INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

NOTE TO USERS

This reproduction is the best copy available.

UMI

Functionally Homogeneous Clustering: a Framework for Building Scalable Data-intensive Internet Services

Yasushi Saito

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

2001

Program Authorized to Offer Degree: Computer Science and Engineering

UMI Number: 3014023

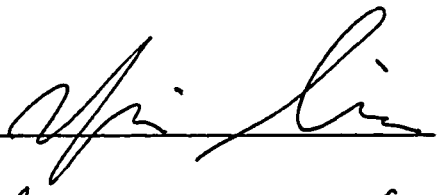
UMI Microform 3014023

Copyright 2001 by Bell & Howell Information and Learning Company.
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

Bell & Howell Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

© Copyright 2001
Yasushi Saito

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Bell and Howell Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature 

Date 6-5-2001

University of Washington

Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Yasushi Saito

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Co-Chairs of Supervisory Committee:

Brian N. Bershad

Henry M. Levy

Reading Committee:

Brian N. Bershad

Henry M. Levy

Steven D. Gribble

Date: 6-5-2001

University of Washington

Abstract

Functionally Homogeneous Clustering: a Framework for Building Scalable Data-intensive Internet Services

by Yasushi Saito

Co-Chairs of Supervisory Committee:

Professor Brian N. Bershad
Computer Science and Engineering

Professor Henry M. Levy
Computer Science and Engineering

This dissertation proposes *functionally homogeneous clustering* (*FHC*), a new software architecture for building data-intensive Internet services that are manageable, available, fast, and inexpensive. FHC lets any node in the cluster manage any function and any piece of data, freeing humans from making specific decisions about the workload distribution. Its dynamic and self-regulative nature is the key to its scalability.

This dissertation also presents three mechanisms that synergistically realize this architecture: automatic reconfiguration, high-throughput replication, and fine-grain load balancing. FHC offers an efficient and scalable automatic reconfiguration mechanism for redistributing functions and data after configuration change. It ensures that users can access all the data on live nodes after any number of failures. The replication mechanism stores important data on multiple disks with small overhead, while ensuring the consistency of their contents. The load balancing mechanism distributes incoming data evenly among nodes and masks the non-uniformity in the workloads and the cluster configuration.

FHC scales without sacrificing its service quality by taking advantage of the semantics of data-intensive Internet services. For example, the name database used to locate on-disk data is stored

only in memory and is recomputed after failure by scanning disks. While such a design makes the contents and operations of the name database application-specific, it makes the system fast and robust. Our replication algorithm also takes advantage of application semantics and ensures only eventual data consistency. In return, this strategy makes the system extremely resilient against failures.

We develop the Porcupine email server as proof of the concept of functionally homogeneous clustering. Porcupine distributes user management and email message storage dynamically to maximize system throughput and ensure continuous service to all users. It replicates the user profile and email messages to ensure their availability. We evaluate the manageability, availability, and performance of Porcupine on a 30-node PC cluster. Through the evaluation, we show that Porcupine's performance indeed scales well and that it reacts to configuration changes gracefully and quickly. We also show that Porcupine's load balancing service efficiently utilizes heterogeneous hardware resources and handles non-uniform workloads by automatically discovering idle resources in the cluster.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I am very fortunate to have worked with two great mentors, Brian Bershad and Hank Levy. Brian guided me, especially through my early graduate career, when I knew little about research. He taught me everything that I needed to start doing research, including how to select a project, search the literature, and use performance evaluation techniques. He emphasized that systems research is all about making users happy, a simple concept I had yet to realize. Hank has an amazing ability to understand my (usually confusing) explanation of issues, identify relevant and challenging problems, and present them in a way that I can comprehend. he taught me the importance of always retaining the project's "big picture" even when working on a tiny part of an algorithm. Both professors also helped me immensely in peripheral but critical areas, such as project scheduling, improving English skills, and making connections with people outside the school. Without Brian and Hank, this dissertation would not have happened.

I really enjoyed working with my colleagues in the SPIN and Porcupine project groups. I am especially thankful to Marc Fuiczynski, Gün Sirer, Mike Swift, Robert Grimm, Stefan Savage, and Przemek Pardyak for their sharp, knowledgeable, and honest discussions with me. I learned from these people not just a great deal of opinions and factoids, but also how logically I must think and present to persuade them.

I found the Department of Computer Science and Engineering at the University of Washington to be the best teaching and research facility I have ever encountered. The professors always spent time with me generously whenever I needed assistance. The computing support staff, especially David Becker, has been quick, efficient, and tireless. Having been a part-time system administrator myself, I know the level of skill and dedication it takes to maintain the machines as well as they are maintained here. Administrative staff, especially Melody Kadenko and Frankye Jones, handled campus bureaucracy so efficiently that I never perceived its existence.

Many companies and agencies supported me through donations and grants. Intel, Dell, and Digital (Compaq) donated numerous computers, many of which became the core of the Porcupine cluster. DARPA and NSF funded the Porcupine project, which became the basis of my dissertation.

I owe a debt to my mentors while in Japan, especially Ken Sakamura, Hiroaki Takada, and Satoshi Matsuoka. They introduced me to the joy of operating systems study and taught me that you do not do research in an ivory tower. Without their motivation and encouragement, I would not have come to the United States in the first place.

Finally, I thank my parents. They have quietly supported me throughout the many years that I have been away. I now understand how tough that has been.

Chapter 1 Introduction

This dissertation proposes *functionally homogeneous clustering* (*FHC*), a new software architecture for building scalable yet inexpensive data-intensive Internet services. FHC lets any node manage any function and any piece of data, freeing humans from making specific decisions about the distribution of workloads. Its dynamic and self-regulative nature is the key to its manageability, availability, and performance. This dissertation also presents three mechanisms that synergistically realize this architecture: automatic reconfiguration, high-throughput optimistic replication, and fine-grain load balancing. The automatic reconfiguration mechanism redistributes functions and data transparently among nodes after configuration change to permit continuous system operation. The replication mechanism stores data on multiple nodes for availability, while ensuring the consistency of replicated contents in the face of failures. The load balancing mechanism distributes incoming workloads evenly among nodes and masks non-uniformity in the incoming workloads or the cluster configuration. To prove the concept of FHC, this dissertation designs and implements *Porcupine*, a functionally homogeneous, cluster-based email server. Through measurement and analysis, the dissertation shows that Porcupine can scale to handle billions of email messages per day on a cluster built of several hundred off-the-shelf PCs.

Data-intensive Internet services, exemplified by email, electronic bulletin board systems (BBS), and wide-area collaboration services (e.g., Yahoo photo book and equill.com), are a class of services that experience frequent updates to their contents. They are not new. In fact, many of them predate the Internet. Until recently, however, their scale remained small, and a single computer sufficed to serve the entire user population. The explosive growth of the Internet over the last decade changed the picture. The commercialization of the Internet, especially the shift of the user population from companies and colleges to Internet service providers, caused the heavy concentration of services [129]. Portal sites, such as Yahoo, Excite, and Lycos, are becoming increasing popular, resulting in their continued exponential growth [69]. These sites are no longer mere web directories. To

accommodate user demand, they have become dynamic and personalized, featuring such services as email, BBS, auctions, and calendars. This rapid growth in the demands for such data-intensive services is what motivates our study.

Internet services in today's competitive market must minimize their operating costs while maintaining a satisfactory quality of service to users. We want large-scale Internet services to resemble power plants — not only do they need to scale, but they should also be highly automated and manageable by people without much knowledge of the underlying circuitry [59]. The chores today's administrators face, such as monitoring system health and moving data around to balance system load, should be delegated to the software itself; people should be responsible only for occasionally replacing broken parts and adding nodes or disks as system demand grows. Such a vision, however, is still far from reality. Most of the efforts to build scalable and manageable Internet services have concentrated on static, read-only services, such as traditional WWW or search engines [33, 61, 119, 161]. The techniques developed for scaling such static services, e.g., caching and stateless data transformation, are not applicable to data-intensive services. As a result, today's data-intensive Internet services are still built by "brute-force," i.e., grossly over-investing in hardware and employing a large number of administrators to tune and troubleshoot the system.

Functionally homogeneous clustering alleviates this situation. Migrating function and data automatically in response to changes in the workload and system configuration not only improves performance, but it also frees humans from continuous system monitoring.

In the remainder of this chapter, we first discuss the requirements for scalable Internet services and define three dimensions of scalability. Section 1.2 defines data-intensive Internet services by contrasting their workload with other type of services, especially web and database systems. Section 1.3 reviews the structure of existing data-intensive services and studies why they fall short of achieving our scaling goals. Finally, we outline the contributions of this dissertation and lay the roadmap for the remainder of the thesis in Sections 1.4 and 1.5.

1.1 Scaling Internet Services

Data-intensive Internet services have been popular for a long time. Email was first deployed in 1971 over the ARPAnet [5, 25]. Usenet (the Internet BBS) was first deployed over the Matrix in 1979

[5, 137]. Commercial BBS companies, such as CompuServe, Prodigy, and AOL, have served users since the early '80s. Until recently, however, the scale of these services was relatively small, and a single fast computer could handle the workload.

Such a simple approach is no longer possible. The use of data-intensive services has grown an order of magnitude larger than just a few years ago. Figure 1.1 shows the recent and projected growth of the U.S. Internet population and its use of email [101]. The figure shows at least a linear growth of email usage over the next few years. The largest mail server in the year 2002 will handle about 250 million messages per day for 50 million users[1]. Moreover, many new data-intensive services have emerged as people's lives have begun to depend heavily on the Internet. Web-based BBS services (delphi.com, egroups.com, slashdot.org, etc.), personal information services (calendar.yahoo.com, address books, fantasy sports tracker, etc.), and wide-area collaboration services (equill.com, crit.org, etc.) are growing rapidly.

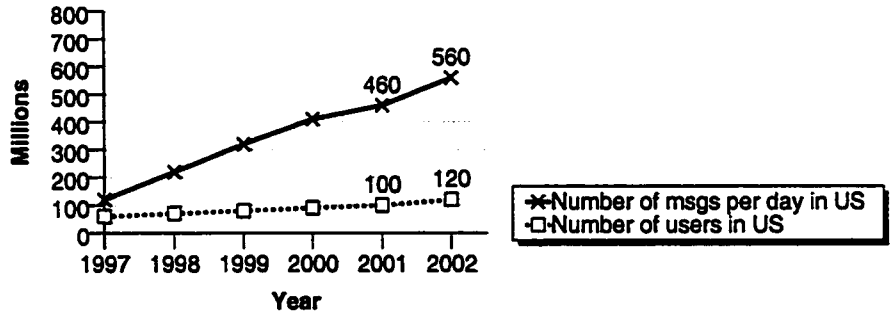


Figure 1.1. *Projected growth of U.S. Internet user population and its email usage.*

Internet services must *scale* to accommodate growing user demands. The term "scale" is traditionally associated with speed — a system scales when its speed improves linearly with its size. This narrow definition, however, misses the real issue Internet service providers face: providing the maximum value to users with the minimum cost. With system administration being a major expenditure in modern Internet services, just making the system faster yields only a marginal benefit

[1]This number is derived by assuming that the largest ISP in the world, i.e., AOL, maintains a 45% share of the U.S. Internet market in the year 2002 [77].

to users and service providers. This dissertation thus defines scalability in terms of three essential system aspects: manageability, availability, and performance.

Manageability: Although a system may be physically large, it should be easy to manage. A manageable system must react automatically to two types of changes.

The system must react to changes in the volume or the distribution of the workload. Events such as the Superbowl or the presidential election may drastically increase the activity of certain newsgroups in a BBS service for a short period. The system must *self-configure* in response to such events by automatically dispersing "hot spots" to nodes with spare capacity and freeing administrators from continuous performance twiddling.

The system must also react to changes in its configuration, such as node and disk failures, recoveries, retirements, or additions. The system must *self-heal* in response to such changes by moving the functions and data managed by failed or decommissioned nodes to live nodes, especially to newly commissioned nodes.

In a self-managing system, the administrator can simply add more nodes or disks to improve throughput and replace them when they break, with the same urgency with which one replaces light bulbs. Over time, the system's nodes will perform at differing capacities, but these differences should be masked and managed by the system.

Availability: In a large system, some of the hardware components will always be unavailable. For example, with per-node mean-time-between-failures (MTBF) of 100 days and mean-time-to-repair (MTTR) of one day — an optimistic assumption for PCs — a cluster of 500 nodes will have five nodes unavailable on average at any moment [19]. Moreover, failures exhibit in a variety of forms: node crashes, disk crashes, thrashing, network partitioning, or false alarms.

The system should always deliver good service to all of its users at all times, regardless of the number or type of failures, with the level of performance proportional to total system capacity.

Performance: The system's throughput should grow linearly with its size. The system should also be able to grow incrementally by adding components to the existing system, rather than having to scrap a running system and replacing it with new hardware.

1.2 Data-intensive Internet Services

Large, scalable Internet servers are in great demand. The demand has resulted in the development of many research systems and commercial products, particularly in the context of traditional, read-only web services [61, 119, 8, 161, 36, 58] and database systems that maintain highly consistent data [12, 117, 71]. This section defines *data-intensive Internet services* by their workload characteristics and service requirements. We contrast them with traditional services and argue that many of the techniques developed for traditional services do not apply to data-intensive ones.

Frequent updates: Data-intensive services update their contents far more frequently than traditional "static" services, such as WWW or FTP. For example, an email server often writes on disk almost as often as it reads, because many email messages are read just once before being deleted. A typical Usenet server often writes on disk more often than it reads, because Usenet's NNTP protocol pushes articles proactively, whether or not the articles are read by clients [137, 81].

Workloads with frequent updates impose significant pressure on the disk and comparatively less pressure on the CPU and network, in contrast to static systems, which exhibit the opposite workload characteristics [61]. Techniques such as proxying [33, 161] and stateless transformation [61] that try to divert the CPU and network load from storage nodes are not as effective as in traditional WWW services.

Low access locality: Data-intensive services access data with low locality. Email messages, newsgroup articles, event schedules, and auction items are not as heavily shared as Yahoo's web directories. For example, on a typical Usenet server, even the most popular articles are read just a few times before being deleted, due to a small overlap in newsgroup subscriptions among users [137].

In contrast, traditional web services often experience a highly skewed Zipf access distribution, in which a small number of objects are accessed exponentially more often than others [161, 20]. These properties allow static services to scale by caching their contents on intermediate proxy nodes and off-loading the back-end storage subsystem [61, 33]. Caching, however, is

less effective in data-intensive services with low access locality.

Large workload concurrency: Internet services face highly concurrent and independent workloads. For example, Yahoo serves around 90,000 concurrent sessions at any given instant [70]. This is in contrast to database systems, which often experience severe access contention to popular data items [12].

Weak data consistency: Many Internet services demand only weak data consistency. That is, they can sometimes serve stale contents. Some of the inconsistencies are anticipated in the network protocols themselves. For example, SMTP (for email) [124, 84] and NNTP (for Usenet) [81] allow data delivery to be delayed arbitrarily, and HTTP explicitly allows stale web pages to be cached on the client side [55]. Data inconsistency also happens at the application level. Duplicate email messages are sent when a user pushes the "Submit" button twice, and articles often fail to arrive at a Usenet server because of Usenet's ad-hoc transport mechanism and many administrative boundaries [137]. Fundamentally, weak consistency is often a necessity in wide-area distributed systems such as the Internet. Such systems must incorporate a "slack" in the quality of data, because remote sites are not always functional and reachable [60, 29, 165]. No slack leads to an unavailable service, a state usually worse than a degraded service. For these services, guaranteeing strong data consistency within a server does not help to improve the end-to-end quality of service.

A weak consistency requirement and a large workload concurrency stand in contrast to the type of workload anticipated by the traditional solution to data-intensive services, i.e., relational database systems with ACID (atomic, consistent, isolated, and durable) semantics [13, 12, 68]. Database systems could be used to build services with weak consistency requirements, but their complexity, including locking and transactions [68, 12], make them harder to scale [21, 60]. Such workloads are better served by loosely connected clusters, because they can be distributed among many nodes with little coordination.

Notice that we do not claim that all Internet-based services are allowed to serve stale data, because some of them, such as online banking and stock trading, are not. Rather, this dissertation restricts its focus to services with weak consistency requirements.

1.3 Existing Solutions and Their Problems

This section reviews today's popular solutions for building data-intensive Internet services. We show that although they offer some practical benefits, e.g., quick deployment, none of them satisfies our goal of a high level of manageability, availability, and performance.

1.3.1 Monolithic Systems

The most straightforward way to build a data-intensive service is to use a large computer with reliable data storage, such as RAID [34] (Figure 1.2). This solution can run all the software designed for single-node use, and the hardware automatically maximizes system performance. This approach is still popular, especially for legacy mainframe servers and today's small-scale servers, because of its simple software architecture and ease of management.

The monolithic solution, however, is not common among large Internet services for two main reasons: its high cost and the scaling limit. First, a large server computer is far more expensive than a set of small machines with the same aggregate performance, because the former must use reliable (and expensive) hardware to avoid a component failure that would crash the entire node. That the market for mainframes is small and less competitive pushes the cost higher. Second, monolithic servers offer limited upgrade options — for example, most symmetric multi-processor machines can support only 8 to 64 CPUs — and the entire node needs to be scrapped and replaced to scale the system beyond the limit. This limitation is fatal for services that face rapidly changing workload demands.

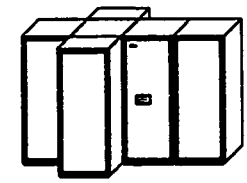


Figure 1.2. *A monolithic server.*

1.3.2 Statically Partitioned Clusters

An alternative to the monolithic architecture is *clustering*, which coordinates multiple computers connected by a low-latency, high-throughput network into a virtual server [122]. As discussed in the previous section, highly parallel workloads and weak consistency requirements make clustering attractive for Internet services.

Clustering can take several forms. The most popular among today's large Internet servers is

statically partitioned clusters, in which the administrator statically partitions the system's function and data among nodes (Figure 1.3). Here, each back-end storage node (e.g., running NFS [139, 143]) is designated to manage data belonging to a specific set of users or newsgroups. Several nodes are also dedicated to the management of user profile. A number of stateless front-end nodes, running stock software (e.g., Sendmail [142] for email, or INN [38] for Usenet), handle requests from users and access back-end nodes using a priori knowledge of user-to-node mappings.

This architecture has been successful in services that deliver mostly read-only data, such as web servers or search engines. In these services, the front-end nodes can take a significant load off the back-end nodes and help the system scale well by utilizing the buffer cache. The system remains relatively manageable, because administrative resources can be concentrated on a small number of back-end nodes. The front-end PCs require little attention despite their low availability, because they can be added or replaced trivially due to their stateless nature.

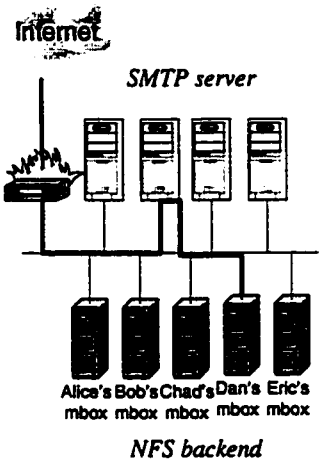


Figure 1.3. *An example of statically partitioned email servers.*

Statically partitioned clusters, however, scale poorly when the service is data intensive. In particular, as the user base grows, so does service demand, which can be met only by adding more machines. Unfortunately, each new machine must be configured to handle a subset of the data, which must be migrated from older machines. As more machines are added, the likelihood that at least one of them is inoperable grows, diminishing availability for users with data on the inoperable machines. In addition, users whose accounts are on slower machines tend to receive worse service than those on faster machines. Finally, a statically partitioned system is susceptible to overload when traffic is distributed non-uniformly across the user base.

To date, this solution has worked for two reasons. First, service organizations have been willing to substantially overcommit computing capacity to mitigate short-term load imbalances. Second, organizations have been willing to employ people to reconfigure the system manually in order to balance load over the long term. For small static systems, the costs have not been substantial. However, once the number of machines becomes large (i.e., on the order of a few dozen), disparate

(i.e., fast/slow machines, fast/slow disks, large/small disks), and continually increasing, this gross overcapacity becomes unacceptably expensive in terms of hardware and people.

1.3.3 Clustering using Network-attached Storage Devices

Another popular approach for building large data-intensive services is to attach storage devices to a system area network (SAN) and let nodes share these devices directly (Figure 1.4) [152, 52, 114, 122, 76]. This architecture is often used in conjunction with a cluster-aware operating system that arbitrates device ownership among nodes [85, 157, 148, 74, 75].

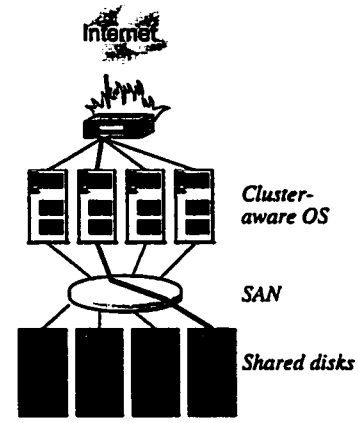


Figure 1.4. *An example of SAN-based clusters.*

This architecture can be seen as a fault-tolerant variant of statically partitioned clusters — the data remain available as long as the disks remain operational, even when some of the nodes are down. It also automates some of the data assignment tasks — a file system can grow automatically as long as there are unused disks attached to the SAN. However, the SAN architecture still suffers from the problems of statically partitioned clusters, albeit to a lesser degree. The function distribution must still be determined manually. Data distribution is neither totally dynamic nor automatic — once a user or a data item is assigned to a disk, it cannot be moved transparently to accommodate changing workload or configurations. Moreover, it is far more expensive than other solutions, because these systems run only on uncommon, proprietary hardware. They also have a limited scalability, only up to low tens of nodes.

1.3.4 Clustering using Distributed Database Systems

A cluster can be also built using a distributed database management system [118, 117]. This solution is similar to statically partitioned clustering in that data and function are still distributed statically by the administrator. The main advantage of this architecture is that it can maintain the consistency of data structures that span multiple nodes, giving the system a latitude of performance and data-layout tuning. For example, AOL uses a database system in part to consolidate the body of mass-mailed messages (i.e., to implement a single-instance store).

This solution, however, still suffers from the problems of statically partitioned clusters. Moreover, managing a consistent, distributed database is far more complex and expensive than managing a group of single-node storage systems. For these reasons, this architecture is not commonly seen in Internet services.

1.3.5 Clustering using Distributed File Systems

One could build a cluster by using a distributed file system to store all data and running a number of stateless, front-end nodes to handle all client sessions. This approach can be seen as a software-only version of the SAN-based architecture. Using this architecture in data-intensive services requires the file system to be highly scalable under changing workload and system configurations. Such file systems do exist [6, 7, 150], but they are still in the basic research stage due to their sheer complexity. Even if they were available now, their level of manageability and availability would not be optimal, because they offer generic, single-copy semantics and sacrifice availability. For example, they tolerate only a limited number of node failures, beyond which the entire system crashes, and they stop functioning when the network is partitioned. In this dissertation, we desire more — we want to tolerate any number of node failures and continue serving users even during network partitioning.

1.3.6 Summary

The traditional solutions for building large, data-intensive Internet servers trade off cost, performance, availability and manageability. For example, monolithic servers are easiest to manage, but they are expensive, whereas statically partitioned clusters are cheap, but they are hard to manage.

1.4 Hypothesis and Contributions

In the preceding sections, we demonstrated the growing needs for an infrastructure for Internet services that scales in three dimensions: manageability, availability and performance. We also showed that existing solutions for building large data-intensive services sacrifice one of the scalability goals to achieve another, and they consequently drive up the cost of system ownership.

The hypothesis of this dissertation is that it is possible to build a scalable data-intensive service by reconsidering the cluster system design from the ground-up. We identify the static function and data placement of traditional clustering architectures as the source of their scalability limitations. We propose a new clustering architecture, *functionally homogeneous clustering* (*FHC*), for building manageable, available, fast, and inexpensive data-intensive services. Functionally homogeneous clusters treat all nodes in the cluster equally and assign both computational and data storage tasks to the nodes dynamically. For example, in a functionally homogeneous email service, a session with a client can be hosted at any node, a user's account information can potentially be managed at any node, and an email message of any user can be stored on any node. This dynamic architecture improves the system's availability by allowing data and functions to be moved automatically to mask failures. It also improves performance by dispatching functions and data dynamically in response to changing workloads. Finally, it improves manageability by automating most of the administrative tasks that require human attention in today's clustering solutions.

This dissertation further develops three specific techniques for realizing the FHC architecture:

Automatic reconfiguration: FHC migrates function and data across nodes to ensure a continued service whenever a node is added or removed, intentionally or accidentally.

This dissertation develops a suite of scalable reconfiguration protocols for FHC. It ensures that the function and data managed on the removed nodes migrate to live nodes, and that all the live nodes agree on how the data are distributed, even after such failures as network partitioning and sudden node retirement. We also prove the correctness of the protocols.

High-throughput optimistic replication: FHC keeps replicas (copies) of important data on multiple nodes for availability. This dissertation develops an efficient replication algorithm suitable for data-intensive Internet services.

This algorithm has several desirable properties for use in FHC. It can tolerate the most severe types of failures, including multiple node failures, network partitions, and long-term failures. It also allows the location of replicas (the *replica set*) to be changed dynamically on a per-object basis, enabling the system to move or add replicas in response to configuration changes and to support heterogeneous clusters that contain disks with different speed and capacity.

Finally, it is efficient both computationally and spatially. These advantages are achieved by allowing replica contents to diverge in the short term. We also prove the correctness of the algorithm.

Dynamic load balancing for masking changes in workload and configuration: We develop two load-balancing mechanisms for FHC. The first mechanism, the *load balancer*, chooses the best set of nodes for storing each incoming data item (e.g., an email message). It ensures that the incoming workload is distributed optimally to nodes and that data delivery is never blocked by failures. The second mechanism, the *rebalancer*, runs periodically on each node to improve the long-term health of the system. It coalesces over-fragmented data for better performance and adds or removes replicas of objects to achieve the proper level of availability.

These mechanisms not only improve system performance, but more importantly, they improve system manageability by masking non-uniformity in both the system configuration and workload distribution.

To prove the concept of FHC, we design and implement *Porcupine*, an email server that incorporates the aforementioned principles and mechanisms. We prove the hypothesis by rigorously studying the behavior of Porcupine in a 30-node PC cluster and showing that it indeed scales in all three dimensions: performance, availability, and manageability. Furthermore, we show, through analysis and simulation, that Porcupine will be able to grow to a far larger scale.

1.5 Thesis Roadmap

Chapter 2 introduces the concept of functionally homogeneous clustering and discusses its potential benefits — manageability, availability, and performance through dynamic function and data placement. This chapter also points out the challenges FHC faces and introduces three key techniques for overcoming the challenges: naming database with automatic reconfiguration, optimistic replication, and load balancing. Chapter 3 reviews previous research in these areas. Chapter 4 presents data structures and managers that constitute functionally homogeneous clusters and describes how these components cooperate to become an Internet service, using the Porcupine email server as an example.

