same machine. In REP, there is constantly high memory pressure in all experiments. It was expected because on-disk databases are known to be less efficient in memory management than in-memory databases [86] and REP does not exploit locality in the request distribution.

5.8 Related work

TimesTen Front-tier [85] is a commercial product that uses an in-memory database as the cache for a manually defined view in an on-disk database. ABM extends the existing work by automatic and dynamic management of clustered in-memory databases.

An alternative to caching raw data is to cache query results in web or proxy servers [66] [67]. This approach has the advantage in situations where generating the results is computationally intensive. However, its use is largely limited to read-only data and proprietary environments.

Materializing views inside or outside database systems has been studied in the web context [96]. Compared to application-level result caching, this approach is more generic and reusable; however, it does not eliminate the inefficiency in cache space usage due to possible overlaps in content across different views.

The replication of search keys in ABM is related to caching indices in clients [98] since both are intended to enable clients to locally perform as much selecting or scanning as possible. The essential difference is that index caching is part of the database systems while key replication is done externally without modifying the database systems. This leaves ABM an open and non-intrusive management system for a variety of database products.

Effective front-end, content-based request distribution strategies [3] [64][61] have been developed to improve the performance and scalability of clustered web servers with static content. However, exploiting locality for dynamic content is difficult in the front end because URLs do not carry enough information for identifying the data that needs to be accessed. In the HACC project [68], there is an extension to content-based distribution for dynamic requests in Lotus Domino, a web server product from IBM Lotus. It makes decision based on requested Notes objects and actions, but does not address distribution of dynamic content in general. ABM extends the existing work by postponing the decision on query distribution till the queries are executed and by exploiting natural affinity in a wide range of Internet applications.

Apers [62] presented and compared methods for obtaining optimal and heuristic solutions to the data allocation problem in distributed databases. The optimal methods basically search the large solution space for determining data allocations to minimize total transmission cost. The heuristic methods typically start with an initial data allocation and iteratively reallocate fragments to decrease the total transmission cost in a greedy fashion until the cost can no longer be decreased. Due to their complexity, those techniques have to be applied off line or statically. Therefore, they are not readily applicable to those Internet services with dynamic changes in access patterns and loads. Brunstrom et. al. [63] addressed how to dynamically reallocate data for databases with changing access patterns and how to incorporate individual site loads in the reallocation decision. Rather than complicated and expensive optimization algorithms, they use a simple heuristic that keeps track of accesses to each data block on each site and periodically moves data to the site where it is accessed most without causing load imbalance across sites. The choice of

the data movement interval is critical to the performance of this method: excessively large values will prevent the system from responding to workload changes in time while excessively small values will result in oscillation of data between sites and increase the total transmission cost. The right interval value is often application specific and varies access pattern changes.

The main contribution of ABM is the combination of its effectiveness comparable to that of the data allocation approaches, and its light-weighted, dynamic nature comparable to that of the request distribution approaches. The goal of ABM is achieved by a novel combination of two-stage execution, vertical replication and horizontal fragmentation.

5.9 Summary

In this chapter, I discuss how to improve the performance of Internet application servers in a cost-effective way by using clustered in-memory databases as the dynamic content cache. In particular, I design an affinity-based management (ABM) system for clustered in-memory databases that strives to maximize effective cache capacity and minimize synchronization cost.

With trace-based simulations, I compare the data/query distribution strategy in ABM to weighted round-robin strategy, basic query-based strategy, enhanced query-based strategy and an ideal case. The results show that ABM outperforms alternative strategies by a factor of 1.3 to 9 and achieves up to 90% of the ideal performance. The throughput comparison indicates that ABM only requires $\frac{1}{4}$ to $\frac{3}{4}$ of the resources in order to achieve the same throughput as the alternative strategies. The results also suggest that applications with dynamic content, especially write-shared content, can potentially benefit more from a good distribution strategy than applications with static content, due to the tendency for more data sharing and the synchronization cost that cannot be reduced by simply increasing memory size.

I have implemented a prototype cluster with the clustered in-memory database and ABM (called the ABM cluster) and compared its performance to that of a replicated on-disk

database with large buffer cache (called the REP cluster). Experimental results show that the ABM cluster outperforms the REP cluster by a factor of 2 to 7 with 8 nodes. The throughput comparison indicates that the ABM cluster only requires $\frac{1}{3}$ to $\frac{1}{2}$ of the resources in order to achieve the same throughput as the REP cluster. The gain of price-performance ratio comes both from efficient memory management and from increased effective cache capacity. The results also suggest that the database manager in ABM will not likely become a bottleneck for clusters with less than 20 processing nodes.