The next three chapters delve into the details of the three techniques that realize FHC. Chapter 5 describes a suite of proven-correct distributed protocols — membership agreement, user map recomputation, and profile and fragment recovery — that recovers the cluster's name database scalably after any number and types of configuration changes. Chapter 6 presents the optimistic replication algorithm that supports dynamic changes to replica placement and handles most types of failures gracefully. We describe the rationale and operations of the algorithm and introduce a set of implementation techniques for performance improvement. Chapter 7 discusses two mechanisms for distributing workloads among cluster nodes: load balancing for choosing the best set of nodes to store incoming data, and rebalancing for periodically moving data among nodes to improve the long-term throughput and availability of the system.

Chapter 8 evaluates FHC by studying the behavior of the Porcupine email server on a 30-node PC cluster. We measure Porcupine's steady-state performance under uniform workload. We also run Porcupine with a variety of unbalanced configurations and workloads and show that it can achieve optimal performance automatically even under these conditions. Chapter 9 points out several open issues we discovered in the course of the study, and Chapter 10 concludes the dissertation by summarizing its findings and contributions.

This dissertation also includes several appendices to discuss topics deemed too detailed for the main chapters. Appendix A describes the implementation of the Porcupine email server. In particular, it focuses on issues that required extra attention to make the system scalable and robust, including its multi-threading structure, inter-node networking, and storage architecture. Appendix B defines the notational conventions used by pseudo-code that appears throughout the dissertation. Appendix C formally describes the recovery protocols introduced in Chapter 5, and Appendix D formally describes the replication protocol introduced in Chapter 6. Appendices E and F present the correctness proofs of the recovery and replication protocols.

Chapter 2 Functionally Homogeneous Clustering: Principles and Strategies

This chapter reviews the principles and strategies that guide functionally homogeneous clustering. We first review the benefits of clustering for large-scale Internet services and simultaneously point out the inherent managerial and architectural complexities that have eluded satisfactory solutions. Next, we introduce the principle of functionally homogeneous clustering for solving these problems and discuss its high-level strategy — the exploitation of application semantics — for scaling without sacrificing quality of service. Finally, we present the three key techniques — automatic recovery, replication, and load balancing — that together realize the FHC architecture.

2.1 Clustering: Pros and Cons

Clusters offer many benefits over monolithic mainframe computers, especially for building large-scale Internet services [122]. First, a cluster can incrementally scale its throughput by adding new nodes or disks, provided that the software can distribute the workload efficiently to the new hardware components. Nodes in a cluster also form a natural fault-isolation boundary; this allows the cluster to continue operation even after some of the nodes fail, provided that the software can transfer the functions and data handled by the failed node quickly to other nodes. Contrast this with monolithic mainframe computers that usually crash entirely when a CPU or a memory block fails. Finally, clusters are inexpensive. PCs are sold in a larger and more competitive market than that of mainframes, and the fault isolation property allows clusters of inexpensive hardware components to be built.

These benefits naturally do not come free. The primary disadvantages of clusters are the complexity of both the software architecture and the administration.

Clusters contain a heterogeneous combination of nodes as the result of their continuous growth. With today's rapid hardware evolution, it is not unusual for one node to be 5 times faster, or to have 10 times more disk capacity than another in the same cluster. The software or the administrator

needs to gauge the performance of the nodes and distribute the workload to utilize cluster resources efficiently. In today's data-intensive Internet services, this task falls mostly on humans because of the static cluster architecture [35, 47, 79].

The fault-isolation property, while beneficial to users, complicates both the software architecture and the administration. The software must replicate data across multiple nodes to keep data available after node failures. Maintaining the consistency of these copies in the face of failures is complex, because cluster components can fail in a myriad of ways: power failures, disk failures, link failures, network partitioning, and thrashing.

These complexities drive up the total cost of today's clusters [101].

2.2 Functionally Homogeneous Clustering

This dissertation proposes *functionally homogeneous clustering* (*FHC*) for building scalable data-intensive Internet services. FHC allows any node to manage any function and any piece of data. For example, in a functionally homogeneous email server, each node in the cluster can handle every email protocol, including SMTP, POP, or IMAP. A particular user's mailbox can be split and replicated across any set of nodes, and the user's profile information can potentially be managed by any node. Before asking how such an architecture can be realized, let us discuss its potential benefits.

Better performance: Unlike traditional, statically partitioned clusters, FHC is a fully dynamic architecture. The software, not the administrators, determines the placement of the data and the functions using dynamic measures, such as current load level and disk usage. Thus, FHC can *adapt* to the changes in the incoming workload and the system configuration by distributing more work on faster (or less loaded) nodes automatically. Such dynamism improves FHC's performance.

Better availability: Because each node runs all the functions, an FHC-based cluster can guarantee service to users as long as one node is alive in the cluster. Users' data can be replicated on a dynamically chosen set of nodes (replicas), guaranteeing their availability as long as one replica is alive. When some node remains down for too long, the system can automatically add new replicas on other nodes to maintain a proper level of availability.

Better manageability: FHC simplifies cluster management and frees administrators from continuous system monitoring and manual workload distribution. Newly added nodes will automatically receive their share of tasks, and crashed (or retired) nodes will be excluded from the cluster automatically, leaving no residual information on other nodes. The system's capacity grows and shrinks with the number and the aggregate power of the nodes, and the system's service quality remains the same regardless of failure. For example, in a functionally homogeneous email service, even when a user temporarily loses her old email messages after the unlikely event of a crash of all of their replicas, she can still receive new incoming messages (which are usually more important than already-read messages), because they are always delivered on live nodes.

2.3 Function Distribution Strategies

FHC poses many architectural challenges before its potential advantages are realized. Allowing any node to perform any function is achieved, at least from the mechanism viewpoint, by running every software module on every node. This is indeed what FHC does. The system may still need a mechanism for scheduling system activities intelligently, especially when the cluster runs a multitude of services that stress the CPU, network, and disks in unpredictable ways. This dissertation, however, focuses on running a single service well in a cluster and does not investigate the issue of function distribution any further. Mixed-resource, distributed scheduling is our future work (Section 9.4.2).

2.4 Data Distribution Strategies

Dynamic data distribution in FHC is the main focus of this dissertation. Letting any node manage any piece of data and ensuring that it remains available after failure is far more difficult to achieve than ensuring the same for functions. To survive failures, data must be copied on multiple nodes, which introduces the complexity of keeping the copies consistent, as discussed in Section 2.1.

The key principle that guides FHC's data management strategy is to *exploit the application semantics*. FHC is not a general-purpose computing platform. Rather, it is an infrastructure specialized for data-intensive Internet services. We take advantage of the properties of the services whenever that allows the system to scale without compromising its service quality. This principle is

specifically stated in terms of two strategies: (1) *be optimistically consistent*, and (2) *use soft state whenever possible* [61, 71, 127].

2.4.1 Be Optimistically Consistent

One example of the exploitation of application semantics is the type of data consistency FHC guarantees. There exists a fundamental trade-off between the consistency and availability of data in a distributed system: the higher the consistency, the lower the availability [60, 164, 71]. For example, in an asynchronous network[1], replication algorithms with non-blocking, single-copy semantics (i.e., replicas are always kept identical) cannot tolerate even a single node failure [56, 100]. Consistency and availability become compatible only by investing in redundant hardware to bound the transmission delay. However, this solution gives only a probabilistic guarantee, which unfortunately diminishes as the system scales up. On the other hand, optimistic replication algorithms that allow non-blocking accesses to replicas sometimes sacrifice data consistency by serving arbitrarily stale data [149, 121, 128, 65, 46]. FHC takes the availability end of the spectrum and guarantees only that all the data it manages become *eventually* consistent with each other, provided that no new failure happens for a long period. FHC's weak data consistency excludes its use in applications that demand high reliability, such as banking and e-commerce, but we believe that in most data-intensive Internet services, users prefer being able to access data even when the data might be old.

This strategy permeates many aspects of the FHC architecture. For example, on-disk data, such as email messages and user profiles, are updated only asynchronously. Thus, a user may temporarily see old contents even after she receives a confirmation for an update. The same idea is also applied to maintain the name service components in FHC. For example, email messages are discovered after a node failure asynchronously, creating a small window in which a user may temporarily lose her messages.

---

[1] An asynchronous network guarantees no upper bound for end-to-end message transmission delay, with one important implication: a failed node cannot be accurately distinguished from a slow node [100]. All the network media used in practice are asynchronous.

2.4.2 *Use Soft State Whenever Possible*

FHC distinguishes two types of data it manages: *hard state* and *soft state*. Hard state is the information maintained persistently (e.g., on hard disks). An email message, a Usenet article, and a user's password are examples of state that must be hard (i.e., these data must not be lost even after a crash). Hard state must be replicated on multiple nodes to survive failures. On the other hand, soft state consists of information that, if lost, can be reconstructed from the hard state [37, 61]. The list of nodes containing mail for a particular user (*mail map*) is soft state, because such information can be obtained by scanning all the disks in the cluster and discovering the user's messages, at least in theory. The list of live nodes in the cluster (*membership list*) is also soft state, because it can be reconstructed by polling all nodes in the cluster.

The concept of soft state is application-specific, but soft state is fundamentally easier to maintain than hard state. It can dispense with the complexity of managing multiple, persistent replicas, because it can always be recovered from hard state, regardless of the number or type of failures. It is also resilient against catastrophic disasters. For example, suppose that some hard disk sectors crash. When the location of mailboxes is managed as soft state, the system can still automatically access whatever mailboxes have survived the crash. In contrast, if the location is maintained as hard state, the administrator is forced to scan the broken disk manually and repair names stored on other disks (on other nodes), a task next to impossible.

FHC tries to use soft state whenever possible. For example, the FHC's name database that locates on-disk objects consists entirely of soft state. The cluster membership list and the node status (used for load balancing) are also soft state, obtained on demand from the cluster nodes. Most soft state in FHC is maintained on only one node at a given instant and is reconstructed from hard state after failure. The exception is when directories that name and locate state are themselves soft state (the cluster membership list is an example). Such directories are replicated on every node to improve performance.

2.4.3 *Trade-offs*

The aforementioned two strategies make the system simple, efficient, yet highly available. There are potential disadvantages, though. Soft state needs to be reconstructed after configuration change,

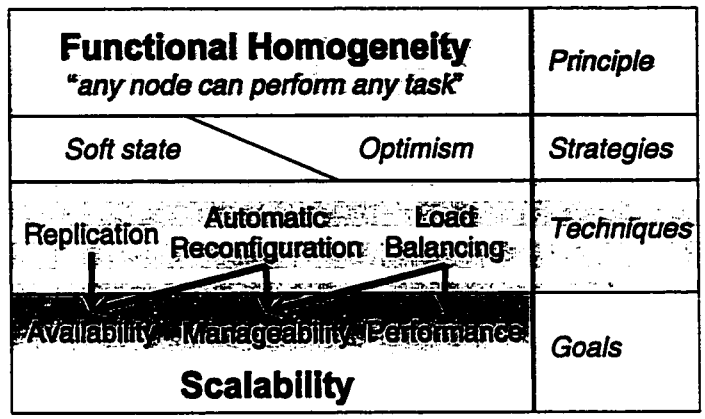


Figure 2.1. *The primary goal of functionally homogeneous clustering is scalability, defined in terms of manageability, availability, and performance requirements. In turn, these requirements are met through combinations of the three key techniques shown above.*

and the reconstruction consumes memory, computation, and disk bandwidth. Optimistic replication algorithms are also known to be less efficient than single-copy counterparts in the common case [136]. One of the key contributions of this dissertation is the development of a set of practical techniques to solve these problems. The following sections introduce these techniques.

2.5 Key Techniques

This section introduces the techniques we have developed for realizing the FHC architecture. Functionally homogeneous clusters must be able to store any object on any node and later retrieve the object. This problem decomposes into three sub-problems: tracking data locations, ensuring the availability of data, and deciding on data placement. They are solved by the following three techniques: *naming mechanism with automatic reconfiguration*, *optimistic replication*, and *dynamic load balancing*. Figure 2.1 shows how these techniques synergistically achieve our three goals of scaling: manageability, availability, and performance.

2.5.1 Naming and Automatic Recovery

FHC provides a distributed naming service for locating on-disk data. This mechanism locates all the hard state, such as the user profile and email messages, stored on live nodes in the cluster and

recovers automatically from any number or type of failures.

This service needs to achieve several goals: (1) spread the name management tasks evenly among nodes for balanced loads, (2) have good common-case performance, (3) tolerate many types of failures and always locate all the live objects in the cluster, and (4) demonstrate a low and constant cost of recovery.

Three observations lead to the design of the naming service. First, names are inherently soft state. In other words, names themselves do not convey any new information that cannot be obtained from the hard state. Even if (a part of) the name database is lost because of a node crash, it can be rebuilt on another node by letting all nodes scan their disks and discover the names of locally stored objects. Second, the naming service can be optimistic. In the event of a failure, we only demand that all objects on live nodes be eventually discovered, but we can tolerate lost or phantom names as long as they are fixed eventually. Third, data-intensive Internet services access data in a structured and restricted fashion. Because of the frequent updates they experience, these services first need to discover the information the user wants before retrieving or modifying an individual data item. For example, both POP and IMAP email retrieval protocols demand clients to log in and obtain the digest of the mailbox before reading an individual message [113, 39]. Similarly, Usenet and BBS require users to obtain the digest of a newsgroup before reading an individual article [81]. These structured access paths allow the name service to become *lazy* and forego remembering the names of all objects.

Based on these observations, FHC's naming service is designed to be purposefully application-specific. It sorts objects into two levels of hierarchy: randomly accessed names and dynamically discovered names. For example, for an email service, the first level manages the user profile, which is directly consulted when an email session starts, and the second level corresponds to the users' email messages, which are discovered dynamically during the email session. For a BBS service, the first level manages newsgroups, and the second level manages articles in the newsgroup. For the sake of conciseness, we use email terminology — the *profile* for the first level and *fragments*[2] for the second level — for describing the naming service in this dissertation.

Figure 2.2 shows a schematic view of the naming data structures. The first level in the name

[2]The set of messages belonging to a particular user on a node is called a *fragment*, because it is a fragment of a logical mailbox. We describe the data structures in more detail in Section 4.3.1.

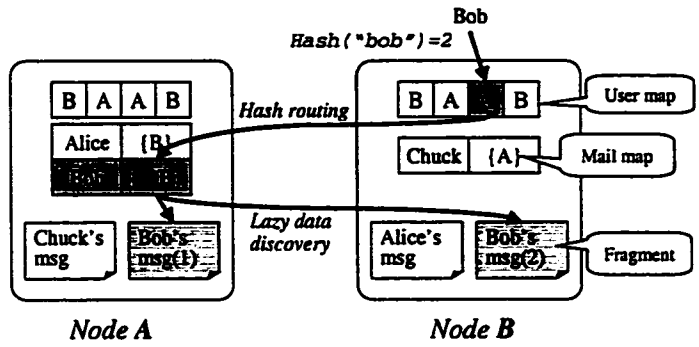


Figure 2.2. *This picture shows how the FHC's naming service is structured. The names in the first level (i.e., user names) are partitioned using the user map replicated on each node. Because user "Bob"'s name hashes to 2 and the entry number two of the user map is A, he is managed on node A. The manager of each user only knows the list of nodes that stores the fragments (data) for the user. Thus, to obtain Bob's fragments, one requests nodes A and B to discover all his fragments.*

space is managed by hash routing [119, 134, 154]. Here, the user population is partitioned into a fixed number of equivalence classes (also called *buckets*) by applying a well-known hash function on user names. The *user map*, a small in-memory table replicated on each node, dynamically assigns a class to a node. By spreading the responsibility for user management evenly across all nodes in the system, larger user populations can be supported simply by adding more machines.

Objects in the second level, e.g., email messages belonging to a particular user, are located lazily. The naming service itself only keeps the list of nodes that store objects for a particular profile. This list, called the user's *mail map*, is maintained as a part of the user's profile; i.e., mail maps are also hash-partitioned and are managed by nodes determined by the user map[3]. To read a particular email message, the user first obtains her mail map and asks each node in the map to generate the digest of messages for the user. Next, the user finds the message she wants from the digest and asks the node that stores the message to retrieve the text. This seemingly inefficient operation actually does not cause a slowdown because, as discussed before, most data-intensive services already perform dynamic data discovery as a part of the sessions.

This design is in contrast with most other naming services for clusters that replicate names

[3]In practice, a user can have multiple mailboxes. Thus, a mail map actually maps a particular mailbox of a particular user to the list of nodes. The node that manages a user's profile also manages the mail maps for all the mailboxes for the user.

themselves on persistent storage (e.g., xFS [7], Active Directory [103], DNS [109, 110], and DDS [71]). Unlike them, all the elements of FHC's naming service — i.e., the user map, the profile, and mail maps — are the soft state that resides only in nodes' memories. After a node crashes, one of the remaining nodes computes the new user map by reassigning buckets managed by the failed node to live nodes. The new bucket managers then rebuild the name database by having all nodes discover the profile and fragments from their disks. This recovery mechanism guarantees that all the profile and email messages on live nodes are eventually discovered and placed in the name database, regardless of the number or the type of failures.

This approach simplifies the system and minimizes persistent store updates and message traffic. The potential disadvantages include the memory overhead of the soft state and the consumption of the disk, the network, and the CPU during failure recovery. Chapters 5 and 8 describe a distributed set of protocols for scalable reconstruction of the soft state and show that these problems can be solved with a negligibly low cost.

2.5.2 High-throughput Optimistic Replication

The hard state, such as email messages, login names, and passwords, can survive failures only by having multiple copies. FHC uses the replication service for this purpose. This service faces several challenges. Being a key piece of functionally homogeneous clusters, it must be fine-grained and dynamic. It must allow each small data object, such as an email message, to be stored on its own set of replica nodes, giving the system the maximum freedom of data placement. It should also support adding or removing replicas dynamically to react to node failures and recoveries without taking the system off-line. Second, the service must be highly fault tolerant, ensuring continuous access to the hard state even when the majority of the replicas are down, or when nodes remain failed for a long period. Finally, it must be efficient, using little CPU power and disk space for consistency maintenance.

We address these challenges by developing a new replication algorithm that is decentralized, non-blocking, and optimistic. FHC's replication algorithm allows an update to be issued at any replica at any time, and the update is propagated to others asynchronously. Being optimistic, replica consistency is maintained within no guaranteed time frame, although the window of inconsistency is

usually smaller than what a user can detect. This optimism causes little problem in typical Internet services because of their weak consistency requirement. In return, our algorithm becomes extremely fault tolerant and efficient. Chapter 6 discusses our replication algorithm in more detail.

2.5.3 Load Balancing

The final component of functionally homogeneous clustering is the load-balancing mechanisms for determining the data placement and optimizing the workload distribution. FHC runs two services for this purpose, the *load balancer* and the *rebalancer*.

The load balancer, running on each node, is responsible for choosing the best set of nodes for storing (replicating) incoming data. The primary challenge in designing the load balancer is to resolve the tension between affinity and load balance. For example, while splitting a user's mailbox across many nodes helps equalize the system load, doing so excessively will harm system performance: not only does it increase the size of the name database and add to memory pressure, but it reduces disk throughput by increasing the number of disks accessed to read the mailbox. The load balancer resolves this tension by limiting the number of nodes a user's data can be stored on (called the *spread limit*), widening it primarily to deal with failure. In this way, the system behaves and performs like a statically partitioned system when there are no failures and load is balanced well, but like a dynamically partitioned system otherwise.

The rebalancer improves the long-term performance and availability of the system. The load balancer is inherently "short sighted" in that it only works for newly incoming data. It may sacrifice the long-term health of the system, e.g., by excessively fragmenting mailboxes to mask transient failures. The rebalancer, running periodically on each node, addresses such problems by coalescing fragmented mailboxes and adding or removing replicas to maintain a proper level of availability for each replicated object.

Chapter 7 discusses FHC's load balancing services in more detail.

2.6 Summary

This chapter introduced functionally homogeneous clustering. This architecture is characterized by dynamic scheduling of both function and data distribution and exploitation of application seman-

tics. FHC becomes manageable by the combination of the automatic reconfiguration and the load balancing mechanisms that let the system react automatically to changes in the number or quality of machines, users, and workload. The system becomes available by the combination of automatic reconfiguration and the replication mechanisms that keep data (such as email messages and the user profile) accessible after failures. Finally, FHC improves performance by the load balancing mechanisms that distribute workloads optimally to nodes in the cluster.

The design of these three mechanisms is guided by two key strategies, optimism and the use of soft state. The recovery mechanism tolerates failures by making the entire name database soft. It relies on asynchronous and optimistic name discovery mechanisms to recover the soft state scalably from failures. Our replication algorithm is also inherently optimistic, but it allows any user to retrieve any data at any time. Finally, the load balancing service relies on soft cluster state to decide on data placement and on a two-tier optimistic strategy to correct data placement anomalies in the background.

Chapter 3 Related Work

This dissertation builds on several areas of research, including clustering architecture, load balancing, fault tolerance, and replication. This chapter reviews the prominent previous work in these areas and contrasts it to the approach we adopted.

3.1 Data-intensive Internet Services

The prototypical distributed mail service is Grapevine [140], a wide-area service intended to support about ten thousand users. Grapevine users are statically assigned to user-visible registries, just as today's Internet email users are assigned to user-visible domains. The system scales through the addition of new registries having sufficient power to handle their populations. Nevertheless, Grapevine's administrators are often challenged to balance users across mail servers. In contrast, Porcupine implements a flat name space managed by a single cluster and automatically balances load. The flat name space also benefits users by letting them roam without changing their email addresses. Grapevine provided an optimistically replicated user profile database, but it did not replicate mail messages. Porcupine uses optimistic replication for both mail and the user database.

As discussed in Section 1.3.2, most contemporary email clusters partition function and data statically among nodes in the cluster. Some of them deploy a set of file system nodes (e.g., NFS [139, 143]) in the back-end and let each node manage a specific set of users. They run off-the-shelf server software (e.g., sendmail [142], popd [80], and imapd [116]) on a number of front-end nodes and route each client session to one of the back-end nodes using a priori knowledge of user-to-node mappings. Others run off-the-shelf email software at the back-end and protocol redirectors at the front-end that forward email requests to the back-end [18, 47].

Usenet servers and web-based BBS servers have traditionally been built as monolithic servers (Section 1.3.1) or as clusters that combine a back-end file-system node and stateless front-end nodes (Section 1.3.5), mainly because of their relatively small size. Some exceptionally large servers (e.g., the main news server in our college, delphi.com, and egroups.com) run clusters that statically assign

newsgroups to nodes [79, 45].

As we demonstrate in this dissertation, the static approaches adopted by these servers are difficult to design and manage, adapt poorly to changes in workloads and configuration, and have limited fault tolerance.

3.2 Distributed Storage Management

Distributed, persistent data structures are an active area of study. For example, LH* [97] and RP* [96] provide distributed tables that can grow dynamically and asynchronously. LH*$_{RS}$ [98] and dPi-tree [99] further replicate tables for fault tolerance. DDS [71] provides a distributed, persistent, and replicated hash table, on which applications such as web servers or protocol gateways are constructed. DDS maintains single-copy semantics using a combination of distributed transactions and whole-database copying on reboot.

Several distributed file systems address scalability and fault tolerance goals that are similar to ours. Echo [17] and Harp [94] provide fault tolerance by primary-copy replication and scale by a form of static partitioning. Newer systems, such as Frangipani [90, 150] and xFS [7], provide dynamic data partitioning using distributed tables.

The differences between these systems and ours illustrate the trade-off made between generality and availability. The aforementioned work provides general-purpose persistent data structures that can manage any number of objects with a constant overhead, but at the cost of less fault tolerance. For example, they assume a synchronous network (i.e., a bounded end-to-end messaging latency), which is theoretically impossible and requires a large hardware investment to realize practically [14, 30, 32, 56, 145]. As another problem, a persistent data structure is only as available as the nodes that replicate it, but having more replicas lowers the throughput of updates. Thus, using a persistent structure, one is forced to trade off between availability and performance. In contrast, our naming mechanism is designed specifically for data-intensive Internet services. It limits the number of entries that a node can handle (Section 8.3.5), but it is more efficient and available than these systems.

3.3 Clustering Infrastructure

3.3.1 Clustering for Reliable Storage Management

Numerous fault-tolerant, cluster-based computing products have been developed in the past, with supports for membership agreement, distributed locking, and atomic resource fail-over. Examples include VMS clusters [85], Microsoft cluster service [157], Sun clusters [148], HP high-availability clusters [74], and IBM HA-CMP [75]. Often coupled with these systems are reliable, network-attached storage systems [152], such as EMC Celerra [52], Network Appliance Filer [114], Compaq ServerNet [122], and IBM SSA [76].

These clusters are designed primarily for relational database systems, although some are used to build email servers as well (e.g., Microsoft Exchange server [159] and Sun Internet mail server [104]). Their main goal is high availability, with only marginal support for performance scaling (in fact, many installations are two-node clusters with one node working as a hot backup). Scaling the performance of such systems requires static partitioning, with performance and managerial problems already discussed in Section 1.3.2. As another downside, they require proprietary hardware and software to implement accurate resource fail-over. Unlike these systems, the goal of FHC is to scale to hundreds of nodes using standard off-the-shelf hardware.

3.3.2 Clustering for Internet Services

Fox and others [61, 21] describe an infrastructure for building scalable cluster-based Internet services. They introduce a data semantics called BASE (Basically Available, Soft-state, Eventual consistency) that offers advantages for web-search and document-filtering applications. The architecture Fox describes is, in fact, quite popular in today's large-scale web servers [23, 43, 119, 123, 144]. Our work shares many of their goals: building scalable Internet services with a semantics weaker than traditional database systems. As in Fox's work, we observe that ACID (Atomic, Consistent, Isolated, and Durable) semantics [13, 68] adopted in traditional database systems may be too strong for our target applications and define a data model that is equal to the non-transactional model used by the system's clients. However, unlike BASE, our semantics support write-intensive applications requiring persistent data. Our services are also distributed and replicated uniformly across all nodes

for greater scalability, rather than statically partitioned by function and data.

3.4 Load Balancing

3.4.1 Theory of Load Balancing

Load balancing has been studied extensively in the context of scientific computation [11, 24, 51]. Many of these studies, however, are not directly applicable to Internet servers. Scientific applications are characterized by centralized admission control, deterministic parallelism, long yet predictable runtime, and CPU-bound workloads, all of which make these applications a rather easy target for load balancing. In contrast, Internet servers have the opposite set of characteristics: complete decentralization, unpredictable and short-lived transactions, and workload that pressures the CPU, the network, and disks equally. Among the studies in this area, the work by Eager and others [51] most relates to ours. They describe a group of algorithms that poll a random set of nodes and pick the least loaded among them. They show that such simple algorithms probabilistically perform as well as more complicated algorithms with far smaller overhead.

This random-polling class of algorithms has recently received a renewed interest due to increasing demand for scalable, cluster-based Internet services. Mitzenmacher and Dahlin extend Eager's algorithm to let each node cache other nodes' load information (rather than polling every time) [42, 107, 108]. In particular, they study how the cache-flush interval and the size of the candidate node set affect the quality of load balancing. They present two main findings. First, when the cache flush interval is large (i.e., load information is allowed to remain rather stale), a larger candidate set exacerbates "herd behavior", in which the globally least-loaded node is flooded with requests because it is picked by all nodes at once. Second, the candidate-set size of two is optimal for a wide range of settings. These findings match what we observe in Chapters 7 and 8.

3.4.2 Load Balancing in Clusters

In the context of the web and clustering, several commercial products transparently distribute requests to cluster nodes (e.g., Cisco Local director [36], F5 BigIP [115], Foundrynet ServerIron [58], and Resonate Central Dispatch [130]). Some cluster-based Internet servers balance the load on the back-end storage servers by dispatching requests intelligently at the front-end nodes [43, 61, 144].

These systems distribute the workload round-robin or by using a simple load measure, such as queue length or average response time. Pai and others describe a LARD (Locality-Aware Request Distribution) mechanism for cluster-based web services [8, 119]. In LARD, the front-end nodes analyze the request contents and attempt to direct requests so as to optimize the use of buffer cache in the back-end storage nodes, while also balancing load. We use the load information, in part, to distribute incoming traffic to cluster nodes. However, unlike previous load-balancing studies that assumed complete independence of incoming tasks, we also balance the write traffic, taking data affinity into consideration.

3.4.3 File Allocation Problem

The problem of choosing the best set of nodes for storing replicas for an object is known as the *file allocation problem.* A common approach is to record the history of accesses to each replica and to add, delete, or move replicas dynamically when a replica is found to be too busy or too idle [9, 78, 125, 160]. These studies focus mainly on balancing the load of the replicas for a particular object. In contrast, FHC's goal is a better overall system throughput; it is less concerned about the load balance of a particular object, because it manages many objects simultaneously. Thus, in our architecture, the replica set of an object is determined on creation and is changed only when nodes subsequently fail or recover.

3.4.4 Hash Routing

Functionally homogeneous clusters use replicated user maps to distribute user management task evenly among nodes. This technique, known as *hash routing*, has attracted wide attention recently — e.g., for data distribution in operating systems [7, 53, 145] and balancing the load of cluster-based web proxies [82, 119, 154] — for its simplicity and ability to balance load transparently without a deep knowledge of workload characteristics. Our work extends the prior art by developing a reconfiguration protocol for hash routing that accurately and optimally redistributes the service contents after configuration change.

3.5 Failure Recovery

3.5.1 Cluster Membership Agreement

Membership agreement has long been a fundamental building block of many cluster-based systems. The simplest and most widely used protocol is the *independent assessment protocol* [61, 144, 93, 155]. In this protocol, each node broadcasts "*I am alive*" packets periodically. When a node fails to hear "*I am alive*" from another node for a while, the former assumes the latter dead. While being simple, this protocol has one shortcoming: because it works independently on each node, it cannot coordinate nodes into an action after change (e.g., recomputing a new user map in FHC or transferring the device ownership in device-sharing clusters.). Thus, this protocol is applicable only when the task is stateless, e.g., for managing HTTP proxies.

Systems that need to coordinate nodes into a recovery action use phased protocols similar to ours. Device-sharing clusters, including Tandem Non-Stop Computers, VAXclusters [85, 145], and Microsoft Cluster Service [157], use a six-round membership protocol to coordinate resource fail-over during configuration changes. Atomic broadcast protocols also use membership protocols similar to ours [4, 49, 131].

The theoretical study of membership protocols has begun only recently. Chandra and Toueg proved the impossibility of membership agreement in asynchronous networks [31]. Cristian and Schmuck defined the timed-asynchronous distributed system model, in which messaging delay is bounded in the steady state, and they presented formal requirements for membership protocols [40, 41]. They also developed three proven-correct membership protocols, including the three-round protocol on which FHC's membership protocol is based.

3.5.2 Soft-state Recovery

In a functionally homogeneous cluster, the location of on-disk objects is maintained as soft state that is reconstructed after configuration changes. This idea of simplifying recovery using state soft is not novel by itself. Data sharing protocols, such as NFS and HTTP, treat client caches as soft state and recover them after server crash by polling [55, 139, 143]. Naming services for mobile computers track host locations using periodic polling [1, 127]. In SNS (Scalable Network

Service) that centrally manages nodes in a cluster-based web server, its entire state (e.g., the cluster membership list and the load of other nodes) is soft and is updated periodically so that its function and data can easily be transferred to another node after crash [61].

These systems rely on periodic polling (or pushing) to maintain the soft state. As such, they can be used only when the size of the state is small. In contrast, FHC obviates the need for periodic polling and instead relies on a sophisticated combination of the membership and user-map recomputation protocols to minimize the amount of state recovered after change. Thus, our solution scales better to larger clusters that handle stateful workloads.

3.5.3 Transactional Recovery

The naming and replication services in FHC can be viewed as fault-tolerant mechanisms for maintaining the consistency of distributed data structures (e.g., a fragment and its name or the replicas of an object). Distributed data structures are commonly maintained using a consensus algorithm that lets nodes reliably agree on the outcome of an operation. Examples of such algorithms include two-phase commit [13, 68], three-phase commit [13], and Paxos [86, 89, 90]. They are the building blocks of general-purpose distributed systems because of their ability to coordinate nodes to execute an arbitrary sequence of operations. Their main downside is the performance penalty, because they require a synchronous multiple-round voting for each operation. They also have fault-tolerance problems, in that a node crash may stall the progress of the entire system. In contrast, FHC's name space is maintained using soft state. It is inherently application-specific, but it is efficient and able to recover from any number of failures without ever stalling the system. Our replication algorithm also uses an optimistic, background update propagation to support non-blocking accesses to replicas after any number of failures.

3.5.4 Hardware-based Recovery

Transparent automatic reconfiguration has also been studied in the context of disks and networks. AutoRAID [158] is a disk array that moves data among disks automatically in response to failures and usage pattern changes. Autonet [132] and SmartBridge [133] are network bridges that automatically reconfigure in response to node failures and recoveries.

3.6 Replication

Replication has been studied and deployed for decades. Previous work in the area of replication, however, has addressed our goals only partially, and no single system has solved all the problems that we face in FHC.

3.6.1 Single-copy Replication Algorithms

Traditional replication algorithms achieve single-copy semantics, that is, they give users an illusion of having a single, highly available copy of an object. This goal can be achieved in several ways, but the basic concept remains the same: they prohibit accesses to a replica unless it is provably up-to-date. Primary-copy algorithms [44, 117, 48, 12, 33], used widely in commercial systems, designate one node as the primary, responsible for receiving, applying, and answering all the requests for a particular data item. Other nodes act as secondaries that only receive changes to the object from the primary node. When the primary fails, the system elects a new primary among the secondaries [48, 85]. Atomic broadcast protocols (e.g., Isis [15], Totem [111], and Transis [4]) coordinate a cluster by letting nodes receive messages in the same order. By running an ordinary database system on each node and feeding client requests through an atomic broadcast service, one can effectively build a replicated database server [120]. Quorum consensus algorithms make replicas vote every time an object is read or updated, and they require that a majority of the replicas agree on the newest contents [63, 151, 73].

Single-copy algorithms are suited to applications that require an extremely high data consistency, such as banking or e-commerce. However, they fail to address the problems we face: frequent failures and frequent changes. For example, primary-copy algorithms demand accurate failure detection, which is theoretically impossible [56, 32, 30]. Many atomic broadcast protocols also suffer from the same problem [15, 16] (recent systems, e.g., Transis [4], solve this problem by blocking message delivery and stopping the system when cluster membership is uncertain). Quorum-consensus algorithms are not only slow in the common case, but also expensive, because they require that the majority of the replicas be live; put another way, they require at least three replicas for each object.

In contrast, the semantics of our replication algorithm are tuned specifically for services with

weak consistency requirements, allowing any users to access any data most of the time even if that results in offering the user stale data. As a result, it can tolerate a far wider variety of failures than can single-copy algorithms.

3.6.2 *Mobile Database Systems*

Mobile replicated database systems (e.g., Bayou [121], Coda [83, 112, 91], Roam [128], and Refdbms [64]) share many properties with our replication service. In these systems, each replica can be read or updated even when no other replicas are communicable, and updates can be propagated between any pair of replicas to facilitate peer-to-peer data exchange. These systems are also optimistic, i.e., they sacrifice replica consistency for better availability. The primary difference between their solutions and ours is that these systems focus on minimizing communication overhead, whereas we focus on minimizing space and computation overhead. For example, many of these systems keep a log of changes (an operational log) for each object to minimize the amount of change exchanged between replicas, but such a design incurs unbounded space overhead and also complicates update conflict resolution. As another example, most mobile database systems use polling (e.g., connecting two laptops manually) to exchange updates between replicas, but polling becomes prohibitively expensive when each node replicates many objects that need to be reconciled with hundreds of other nodes. FHC addresses these problems by using contents-transfer and proactive update pushing (Chapter 6).

3.6.3 *Wide-area, Multi-master Replication Services*

The replication algorithm developed in this dissertation is most closely related to wide-area, multi-master replicated services, including Xerox Clearinghouse [46], Usenet [81, 92, 137], Active Directory [103], and several other experimental systems [2, 3, 163]. These algorithms achieve fault tolerance by being optimistic, and they alleviate the performance problem of mobile database systems by proactively pushing changes to peer replicas rather than polling them periodically. Pushing also supports many small objects replicated on a diverse set of nodes. They do not, however, support dynamic replica set changes and provide only weak fault tolerance. For example, one failed node can stall the update propagation of the entire system. Moreover, the existing systems do not support

quick object deletion and require storing update records (often called "death certificates" [46]) for an indefinitely long period.

3.7 Summary

This chapter reviewed the previous work related to FHC. The previous clustering solutions have mostly deployed static partitioning to scale. FHC proposes dynamic function and data distribution to solve the problems of static partitioning. The techniques developed in this work, such as automatic recovery, optimistic replication and load balancing, are not entirely new. They are evolutions of past ideas. It is the synergy of these techniques that differentiates our work from past systems, which deployed these techniques only individually.

Chapter 4 Functionally Homogeneous Clustering: A Structural Overview

This chapter overviews the structure and operations of functionally homogeneous clusters. We introduce the key data structures and managers that comprise a cluster. We also explain how these components cooperate to present an image of single, coherent system to users, using examples from Porcupine's email sessions.

4.1 Rationale for Email

FHC is fundamentally a general-purpose infrastructure for building scalable, data-intensive Internet services. In this thesis, however, we emphasize its use in an email service for two reasons. First is the practical demand for large-scale email services. Email is the oldest and most important among the data-intensive Internet services. Email has also seen the highest degree of service concentration (e.g., Hotmail already serves more than 100 million accounts) due to a historic drag and the practical advantage of having a single email address that one can carry over time and place. The second reason is the architectural challenge: email has the most update-intensive workloads and exhibits very little access locality.

Thus, to present the main data structures and managers that comprise FHC, we use an email-inspired terminology, such as "users" and "messages". Our prototype implementation of FHC, Porcupine, is also primarily an email server, although it supports WWW, FTP, and BBS services experimentally. We use examples from Porcupine's sessions to illustrate how these components fit together coherently to form an Internet service. Section 9.3 discusses how other data-intensive services are supported.

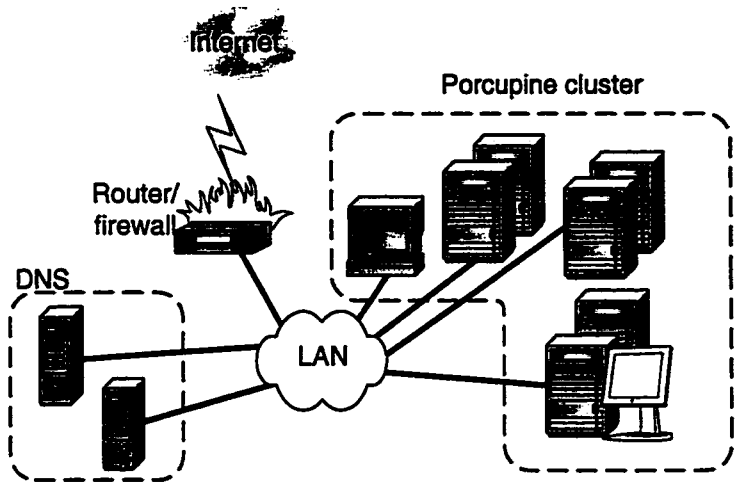


Figure 4.1. *Structure of a typical FHC cluster. Most of the work is handled by the cluster of functionally homogeneous nodes. In addition, DNS servers help clients discover the nodes, and routers connect the cluster to the Internet.*

4.2 Structure of a Cluster

Figure 4.1 shows the typical configuration of a functionally homogeneous cluster. As discussed earlier, most of the work — e.g., interaction with clients, user authentication, and data storage — is handled by a set of functionally homogeneous nodes running a identical set of software modules. All nodes in the cluster trust one another.

Nodes in the cluster are located by remote clients using a border routing service. The routing service is not a part of the functionally homogeneous structure, because it needs to be directly reachable from clients. One solution to the routing service is to register nodes in the cluster in a DNS domain and let clients pick the nodes in a round-robin fashion [22]. Dynamic DNS [156] can be used to update the DNS database after configuration change. An alternative solution is to deploy intelligent routers that support transparent request distribution (e.g., Cisco Local director [36] and Foundry networks ServerIron [58]). Their quick fail-over facility helps minimize the effect of failures to remote clients, but their other features, such as load balancing, will not provide much benefit in FHC.

4.3 Structure of a Node

FHC consists of a collection of data structures and a set of internal operations provided by managers running on *every* node in the cluster. Figure 4.2 shows the major components of a node. The components are broadly divided into three groups: the front-end components that interact with clients, the middle-tier components that distribute the workload among nodes, and the back-end components that manage the name database and on-disk data structures.

4.3.1 Major Data Structures

Each node manages two types of on-disk data structures (hard state): the profile of users and the data that the users own. Appendix A describes their actual format in Porcupine in more detail.

Profile database: This database describes the client profile, e.g., login names, passwords, etc. It is persistent and changes infrequently for a given user. The profile database is partitioned into buckets using the hash function, as are mail maps and the profile bank (Section 2.5.1). Each bucket, usually in its entirety[1], is replicated on several nodes for maximum availability.

The profile database is used only to bootstrap the profile bank, which is introduced later. A query to a user's profile is handled exclusively by the node managing the user's profile bank. As discussed in Section 4.4.4, an update to a user's profile is first accepted by the database node(s) and is then forwarded to the node managing the user's profile bank.

The replica placement policy for the profile database is currently left up to the administrator: however, a random assignment usually suffices, because the speed of access to this database is not critical to the system's overall performance.

Fragments: The collection of on-disk data a user owns (e.g., email messages) at any given node is called a *fragment*. This term is chosen because it is a fragment of a larger, logical object visible to the user (e.g., a mailbox). In other words, there is no single, persistent, mailbox

[1] A bucket is sometimes split into multiple groups, each of which is replicated on multiple nodes. The split happens only when a background maintenance process and a foreground account management job compete for an update, and it is repaired by the rebalancer. We discuss this issue in more detail in Section 7.2.3.

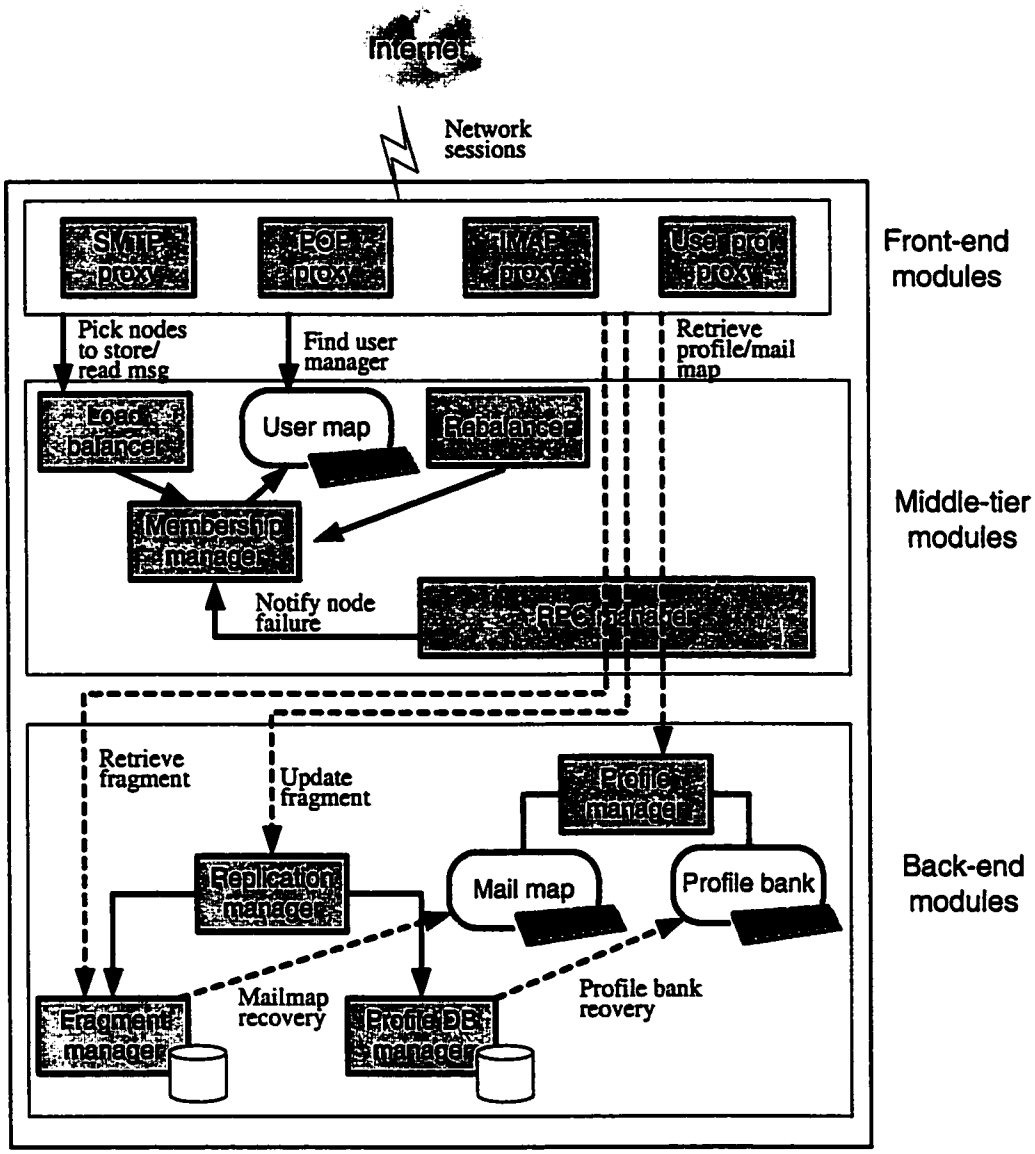


Figure 4.2. *The structure of an FHC node. Each node in Porcupine runs the same set of modules shown in this picture. A solid arrow shows a use relationship within a node, and a dotted arrow shows a use relationship across nodes via remote procedure calls. Data structures with the "memory-module" logo are in memory, and those with the "disk" logo are on disk.*

structure containing all of a user's mail. A mailbox in FHC is therefore a *logical* entity consisting of a user's fragments distributed and replicated across a number of nodes. Fragments are collected and presented to the user only when she requests so.

In addition to the on-disk data structures, each node manages the following data structures as soft state for tracking and distributing on-disk data. Being soft, they are stored only in nodes' memories and are reconstructed from hard state as nodes fail and recover. We discuss the soft state recovery protocols in Chapter 5.

Profile bank: Porcupine separates the storage and management of the user profile. A node that stores the profile of a user (i.e., the profile database) does not necessarily authenticate the user. Each user's entry in the profile database is copied as soft state on one live node in the cluster. We call this soft state the *profile bank*. This separation ensures that each user is managed by a live node and that the workload of user management is balanced. The mapping between users and nodes is determined by the user map.

Mail map: This map describes the nodes storing fragments for a given user[2]. The mail map for a user is managed by the same node that manages her profile bank. The mail map is also soft state. It is updated as fragments are created or deleted on remote nodes and as nodes crash or recover.

User map: The management of the profile bank and mail maps is distributed using the user map, a small in-memory table replicated on every node in the cluster. The user population is partitioned into a fixed number of equivalence classes (or *buckets*) using a hash function, known a priori to all nodes in the cluster. The user map translates a hash value to the node that manages the bucket. In other words, all the users whose login-names hash into a particular value are managed by the same node determined by the user map. The user map ensures that exactly one live node in the cluster manages a particular bucket; however, a node may manage multiple buckets, because the size of the user map is set to be many times larger than the

[2]In Porcupine, a user may own multiple mailboxes. A mail map actually describes the nodes storing a mailbox for a user. Thus, the profile manager may manage multiple mail maps for a particular user. Throughout this thesis, however, we identify a mailbox with a user to simplify the explanation.

size of the cluster. The contents of a user map are recomputed automatically as nodes fail or recover (Section 5.4).

The size of the user map is one of the few pre-defined constants that must be set by the administrator. Porcupine currently uses a 256-entry user map, but the range of reasonable values is wide in practice. It should be several times larger than the maximum possible cluster size so that the system can assign an even number of entries to each node and balance the CPU and memory load on nodes. On the other hand, too large a user map increases the cost of soft-state recovery, albeit in a minor way (Section 5.7).

Membership list: The membership list describes which nodes are live or dead to the best of the node's knowledge. The membership list is used for many purposes, for example, to update the user map (Section 5.4) and to distribute the incoming workloads (Section 7.1). The membership list is automatically kept identical across the nodes most of the time by the membership manager, although a node's arrival or departure may cause short-term inconsistencies. During a network partition, inconsistencies may last for a long time.

4.3.2 Data Structure Managers

The data structures introduced in the previous section are distributed and maintained on each node by several essential managers.

Profile database manager and fragment manager: These two managers maintain persistent storage. They allow client sessions to retrieve the profile database and fragments and the replication manager on the same node to update these objects.

Profile manager: This manager maintains the profile bank and mail maps for buckets designated by the user map. It authenticates users and provides accesses to mail maps on demand from the front-end proxies.

Membership manager: This manager maintains the node's view of the overall cluster state, including the membership list and the user map. It cooperates with the counterparts on other

nodes to detect node crashes and recoveries and to let the nodes agree on the new cluster state. Section 5.3 discusses the membership agreement in more detail.

Replication manager: This manager, running on each node, ensures the consistency of replicated on-disk objects, such as email messages and the user profile. Only updates are handled by the replication manager: retrieval requests are sent to the objects directly because of FHC's weak data consistency requirement. Chapter 6 discusses replication in more detail.

Load balancer: This manager picks the best set of nodes to store or read messages, using the information about the load and disk usage of other nodes. Chapter 7 discusses the load balancer in more detail.

Rebalancer: The rebalancer, running on each node during the night, restores the cluster's long-term health. It scans the mail maps managed on the node and coalesces or moves fragments that are split more than necessary. It also adds or deletes the replicas from fragments to maintain a proper level of availability for them. Chapter 7 discusses the rebalancer in more detail.

Remote procedure call manager: The remote procedure call (RPC) manager supports inter-node communication. It provides high-throughput, parallel remote procedure calls to any node in the cluster. Appendix A.2 describes Porcupine's RPC design in more detail.

Proxies: On top of the aforementioned managers, each node runs proxies that handle client sessions. Proxies are application-specific by nature. For an email service, proxies include the SMTP proxy for handling mail delivery, the POP and IMAP proxies for handling mail retrieval, and a user account proxy for managing the user profile. These proxies accept requests from remote clients, retrieve user profile information from the naming service, ask the load balancing service to decide the data placement, and call the replication service to update (or create) on-disk data.

The structure of the proxies is simple because they are stateless. All the state that must persist across sessions, e.g., email messages and passwords, is handled by the back-end modules and accessed by proxies via remote procedure calls.

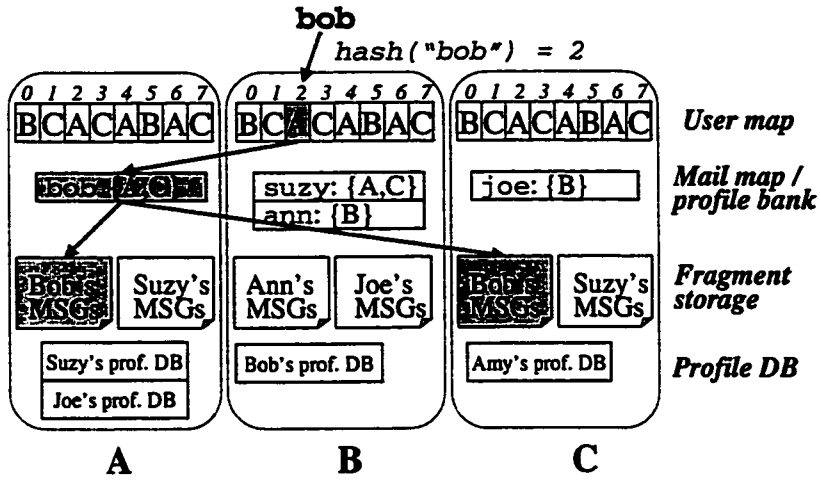


Figure 4.3. *This picture shows how a three-node cluster might distribute email messages for four users. The eight-entry-wide user map is replicated on each node. For example, a node learns that Bob is managed by node A, because the string "bob" hashes into 2, and the entry number two of the user map is A. To read Bob's messages, the POP or IMAP proxy consults the profile manager on A to obtain Bob's mail map (i.e., $\{A,C\}$) and contacts nodes A and C to read his messages.*

4.3.3 Data Distribution Example

Figure 4.3 shows a sample FHC configuration consisting of three nodes and four users. For simplicity, messages are not shown as replicated. The profile manager on node A maintains the buckets 2, 4, and 6, which include user Bob. To retrieve Bob's data, for example, the proxy (on any node) first contacts node A to retrieve Bob's profile and mail map. Then, the proxy contacts nodes A and C, because Bob's mail map contains the set $\{A,C\}$.

4.4 Common Operations

To illustrate the operation of FHC's internal components, we introduce the Porcupine email server in this section. In particular, we show how three common tasks in the email service — mail delivery, mail retrieval, and password change — are carried out in failure-free scenario. The behavior of the system after failure is discussed in Section 4.5.