6 Conclusion

6.1 Results

The objective of this dissertation is to address the availability, scalability and cost-effectiveness issues in cluster-based Internet infrastructures. The approach taken in this dissertation is to investigate the data and query distribution strategies of the storage systems in the infrastructures.

I study a failure isolation approach that improves the availability and scalability of the storage systems for large-scale Internet services. This approach is complementary to redundancy-based methods. Based on this approach, an *island-based file system* is designed with the principle that as many operations as possible should involve exactly one island. I evaluate the island-based file system by statistical analysis of the content structures and access patterns of existing file systems. The results show that failure isolation is indeed achievable in the island-based file system. On average 99.8% file system operations involve a single island. The results also show that the availability model offered by the island-based file system is useful to Internet services. In one of the examples, if 1 out of 32 islands is down for an hour, it is expected that 93.8% clients during that hour will not notice the temporary partial failure. I have implemented a prototype island-based file system called *Archipelago* on a cluster of PCs running Windows NT. Measurements with trace-based operation mixes show a speedup of 15.7 on 16 islands. In an analytical model, the system is estimated to scale up to 702 islands with speedup efficiency higher than 50%.

I also study management strategies for improving the price-performance ratio of clustered in-memory databases as the dynamic content caches for Internet applications. I observe that, despite the conflicts across queries for dynamic content, many applications exhibit query affinity, which can be exploited for a good management strategy. I design an *affinity-driven management* (*ABM*) system for clustered in-memory databases that strives to maximize effective cache capacity and minimize synchronization cost. The goal of ABM is achieved by a novel combination of two-stage execution, vertical replication and

horizontal fragmentation. I evaluate the distribution strategy in ABM with simulations on two example applications: white pages and auctions. The simulation results show that, in a cluster of 16 nodes, ABM outperforms the basic query-based strategy by a factor of 2 to 3, outperforms the weighted round-robin strategy by a factor of 6 to 9, and achieves up to 90% of the ideal performance. The throughput comparison indicates that ABM only requires $\frac{1}{4}$ to $\frac{3}{4}$ of the resources in order to achieve the same throughput as the alternative strategies. I run experiments with two prototype clusters on Linux servers, one with a clustered in-memory database and ABM (called the ABM cluster), the other with a replicated database and large buffer cache (called the REP cluster). Measurements with trace-based benchmarks under various access patterns show that, in a cluster of 8 nodes, ABM outperforms the REP cluster by a factor of 2 to 7. The throughput comparison indicates that the ABM cluster only requires $\frac{1}{3}$ to $\frac{1}{2}$ of the resources in order to achieve the same throughput as the REP cluster. The experimental results also suggest that the centralized manager in ABM will not likely become a bottleneck for clusters with less than 20 nodes.

I have designed and implemented a consistency protocol for the replicated metadata in the island-based file system and the clustered in-memory database. The correctness of the protocol is checked in a randomized test engine. The impact of the consistency protocol on the performance and scalability of Archipelago is studied in micro benchmarks and trace-based operation mixes. The results show that the protocol does not have a noticeable impact on the common operations. This leads to my belief that it is possible to distribute data in a cluster under such a protocol that the system can both achieve high availability and strong consistency, and scale efficiently with the cluster size.

In summary, I learn from my dissertation work that a good distribution strategy for cluster-based Internet infrastructures should strive to partition and replicate data in such a way to address locality, load balance, consistency and autonomy with balanced efforts. To design and implement such a system, it pays to gain insights into the content structures and access patterns of representative applications, then generalize and exploit

those insights for a wider range of applications.

6.2 Future work

In the near future, I would like to extend my work on the data distribution strategies to global or geographical distribution of Internet services with dynamic or multimedia content. Scaling a local, cluster-based server and delivering content with globally replicated servers are complementary approaches to improving the overall availability and scalability of Internet services. I am interested in studying the interaction of the two approaches, especially for dynamic data that is updated under various consistency requirements and for confidential data that cannot be replicated to un-trusted sites. I expect that global distribution of services with dynamic content involves partitioning and replication of functionality as well as data.

I am also interested in exploring a related research area, system management of large server farms, e.g. dynamically installing, configuring, booting and launching operating systems and software. I learn from my personal experience that it is unexpectedly painful to run even a small cluster, and that the amount of system management work grows as a function of the number of tasks for the cluster as well as of the number of nodes in the cluster. Despite the availability of various management software with GUIs in the market, I believe that some fundamental research needs to be done in operating systems and/or network protocols in order to make a large cluster manageable.

Reference

[1] C. A. Thekkath, T. Mann, and E. K. Lee, “Frangipani: A Scalable Distributed File System”, in Proceedings of the 16th ACM Symposium on Operating Systems Principles, Octobor 1997.

[2] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web", in Proceedings of the 29th ACM Symposium on Theory of Computing, May 1997.

[3] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum, "Locality-Aware Request Distribution in Cluster-Based Network Servers", in Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1998.

[4] P. F. Corbett, and D. G. Feitelson, "The Vesta Parallel File System", in ACM Transactions on Computer Systems, Vol. 14, No. 3, August 1996.

[5] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang, "Serverless Network File Systems", in Proceedings of the 15th ACM Symposium on Operating Systems and Principles, December 1995.

[6] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath, "Implementing Global Memory Management in A Workstation Cluster", in Proceedings of the 15th ACM Symposium on Operating Systems and Principles, December 1995.

[7] E. K. Lee, and C. A. Thekkath, "Petal: Distributed Virtual Disks", in Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1996.

[8] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and Performance in A Distributed File System",

in ACM Transactions on Computer Systems, Vol. 6, No. 1, February 1988.

[9] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network File System", in Proceedings of USENIX Summer Technical Conference, Summer 1985.

[10] J. L. Carter, and M. N. Wegman, "Universal Classes of Hash Functions", in Journal of Computer and System Sciences 18, 1979.

[11] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible Hashing - A Fast Access Method for Dynamic Files", in ACM Transactions on Database Systems, Vol. 4 No. 3, 1979.

[12] B. E. Keith, and M. Wittle, "LADDIS: the Next Generation in NFS File Server Benchmarking", in Proceedings of USENIX Summer Technical Conference, June 1993.

[13] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol", in Proceedings of ACM SIGCOMM, February 1998.

[14] R. Hagmann, "Reimplementing the Cedar File System Using Logging and Group Commit", in Proceedings of the 11th ACM Symposium on Operating System Principles, November 1987.

[15] "The Common Internet File System (CIFS) Specification Reference", Microsoft, 1996.

[16] "Windows NT IFS Kit", Early Release, Microsoft, March 1997.

[17] B. Gronvall, A. Westerlund, and S. Pink, "The Design of a Multicast-based Distributed File System", in Proceedings of the 3rd Symposium on Operating Systems Design and Implementation, February 1999.

[18] H. Custer, “Inside the Windows NT File System”, Microsoft Press, 1994.

[19] R. Vingralek, Y. Breitbart, and G. Weikum, "Distributed File Organization with Scalable Cost/Performance", in Proceedings of ACM SIGMOD, May 1994.

[20] N. Nienwejaar, and D. Kotz, "The Galley Parallel File System", in Proceedings of the International Conference on Supercomputing, July 1996.

[21] M. Devarakonda, A. Mohindra, J. Simoneaux, and W. H. Tetzlaff, “Evaluation of Design Alternatives for a Cluster File System”, in Proceedings of USENIX Winter Technical Conference, Winter 1995.

[22] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg and J. Zelenka, “File Server Scaling with Network-Attached Secure Disks”, in Proceedings of ACM SIGMETRICS, June 1997.

[23] L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System”, in Communications of the ACM, July 1978.

[24] J. Gray, “Notes on Database Operating Systems”, in Operating Systems: An Advanced Course, 1978.

[25] M. N. Nelson, B. B. Welch, and J. K. Ousterhout, "Caching in the Sprite Network File System", in ACM Transactions on Computer Systems, February 1988.

[26] Watson, and P. Benn, "Multiprotocol Data Access: NFS, CIFS, and HTTP", Technical Report 3014, Network Appliance, May 1999.

[27] H. C. Rao, and L. L. Peterson, "Accessing Files in an Internet: the Jade File System", in IEEE Transactions on Software Engineering, IEEE, Vol. 19 No. 6, June 1993.

[28] Virtual Interface Architecture, http://www.viarch.org

[29] T. Mann, A. Birrell, A. Hisgen, C. Jerian, and G. Swart, “A Coherent Distributed File Cache with Directory Write-Behind”, in ACM Transactions on Computer

Systems, Vol. 12, No. 2, May 1994.

[30] N. Lynch, and M. Tuttle, “An Introduction to Input/Output Automata”, CWI-Quarterly, 2(3), September 1989.

[31] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol Verification as a Hardware Design Aid", in Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors, 1992.

[32] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline, “Detection of Mutual Inconsistency in Distributed Systems”, in IEEE Transactions on Software Engineering 9(3), May 1983.

[33] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, “Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System”, in Proceedings of the 15th ACM Symposium on Operating Systems and Principles, December 1995.

[34] M. Baker, and J. K. Outsterhout, “Availability in the Sprite Distributed File System”, in ACM Operating Systems Review, 25, 2, April 1991.