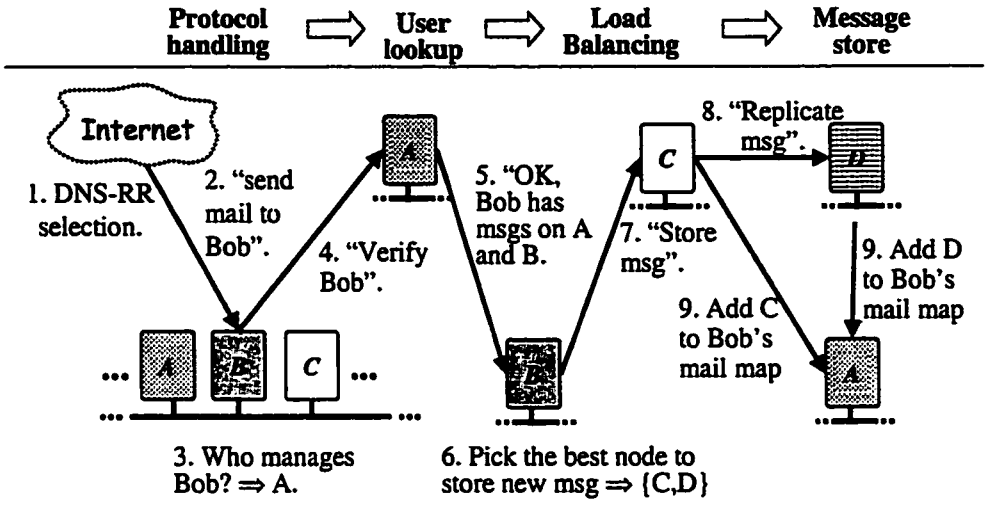


Figure 4.4. *The flow of control during message delivery. The remote server begins message delivery by picking a proxy node in the cluster (step 1) and starting an SMTP session (step 2). The proxy retrieves the profile and the mail map of the recipient from the profile manager (steps 3 and 4). The load balancer picks the set of nodes to replicate the message (step 6), and the proxy asks the replication manager on one of the storage nodes (C) to replicate the message (steps 7 and 8). Finally, the storage nodes (C and D) add themselves to the recipient's mail map, if these nodes have newly created the fragments (step 9).*

4.4.1 Porcupine Email Server from User Viewpoint

Porcupine is a mail transfer agent (MTA) that supports three email protocols and one protocol for managing user accounts. SMTP is used between MTAs to exchange email messages [124, 84]. POP and IMAP are used by mail user agents (MUA) to read and edit messages. POP is an older and simpler protocol that only supports listing, reading, and deleting messages [113]. IMAP is a newer protocol designed to support thin clients that keep messages on the server [39, 116]. In addition to the features found in POP, IMAP supports multiple mailboxes per user, mailbox editing, selective digest listing, searching, and other advanced commands. Porcupine also supports a proprietary protocol for adding, deleting, and changing the user profile.

4.4.2 *Mail Delivery*

Figure 4.4 shows the flow of control during mail delivery to a user managed in a Porcupine cluster. To send mail, a remote MTA first picks a node in the cluster using DNS round robin or an equivalent mechanism discussed in Section 4.2 (step 1). The MTA starts an SMTP session with the proxy on the designated node (step 2). The proxy's job is to interact with the MTA, obtain the recipient's name and the message body, and store the message on disks (possibly on other nodes). The first step for doing this is to obtain the profile of the recipient. The proxy applies the hash function on the recipient's name, looks up the user map (step 3), and asks the profile manager to retrieve the recipient's profile and mail map (steps 4 and 5). The proxy then passes the recipient's mail map to the load balancer and asks it to pick the best set of nodes to replicate the new message (step 6). If the mail map is empty or all choices are poor (for example, overloaded or out of disk space), the load balancer is free to select any other nodes, which actually happens in this example. The proxy then arbitrarily picks one of the nodes chosen for message storage (node *C*) and asks the replication manager on that node to coordinate message replication (step 7). Node *C* forwards the message to the peer replica node *D* (step 8). When these nodes create fragments (which is true in this example, because Bob had messages only on nodes *A* and *B* before), they ask the profile manager (node *A*) to add them to the user's mail map (step 9).

4.4.3 *Mail Retrieval*

Figure 4.5 shows the flow of control during mail retrieval. A remote mail user agent (MUA) picks a node in the cluster and initiates a POP or IMAP session (steps 1 and 2). The proxy on the contacted node authenticates the user through the profile manager and obtains the user's mail map (steps 3 to 5). The proxy then asks the fragment manager on each node in the mail map to retrieve the digest (message IDs, subjects, file locations, etc.) of the fragments (step 6), which is then presented to the MUA. In response to a message retrieval request from the MUA, the proxy searches the digest to find the node storing the message and asks the node to retrieve the message. In response to a message deletion request, the proxy forwards the request to the node storing the message (steps 7 and 8). When the last message in the fragment is deleted from the node, the node removes itself from the user's mail map (step 9).

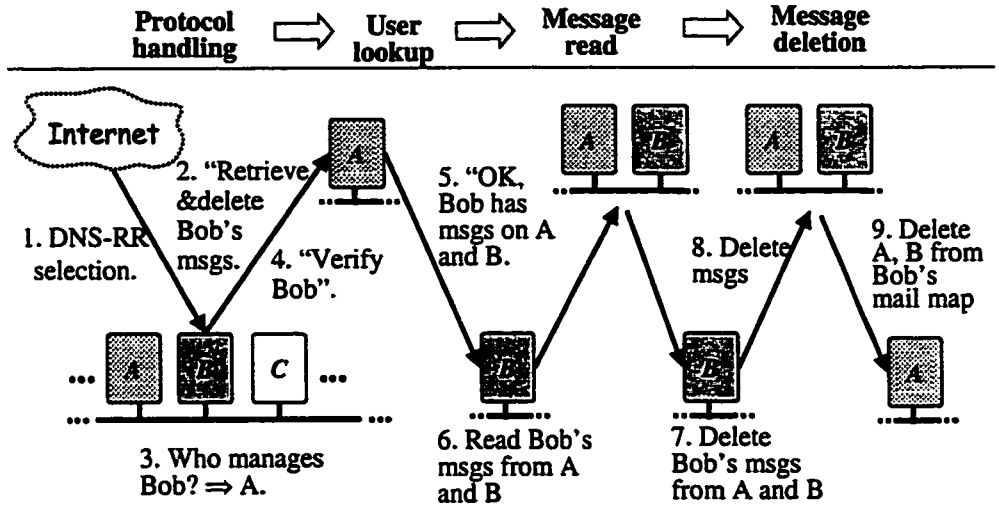


Figure 4.5. *The flow of control during message retrieval. The mail user agent picks an arbitrary proxy node in the cluster (step 1) and starts an IMAP or POP session (step 2). The proxy node retrieves the client's status, including the mail map, from the user manager (steps 3 and 4). The proxy reads and deletes the messages on the nodes in the mail map. After a node deletes the final message in a fragment, it asks the user manager to delete the node's entry from the client's mail map.*

4.4.4 Changing a Password

Figure 4.6 shows the flow of control during password change. First, an MUA or the account management software picks a proxy in the cluster and initiates a session. After authenticating the user through the profile manager, the proxy asks the replication manager on one of the nodes that replicate the user's profile database bucket to coordinate updating the profile. The coordinator runs the replication protocol to propagate the update to all other nodes that replicate the profile database. Finally, the profile database manager on these nodes upload the new profile to the profile manager so that the user can reference the new password in the future.

A bucket for the profile database is sometimes split into multiple groups (Section 4.3.1). In this case, the proxy keeps sending the update to each group until it finds the user. Database splitting is rare enough that this sequential search process does not in practice slow the system down (Section 7.2.2).

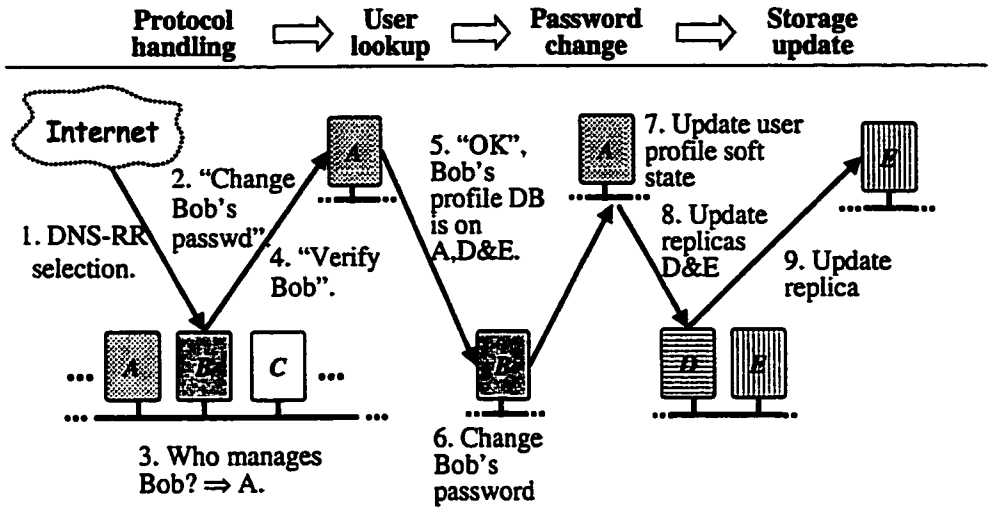


Figure 4.6. *The flow of control during password update. The MUA picks a proxy in the cluster (step 1) and starts a profile management session (step 2). The proxy node authenticates the user (steps 3 and 4) and learns the locations of the user's profile database (step 5). The proxy asks one of the profile database nodes to change the profile, and the node propagates the update to other replicas of the database.*

4.5 Reacting to Configuration Changes

The key goal of any cluster-based system is the *single-system image*, i.e., to hide the internal structure as much as possible and present the cluster as a single, large node. The previous sections described how FHC's components cooperate to provide the single-system image to clients in the common case without configuration change. This section describes the FHC's strategy for maintaining the single system image after a node fails or recovers. We define the types of failures the system must handle and overview the FHC's recovery strategy. Detailed discussions of the recovery mechanisms appear in Chapters 5 and 6.

4.5.1 Failure Assumptions

The challenge in building a fault-tolerant system is the diversity of failures the system experiences. The network may partition, packets may be delayed or lost, and a node may crash or thrash and become non-responsive. The diversity, severity, and the number of failures grow as cluster size

grows.

Because our goal is to build a scalable service from inexpensive hardware components, we do not pretend that we can distinguish or eliminate these failures. In this dissertation, we thus adopt the timed-asynchronous distributed system model [40], a failure assumption weak enough to be satisfied by all the computers and network media in use today. We assume that live nodes in the cluster behave correctly with bounded end-to-end response time *most of the time*, but we allow anything to happen in the short term. For example, in the short term, the network may drop packets or partition, and a node may thrash to the extent that it is falsely diagnosed to have crashed[3]. We hope that situations such as network partitioning and thrashing are rare. Our claim, however, is that unusual failures inevitably happen in large clusters. In fact, while developing Porcupine, we often experienced virtual network partitions and false failure detections, because the operating system's disk head scheduling sometimes delayed some tasks significantly. Any system unable to tolerate such failures becomes unmanageable, because these are the most difficult kinds of problems for humans to handle.

We also assume that nodes fail mostly independently (i.e., failure is a Poisson process). By assuming that mass failures are rare, the system can be made available by replicating data inside the cluster. Mass-failures are tolerated only by placing nodes in multiple geographical locations, which we do not assume at this moment. Geographically distributed clustering is our future work (Section 9.2).

Our goal in FHC is to build a system that keeps its data consistent regardless of the type or the severity of failures it may have experienced in the past. On the other hand, we allow the system to be in an arbitrary (sometimes useless) state while it is recovering from failure.

Byzantine failures [88] are excluded from our model. We assume that nodes and network links may stop or slow down, but otherwise they do not act arbitrarily or counterfeit network packets. Byzantine failures are excluded because they are prohibitively expensive to tolerate (e.g., at least $3f+1$ nodes are needed to tolerate f malicious nodes [27, 28]).

---

[3]We assume that the cluster is reliable "most of the time" only because no useful work can be done otherwise (i.e., this assumption ensures system's liveness). Making this assumption allows us to use, for example, RPC timeouts to detect a node crash, with hope that such a detection technique gives a right answer and the system can make progress most of the time.

4.5.2 *Overview of Recovery Mechanisms*

The data structures introduced in this chapter, including the user map, mail maps, email messages, and the user profile, are dynamic entities that need to be recomputed or recovered after change. FHC employs two different mechanisms for two different types of data it manages: soft state and hard state.

The soft state (the user map, the profile bank and mail maps) is recovered by re-calculating from hard state. When a node fails or recovers, the membership protocol first detects the change and lets all the live nodes agree on the new membership list. It then recalculates the new user map by replacing references to the dead nodes by live nodes, while making sure that each live node manages approximately the same number of entries for balanced load. The new user map ensures that each user is managed by a live node, but the profile bank and mail map of the users whose managers are reassigned are still empty. The profile bank and mail maps are recovered by letting *every* live node scan its disks and discover the profile and the fragments of all the users with management changes. The information about these users is pushed to their new managers to complete the soft-state recovery. The soft-state recovery mechanisms are discussed in more detail in Chapter 5.

The hard state (fragments and the profile database) is made available by replicating it on multiple nodes. A user can read or write an object as long as one replica is alive. When a node crashes, the replication managers running on other nodes remember updates issued for the objects stored on the crashed node. These updates are pushed to the node as soon as the node recovers. When the node remains failed for a long period (e.g., a week), the objects replicated on the node are re-created on other nodes to keep them available. A configuration change may induce the loss or the imbalance of the hard state. For example, a node's retirement causes replicas stored on the node to be lost. A node's recovery may cause some objects to be over-replicated. These problems are solved by the cooperation of the rebalancer (Chapter 7) and the replication service (Chapter 6) that adds or moves replicas of objects to restore the proper level of availability and load balance.

4.5.3 *Supporting Node Addition and Retirement*

The same set of mechanisms introduced in the previous section is used to support node addition and retirement. To add a node to an existing cluster, the administrator need only install and start

the server software on the node[4]. The membership manager will discover the node and assign user management tasks to it. The load balancing service will distribute data storage tasks to the node automatically. Retiring a node is also automatic. The administrator simply pulls the node off the cluster. The remaining nodes will remove the node from their user map, mail maps, and the replication data structures, and the cluster will behave as if the node had never existed in the first place.

4.5.4 Consistency Semantics

The recovery mechanisms described so far guarantee the *eventual consistency* of the data managed in the cluster.

The soft state recovery mechanism ensures that the combination of the user map, the profile bank, and mail maps can locate *all* the on-disk objects on live nodes, no matter what type or number of failures the system may have experienced in the past, provided that no new change happens for a long enough period[5].

The replication mechanism ensures that an on-disk object is readable and writable if at least one of its replicas is stored on a live node. The replication mechanism ensures, in most cases, that all replicas on the live nodes will agree on the newest contents among them, if no new change happens for a long enough period. Replicas may become permanently inconsistent when network partitioning persists for a very long period, but we believe that this situation is exceedingly rare (Section 6.7.3).

Thus, a particular user can always find her newest profile and fragments if: (1) at least one of the replicas of her profile is accessible, (2) at least one of the replicas of each of her fragments is accessible, and (3) no new configuration change happens for a long enough period. This is guaranteed no matter how many nodes are down, and no matter what type and number of failures the system has experienced in the past.

---

[4] Actually, the administrator often needs to update the border naming and routing services (e.g., the DNS server) and announce the addition of the node to the world. This task may not be necessary if the cluster is attached to the Internet via an automatic traffic redirector. See also Section 4.2.

[5] The "long enough period" mentioned above is the time needed to reconstruct the soft state and propagate outstanding updates among the replicas. Its exact length depends on the system configuration, but the soft state reconstruction takes about 10 seconds, whereas the update propagation takes minutes or hours, depending on the number of updates issued while a node is down.

4.6 Performance Analysis

This section analyzes the steady-state performance of FHC. We show that FHC will perform as well as conventional cluster architectures even when the cluster configuration and the incoming workloads are uniform, the situation in which FHC's dynamic architecture does not help improve its performance. The true benefits of dynamic data and function distribution become apparent under a non-uniform workload or system configuration. The behavior of FHC clusters under such environments will be studied in Chapter 8.

4.6.1 Performance Scalability

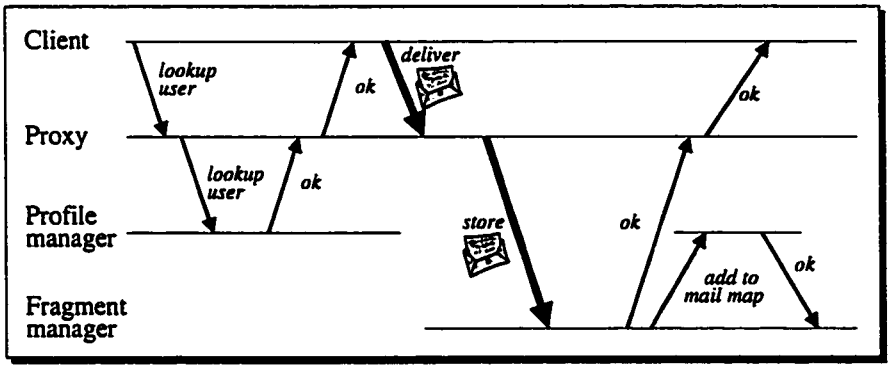
FHC is designed for performance scaling in the first place. Its load balancing service lets each node receive data-storage traffic according to its speed. The user map allows the user management tasks to be spread evenly among live nodes. The skew in user activity (e.g., some group of users receiving a huge amount of email messages, or the hash function not partitioning the user population evenly) may create an imbalance in user management workload distribution, but such an imbalance is absorbed by the load balancing mechanism, because system performance is disk-bound overall (Section 8.3.3). In addition, the workloads are highly parallel with little dependency between sessions. For these reasons, the cluster is able to increase its total throughput as its size grows. We measure the actual performance of Porcupine in Section 8.3 and prove our claim.

4.6.2 Overhead of Data Structure Management

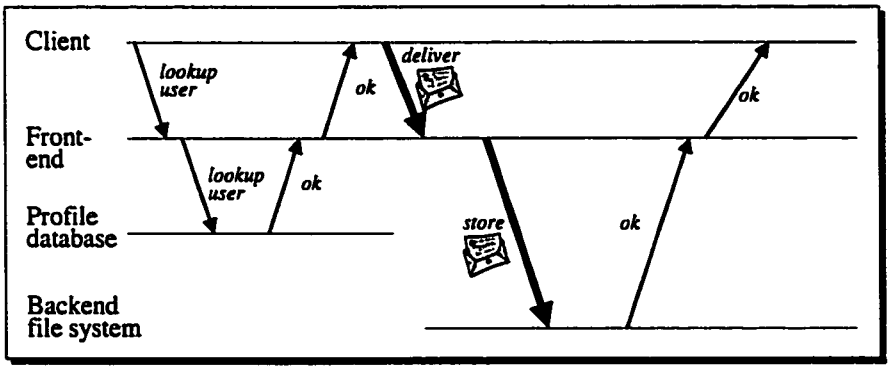
Let us look first at the networking overhead. Figure 4.7 shows the sequence of network messages exchanged to deliver an email message to a user in Porcupine (upper figure) and in a traditional, statically partitioned cluster (lower figure). The thick lines show large network messages that contain the email text, and the thin lines show smaller control messages. Overall, the two figures are remarkably similar, with the same number and type of messages being exchanged before the client receives the acknowledgement. The only difference is the additional messages Porcupine exchanges to update the recipient's mail map. These messages, however, do not delay the session, and their overhead is far smaller than the rest of the session that sends the actual email text twice.

Moreover, in most data-intensive services, neither the CPU nor the network is the major source

of overhead in data-intensive services. Disk I/O is what determines the system's throughput (Section 8.3.3). Regarding disk overhead, Porcupine and traditional clusters perform identically, because both store and read each email message the same number of times.



(a) Email delivery in Porcupine.



(b) Email delivery in a statically partitioned cluster.

Figure 4.7. *Comparison of email delivery processing between Porcupine and a statically partitioned cluster. The thick lines represent large network messages that contain the email message body, and thin lines represent small control messages. The only difference is the mail-map update performed asynchronously in Porcupine.*

4.7 Summary

This chapter introduced the major components of functionally homogeneous clusters. The cooperation of these components leads to a rich distribution of information in which data storage is decoupled from user management with little or no performance overhead. User profile management responsibility is assigned equally to live nodes using the user map, ensuring a continuous service to any user in the system. Storage of fragments is scheduled completely dynamically, allowing maximum throughput and availability for any configuration after any number of failures.

The common-case performance of FHC is comparable to other architectures, even when the system configuration and the workload are uniform and FHC's dynamic architecture does not help improve system performance.

The next three chapters discuss the challenges FHC faces and our detailed solutions to them. The foremost challenge is tolerating configuration changes, such as node crashes and additions. Chapter 5 discusses the recovery of soft state (the user map, mail maps, and the profile bank). Chapter 6 describes the replication protocol for storing on-disk data (email messages and the user profile) on multiple disks for availability. Chapter 7 discusses how FHC determines the placement of data items using load balancing services.

Chapter 5 Automatic Recovery

This chapter discusses FHC's strategies for restoring the soft state — the user map, the profile bank, and mail maps — after configuration change. We discuss the design and implementation of three proven-correct mechanisms developed for this purpose: membership agreement, user-map recomputation, and soft-state delivery. These mechanisms ensure that all the on-disk objects on live nodes are reliably located. We also analyze the cost of recovery and show that these mechanisms scale well, i.e., the cost averaged over time is mostly a constant regardless of cluster size.

The correctness proof of these mechanisms is deferred until Appendix E. This chapter describes the soft-state recovery only. The recovery of on-disk data (replication) is discussed in Chapter 6.

5.1 Goals of Soft-state Recovery

FHC's soft-state recovery mechanisms are designed with the following three high-level goals in mind:

Strong fault tolerance: The soft state maintains the system's name database, the sole gateway to on-disk data. The naming service must be extremely fault tolerant, being able to locate all on-disk objects on live nodes, regardless of the type or the number of failures it has experienced in the past (Section 4.5.1).

Scalability: The recovery process should finish in constant time with constant CPU and I/O costs, independent of cluster size. This goal, in particular, affects the recovery of mail maps and the profile bank. Because these two steps force nodes to scan their disks to discover on-disk data, they need to limit the amount of scanning to scale.

Graceful performance degradation and improvement: The recovery mechanisms must reconfigure to allow the system to run at a throughput proportional to its aggregate capacity. In other words, they must distribute the soft-state management responsibility evenly across nodes, regardless of the number or type of failures.

5.2 Overview of Soft-state Recovery Mechanisms

At a high level, soft-state recovery proceeds in the following four distributed and asynchronous steps.

1. When a node fails or recovers, another node notices the change and starts the membership protocol. The membership protocol lets nodes agree on the new membership list in a constant time, regardless of the number of failures or the previous state of the cluster.

2. The coordinator of the membership protocol computes the new user map as a side effect of the protocol. It removes dead nodes and adds newly recovered nodes to the user map, while assigning approximately the same number of buckets to each live node for a balanced load. The new user map is distributed to nodes along with the new membership list.

3. The profile managers with fresh bucket assignments fill their profile bank. To achieve this, upon receiving the new user map in the previous step, each node discovers the map entries with management changes by comparing the old and new maps. For each changed entry, the node scans its profile database and sends the contents to the new profile managers.

4. The profile managers with fresh bucket assignments also fill their mail maps. To achieve this, each node discovers the set of user map buckets with management changes and discovers in the local disks all the fragments that belong to these buckets. The names of the fragments are pushed to the new profile managers. This step runs in parallel with step 3.

This design achieves our two goals naturally. Because all information in the name database — the user map, profile bank, and mail maps — is computed or discovered from scratch from the hard state, the system can tolerate any type and number of failures. It scales well, as we see in Section 5.7: the per-node cost of recovery actually decreases with cluster size, because each node scans only a portion of its disks that corresponds to the user-map buckets with assignment changes. Finally, because the user map always assigns the same number of entries to each live node, system performance improves or degrades gracefully as nodes are added or removed from the cluster.

In the following sections, we first discuss the membership agreement. Section 5.4 then describes the user-map recomputation. Section 5.5 discusses the reconstruction of the profile bank and mail maps.

5.3 Membership Agreement

FHC's cluster membership manager provides the basic mechanism for tolerating change. It maintains the membership list, detects node failures and recoveries, notifies other services of changes in the system's membership, and distributes the new system state.

FHC's membership service is an extension of the three-round membership (TRM) protocol originally described in [41]. TRM is an efficient protocol that sends $O(1)$ messages in a failure-free situation, $O(N)$ messages during reconfiguration, and completes in a constant time (N is the number of nodes in the cluster). The next section describes the original TRM, and Section 5.3.2 describes the extensions we applied to the TRM to improve its robustness. A formal description of the membership protocol appears in Appendix C.1.

5.3.1 Three Round Membership Protocol

The TRM exchanges three rounds of messages to converge membership views of nodes. A sample run of the protocol is shown in Figure 5.1.

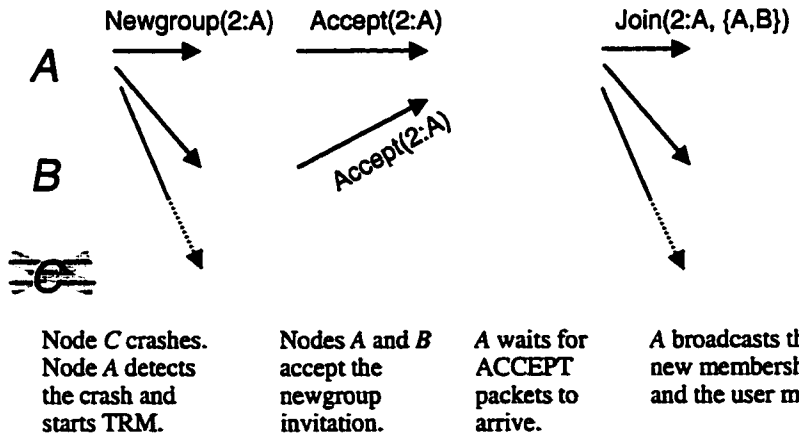


Figure 5.1. *A sample run of the membership protocol.*

1. The first round begins when any node detects a change (node crash or recovery) and elects itself to become the *coordinator*. The coordinator broadcasts a *NEW_GROUP(gid)* message, where *gid* is the group ID that uniquely identifies this reconfiguration attempt. In particular, it is the node's Lamport clock value [87] combined with the node's own name (IP address) to break ties. The significance of the use of Lamport clocks (and why, say, random bits cannot be used) will become clear when we discuss coordinator conflicts later in this section.