[35] M. Devarakonda, B. Kish, and A. Mohindra, “Recovery in the Calypso File System”, in ACM Transactions on Computer Systems, Vol. 14, No. 3, August 1996.

[36] R. Golding, J. Wilkes, and A. Veitch, private communications, August 1999.

[37] E. M. Clarke, O. Grumberg, and D. E. Long, “Model checking and abstraction”, in Proceedings ACM Symposium on Principles of Programming Languages, January 1992.

[38] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta, "Hive: Fault Containment for Shared-Memory Multiprocessors", in Proceedings of the 15th ACM Symposium on Operating Systems and Principles, December 1995.

[39] D. Roselli, and T. E. Anderson, "Characteristics of File System Workloads", Technical Report UCB//CSD-98-1029, 1998, and personal communications, April 1999.

[40] K. W. Shirriff, and J. K. Ousterhout, "A Trace-Driven Analysis of Name and Attribute Caching in A Distributed System", in Proceedings of USENIX Technical Conference, 1992.

[41] "Platform SDK: Windows Base Services: Files and I/O", in MSDN Library Visual Studio 6.0, Microsoft, 1998.

[42] Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams, "Replication in the Harp File System", in Proceedings of the 13th Symposium on Operating Systems Principles, October 1991.

[43] J. R. Douceur and W. J. Bolosky, "A Large-scale Study of File-system Contents", in Proceedings of ACM SIGMETRICS, May 1999.

[44] P. Chen, E. Lee, G. Gibson, R. Katz, and D. Patterson, "RAID: High-Performance, Reliable Secondary Storage", ACM Computing Surveys, June 1994.

[45] D. A. Patterson, G. A. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", in Proceedings of ACM SIGMOD, June 1988.

[46] T. F. Sienknecht, R. J. Friedrich, J. J. Martinka, P. M. Friedenbach, "The Implications of Distributed Data in a Commercial Environment on the Design of Hierarchical Storage Management", Performance Evaluation, 1994.

[47] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier, "Cluster-Based Scalable Network Services", in Proceedings of the 16th ACM Symposium on Operating Systems Principles, Octobor 1997.

[48] A. Fox, and E. A. Brewer, "Harvest, Yield, and Scalable Tolerant Systems", in Proceedings of HotOS-VII, March 1999.

[49] P. Chundi, D. J. Rosenkratz, and S. S. Ravi, "Deferred Updates and Data Placement in Distributed Databases", in Proceedings of 12th International Conference on Data Engineering, 1996.

[50] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz, "Update Propagation Protocols for Replicated Databases", in Proceedings of ACM SIGMOD, 1999.

[51] G. J. Popek, R. G. Guy, T. W. Page Jr., J. S. Heidemann: "Replication in Ficus Distributed File Systems", in Proceedings of Workshop on the Management of Replicated Data, November 1990.

[52] Hisgen, A. Birrell, C. Jerian, T. Mann, M. Schroeder, and G. Swart, "Granularity and Semantic Level of Replication in the Echo Distributed File System", in Proceedings of Workshop on Management of Replicated Data, November 1990.

[53] Walker, G. Popek, R. English, C. Kline, and G. Thiel, "The Locus Distributed Operating System", in Proceedings of 9th ACM Symposium on Operating Systems Principles, October 1983.

[54] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, D. C. Steere, "Coda: A Highly Available File System for a Distributed Workstation Environment", in IEEE Transactions on Computers 39(4), April 1990.

[55] P. Triantafillou and C. Neilson, "Achieving Strong Consistency in a Distributed File System", in IEEE Transactions on Software Engineering, Vol. 23, No. 1, January 1997.

[56] DBC/1012 database computer system manual release 2.0. Technical Report Document No. C10-0001-02, Teradata Corporation, Nov 1985.

[57] Apache Web Server, http://www.apache.org

[58] MySQL Database Server, http://www.mysql.com

[59] http://www.akamai.com

[60] http://www.sandpiper.com

[61] K. P. Eswaran, "Placement of Records in a File and File Allocation in a Computer Network", in Proceedings of IFIP Congress on Information Processing, 1974.

[62] P. M. G. Apers, "Data Allocation in Distributed Database Systems", in ACM Transactions on Database Systems, Vol. 13, No. 3, September 1988.

[63] A. Brunstrom, S. T. Leutenegger, and R. Simha, "Experimental Evaluation of Dynamic Data Allocation Strategies in a Distributed Database with Changing Workloads", in Proceedings of International Conference on Information and Knowledge Management, November, 1995.