2. In the second round, all nodes that receive the *NEW_GROUP(gid)* message reply to the coordinator with the *ACCEPT(gid)* message. After a timeout period, the coordinator defines the new membership to be those nodes from which it received *ACCEPT(gid)* messages.

3. In the third round, the coordinator broadcasts the *JOIN(gid,list)* message to all nodes, with *list* being the new membership list computed in the previous step.

TRM handles network partitioning naturally by creating a separate group for each partition. Network partitioning is healed by periodic probing. Here, the coordinator periodically broadcasts *PROBE* packets once the membership is established[1]. When a coordinator receives a probe packet from a node not in its current membership list, it initiates the TRM protocol. Probing is also conveniently used to merge newly booted nodes in the cluster. A newly booted node acts as the coordinator for a group in which it is the only member. Its probe packets are sufficient to notify others in the network that it has recovered.

A node may discover the failure of another node in several ways. The first is through a timeout that occurs normally during part of a remote procedure call[2]. In addition, nodes within a membership form a virtual ring by connecting nodes in an ascending IP-address order (the largest IP address connects to the smallest IP address). Each node periodically sends to the next member in the ring an *ARE_YOU_ALIVE* message, to which the recipient replies with an *I_AM_ALIVE* message. If a node fails to receive *I_AM_ALIVE* from the next member in the ring, it initiates the TRM protocol.

[1]The node that sends probe packets need not be the coordinator that declared the newest membership. The former can be any node deterministically chosen from the membership list. In fact, the original TRM designates the node with the smallest IP address (among the current membership list) for this task. We chose our design merely because of its simplicity.

[2]Remote procedure calls are sent over TCP (Appendix A). The termination of the TCP channel also triggers TRM.

Two corner cases remain to be addressed. First, because failures are detected independently, two or more nodes may attempt to become coordinators at the same time. In such an event, the one proposing the largest group ID wins. More precisely, when node *A* receives from another node *B* a *NEW_GROUP(gid')* message for which *gid' < gid* (*gid* is largest group ID that node *A* has ever received), node *A* forces node *B* to resign by forwarding *NEW_GROUP(gid)* to *B*. Notice that the use of Lamport clocks is essential in avoiding livelocks during coordinator conflict resolution. The coordinator of an aborted reconfiguration attempt will receive the latest Lamport clock piggybacked on *NEW_GROUP*, and it will "win" in a new reconfiguration round it may start in the future.

The second corner case is when a coordinator crashes after it sends *NEW_GROUP* messages. To handle such a case, each node starts a timer after it receives *NEW_GROUP(gid)*. If the node fails to receive *JOIN(gid,list)* after a certain period, it presumes that the coordinator has crashed and starts the TRM protocol.

TRM runs in parallel with other tasks, such as client interaction and storage management. These tasks continue to use old membership information until the new membership is determined.

5.3.2 Improving the Robustness of the Membership Protocol

The TRM protocol is proven correct under the conditions defined in Section 4.5.1 [41]. We found one problem with it, however: at a large scale, the TRM may stress the network so much that, without any real hardware malfunction, it creates artificial network failures that force the protocol to restart. This problem is especially serious in the *ACCEPT* round, because hundreds of nodes send packets to a single coordinator simultaneously. Worse, this cycle may repeat perpetually, because the failures are caused by the protocol itself. We extended the TRM in two ways to improve its robustness at large scale. The effect of these improvements will be evaluated in Section 8.5.3.

Message Retransmission

In FHC, each type of request (*NEW_GROUP*, *JOIN*, and *ARE_YOU_ALIVE*) is sent multiple times (three in Porcupine) with a fixed interval in between (500ms in Porcupine), with a hope that nodes will receive at least one of them. The message retransmission, in particular, alleviates the coordinator flooding problem during the *ACCEPT* round. Here, each *NEW_GROUP* message now includes

the *acceptors* parameter containing the set of nodes from which the coordinator has received *ACCEPT(gid)* messages. A node responds to *NEW_GROUP* with *ACCEPT* only if the node is missing from *acceptors*. Thus, even if the coordinator drops *ACCEPT* messages from some nodes after the first *NEW_GROUP* message, the chance of it receiving the *ACCEPTs* will improve in the later resending of *NEW_GROUPs*, because the coordinator will receive replies from far fewer nodes.

Adding Random Delays

Coordinator flooding is alleviated further by letting nodes wait for a random period (0 to 200ms in Porcupine) before sending *ACCEPT* messages. This delay flattens the load of intermediate routers and the coordinator.

Each node also waits for a random period before becoming a coordinator. This reduces the chance of coordinator conflicts especially after a node failure, which many nodes are likely to detect simultaneously [57].

5.4 User Map Recomputation

The user map, replicated on each node, distributes the responsibility of profile management across live nodes in the cluster. Thus, whenever cluster membership changes, the system must reassign that management responsibility while satisfying the following three requirements:

Reliable change detection: The new user map must facilitate bootstrapping profile managers for buckets with assignment changes. Each node should be able to accurately determine the set of profile managers that needs to be bootstrapped by just comparing the old and new user maps. Alternatives, such as letting nodes negotiate on a peer-to-peer basis after receiving the new membership list, are complicated and not scalable.

Minimal changes: The new user map should differ minimally from the old user map, because the cost of soft-state recovery grows proportionally with the number of changes in the old and new user maps (Section 5.7).

Balanced load: The user map should always distribute the profile management responsibility evenly

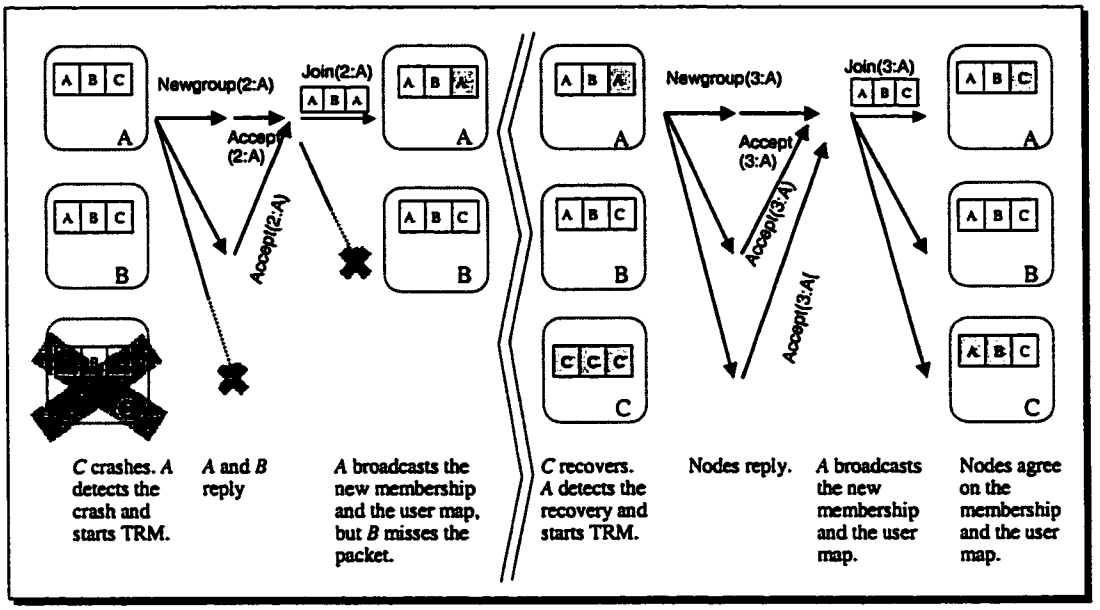


Figure 5.2. *A scenario that results in a permanently corrupt soft state. Here, node C crashes in the beginning, and node B misses all the JOIN messages sent by the coordinator (node A). Later, node C recovers. Here, let us reasonably assume that membership coordinator A creates a user map identical to the initial one. Upon receiving JOIN from A, node B wrongly concludes that the last entry in the user map remains unchanged, when in fact node C has rebooted, and C must be considered as the new profile manager.*

to live nodes. Without knowledge of how active the users in each bucket are, this means that the system should assign approximately an equal number of buckets to each live node.

The last two goals are achieved by centralizing the user map recomputation. The user map is recomputed by the membership coordinator as a side effect of the membership protocol. An alternative would be to let nodes independently and deterministically compute the new user map from the new membership list (e.g., consistent hashing [82]), but that fails to satisfy our first goal, as we discuss next.

Achieving the first goal requires more than just comparing the names of managers (i.e., IP addresses) in the old and new user maps bucket-wise, because such a strategy sometimes causes false-negative change detection, especially when nodes miss membership *JOIN* messages. Figure 5.2 demonstrates a problematic scenario.

We solve this problem by checking not just who the profile manager is for each bucket, but also *when* the manager first undertook its responsibility. In particular, we associate with each user map entry an *epoch ID*, which shows the ID of the first group (Section 5.3.1) that assigned the bucket management responsibility to the profile manager (Figure 5.3). If, on node X, both the manager ID (say, node Y) and the epoch ID match for a particular bucket between the old and new user maps, then node Y has continuously managed that bucket since X received the old user map. Thus, X need not send the profile bank and mail maps for that bucket to Y, provided that X pushed the soft state to Y when (or before) it received the old user map.

The actual user map reconstruction protocol proceeds as follows. Appendix C.2 shows a more formal specification, while Appendix E.2 proves that the protocol always maintains a soft state consistent with the on-disk data. A sample run of the protocol is shown in Section 5.5.3.

1. After receiving a *NEW_GROUP* message, each node examines its current user map and piggybacks the entries it manages (the bucket numbers and the epoch IDs) on the *ACCEPT* message.

2. The coordinator remembers the set of tuples ⟨*nodeid*, *bucket-number*, *epoch-id*⟩ as it receives the *ACCEPT* messages.

3. After the second round, the coordinator computes the new user map as follows. Keep in mind that "the most- (least-) loaded node" is the node that manages the most (fewest) entries in the user map, with ties broken arbitrarily.

   The coordinator first removes failed nodes from the old user map and assigns the least-loaded node(s) to the slots vacated by the failed nodes. Next, the coordinator picks an arbitrary entry managed by the most-loaded node and reassigns the entry to the least-loaded node. This reassignment procedure is repeated until the difference between the numbers of buckets managed by the most- and least-loaded nodes is two or less. The second step is needed to give nodes their fair share of user management responsibility after a node recovers or a network partition heals. This scheme minimizes changes to the user map while assigning approximately the same number of buckets to each live node.

For each bucket with an assignment change, the coordinator sets its epoch ID to be the ID of the new group (Section 5.3.1). For other buckets, the coordinator reuses the epoch IDs obtained in step 2.

4. The new user map is piggybacked on the *JOIN* message and is distributed to all nodes.

```
type UserMap = array [0 .. USER_MAP_SIZE-1] of record
  manager: NodeID;
  epoch: GroupID;
end
```

Figure 5.3. *Definition of the user map. The "manager" field shows the node ID of the bucket's profile manager. The "epoch" field shows the first time the node became the profile manager. USER_MAP_SIZE is a constant chosen by the administrator (256 in the Porcupine).*

5.5 Reconstructing the Profile Bank and Mail Maps

When a node newly assumes management of a bucket in the user map, it needs to learn about the profile and the mail maps for users in the bucket. The reconstruction of these data structures starts as soon as the node receives the new user map. This section describes a simple protocol for deciding which buckets are to be pushed, as well as several techniques for improving the protocol's parallelism.

5.5.1 Reconstructing the Profile Bank

FHC divides the profile database of user population into a fixed number of hash buckets. Each bucket, in turn, is divided into a few (usually one) groups, each of which is replicated on several nodes for availability (Section 4.3.1).

Upon receiving a new user map, the node compares the old and new user maps bucket-wise. For each bucket with an assignment change (i.e., either the manager ID or the epoch ID has changed), nodes storing a group of the bucket push the group's contents to the new profile manager.

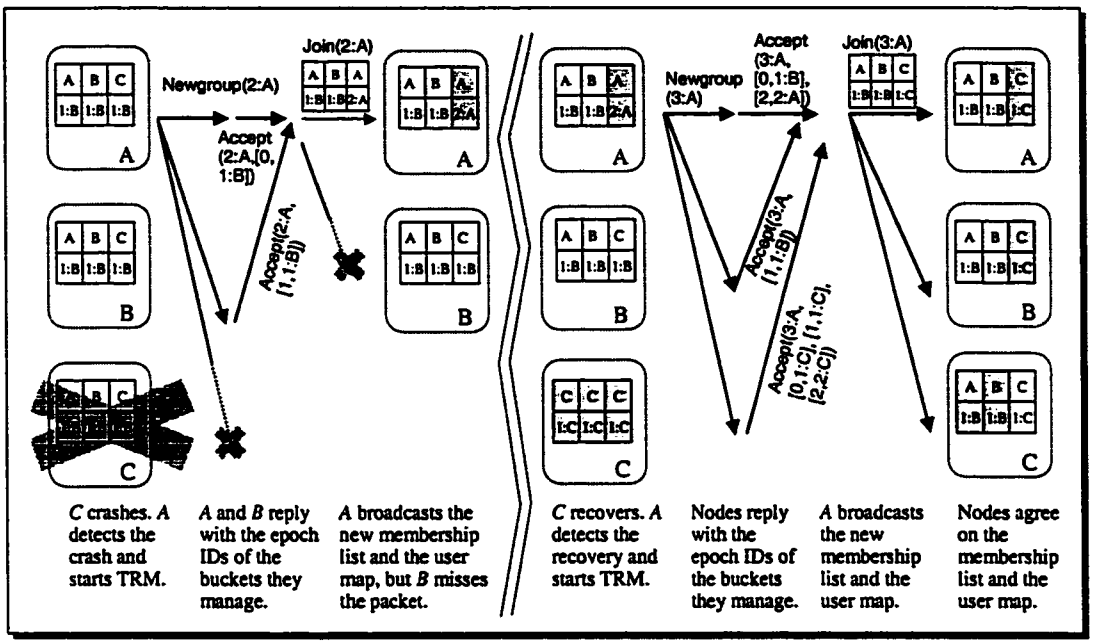


Figure 5.4. *An example of soft state recovery. In this example, node C crashes and then recovers. Node B fails to receive the membership renewal after C's crash. Arrows show messages exchanged among the nodes. Rectangular boxes are the user maps sent as a part of membership messages. Upper boxes in the user map show the manager nodes, and lower boxes show the epoch IDs. Shaded boxes are entries that nodes recognize as being changed.*

5.5.2 Reconstructing Mail Maps

The profile database stores only "hard" information about users — login names, passwords, full names, etc. Thus, the mail maps of the users, being soft, must be reconstructed separately. The mail map reconstruction procedure resembles that of the profile reconstruction. For each bucket with a new assignment, a node discovers all the local fragments for the users in the bucket and sends the names of the users to the new manager. The new manager in turn adds the node's name to the mail maps of these users. In Section 5.7, we show that this seemingly inefficient procedure actually is inexpensive and scalable.

5.5.3 Reconfiguration Example

Figure 5.4 shows an example of membership reconfiguration under the scenario that appeared in Figure 5.2. First, node *C* crashes. The new membership list and the new user map are computed on

the coordinator node *A*, but the *JOIN* messages that contain them fail to reach node *B*. Later, node *C* recovers, and node *B* receives the new user map, which happens to be identical to the initial one except for the epoch ID of the last bucket. The change in the epoch ID allows node *B* to detect *C*'s reboot. Node *B* subsequently searches its disk for fragments in the last bucket and pushes the names of users that own the fragments to *C*. At the same time, if node *B* stores (a group in) the profile database for the last bucket, it also pushes the database contents to *C*. Node *A* does the same for the last bucket, and node *C* does the same for the first three buckets.

Notice that different nodes scan different sets of buckets in this example. In this recovery mechanism, each node scans the minimal set of buckets to restore the global consistency of the profile bank and mail maps. This is one of the reasons our algorithm scales to a large cluster.

5.5.4 Implications of Asynchronous Soft-state Reconstruction

The profile bank and mail maps are reconstructed in parallel with no explicit synchronization between them. Thus, a profile manager may receive mail maps for users before receiving their profile. The profile for such users is marked "void" and is hidden from clients until their profile arrives. Sometimes, a user's profile may never reach the profile manager, e.g., when her account was deleted from the system but somehow her mailboxes remain in the cluster[3]. The rebalancer will eventually delete such orphan messages (Section 7.2).

Sometimes, a new profile manager may receive a user's profile or a mail map entry from another node before it receives the *JOIN* message that caused the node to push the soft state, since causal ordering among messages is not necessarily preserved. To address this issue, each node tags the packet containing the profile bank or mail maps with its current group ID. The profile manager accepts the packet only if its current group ID matches the one in the packet. Otherwise, the manager asks the sender to resend the soft state after some wait.

---

[3]If a node that stores a user's fragment fails, the fragment becomes invisible to the rest of the system. If the user's account is deleted while the node is down and later the node recovers, the fragment will be recovered on the profile manager without a corresponding profile.

5.5.5 *Parallelizing Updates to Mail Maps and the Profile Bank*

The profile bank and mail maps need to be kept consistent with fragments and the profile database stored on other nodes, while allowing concurrent accesses for better system throughput. This section describes techniques for parallelizing soft-state updates.

Mail maps are modified in two ways. The first is when fragments are created or deleted during the normal course of operation (e.g., when an email message is delivered). The second is by the mail map recovery algorithm after a node failure or recovery. For a mail map to locate all fragments for the user, an access to a fragment and to the corresponding mail map must be logically atomic. Consider the following sequence of events, running on a single node:

1. POP session *A* deletes a user's fragment.
2. SMTP session *B* creates the user's fragment.
3. Session *B* adds an entry to the user's mail map (on the remote profile manager).
4. Session *A* deletes the entry from the user's mail map (on the remote profile manager).

This user will not be able to read the message delivered by session *B* in this example. The problem here is that step 4 must be performed before step 3. A naive solution, such as locking a fragment until the mail map is updated, lowers the system's parallelism and increases session latency. Fortunately, we have a better solution that uses timestamp-based concurrency control (i.e., the Thomas write rule) [151, 12].

In our design, a mail map is actually a table that partially maps a node ID to a wall-clock[4] time that shows when the fragment was last accessed on that node. We assume that the clock resolution is fine enough to distinguish between any two events that happen on a node (Appendix B).

After a node creates or deletes a fragment belonging to a user, it sends the node's current wall-clock value to the profile manager to update the user's mail map. The mail map recovery protocol (Section 5.5.2) also sends the node's wall-clock value along with the list of user names.

When adding a node to the mail map, the profile manager sets the entry's timestamp to the wall-clock value supplied by the node. If the entry already exists, then the timestamp is set to the *newer*

[4] "Wall-clock" is a computer-science jargon that actually means the computer's quartz clock that tells the time and date.

of the current value and the caller-supplied value. On the other hand, the profile manager deletes a mail-map entry only when the entry's timestamp is older than the one provided by the caller. This allows the update in step 4 of the previous example to be ignored silently, ensuring that the user can read the message created in step 2. This algorithm is described more formally in Appendix C.3.

This technique sometimes causes update-delete conflicts [46]. Suppose a session with wall-clock value wc_n deletes a mail-map entry, and an older session on the same node with wall-clock value $wc_o (< wc_n)$ later tries to add an entry. In this case, the older update is wrongly applied, and a dangling link is created. Dangling links, however, cause no harm. A node will delete itself from the mail map when it receives a retrieval request for a non-existent fragment.

The profile bank is also updated on two occasions: (1) by front-end proxy sessions updating the profile (Section 4.4.4), and (2) by the profile recovery mechanism. Updates to the profile bank are parallelized using exactly the same technique as above.

5.6 End-to-end Effects of Failures

When a node fails, all sessions hosted by proxies on the node abort — an unavoidable result given the difficulty of TCP session fail-over. Aborting these sessions is no different from abortions due to wide-area network problems, which are far more common than server failures [29, 165]. The effect of a session abortion depends on the nature of the service. For example, in services such as SMTP, the client reconnects to another node in the cluster automatically. The abortion may cause a potential anomalous outcome on the recipient side: if the failure occurs after the message has been written to a disk but before any acknowledgment has been sent to the client, the recipient may read the same message more than once. Duplicate message delivery, however, is not a major problem in data-intensive Internet services (Section 1.2), and we believe that this problem is a reasonable price to pay for service that is continually available. For other services, e.g., POP, IMAP, and NNTP, the users must reconnect to the cluster manually.

The crash of a node acting as an on-disk data manager (Section 4.3.2) is handled similarly. For example, if a candidate fragment manager fails during email delivery in an email service, the proxy will abort, forcing the remote client to retry later. This action may cause duplicate message delivery for the same reason as above. If a fragment manager crashes during data retrieval (e.g., during a POP

session), the proxy will transparently retrieve the data from another replica for the same fragment. If all replicas for the fragment are down, then an error will inevitably be returned to the client, but the session itself continues retrieving other fragments.

5.7 Cost of Recovery

This section analyzes the cost of the recovery mechanisms and shows that these mechanisms can scale to large clusters. As we will show, three main determinants of the cost are the size of the cluster (N), the number of entries in the user map (U), and the per-node mean time between failures ($MTBF$). The per-node, time-averaged cost of membership agreement and user map recomputation grows at $O(N/MTBF)$, but its magnitude is negligibly small. The per-node, time-averaged cost of mail map and user profile reconstruction grows at $O(U/MTBF)$, and its magnitude is also negligibly small. Section 8.5.1 studies the actual behavior of Porcupine's recovery mechanisms.

5.7.1 Cost of Membership Agreement

FHC's membership protocol sends R broadcast packets each for *NEW_GROUP* and *JOIN*, and N packets for *ACCEPT*, where N is the number of live nodes in the cluster and R is the number of times each type of packet is broadcast ($R = 3$ in Porcupine, as shown in Section 5.3.2). In total, $2NR+N$ packets are received by each node per reconfiguration. On the other hand, the frequency of failures increases at the rate of $N/MTBF$ (assuming independent failures). Thus, the number of packets received by all nodes in the cluster averaged over time will be:

$$C_{membership} = (2R+1) * N^2/MTBF.$$

While this cost grows at $O(N^2)$, it is practically negligible. Let us make the following pessimistic set of assumptions and calculate the cost of membership agreement.

- $MTBF$ = 1 day per node.
- $R = 3$.
- The mean size of membership packets is 4200 bytes.

Under this assumption, in a 1024-node cluster connected by a 1Gb/s network, the nodes receive $7 * 1024^2/(24 * 3600) \approx 81$ packets per second, or 0.27% of the available network bandwidth. In a 128-node cluster connected by a 100Mb/s network, the network handles about 1.33 packets per second, or 0.036% of the available network bandwidth.

5.7.2 Cost of Mail Map and Profile Bank Reconstruction

The per-node cost of reconstructing mail maps and the profile, averaged over time, is independent of cluster size. The reason is twofold. First, as we will show, the cost of reconstruction per node per failure is inversely proportional to cluster size. Second, the frequency of reconfiguration increases with cluster size. These two factors cancel each other out to make the reconfiguration cost independent of cluster size.

Why does the reconstruction cost per node per failure decrease with cluster size? We consider only mail map reconstruction, but the same argument can be applied to user profile recovery as well. Let us assume for the moment that the cost of mail map reconstruction per node per failure is proportional to the number of reassignments to the user map. The number of user map reassignments per failure is inversely proportional to N (i.e., cluster size). To understand why, consider what happens when a node crashes. Because the user map recomputation algorithm evenly distributes nodes to buckets (Section 5.4), the node should have managed $B_c \approx U/N$ buckets in the old user map (U is the number of entries in the user map, and N is the number of nodes in the cluster). When a node recovers, it will newly manage $B_r \approx U/(N+1)$ buckets in the new user map. B_c or B_r is the number of entries reassigned in the new user map. The cost of recovery per node per failure thus becomes inversely proportional to N.

Now, let us examine our assumption — the cost of mail map reconstruction is proportional to the number of fragments to be discovered — more closely and derive the actual cost of mail map reconstruction. We will see that the assumption is mostly valid, although it depends on the size of the user map in a minor way.

We store all the fragments for a particular user-map bucket in a single directory to discovery their names quickly (Appendix A.3.1). For example, fragments in bucket 0 are stored in the directory "`/spool/0/`", those in bucket 1 are stored in the directory "`/spool/1/`", and so on. Thus,

discovering the names of fragments in a particular bucket becomes a single directory scan, which is mostly a sequential disk read. By pessimistically assuming that the operating system kernel lacks a buffer cache[5], the cost of scanning a single bucket directory, $C_{bucketscan}$ can be approximated as below, with the measured values of the variables shown in Table 5.1.

$$\begin{aligned} C_{bucketscan} &= C_{startup} + C_{seq} * NR\_FRAGS\_IN\_BUCKET/E. \\ NR\_FRAGS\_IN\_BUCKET &= NR\_FRAGS/U. \end{aligned}$$

Table 5.1. Meaning and values of variables. The values are measured on a PC with Pentium II 300 MHz and a fast SCSI disk.

Variable	Value	Description
$C_{startup}$	14.3ms	The average cost of initiating a disk read
C_{seq}	0.0957ms	The marginal cost of reading a block (1KB) from a disk
E	40	The average number of fragment names that fit in a block
U	variable	The size of the user map
NR_FRAGS	variable	The total number of fragments in a node

The total cost of mail-map recovery per node averaged over time, $C_{mailmap}$, is $C_{bucketscan}$ multiplied by the number of buckets to be scanned, i.e.:

$$\begin{aligned} C_{mailmap} &= C_{bucketscan} * U/(N * MTBF) \\ &= (C_{startup} + C_{seq} * NR\_FRAGS\_IN\_BUCKET/E) * (U/(N * MTBF)) \\ &= (C_{startup} * U + C_{seq} * TOTAL\_FRAGS/E)/(N * MTBF). \end{aligned}$$

Thus, the cost of mail map recovery per node per failure decreases with cluster size N (which is

[5]In reality, the kernel often fully caches the spool directories, since they are accessed very frequently.

expected) and slowly increases with the user map size U, assuming that the number of fragments on a node remains constant.

Figure 5.5 summarizes the recovery cost as a function of the user map size, assuming *TOTAL_FRAGS* = 20 million and $N = 128$. Figure 5.6 plots the recovery cost for various per-node MTBF and user map sizes. These graphs show that the recovery cost is very small — each node spends less than 3% of the time discovering mail maps even when each node fails every hour — and that the size of the user map has only a minor impact on the cost of recovery.

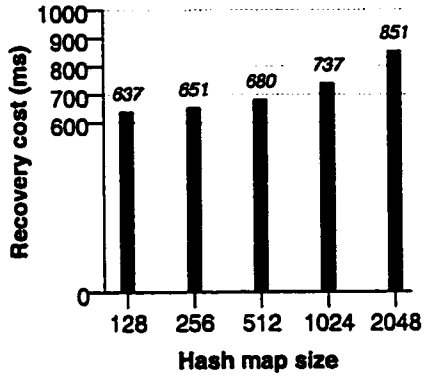


Figure 5.5. *The recovery cost (total disk I/O time) per node per failure for various user map sizes.*

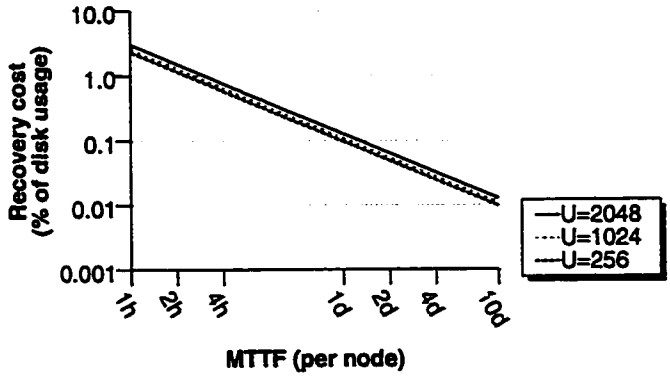


Figure 5.6. *Estimated recovery cost for various user map sizes. The Y axis shows the percentage of time each node spends in mail map recovery.*

5.8 Summary

FHC's soft-state recovery mechanisms ensure that the system's service is always available for any user after any number or type of failures. Three provenly correct protocols are developed for this purpose: membership agreement, user map recomputation, and reliable soft-state delivery. Client activities are affected minimally by failures, and they continue reading or updating messages on live nodes.

The cost of recovery is dominated by the cost of the disk I/Os performed to discover the profile database and fragments. The cost grows slowly with the size of the user map, but it is otherwise independent of cluster size. Overall, each node spends less than 3% of the time reconfiguring, even when it fails every hour, proving that our naming service with automatic recovery is a practical mechanism for building a large, scalable service in an unreliable environment.

Chapter 6 Replication

FHC replicates on-disk objects, such as the user profile and email messages, to improve their availability. Our replication service is efficient, dynamic, and fault tolerant. It allows each object to be replicated on its own set of nodes (called the *replica set* of the object) so as to support fine-grain, dynamic data distribution. It can gracefully handle most failure types, including multiple node failures, network partitions, and sudden node retirements, while guaranteeing eventual consistency of the replicas. Such versatility and performance are achieved by defining semantics tuned to FHC's service requirements.

This chapter discusses the design, implementation, and use of the replication algorithm. A formal presentation of the algorithm is deferred until Appendix D. The correctness proof of the algorithm appears in Appendix F. In particular, the proof focuses on how the system is able to propagate updates when replicas are added or removed concurrently.

6.1 Goals of Replication

Being the key component of FHC, our replication service needs to handle dynamic distribution of data for better manageability, availability, and performance. Below, we list the requirements for our algorithm and reasons for the development of a new algorithm.

Strong fault tolerance: Large clusters inevitably experience unusual types of failures (Section 4.5.1). Our algorithm must ensure the consistency of replicas against such failures as network partitioning, multiple node failures, and sudden node retirement. It must also be *non-blocking*; that is, it must allow reading of and writing to any replica at any time regardless of whether other replicas are available or not.

Fine-grain replication: Many traditional algorithms replicate objects at a very large unit, such as the entire disk or the entire database table. Such a strategy is inapplicable to FHC, which distributes incoming data items dynamically at a fine grain. Instead, our algorithm must allow

each small object (e.g., the average size of an email message is 4.7 kilobytes) to be stored in its own set of replicas and give the system maximum freedom of data placement.

Dynamic replica-set changes: Large clusters experience frequent configuration changes. Replicas often need to be added, deleted, or moved to react to node addition or retirement. (This feature is used, for example, by the rebalancer; see also Section 7.2.) Our replication service should support such operations without taking the system off-line.

Efficiency: Because of fine-grain replication, the system must manage billions of small objects. Thus, the algorithm should consume few computational and disk resources for each object. It also should be *quiescent*; that is, it should incur no computational overhead when no update is in progress for an object.

Quick object deletion: Traditional replication algorithms assume that objects are long-lived. Deleting objects often requires an off-line intervention [109, 110, 147] or keeping residual records indefinitely [46, 103]. This is in contrast with FHC's data (e.g., email messages and the user profile), which are created and deleted frequently. The algorithm needs to delete objects quickly without leaving any residual information behind.

The key strategy we deploy to achieve these goals is the exploitation of application semantics. The replication algorithm propagates updates among replicas without a global lock, allowing the algorithm to be extremely fault tolerant. This design may cause a short-term data inconsistency, but this problem has little effect on the system's quality of service, because the possibility of inconsistent data is already prevalent in the environment (Section 1.2). As another example, when multiple updates are issued simultaneously for the same object, we simply overwrite the object entirely with the contents of the newest update, relying on the fact that updates are rarely issued concurrently and that the objects (email messages, the user profile, etc.) are not subject to complex manipulations.

6.2 Overview of the Replication Algorithm

FHC's replication algorithm is based on four principles: *state transfer*, conflict resolution using the *Thomas write rule*, update retirement using *synchronized clocks*, and *unified treatment of contents*

and replica-set updates.

In its basic form, our algorithm resembles those used in systems such as Active Directory [103], Usenet [137], and Xerox Clearinghouse [46]. Any replica (or any node for a newly created object) can issue an update at any time. A *coordinator*, usually the issuer of the update, propagates the update by pushing the object's new contents to others. Since objects are usually small and are modified in their entirety (most common operations for email messages are "store" and "delete"), whole-state transfer is a reasonable restriction that simplifies update propagation and replica reconciliation, while also keeping overheads low. In Section 6.6.3, we discuss a technique for reducing the overhead of whole state transfer in some situations.

Because updates are issued independently, more than one update may be issued for an object on different nodes at the same time. Such conflicting updates are resolved by using the Thomas write rule [151], that is, by attaching loosely synchronized wall-clocks to the updates and accepting and applying only the newest update. This rule lets all the replicas' contents converge on the globally newest contents.

Loosely synchronized clocks are also used to retire updates [95]. After the coordinator completes update propagation, it sends out *retirement notices* to the replicas. Upon receiving a retirement notice, the node deletes the update from disk after waiting for a fixed period. This wait ensures that the node rejects stale updates that may arrive out of order.

The unique aspect of our algorithm is its uniform handling of replica-set updates (i.e., replica addition or deletion) and contents updates. In fact, an update is actually a tuple consisting of the new object contents and the new replica set. For an update that changes the replica set, the coordinator pushes the update to the union of the old and new replica sets (called the *targets* of the update). A node receiving the update modifies, creates, or deletes a replica depending on whether it appears in the new replica set. The Thomas write rule is again used to resolve conflicts among replica set changes: the older updates are canceled by forwarding the newest update to their target nodes and letting them be overwritten.

This combination of state transfer, the Thomas write rule, quick update retirement, and unified treatment of updates allows our algorithm to meet our goals effectively, as follows:

- Our algorithm's basic design directly achieves the five goals — non-blocking access, fine-

grain replication, dynamic replica-set changes, eventual consistency, and computational efficiency.

- Strong fault tolerance is achieved by decentralizing the algorithm. The algorithm lets any node take over the task of the coordinator at any time, leaving no single node permanently responsible for maintaining an object's replica consistency. Decentralization improves availability, since updates need not await the revival of the node that coordinates updates. It also eliminates the requirement that failure detection be precise, since there need not be agreement on which is the primary node, thereby allowing the system to use inexpensive (and unreliable) hardware components. Moreover, as we discuss in Section 6.7.3, our replica-set update mechanism lets all the replicas quickly learn about added (or removed) replicas, even when multiple updates are issued concurrently.

- Our algorithm's space overhead is small for three reasons. First, the size of the update record, maintained during propagation, is very small because it omits new object contents (the contents are read from the replica directly). Second, the system never keeps more than one update for a single object, because it filters out conflicting updates using the Thomas write rule. Finally, our algorithm quickly reclaims the space occupied by update records by retiring them as soon as they finish propagation.

6.3 Use of Replication in FHC

The general unit of replication is the *object*, which is simply an opaque piece of data that corresponds to a single email message, a user's profile, or whatever the client of the replication service means. Each object is named by the *object ID*, which is also an opaque string. For example, for email messages, object contents are their text, and object IDs are of the form ⟨*type*, *user*, *msgid*⟩, where *type* is the type of object (mail message), *user* is the owner of the message, and *msgid* is a unique identifier found in the mail header ("`Message-ID:`" field). For the users' profile, objects are the binary representation of the account information (login names, passwords, forwarding addresses, etc.), and object IDs are the users' login names.

6.3.1 The Replication Manager

The consistency of replicas is maintained by the *replication manager* running on each node (Section 4.3.2). This manager has a limited role. Being optimistic, it does not intervene on object retrieval requests. The clients, e.g., proxies, are free to pick any replicas and read them directly. It also does not manipulate objects directly. On-disk data managers define the format of objects and their IDs, along with procedures for reading and writing on-disk replicas (Section 4.3.2). Finally, it does not define the policy regarding when and where replicas are created. The load balancer and the rebalancer define such policies (Chapter 7). Thus, the replication manager is responsible only for accepting object-update requests from clients, cooperating with its counterparts on other nodes, and ensuring the consistency of replica contents by calling the on-disk data managers.

The replication manager is used mainly by proxies on behalf of users, e.g., by an SMTP proxy trying to store a message or by an IMAP proxy trying to delete a message[1]. For updating an object, it exports a single procedure with three parameters: the object ID, the new object contents, and the new replica set (Appendix D.2). For example, to create a new replicated object (as would occur with the delivery of a mail message), a proxy generates object contents (from the email client), an object ID (by extracting the recipient name and `Message-ID:`), and the replica set (by asking the load balancer) and passes them to the replication manager.

6.3.2 Locating Replicated Fragments

When fragments are replicated, the user's mail map reflects the set of nodes on which each fragment is replicated. For example, if Alice has two mailbox fragments, one replicated on nodes *A* and *B* and another replicated on nodes *B* and *C*, Alice's mail map will be $\{\{A,B\},\{B,C\}\}$. To read a fragment, the retrieval proxy contacts the least-loaded node for each replicated fragment to obtain the complete contents of the user's mailbox (e.g., for Alice, nodes A and C). Notice that the definition of fragment changes slightly when data are replicated (Section 3). Now, a fragment is the set of messages stored on a single node *and* replicated on the same set of nodes.

---

[1]The replication manager is also used internally, e.g., by the rebalancer to move replicas (Section 7.2).

6.4 Description of the Replication Algorithm

This section presents only the gist of the algorithm. A more formal presentation of the algorithm appears in Appendix D.

6.4.1 System Model and Assumptions

We adopt the failure model introduced in Section 4.5.1: nodes may crash or slow down and the network may delay or drop packets, but otherwise they behave correctly with a bounded response time most of the time[2].

In addition, we assume that nodes in the cluster have loosely synchronized wall-clocks. Wall-clocks are used for two purposes: update serialization (Section 6.4.4) and retirement (Section 6.4.3) [94]. As such, the maximum clock skew among nodes should be less than the inter-arrival time of externally initiated, order-dependent operations (e.g., password update). In practice, modern clock synchronization protocols easily keep clock skew in the order of tens of microseconds [105, 106], whereas user operations are separated by networking latencies of at least a few milliseconds. Having non-synchronized clocks not only confuses users, but also corrupts replicas permanently on rare occasions. This problem is discussed further in Section 6.9. We could have used Lamport clocks [87] for update serialization (but not for update retirement), but we chose wall-clocks because they can order logically unrelated events (e.g., a user contacting two nodes in a cluster serially).

6.4.2 Data Structures

Although our algorithm is designed to support many objects replicated on a diverse set of nodes, it is first introduced in the context of a single object. Section 6.6.1 describes a straightforward extension of the basic algorithm to support multiple objects.

Figure 6.1 shows the types and the persistent data structures maintained on each node for each object. Among them, the object contents (*gData*) and the object's replica set (*gPeers*) should be obvious.

---

[2]The replication algorithm itself does not rely on the bounded-delay assumption. It is used only by the membership service, which is called by the replication algorithm to choose a unique coordinator (Section 6.6.2).

```
type Timestamp = record
  time: Wallclock  // wall-clock time.
  nid: NodeID  // a tie-breaker.

type Update = record
  state: {ACTIVE, RETIRING, RETIRED}
  ts: Timestamp    // Uniquely identifies the update
  targets, done, peers: Set⟨NodeID⟩

persistent var
   gData: Content // The contents of the object
   gPeers: Set⟨NodeID⟩ // The replica set of the object.
   gU: Update  // The latest ongoing update, or NIL
```

Figure 6.1. *Three persistent global variables are used by the replication algorithm per object per node. gData is the contents, gPeers remembers the set of nodes that replicate the same object, and gU remembers the latest update being applied to the object, if any. The values of gData and gPeers are identical, and gU is NIL in the steady state in which no update is issued for the object. The value of these variables will diverge while an update is being propagated among replicas.*

The newest update to the object, if any, is stored in the *update record* (*gU*). An update transitions through several states. An update is ACTIVE when it is being propagated among replicas; it is RETIRING at the coordinator node when retirement notifications are sent; and it is RETIRED after the node receives a retirement notice. Fields *targets*, *done*, and *peers* in the update record remember the set of nodes that should receive the update, have acknowledged the update, and should replicate the object, respectively. When an update is newly issued, *peers* is set to be the replica set given by the caller, *done* is set to empty, and *targets* is set to be the union of the current replica set (*gPeers*) and the new replica set (*gU.peers*). As we discuss in Section 6.4.4, *targets* may expand as concurrent updates are found during propagation. Update propagation finishes when *gU.done* = *gU.targets*.

Notice the absence of new object contents in *gU*. The contents are pushed to other nodes by reading from *gData* directly most of the time[3]. This design contributes to reducing the space overhead of the algorithm.

[3]In practice, in certain situations, the new contents must be stored separately from *gData*. This issue is discussed in detail in Appendix D.1.

6.4.3 Common-Case Operations

This section describes the common-case operations of the algorithm, assuming for the moment that no two updates are issued at the same time for a single object. The handling of concurrent updates will be discussed in Section 6.4.4.

Update Issuance

Updates can be issued on any replica (or any node for a newly created object) at any time. The caller supplies the object's new contents and the new replica set. Setting the new replica set to empty deletes the object entirely from the cluster. The replication manager at the issuing node creates an update record (*gU*) and either modifies, creates, or deletes the local replica (*gData* and *gPeers*), depending on whether the node appears in the new replica set.

Update Propagation

The issuer of the update usually coordinates update propagation, although any node that stores an update record can theoretically become a coordinator. This feature makes the algorithm highly fault tolerant. Having multiple coordinators for the same update causes no logical problem, because updates are idempotent, but it does waste network bandwidth. The issue of coordinator selection is discussed further in Section 6.6.2.

The coordinator periodically pushes the update record and the new object contents to the target nodes that have not acknowledged the update (i.e., to nodes in *gU.targets* \ *gU.done*). When some targets are down, the coordinator simply keeps pushing the update until they recover. When some nodes remain failed for a very long period, the other nodes initiate a purge action, as will be described in Section 6.4.5.

The recipient of the update stores the update record in *gU*, creates, modifies, or deletes its replica, and replies to the coordinator. The coordinator expands its *gU.done* as replies arrive. It continues pushing the update until its *gU.done* equals *gU.targets*.

Deleting Retired Updates

While an update record occupies only a small amount of space, it is stored even after the replica itself is deleted. Because each node stores billions of replicas, these records must be erased quickly to keep them from filling the disks. Update records are deleted in two steps.

First, after finishing propagating an update, the coordinator sends *retirement notices* to the target nodes and informs that the particular update has finished propagation.

A node cannot immediately delete gU from the disk after receiving a retirement notice, however. It must keep the record for the newest update to be able to reject stale updates that may arrive out of order (Section 6.4.4). We adopt an algorithm for implementing at-most-once RPCs [95] to avoid accepting older updates. This technique works as follows:

- After receiving a retirement notice, the node waits for at least *WAIT* seconds and then deletes the update record. Here, *WAIT* is the sum of *SKEW*, the maximum clock skew among nodes, and *LIFETIME*, a long-enough period within which almost all the messages would arrive at the destination.
- Each node tags every outgoing packet with its local wall-clock value. The receiver discards the packet when the packet was sent more than *LIFETIME* seconds ago on the receiver's wall-clock.

6.4.4 *Handling Concurrent Updates*

This replication algorithm uses no global locks to serialize concurrent updates. Instead, it relies on the Thomas write rule, or timestamp ordering, to resolve conflicts asynchronously. When the replication manager receives an update request U from a proxy or another node, it checks if U's timestamp is newer than that of the newest update it knows, i.e., gU's. There are four possibilities:

1. Variable gU is empty.

   This is the normal case, in which no concurrent update is found. The node remembers update U in gU and modifies the replica according to U.

2. U's timestamp is equal to gU's.

   This means that the node has received the same update twice. The node just acknowledges

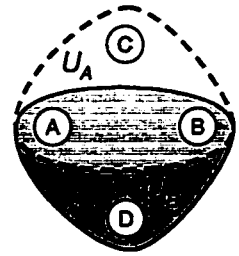


Figure 6.2. *In this example, the object is initially replicated on two nodes, A and B. Node A issues update U_A that adds a replica on node C. Simultaneously, node B issues update U_B that adds a replica on node D. We assume that U_B's timestamp is newer than U_A's. Applying the Thomas write rule, all nodes must agree on U_B, i.e., the replicas must be stored on nodes A, B, and D, but not on C. However, if node A propagates U_A to node C before it learns about U_B from node B, then node C must eventually receive U_B and delete its own replica. For this to happen, node B must learn about node C, which is not in U_B's original target set.*

the coordinator without applying the update.

3. U's timestamp is newer than gU's.

   In this case, two updates are found to be issued simultaneously. The node accepts U, stores it in gU, and modifies the replica according to U.

4. U's timestamp is older than gU's.

   In this case, U must be discarded so that all the replicas can agree on gU, which is the newest update.

One issue remains to be addressed for cases 3 and 4: when updates conflict, the older updates must be canceled on all replicas. If both the new and the old updates keep the object's replica set intact, this issue is solved simply by letting nodes receive the newer update and overwrite the older (which is trivial, because each update comes with the entire object contents). The problem becomes tricky otherwise, because the coordinator of the newer update sometimes must *discover* the targets of the older update, as demonstrated in Figure 6.2. This problem, which we call the *node discovery problem*, is similar to the distributed resource discovery problem [72]. The solution is also similar: a node that receives a newer, conflicting update sends the target nodes of the older update back to the coordinator and lets the coordinator expand its targets transitively.

Suppose a node receives an update U while it still stores gU. First, if U's timestamp is newer than gU's, the node applies U and remembers it in gU. Second, the node sets the *gU.targets* to be the union of the target sets of the two updates, the original gU and U. The node piggybacks this union on acknowledgement to the coordinator. The coordinator, in turn, adds the nodes in the union to its own *gU.targets*. This way, the target set of the newest update grows transitively as a side effect of the propagation, and it will eventually cover all the replicas involved in all the conflicting updates. Section 6.7.1 proves that this node discovery protocol always discovers all nodes added by concurrent updates.

6.4.5 Handling Long-term Failures

Nodes sometimes crash and never recover. Such nodes block update propagation from completing and create a backlog of update records on other nodes that eventually fill up the disks. FHC's replication algorithm solves this problem by purging nodes that remain failed for too long.

When a node finds another node dead for more than a predefined *purge period* (e.g., one week), it pretends that it received affirmative acknowledgements from the dead node for all its jammed updates. The node then purges the dead node simply by removing all references to the dead node from all the replica-set data structures (i.e., *gPeers*) it stores. To avoid having inconsistent data, when a node recovers after being down for longer than the purge period, it clears its disk contents and rejoins the cluster with a new name. Section 6.7.3 shows that this scheme preserves the consistency of replicas in most cases.

6.5 Examples

This section shows two examples to illustrate the behavior of the algorithm. The first example is a simple contents update. The second example demonstrates the node discovery protocol used to resolve concurrent replica-set changes.

Figure 6.3 shows the sequence of steps performed to update the contents of an object replicated on nodes A, B, and C.

(1) At time T (on A's wall-clock), A issues an update U:⟨*ts*=T, *targets*=*peers*={A,B,C}⟩ and modifies its replica.

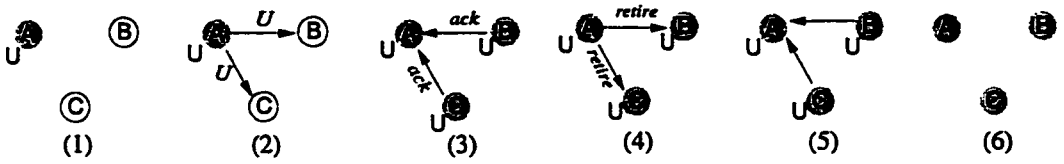


Figure 6.3. *An object replicated on nodes A, B, and C is updated by node A. A gray circle indicates that the node has applied the update. The letter "U" indicates that the update is stored on the node (i.e., on variable gU).*

(2) A pushes U and the new object contents to B and C.

(3) B remembers U in gU, modifies its replica, and returns $\langle ts{=}T,\ targets{=}\{A,B,C\}\rangle$ to A. C similarly applies U and returns $\langle ts{=}T,\ targets{=}\{A,B,C\}\rangle$ to A.

(4) A receives acknowledgements from B and C and changes *U.state* to RETIRING. A sends retirement notice $\langle ts{=}T\rangle$ to B and C.

(5) B and C change *U.state* to RETIRED and reply to A. A receives the replies from B and C and changes *U.state* to RETIRED.

(6) *WAIT* seconds later, A, B, and C erase gU from disk.

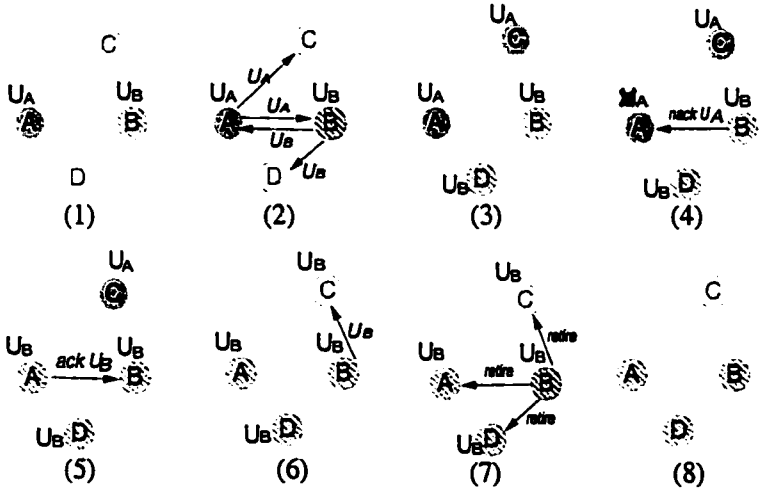


Figure 6.4. *Conflicting updates involving replica addition. Gray circles show the nodes that have applied U^A, and diagonally shaded circles show the nodes that have applied U^B.*

Figure 6.4 shows a scenario in which an object replicated on nodes A and B is updated concur-

rently, first by A, which adds C to the replica set, and next by B, which adds D to the replica set. For the sake of explanation, lets assume that B's update has a newer timestamp than A's. Applying the Thomas write rule, the system should converge on the update issued by B. In other words, the replicas should be on nodes A, B, and D, but not on C in the final state.

(1) At time T_A (on A's wall-clock), A issues U^A:⟨*ts*=T_A, targets=peers={A,B,C}⟩ and modifies its *gPeers*. At time T_B (on B's wall-clock), B issues U^B: ⟨*ts*=T_B, targets=peers={A,B,D}⟩ and modifies its *gPeers*.

(2) A pushes U^A and its object contents to B and C. B pushes U^B and its object contents to A and D. Notice that the object contents has not changed in this example, but nodes A and B push them anyway. Section 6.6.3 discusses the optimistic-delta technique for improving system performance in such cases.

Now, for the sake of explanation, suppose C and D receive the updates before B and A do.

(3) C creates a replica and replies ⟨*ts*=T_A, *targets*={A,B,C}⟩ to A. Similarly, D creates a replica and replies ⟨*ts*=T_B, *targets*={A,B,D}⟩ to B.

(4) B receives U^A from A. Because B stores *gU* (whose timestamp is T_B) and $T_A < T_B$, B rejects U^A and replies ⟨*ts*=T_A, *targets*={A,B,C,D}⟩ to A. B's U^B.*target* becomes {A,B,C,D}. Now, B has discovered the node C.

(5) A receives U^B from B. Because $T_B > T_A$, A modifies *gPeers*, replaces *gU* with U^B, and replies ⟨*ts*=T_B, *targets*={A,B,C,D}⟩ to B. Later, A may receive an acknowledgement for U^A from C, but A ignores it, because its timestamp differs from *gU.ts* on A.

(6) B pushes U^B to C, the only target not yet contacted. Upon receiving U^B, C recognizes that it is not a part of the new replica set, removes its replica, and replies ⟨*ts*=T_B, *targets*={A,B,C,D}⟩ to B.

(7) B receives the replies from A, C and D. B changes U^B.*state* to RETIRING and pushes U^B's retirement to A, C, and D. The three nodes acknowledge the retirement.

(8) *WAIT* seconds later, nodes A, B, C, and D erase U^B from *gU*. In the end, nodes A, B, and D store replicas. Node C created a replica and later deleted it.

```
var members: Set⟨NodeID⟩;
proc IAmCoordinator(u): bool
  return u.ts.nid = Me // ⟨1⟩
    ∨ Picker(members ∩ u.done, u.ts) = Me // ⟨2⟩

proc Picker(nodes, ts): NodeID
  // Apply a hash function on the bit-representation of nodes and ts.
  idx ← MD5(nodes, ts)
  return idx'th (modulo size of nodes) entry in nodes
```

Figure 6.5. *Designated coordinator selection. The membership service stores the set of presumed-live nodes in variable members. Function* IAmCoordinator *decides whether the node should become a coordinator for a particular update. Function* "Picker" *picks one node deterministically from the set of nodes, with an even chance for every node in the set.*

6.6 Extensions

This section describes several implementation techniques for improving the performance and practicality of the algorithm. We first discuss a simple extension for supporting multiple objects, followed by several performance optimizations.

6.6.1 Supporting Multiple Objects

The discussions so far have focused on a single object. In practice, the basic algorithm can easily be extended to support multiple objects. To support multiple objects, instead of variable gU, we now have a persistent table that partially maps an object ID to the update in progress for the object. An update is added to the table when an object is going to be modified. It is deleted from the table when it is removed or superseded by a newer update for the same object.

6.6.2 Designated Coordinator Selection

Section 6.4.3 introduced the algorithm as if any node with an update record could become a coordinator. Such a design, however, can potentially flood the update between every pair of replicas, resulting in $O(N^2)$ propagation cost. However, by carefully deciding when to coordinate an update using cluster membership information, the system can choose a single coordinator node most of the time. This algorithm is shown in Figure 6.5. Here, a node coordinates an update only when it is the original issuer of the update or when it is designated to take over the failed issuer.