[64] C. Yang, and M. Luo, "Efficient Support for Content-Based Routing in Web Server Clusters", in Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems, October 1999.

[65] M. Aron, D. Sanders, P. Druschel and W. Zwaenepoel, "Scalable Content-Aware Request Distribution in Cluster-based Network Servers", in Proceedings of USENIX Annual Technical Conference, June 2000.

[66] A. Iyengar, and J. Challenger, "Improving Web Server Performance by Caching Dynamic Data", in Proceedings of the 1st USENIX Symposium on Internet Technologies and Systems, December 1997.

[67] B. Smith, A. Acharya, T. Yang, and H. Zhu, "Exploiting Result Equivalence in Caching Dynamic Web Content", in Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems, October 1999.

[68] X. Zhang, M. Barientos, J. B. Chen, and M. Seltzer, "HACC: An Architecture for Cluster-Based Web Servers", in Proceedings of the 3rd USENIX Windows NT Symposium, July 1999.

[69] "FastCGI: A High-Performance Web Server Interface", Technical White Paper, Open Market, April 1996.

[70] "The Server-Application Function and Netscape Server API", Netscape.

[71] M. J. Franklin, M. J. Carey, and M. Livny, "Transactional Client-Server Cache Consistency: Alternatives and Performance", in ACM Transactions on Database Systems, Vol. 22, No. 3, September 1997.

[72] M. J. Franklin, M. J. Carey, and M. Livny, "Global Memory Management in Client-Server DBMS Architectures", in Proceedings of the 18th VLDB Conference, August 1992.

[73] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson, "Cooperative Caching: Using Remote Client Memory to Improve File System Performance", in Proceedings of the USENIX Conference on Operating Systems Design and Implementation, November 1994.

[74] J. R. Davis, "DataLinks: Managing External Data with DB2 Universal Database", IBM Data Management White Paper, http://www-4.ibm.com/software/data/pubs/papers/datalink.html, February 1999.

[75] Cisco Systems, LocalDirector, http://www.cisco.com.

[76] Compaq, NonStop™ Himalaya Server, http://www.tandem.com.

[77] L. Wirzenius and J. Oja, "The Linux System Administrators' Guide", Version 0.6.2, Chapter 5, http://www.linuxdoc.org/LDP/sag/index.html, 1993.

[78] Princeton University Campus Directory, http://www.princeton.edu/Siteware/puphf.shtml

[79] http://www.ebay.com

[80] MySQL Manual, http://www.mysql.com/documentation/mysql/commented

[81] http://www.yahoo.com

[82] http://www.google.com

[83] http://www.amazon.com

[84] Oracle Database, http://www.oracle.com

[85] "Dynamic Data Cache", TimesTen, http://www.timesten.com/products/fronttier/ftconcepts.html.

[86] H. Garcia-Molina and K. Salem, "Main Memory Database Systems", in IEEE Transactions on Knowledge and Data Engineering, December 1992.

[87] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler, "Scalable, Distributed Data Structures for Internet Service Construction", in Proceedings of 4th Symposium on Operating Systems Design and Implementation, October 2000.

[88] A. Iyengar, E. MacNair, and T. Nguyen, "An Analysis of Web Server Performance", in Proceedings of GLOBECOM, November 1997.

[89] A. Moffat and J. Zobel, "Self-Indexing Inverted Files for Fast Text Retrieval", in ACM Transaction on Information Systems, 1996.

[90] IBM WebSphere, http://www-4.ibm.com/software/webservers.

[91] Exodus Communications, http://www.exodus.com.

[92] Web server of department of computer science, Princeton University, http://www.cs.princeton.edu.

[93] M. Ji, E. W. Felten, R. Wang, and J. P. Singh, "Archipelago: An Island-Based File System for Highly Available and Scalable Internet Services", in Proceedings of 4th USENIX Windows Systems Symposium, August 2000. Best student paper award.

[94] M. Ji, "Atomicity, Serialization and Recovery in the Island-Based File System", in Proceedings of 1st IEEE International Conference on Cluster Computing, November 2000. Extended abstract/poster.

[95] O. Shmueli and A. Itai, "Maintenance of Views", in ACM SIGMOD Record, 14(2), 1984.

[96] A. Labrinidis and N. Roussopoulos, "WebView Materialization", in Proceedings of ACM SIGMOD, May 2000.

[97] W. Effelsberg and T. Haerder, "Principles of Database Buffer Management," in ACM Transactions on Database Systems, Vol.9, No.4, December 1984.

[98] M. Zaharioudakis and M. J. Carey, "Highly Concurrent Cache Consistency for Indices in Client-Server Database Systems", in ACM SIGMOD, 1997.