6.6.3 *Optimistic Deltas*

The basic algorithm pushes the entire object contents even when only a byte is modified. This design becomes inefficient in some situations. For example, Porcupine's IMAP proxy sometimes modifies only flags in the email header [39], and the rebalancer often modifies only the replica set of the objects without changing their contents.

FHC's replication algorithm saves the networking and computational costs for such tasks by sending *optimistic deltas* [10]. Here, the coordinator simply pretends that all replicas for the object were consistent before the update was issued; it pushes only the difference between the old and new contents (called the optimistic delta) along with the *fingerprint* of the old replica contents. A node that receives the delta applies it when the replica's current fingerprint matches the one sent by the coordinator. Otherwise, the node requests a full contents transfer from the coordinator. This technique can reduce the cost of the algorithm in the common case without concurrent updates, especially when the update modifies only a small portion of the object.

A fingerprint is any short bit-string that summarizes replica contents. A collision-resistant hash function (e.g., MD5) is one way to compute a fingerprint. An alternative, used by Porcupine, is to store with each replica the timestamp (Figures 6.1) that shows the last time the replica was modified, and to use the timestamp as a fingerprint. The latter technique is faster and more accurate but consumes slightly more space than the former. A formal presentation of the optimistic delta algorithm is shown in Figure D.7.

6.6.4 *Batching Update Propagation*

The cost of the algorithm is reduced by batching (and sometimes delaying) retirement notices for multiple objects to the same node. Delaying retirement notices does not affect replica consistency. It merely delays the deletion of update records, increasing only the size of the update table (assuming that a node manages multiple objects; see Section 6.6.1).

The networking cost could be reduced further by batching the update application, as well. However, we forego this strategy because batching increases the chance of users observing inconsistent data. Batching also reduces disk I/O parallelism and harms system performance.

6.6.5 *Delayed Log Flushing*

This algorithm updates global variables transactionally. A common technique for implementing transactions is *redo logging*. Redo logging synchronously writes updates to the disk at the end of every transaction so that updates can be re-applied once a node recovers from a crash [68, 12]. In our algorithm, a node can sometimes delay log forcing and group log writes with other transactions without sacrificing replica consistency. Specifically, a node receiving an update for which it is the only remaining node (i.e., $U.targets \setminus U.done = \{Me\}$) need not force the update record to the log disk: the other replicas are already up to date, the last node will never have to become the coordinator by itself, and it can lose the update record without compromising replica consistency. Similarly, a node receiving a retirement notice can delay forcing the log. Losing retirement notices causes no harm to the replicas, because both the update application request and the retirement notice are idempotent.

In practice, for the common case of two-way replication, using this technique means that only the coordinator forces its log after update issuance, making the disk I/O overhead of this algorithm comparable to a simpler primary-copy replication algorithm.

Even with this optimization, note that the new object contents themselves still must be forced to the disk to ensure that the object remains up to date after a node crash.

6.7 Correctness of the Replication Algorithm

The correctness of FHC's replication algorithm hinges on two issues: that all replicas always receive and apply the newest update, and that no replica applies a stale update. This section explains how the algorithm solves these two issues, especially in the presence of updates that concurrently modify an object's replica set. A more formal proof appears in Appendix F.

6.7.1 *All Replicas Agree on the Newest Contents*

A key concept in the discussion of correctness is that of "knowing": node A knows another node B when $B \in gPeers$ or $B \in gU.targets$ on node A. The algorithm can distribute the newest contents to all replicas if all the following conditions apply:

- Applying the newest update suffices for a replica to become up to date, even if the replica misses some of the older updates.
- The replicas are always connected through the chain of "knowledge" (i.e., the graph of knowledge is strongly connected), and the coordinator of the newest update can discover all replicas using the node discovery protocol.
- No new edges and vertices are added to the knowledge graph by older updates while the newest update is being propagated.

Our algorithm satisfies all three conditions. The first condition holds, because each update modifies the entire replica state — the contents and the replica set — atomically. The second condition also holds, because a new replica is always created from an existing replica, and a replica disappears (i.e., *gData*, *gPeers*, and *gU* are deleted) only after all other replicas have received and applied the update that requested replica deletion. The third condition holds, intuitively because the Thomas write rule freezes the edges of "knowledge" emanating from a node once that node applies the newest update (i.e., after applying the newest update, the node rejects any older updates).

6.7.2 No Replica Applies a Stale Update

A node rejects older updates as long as the update record for the newest update is stored on it. Thus, it suffices to show that no older updates arrive after the update record (gU) is removed from disk. Below, we show that this indeed is the case. A more formal proof appears in Appendix F.5.

Figure 6.6 shows the problematic scenario. Here, node A accepts the retirement notice for update U at time T from the coordinator C, and later A receives a stale update U' at time T' from node B (all the times are observed at node A). Several causal observations can be made. For node A to receive the retirement of U, the coordinator C must have already received an acknowledgement for U from node B (i.e., $T_3 < T$) [4]. Moreover, node B also must have sent U' to node A before it received U (i.e., $T_4 < T_3$).

Let $T_{U'}$ be the clock value tagged on update U', which is sent from B to A. Then, node A never deletes the update record for U before T' (i.e., $T + WAIT > T'$) from the following chain of inequalities. This guarantees that node A can reject the packet containing U':

[4]Node B is always in $U.targets$; thus, it must have received U because of the argument in the previous section.

$$
\begin{aligned}
T + WAIT &> T_3 + WAIT && \text{(Node never receives retirement before applying } U\text{)} \\
&> T_4 + WAIT && \text{(Node cannot send } U' \text{ after receiving } U\text{)} \\
&> T_{U'} + WAIT - SKEW && \text{(Clock skew is less than } SKEW\text{)} \\
&> T_{U'} + LIFETIME && (WAIT = SKEW + LIFETIME) \\
&> T' && \text{(From the message discard rule)}
\end{aligned}
$$

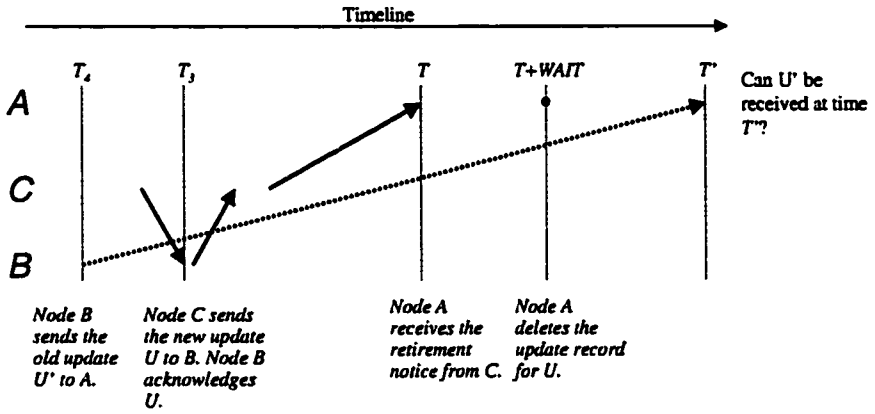


Figure 6.6. *Possible scenario for replica A applying a stale update.*

6.7.3 Node-purging Keeps Replicas Consistent

As discussed in Section 6.4.5, when a node remains failed for a long period, other nodes remove references to it after the purge period. As long as the knowledge graph is connected at the end of the purge period, this scheme does not cause inconsistency, because all remaining replicas agree on the newest content by then.

This purging scheme may render replicas permanently inconsistent when nodes or network links fail in such a way as to make the knowledge graph continuously disconnected until the purge deadline. We argue below that although such an event is possible, it is highly unlikely to happen in practice.

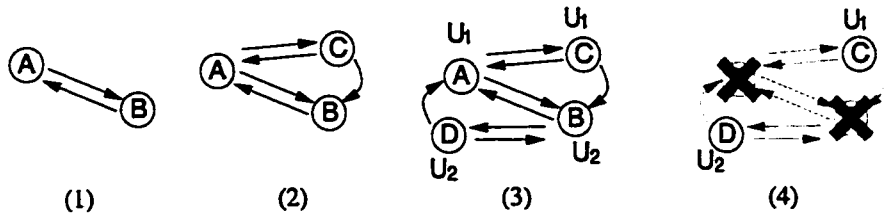


Figure 6.7. *A scenario that disconnects the knowledge graph without network partitioning. The edges show the knowledge among nodes. (1) Replicas A and B initially know each other. (2) A issues update U_A and creates replica C. (3) B issues update U_B and creates replica D. (4) A crashes before U_A is propagated to B or D. B crashes before U_B is propagated to A or C. If nodes A and B remain failed until the purge deadline, two components, $\{C\}$ and $\{D\}$, will be live but disconnected.*

How can a graph become disconnected? One cause of graph disconnection is network partitioning. Another cause is multiple node failures combined with concurrent replica additions, illustrated in Figure 6.7. Can a graph disconnection last until the purge deadline? The answer is no, for all practical purposes. First, network partitioning would never last long, because the network routing topology is usually designed with redundancy, and, even if it were not, it is repaired simply by installing replacement parts. Second, the latter failure scenario is unlikely to happen in practice, because it requires the combination of simultaneous replica additions and coincidental long-term failures of multiple nodes, both of which have very small window of vulnerability.

6.8 End-to-end Effects of Node Failures

The non-blocking and decentralization attributes, combined with the fact that any node in the cluster may deliver data to any other node, means that incoming data delivery is never blocked, assuming at least one node remains functional.

If a non-coordinator replica fails during update propagation, the coordinator simply keeps delivering the update to the replica in the background until it recovers. The initiating agent does not block, because it receives an acknowledgement after all live replicas apply the update.

If the coordinator fails before responding to the initiating agent, the agent will select another coordinator. For updates to a new object, as is the case with a new mail message, the initiating agent will create another new object and select a new, possibly overlapping, set of replicas. This helps to ensure that the degree of replication remains high even in the presence of a failed coordinator. This

design may deliver data to the user more than once, but duplicate delivery is not a major problem, as was discussed in Section 5.6.

The eventual consistency of the algorithm means that earlier updates to an object may "disappear" after all replica inconsistencies are reconciled. This happens, for example, when a user logs onto different nodes in the cluster and issues multiple updates to the same object simultaneously. We believe that just applying the newest update and discarding the older is the correct semantics in such a situation. An update race could also happen between users and the rebalancer that moves replicas to improve the system's performance and availability. The rebalancer, being a low-priority background process, deliberately chooses old timestamps to avoid disrupting the users in such cases (Section 7.2.3).

6.9 Effects of Non-synchronized Clocks

One important assumption made by this algorithm is that the nodes' clocks are loosely synchronized. Clock synchronization is inexpensive and is already deployed widely [105, 106], so there is no reason not to do that. What happens, however, if clock skew among nodes exceeds the limit (Section 6.4.3) for any reason? This section shows that non-synchronized wall-clocks may resolve conflicting updates non-intuitively and may render replicas permanently inconsistent, but that the chance of these problems happening is low.

First, when two updates are issued concurrently, the one issued earlier may win over the one issued later because of clock skew (Section 6.4.4). This phenomenon may confuse users but otherwise keeps replicas consistent (i.e., the replicas converge on the older update). This problem is unavoidable in theory, because clock skew cannot be zero; however, an uncontrolled skew increases its chance of happening.

Second, an uncontrolled clock skew may trick nodes into receiving and applying a stale update that arrives out of order, resulting in permanent replica inconsistency (Section 6.7.2). This problem, however, happens only when: (1) updates are issued concurrently, and (2) the network message is delayed longer than *WAIT* seconds. The latter situation can virtually be eliminated by setting a very long value for *WAIT*, although this solution is still probabilistic.

6.10 Cost of Replication

This section analyzes the steady-state performance of our replication algorithm. We show that the networking and computational overhead is close to optimal, and the disk space used by the algorithm to maintain replica consistency is very small. We analyze the actual behavior of the algorithm in Porcupine in Section 8.3.1.

6.10.1 Networking and Computational Overhead

In the common case, with the optimization described in Section 6.6.2, our algorithm pushes each update to $N-1$ replicas, receives $N-1$ replies, aggregates the retirement notices into one "batched" notice, sends it to $N-1$ nodes, and receives $N-1$ replies (N is the number of replicas for the object). In total, the algorithm sends $(2+\frac{2}{G})(N-1)$ messages per update, where G is the average aggregation factor for retirement notices. In Porcupine, G is measured to be about five to eight under a heavy load (Section 8.3.1). Thus, the networking and computational costs of our algorithm are less than $2.4(N-1)$ per update, which is close to the optimal number $2(N-1)$ for an algorithm that does not batch (and thus delay) update propagation.

6.10.2 Space Overhead

This replication algorithm is extremely space-efficient. Two types of on-disk data structures are stored per replica in addition to the object contents: the replica set (*gPeers* in Figure D.1), and the update record (*gU*). The replica set consumes little space — typically a few bytes per replica — and is stored only when the replica itself is present. The update record is stored on disk only while an update is being propagated. The space overhead by update records on a node is SUD, where S is the average size of *gU*, U is the average number of objects updated per second, and D is the average lifetime of an update, including the deletion wait period (Section 6.4.3) and the delay introduced by retirement-aggregation (Section 6.6.4). In Porcupine, $S \approx 60$ bytes, $U \approx 30$ seconds, and $D \approx 120$ seconds. Thus, the total size of update records on a node at any moment is about 200 kilobytes, regardless of the space occupied by replicated objects themselves. Section 8.3.1 actually measures Porcupine's replication space overhead and verifies our claim.

6.11 Summary

FHC's replication algorithm provides high availability using consistency semantics that are weaker than strict single-copy consistency, but strong enough to service Internet clients. By relaxing single-copy consistency and using a non-transactional protocol, our replication algorithm becomes exceptionally versatile, efficient, and fault tolerant, allowing both object contents and the replica set to be changed on any node at any time. The algorithm ensures that all live replicas will eventually converge on the newest contents, no matter how many updates are issued concurrently, how many replicas are added or deleted by these updates, and how many nodes are down. Inconsistencies, when they occur, are short lived and, by Internet standards, unexceptional.

Chapter 7 Dynamic Load Balancing

This chapter describes FHC's load balancing services for distributing data efficiently across nodes in the cluster. FHC deploys two different mechanisms for this purpose: the *load balancer* for determining the placement of incoming workloads, and the *rebalancer* for moving data in the cluster for better long-term performance and availability.

The load balancer is called mainly from front-end proxies whenever a data item is going to be read or delivered. Its job is to pick the node (or set of nodes) for reading or storing the item for better performance.

The rebalancer is a background maintenance process that runs on each node when the cluster is idle. It scans mail maps managed on the node, coalesces over-fragmented data, and adjusts the replication count of under- or over-replicated fragments. The rebalancing process improves the long-term health of the system by repairing suboptimal data distribution caused by configuration changes or by skewed incoming workloads.

7.1 Load Balancer

The load balancer is responsible for choosing the best set of nodes for storing a new data item or the best node for reading from the replicas of an existing fragment (Section 4.4.2). In developing the system's load balancer, we had several goals.

1. The load balancer must handle short-term fluctuations in the incoming workloads and the cluster load and make fine-grain decisions at the granularity of data delivery and retrieval.
2. It must support heterogeneous clusters, since not all nodes had the same disk capacity or speed.
3. It must handle skew in the incoming workloads, which occurs when some group of users receives a disproportionately large amount of data.

4. It must be automatic and minimize the use of "magic constants," or tuning parameters that would need to be adjusted manually as the system evolved.

5. With throughput as the primary goal, it must resolve the tension between *load balance* and *affinity*.

   Specifically, to balance load better, data should be stored on less-loaded nodes. However, blindly picking the globally least-loaded node for each incoming piece of data increases the number of fragments for a user, creating several performance problems. First, it increases the size of the mail map, leading to a larger memory footprint and recovery cost. Second, reading a single large fragment is usually faster than reading multiple small fragments, because the former collectively requires smaller disk head movements (the contents of a fragment are stored in a single file, as described in Appendix A.3.1). Third, in most file systems, creating a fragment is more expensive than appending onto an existing fragment, because the former requires a directory entry allocation[1]. Similarly, deleting many small fragments costs more than deleting a single large fragment. Finally, pure load-based data distribution may induce *herd behavior*, in which a herd of nodes all choose the same idlest node at the same time, instantly overloading the node [107].

   Consequently, the system should create multiple fragments for better load balance. However, it should also take affinity into account by appending new incoming data to existing fragments when needed.

We achieve the first goal by localizing load-balancing decisions. There is no centralized load-balancing agent in FHC. Instead, each node caches the recent load of other nodes and makes decisions independently. The second and third goals are achieved by carefully determining how the load is measured and distributed. Specifically, we use the number of pending I/O requests as the measure of the load and distribute it among nodes quickly using several complementary mechanisms. The last two goals are achieved by the spread-based load balancing algorithm, which takes both affinity and load into consideration.

---

[1]This is the case with traditional UNIX file systems (FFS) [102], but not on Linux ext2 [153] and several other file systems [141], because they create and delete files without synchronous disk I/O.

7.1.1 Defining Load

The load information on a node has two components: a boolean, which indicates whether the disk is full, and an integer, which indicates the "busyness" of a node. A node with a full disk is always considered "very busy" and is used only for operations that read or delete existing fragments. The busyness is the number of pending remote procedure calls that might require a disk access. For example, calls such as "retrieve message" and "update replica" are counted, while calls such as "add a node to mail map" and "retrieve the user's profile bank" are not. (The list of remote procedure calls in Porcupine is shown in Appendix A.2.2.)

This definition of load differs from the customary definition that measures load by CPU utilization or by the number of active processes [51, 42, 107]. We chose ours because FHC's performance is almost always disk-bound, with the CPU spending most of the time waiting for disk I/Os to complete (Section 8.3.3). Using a CPU activity as the measure of load will yield very low, meaningless numbers. On the other hand, using our load definition allows the system to even the load naturally in a cluster containing disks with different speeds.

We use no "decay" factor, such as computing $load \leftarrow newload * \alpha + load * (1 - \alpha)$. We rather use the instantaneous load of nodes directly. Decay is meaningful only when the unit of load balancing is so large that the future load can be predicted from the current load. Such is not the case in FHC. We found that decay harms performance by aggravating herd behavior.

7.1.2 Distributing the Load Information

Each node in the cluster caches the load of other nodes to reduce the cost of obtaining load information. Load information is updated by two means: RPC piggybacking and ring-based distribution.

First, each node embeds its load on each remote procedure call request or reply, and the receiver caches the sender's load upon receiving the message. The average staleness of the load information using this mechanism is N/S seconds, where N is the number of nodes in the cluster, and S is the average number of messages sent per node per second. Thus, this approach gives a timely view of nodes in a small cluster, but it gives increasingly stale information as cluster size grows.

To alleviate the deficiency of the first mechanism, FHC also circulates load information along the virtual ring organized by the membership protocol (Section 10). The membership coordinator

periodically sends to the next node in the ring a packet containing its knowledge of the nodes' load information. The receiving node updates the slot in the packet corresponding to its own load and forwards the packet to the node next in the ring. This process continues until the packet returns to the coordinator. The average staleness of the load information using this mechanism is $T + LN/2$, where T is the load update frequency, and L is the average node-to-node messaging delay. With $N = 1000$, $T = 5$ seconds, and $L = 2$ms, the staleness of the load information is 3 seconds.

Notice that this ring-based load distribution protocol is different from the ring-based fault detection protocol described in Section 10, because of their different timing requirements. The load distribution mechanism circulates the load packet around the ring as quickly as possible to keep information fresh. On the other hand, the fault detection mechanism lets each node independently and asynchronously probe the next node in the ring, because it cannot rely on the coordinator sending packets in a timely manner.

7.1.3 *Description of the Load Balancing Algorithm*

The load balancer defines for each user a *spread* to reconcile the load and affinity. The spread is a configuration parameter that specifies a soft upper bound on the size of a user's mail map, or the maximum number of nodes on which a given user's fragments are stored[2] (for the moment, fragments are assumed not to be replicated; balancing load with replication is discussed in the next section). The bound is soft to permit the load balancer to violate the spread if one of the nodes storing a user's data is not responding. When a user owns fewer fragments than the spread limit, the load balancer adds a random set of nodes to make up a candidate set (Figure 7.1). Adding a random set of nodes helps the system avoid herd behavior. The algorithm is shown in Figure 7.2.

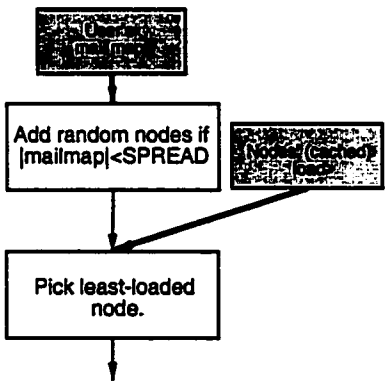


Figure 7.1. *The algorithm for choosing the node for storing new data for a user.*

The use of a spread-limiting load balancer has a substantial effect on system throughput even with a relatively narrow spread. The benefit is that a given user's data will be found on relatively

[2] Each user can in practice own multiple mailboxes in Porcupine. Thus, the spread actually specifies the maximum number of fragments for a particular mailbox.

few nodes, but those nodes can change entirely each time the user reads and deletes data from the server.

How large should the spread actually be? When the cluster configuration is homogeneous and the workload is uniformly distributed, a spread of two has been shown to give a far better load balance over a spread of one; widening the spread beyond two improves balance slightly, but not substantially [107, 108, 42]. The reason for this has been demonstrated previously and is as follows [51]: in any system where the likelihood that a host is overloaded is p, then selecting the least loaded from the spread of s hosts will yield a placement decision on a loaded host with probability p^s. Thus, the chance of making a good decision (avoiding an overloaded host) improves with the spread. In particular, in a nearly perfectly balanced system, p is small, so a small s yields good choices. A smaller spread also lets the system better avoid herd behavior. In FHC, a smaller spread has an additional benefit of more streamlined disk operations (Section 7.1). As we show in Chapter 8, we also achieved good performance with a spread of two in our 30-node homogeneous cluster; however, we also observed that a spread of four gives equally good performance. The reasons are twofold. First, herd behavior is not pronounced with such a small cluster size. Second, because our evaluation platform (Linux) creates and deletes files extremely fast, the penalty of accessing a larger number of files is offset by the slightly better load balance achieved by a spread of four.

On the other hand, when the cluster is heterogeneous, that is, when some disks are much faster than others, a larger spread is required. This is because a small spread cannot include the fast nodes in the candidate sets often enough to give them the chance to handle the load commensurate with their speed. In Chapter 8, we show that when a few nodes are three times faster than others in a 30-node cluster, a spread of at least four is needed to utilize resources on the fast nodes. The optimal spread limit in a heterogeneous cluster is currently determined only by trial and error, unfortunately, because it depends on many factors that elude formal analyses, such as disk-head scheduling policy, average disk seek time, and buffer cache hit rate.

7.1.4 Balancing Load for Replicated Objects

When a new object is to be replicated, the load balancer picks multiple target nodes. The concept behind the algorithm remains the same as before: with a spread limit of S and replication factor of R, the load balancer creates S candidate replica sets, each containing R nodes, and picks the least-

loaded replica set among them. (As described in Section 6.3.2, a mail map already contains a set of replica-sets when the fragments are replicated.) The load of a replica set is defined to be the sum of the loads of the nodes in the set. Figure 7.2 describes the algorithm.

This scheme, however, turns out to work less effectively than that for the non-replicated case. The primary reason is that the load value computed during decision-making usually becomes stale by the time the nodes actually store the data on disk because of the propagation delay in the replication algorithm. It is also debatable whether simply adding the load of nodes is a good measure of the load of a replica set. We experimented with using the minimal and the median load values among the replica set, but they did not show an improvement. Finally, our load balancer does not make an explicit effort to maximize the availability of a particular object. A policy such as allocating at least one replica at a node with historically high availability [19] would be beneficial to our system. Improving load balancing for replicated storage is future work (Section 9.4.1).

```
const
  S = ...  // The spread limit. two or four is common.
  R = ...  // Number of replicas per fragment. One to three is common.

proc PickNodesForStorage(user): Set(nodeid)
  map ← LookupUser(user)
  Filter out all fragments that contain dead replicas from map.
  while Size(map) < S
    rset ← pick R random nodes
    map ← map ∪ {rset}
  maxload ← −1
  foreach ⟨n1,n2,...,nR⟩ ∈ map
    thisload ← Σ_{1≤i≤R} Load(ni)
    if thisload > maxload then
      r ← ⟨n1,n2,...,nR⟩
      maxload ← thisload
  return r
```

Figure 7.2. *The load balancing algorithm. This procedure takes the name of the user to whom the data is delivered, and it returns the set of nodes on which to store the object. Function* LookupUser(*user*) *calls the user's profile manager and obtains her mail map, which contains a set of the replica sets that currently store the user's fragment. Function* Load(*node*) *returns the cached load of the node.*

7.1.5 *Balancing Load on Data Retrieval*

The load balancer is also used when reading a replicated fragment. It again uses the number of pending I/O-inducing RPC requests as the load measure and picks the least-loaded among the live nodes in the fragment's replica set.

7.2 Rebalancer

The rebalancer has two goals. The primary goal is to optimize the long-term data distribution of the cluster by adding, deleting, or moving replicas. Equally importantly, it must avoid disrupting users, since the rebalancer often competes for resources with client activities.

7.2.1 *Rationale for the Rebalancer*

The use of the rebalancer in addition to the load balancer for workload distribution is justified on two accounts: (1) the inability of the load balancer to distribute data optimally in the long term, and (2) the daily workload fluctuation, which allows the rebalancer to exploit idle resources.

Suboptimal Data Distribution by Load Balancer

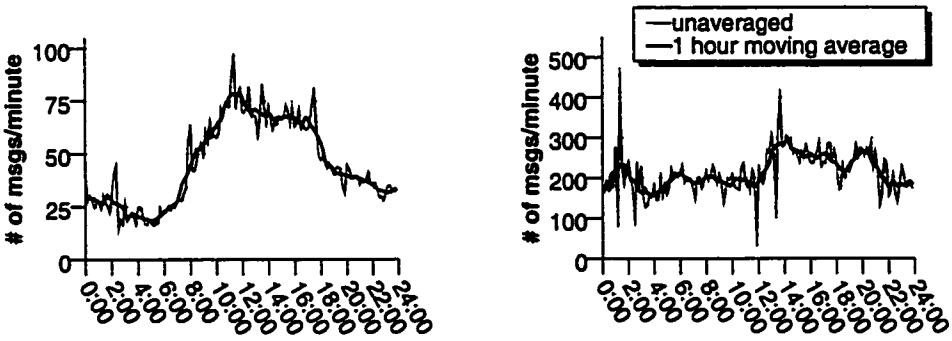
The load balancer maximizes the throughput of data delivery and retrieval only in the short term. The data layout chosen by the load balancer may become non-optimal in several situations:

- The load balancer violates the spread limit to mask failure. Later, if the node recovers, there will be more fragments than the limit, causing performance degradation (Section 7.1). This situation actually happens often because most failures are transient [66, 67].
- The spread limit is sometimes violated because of workload concurrency. For example, when two nodes deliver messages to the same mailbox simultaneously, they may choose different fragments for storage, creating more fragments than are allowed by the spread limit.
- The load balancer cannot change replica locations once fragments are created. Thus, a node crash or retirement drops the replication factor of fragments to a less than desired number.

These problems require a solution by an agent such as the rebalancer, which can distributes data with a longer attention span.

Daily Workload Fluctuation

The charts in Figure 7.3 show the number of email messages and Usenet articles handled during a day on two medium-scale servers [137]. Both of them show 1:2 to 1:5 differences between peaks and valleys. Thus, the rebalancer, by running only at night, is simply using otherwise wasted resources. FHC's two-tier load balancing strategy is an attempt to maximize daytime throughput and correct resulting problems during "off" hours.



(a) The number of email messages handled at the Dept. of Comp. Sci. and Engr., Univ. of Washington, on a weekday in February 1998.

(b) The number of incoming Usenet articles received at Compaq Western Research Laboratory, on a weekday in July 1998.

Figure 7.3. *Daily workload fluctuation observed on two Internet servers.*

7.2.2 *Description of the Rebalancing Algorithm*

The rebalancer runs independently on each node. Each node is responsible for rebalancing the user-map buckets it manages (Section 2.5.1). The rebalancer walks over every mail map in these buckets and adds, deletes, or moves fragments when one of the following conditions is met:

- A user's profile bank is void, but her mail map is non-empty for a long period (three days in Porcupine). This situation usually happens when the nodes storing the user's fragments were down when the user's account was deleted (Section 5.5.4). The rebalancer deletes these fragments.

This case requires caution, because the profile bank can also become void when all the replicas for the user's profile database fail. Fortunately, because the probability of all the replicas failing for a long period is very small (assuming independent failures), the rebalancer can ensure that it only deletes orphan fragments by waiting for a few days.

- The number of replicas for a fragment differs from the value specified by the administrator or the user. This happens for the same reason as above: some nodes storing fragments were down when the replication policy changed. For these fragments, the rebalancer adds or deletes replicas by using the replication service (Chapter 6).
- Some of the replicas for a fragment have been dead for a long period (three days in Porcupine). For these replicas, the rebalancer adds replicas using the replication service described in Chapter 6. If a node remains dead for a truly long period (e.g., a week), the rebalancer asks the replication manager to remove all references to that node (Section 6.4.5).

The rebalancer applies a similar strategy to the profile database to improve its availability and performance.

- When some of the replicas for a profile database bucket have been dead for a long period (three days), the rebalancer adds replicas to the bucket.
- When a database bucket is split into multiple groups, the rebalancer merges the smaller groups into the largest by calling the replication manager. Split buckets slow down profile updates, because proxies need to ask groups in the bucket sequentially to find the user during a profile update (Section 4.4.4).

  In practice, as we discuss in the next section, bucket splitting is a rare phenomenon. It happens only when the rebalancer itself updates the replica-set of a profile database bucket while a proxy is simultaneously updating the same bucket on behalf of a user.

7.2.3 Avoiding Disrupting Clients

The rebalancer runs while the node is handling client sessions. Thus, a possibility of conflicts arises: not only does the rebalancer compete for the node's CPU, disk, and memory resources against

client sessions, but it may move fragments or the profile database while they are being updated by these sessions. The rebalancer employs the following two strategies to avoid disturbing such user activities.

First, the rebalancer runs only during the night, when the level of user activity is generally low (Section 7.2.1). In addition, we start the rebalancer at a random time during the night on each node, allowing the cluster to spread the load imposed by the rebalancer.

Second, the rebalancer tries to *yield* to competing updates by clients by deliberately choosing old timestamps for updates. Below, we discuss the effects of concurrent updates and explain our solution in more detail.

If a proxy adds a new object to a fragment while the rebalancer is changing the fragment's replica set, no problem occurs. Two fragments will be created as the result: one on the original replica-set nodes containing only the new fragment, and the other on the new replica-set nodes, containing the rest. This happens because the rebalancer moves not the fragment itself, but the objects (e.g., email messages) it finds in the fragment. This fragment-splitting is transparent to the user, because she will discover both fragments through the mail map. The split fragments will be coalesced when the rebalancer runs the next time.

On the other hand, the rebalancer and a proxy updating an existing object simultaneously (e.g., by an IMAP session that modifies email flags) can be problematic. When this happens, the update by the proxy is lost if its timestamp is older than the rebalancer's (Section 6.2). While this problem is inherent in any non-blocking replication algorithms, the rebalancer tries to minimize the harm by using two techniques. First, as discussed already, the rebalancer runs during the night, when users are unlikely to be active. Second, the rebalancer deliberately chooses an update timestamp that is older than the actual wall-clock. Remember that timestamps are used to serialize conflicting updates (Section 6.4.2). The timestamp shift is chosen to be slightly longer than the average update propagation delay for replicated objects. This way, the rebalancer can mostly ensure that competing updates issued by proxies will "win" over those by the rebalancer[3].

A similar situation may arise for the profile database. Adding a user while the users' bucket is being moved by the rebalancer causes the bucket to split. Again, no harm is done. The groups will

[3]In effect, this scheme artificially inflates the inter-node clock skew. Thus, the shift must be added to *WAIT* (Section 6.4.3) to prevent an update by the rebalancer from accidentally overwriting newer updates.

be merged when the rebalancer runs the next time. Modifying the profile of an existing user while the bucket is being moved may cause a lost update, but the rebalancer tries to avoid this problem by choosing an old timestamp.

7.3 Summary

The load balancing service in FHC distributes on-disk data among cluster nodes to improve performance. The service is provided by the cooperation of two managers, each specialized for a different goal.

The load balancer maximizes short-term system throughput. It is consulted by local proxies every time a data item is going to be delivered or read. It tries to achieve the best balance between data affinity and load balance by limiting the spread of the data a user owns. This design allows the system to localize data on fewer disks and improve performance when there are no failures, but to spread data dynamically and improve availability otherwise.

The rebalancer moves data within the cluster to achieve better long-term performance and availability. Its key goal, to avoid disrupting user activity, is achieved by running during the night and yielding to concurrent updates issued by clients.

Chapter 8 System Evaluation

This chapter evaluates the FHC architecture using Porcupine as a prototype. We characterize the system's scalability as a function of its size in terms of three key requirements:

Performance: We show that the system's performance scales linearly with additional nodes, and that the system outperforms a statically partitioned configuration consisting of a cluster of standard SMTP, POP, and IMAP servers with fixed user mapping. We also show that Porcupine runs with a small memory overhead even for a large user population.

Availability: We demonstrate that replication and reconfiguration have low cost.

Manageability: We show that the system responds automatically and rapidly to node failure and recovery, while continuing to provide good performance. We also show that incremental hardware improvements can automatically result in system-wide performance improvements. Lastly, we show that automatic dynamic load balancing efficiently handles highly skewed workloads.

8.1 Platform

For the measurements in this chapter, we ran Porcupine on a cluster of thirty nodes running Linux 2.2.7 and glibc-2.1.2 and using the ext2 file system for storage [153]. As would be expected in any large cluster, our system contained several different hardware configurations, as listed in Table 8.1. All nodes were equipped with 100Mb/second Ethernet cards but were connected by 1Gb/second Ethernet hubs. In our experiments, neither the Ethernet cards nor the hubs were ever saturated. Rather, the performance of our system was bound totally by disks, as we show in Section 8.3.3. Moreover, the disks we used were uniformly slow (except for the SCSI disks on type "B" nodes used during the experiments in Section 8.4.3). For these reasons, our baseline configuration can be considered to be a homogeneous cluster.

Porcupine's implementation issues are discussed in detail in Appendix A, but some of its attributes pertinent to the experiments are summarized below.

- The user map contains 256 entries (Section 4.3.1).
- The spread limit for a mailbox is set to four, unless otherwise noted (Chapter 7).
- The number of replicas per email message is either one or two, except in the experiments conducted in Section 8.3.1.
- A mailbox fragment is stored in two files, regardless of the number of messages contained within (Appendix A.3.1). One file contains the email texts, and the other contains the email digest (size, date, sender, message ID, file offset, etc.). Only the "text" file is forced to disk after modification, and the "digest" file is written back to disk lazily. The digest file is reconstructed if found to be corrupt.
- The mailbox fragment files are grouped and stored in directories corresponding to the hash of user names (Section 5.7). For example, if Ann's hash value is 9, then her fragment files are stored in `spool/9/ann` and `spool/9/ann.idx`. This design allows the discovery of mailbox fragments belonging to a particular user-map bucket — a critical operation during system reconfiguration — to be performed by a single directory scan.

For purposes of comparison, we also measure a tightly configured conventional mail system in which users and services are statically partitioned across nodes in the cluster. In this configuration, at the front end, we run protocol redirector nodes that accept SMTP, POP, and IMAP requests and relay them to appropriate back-end storage nodes using a static knowledge of the user-to-node mapping. At the back end, we run modified versions of the widely used Sendmail-8.9.3 [142], ids-pop3d-0.9.2 [80], and UW-IMAPD [116] servers. To keep the front-end nodes from becoming a bottleneck, we determined empirically that we needed to run one front end for every fifteen back ends. The tables and graphs that follow include the front ends in our count of the system size. For these configurations, we assigned users to nodes round-robin, which is near-optimal for our workload. To further optimize the configuration, we disabled all security checks, including user authentication, client domain name lookup, and system logging.

Table 8.1. The number and the specification of nodes used for the experiments. In addition to these machines, we used four type "A" machines to generate the email workloads. Type "B" machines are equipped with slow IDE disks and fast SCSI disks. The SCSI disks are used to test the effectiveness of load balancing but are otherwise left unused.

Type	#	CPU	Clock	Memory	Disk (interface)	Ethernet
A	15	PII	350MHz	128MB	WDC AC24300L (IDE)	3C905B
B	3	PII	350MHz	128MB	WDC AC14300R (IDE)	3C905B
					2*ST34502LW (AHA2940)	
C	2	PII	333MHz	128MB	WDE4360-1807A3 (AHA2940)	i82557
D	6	PII	300MHz	128MB	Qt Fireball ST4.3A (IDE)	3C905
E	4	PPro	200MHz	64MB	ST32151N (BT948)	3C905

8.2 Workload

We developed a synthetic workload to evaluate Porcupine, because users at our site do not receive enough email to drive the system into an overload condition. We did, however, design the workload generator to model the traffic patterns we have observed on our departmental mail servers. The email message size distribution, Figure 8.1, models what was observed on the servers. The distribution has a mean message size of 4.7KB and a fat tail up to about 1MB.

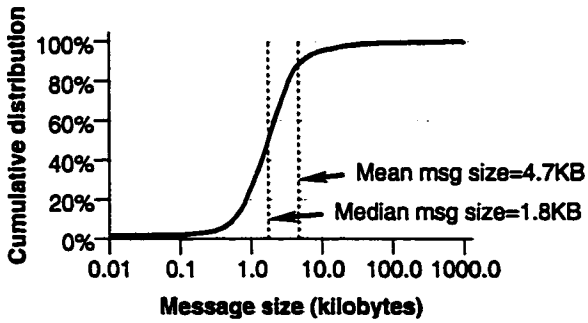


Figure 8.1. *Email message size distribution used by the workload generator.*

The workload consists of a random mix of mail delivery (SMTP) and retrieval (POP or IMAP) sessions. Mail delivery accounts for 90% of the workload. Unless otherwise noted, each SMTP session sends a message to a user chosen from a population according to a Zipf distribution with $\alpha = 10.3$ (Figure 8.2). We defined a user population with size equal to 160,000 times the number of nodes in the cluster (or about 5 million users for the 30-node configuration) unless otherwise noted. As we show in Section 8.3.5, the primary impact of the increase of the total user population to the system is the potential increase of memory usage. Otherwise, performance is not significantly impacted, because the database is distributed in Porcupine and no authentication is performed for any of our platforms.

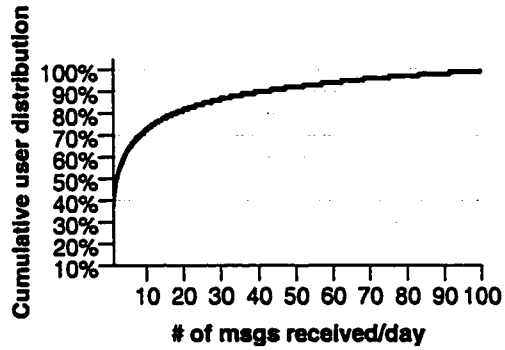


Figure 8.2. *User activity follows a Zipf distribution. A few users receive a disproportionately large number of messages.*

Mail retrieval accounts for the remaining 10% of the workload. In most experiments, mail retrieval consists of POP sessions that select a user according to the same Zipf distribution, collect and then delete all messages awaiting the user. Some experiments run IMAP sessions instead of POP. In the IMAP-based workload, 7% of the sessions retrieve the digest of all messages in the mailbox, read previously unseen messages, stamp the "Seen" flag on these messages, and log out. The remaining 3% of the sessions retrieve the digest of all messages, read previously unseen messages, and delete all messages. The POP-based workload models an environment in which users downloads messages on an ISP server to their PCs, whereas the IMAP-based workload models an environment in which users with thin-clients keep their messages on the server. We will show, however, that the

difference in the workload has little effect on the system's scaling characteristics. For this reason, we will conduct most experiments using the POP-based workload only.

In the Porcupine configuration, the workload generator initiates a connection with a node selected at random from the cluster. In the conventional configuration, the generator selects a node at random from the front-end nodes, and that node forwards the session to the back-end node designated to manage the user on whose behalf the session runs.

The load generator attempts to saturate the cluster by increasing the number of outstanding requests until at least 10% of the SMTP requests fail to complete within two seconds [1]. At that point, the generator reduces the request rate and resumes probing.

We demonstrate performance by showing the maximum number of messages the system receives per second. Only message deliveries are counted, although message retrievals occur as a part of the workload. Thus, this figure really reflects the number of messages the cluster can receive, write, read, and delete per second. In all the experiments, except for the failure recovery experiments, we start the cluster from an empty spool, "warm up" the cluster for 12 minutes to fill the spool, and then measure the cluster's time-averaged throughput for 6 minutes. The error margin is smaller than 5%, with a 95% confidence interval for all values reported in the following sections.

8.3 Steady-state Performance

Figure 8.3 shows the performance of the system as a function of cluster size. Email messages are not replicated in this experiment.

The graph shows four different configurations: Porcupine under a 90%-SMTP, 10%-POP workload; Porcupine under 90%-SMTP, 7%-IMAP-read, 3%-IMAP-read-and-delete workload; and the two conventional configurations running Sendmail+popd and Sendmail+imapd.

Although neither replicates, Porcupine outperforms and outscales conventional Sendmail-based configurations. The difference is primarily due to the conventional system's use of the file system during message delivery. For each incoming message, Sendmail stores the message in a temporary file, forces the file buffer to disk using the "fsync" system call, and forks a helper application

[1] We excluded POP and IMAP sessions from workload throttling measurements, because their latency depends heavily on the number of messages in the mailbox.

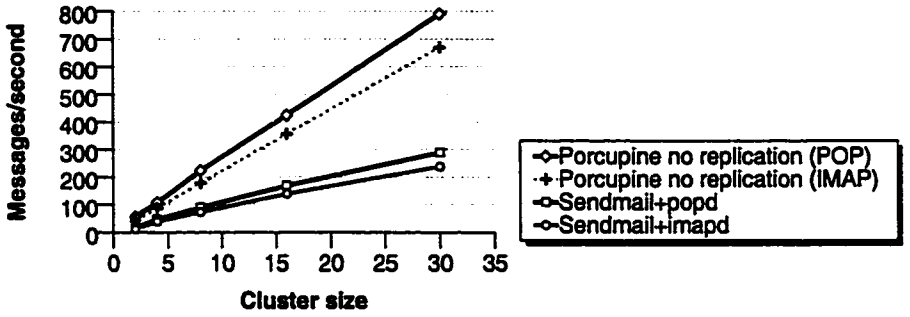


Figure 8.3. *Throughput scales with the number of hosts. This graph shows how Porcupine and the sendmail-based system scale with respect to cluster size under a POP-based workload.*

(`/bin/mail`), which appends the message to the recipient's mailbox, "fsync"s the mailbox, and deletes the temporary file. On the other hand, Porcupine appends the incoming message directly to the recipient's mailbox file and "fsync"s it. Thus, the Sendmail-based system issues twice as many synchronous disk writes as Porcupine. Sendmail incurs several other types of overhead, including excessive process forking and the use of lock-files to arbitrate mailbox accesses among the instances of Sendmail, popd, and imapd; however, these overheads are not apparent in Linux, which handles them efficiently. With some effort, we believe that the conventional system could be made to scale as well as Porcupine. However, the systems would not be functionally identical, because Porcupine allows users to read incoming messages even when some nodes storing the user's existing messages are down.

When workload includes IMAP sessions, both Porcupine and the conventional system run slower than with the POP-based workload, because they perform additional synchronous disk writes when writing the "Seen" flag on messages. The overall trend, however, remains the same: both the configurations scale well, but Porcupine is about twice as fast as Sendmail. Therefore, in the remainder of this chapter, we evaluate system performance mostly using the POP-based workload only.

8.3.1 Replication Performance

Figure 8.4 shows how replication affects the system's performance. Graph (a) is for the 90%-SMTP, 10%-POP workload; graph (b) is for the 90%-SMTP, 7%-IMAP-readonly, and 3%-IMAP-and-delete workload.

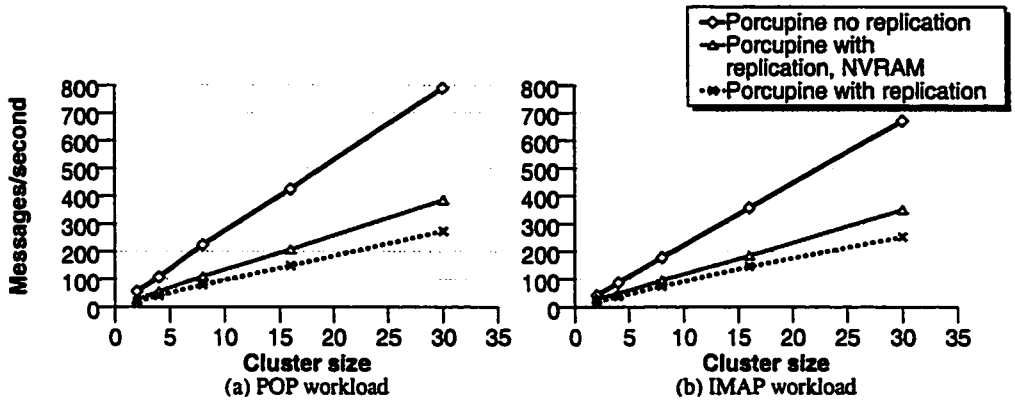


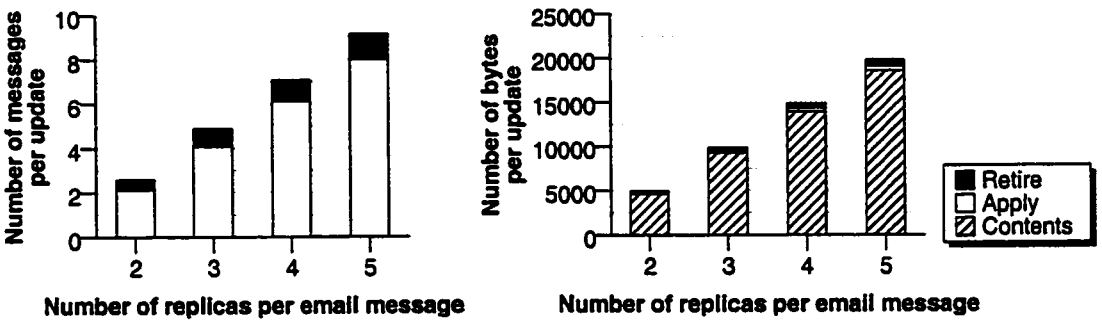
Figure 8.4. *Steady-state scalability of Porcupine when each email message is replicated on two nodes. The graph (a) is for the POP-based workload, and the graph (b) is for the IMAP-based workload. Porcupine with replication still scales linearly with cluster size. Replication slows down the system by more than 50% because of disk traffic imposed by redo-logging. Using NVRAM for logging improves performance to the ideal level.*

If the replication service works ideally, system throughput should scale linearly at half the level of the baseline non-replicated case, because replicating data on two disks at least doubles disk I/O traffic, and the performance of our cluster is disk-bound. The actual performance ("Porcupine with replication") still scales linearly, but at less than half of the baseline case. This is because our replication algorithm runs local transactions to maintain its persistent data structures. Thus, the algorithm performs three synchronous disk writes per update: one for each replica and one to update the coordinator's log file (Section 6.6.5). Moreover, in our hardware configuration, the log and the fragments share the same disk (nodes have only one disk each), forcing the disk heads to bounce more than necessary.

One way to improve the performance of replication is to use non-volatile RAM (NVRAM) for the log. Since updates retire quickly from the log (Section 6.4.3), most of the writes to NVRAM never need go to disk and can execute at memory speeds. Although our machines do not have NVRAM installed, we can simulate NVRAM simply by disabling log flush (log contents are still written to disk in the background). As shown in Figure 8.4, NVRAM improves throughput to half of the non-replicated case, indicating that the replication algorithm works optimally when equipped with NVRAM.

8.3.2 *Space and Networking Overhead of Replication*

The previous sections demonstrated that replication works optimally with the addition of NVRAM to nodes. This section looks at the networking and space overhead of the replication service. Figure 8.5 shows the networking overhead of replication during email message delivery in a 30-node cluster. Here, during the delivery of a single email message, $2(R-1)$ update propagation messages are sent ($R-1$ requests and $R-1$ replies; see also Section 6.6.4). Retirement notices are effectively batched and compressed to the ratio of 1:5 to 1:8. The replication also imposes a negligible overhead in terms of message byte counts, as demonstrated in Figure 8.5(b).



(a) The average number of packets sent per email delivery. The "Apply" packets contain update application requests, and the "Retire" packets contain retirement notices.

(b) The average number of bytes sent per email delivery. "Apply", "Retire", and "Contents" show the number of bytes consumed by each type of information exchanged between replication managers.

Figure 8.5. *The networking overhead of replication in a 30-node cluster running at peak throughput under the POP-based workload. The graphs confirm that the replication manager effectively batches retirement notices and that its networking overhead is negligibly small.*

Figure 8.6 shows the space overhead of replication. As already discussed in Section 6.10.2, the space overhead of the replication manager is very small, and this graph confirms that claim. Even when each message is replicated on five nodes, replication meta-data consumes less than 1% of the disk space.

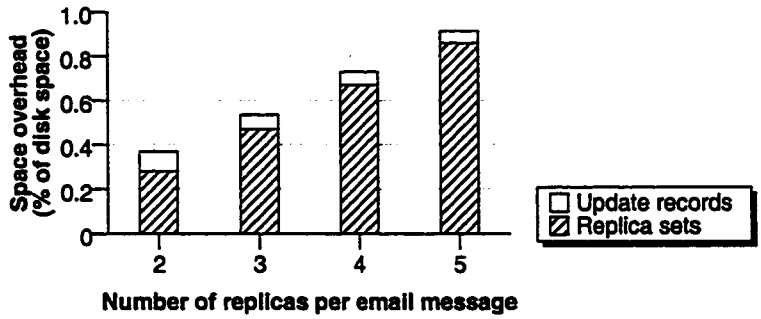


Figure 8.6. *The space overhead by update records and replica sets on a type "A" node, running at peak throughput as a part of the 30-node cluster.*

Table 8.2. Resource consumption on a single node with one disk.

Resource	No replication	With replication
CPU utilization	15%	12%
Disk utilization	75%	75%
Network send	2.5Mb/second	1.7Mb/second
Network receive	2.6Mb/second	1.7Mb/second

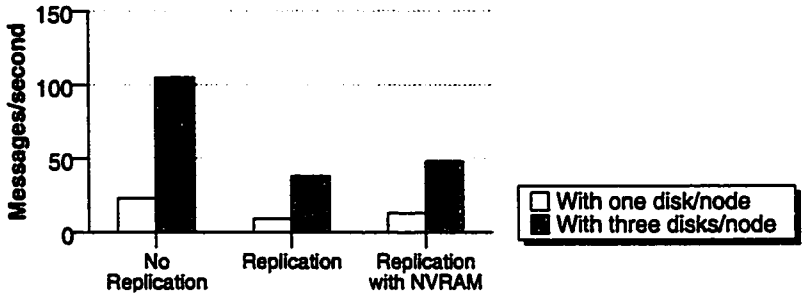


Figure 8.7. *Summary of single-node throughput in a variety of configurations.*

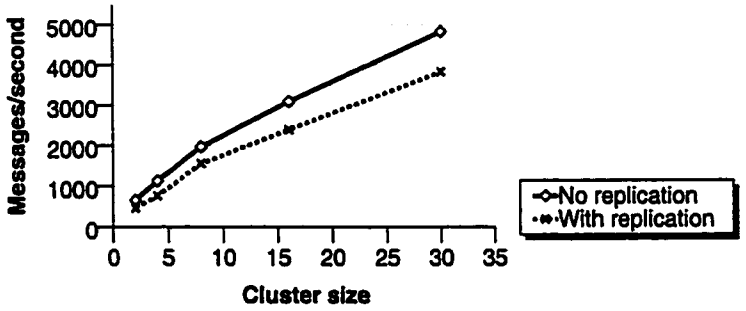


Figure 8.8. *Throughput of the system configured with infinitely fast disks.*

8.3.3 Analyses of Bottlenecks

Table 8.2 shows the CPU, disk, and network load experienced by a type "A" node running at peak throughput. Disk utilization is measured in the same way as CPU utilization; i.e., by measuring how often the disk request queue is found to be non-empty[2]. For this configuration, the table indicates that the disk is the primary impediment to single-node performance.

To demonstrate this, we measured the performance of clusters with one and two nodes with increased I/O capacity. A single type "B" node with one IDE disk and two SCSI disks delivered a throughput of 105 messages/second, as opposed to about 23 messages/second with only the IDE disk. We then configured a two-node cluster, each with one IDE disk and two SCSI disks. The machines were each able to handle 38 messages/second (48 assuming NVRAM). These results (normalized to single-node throughput) are summarized in Figure 8.7.

[2]We developed a kernel module that inspects the queue every 10 milliseconds.

Lastly, we measured a cluster in which disks were assumed to be infinitely fast. In this case, the system does not store messages on disk but only records their digests in main memory. Figure 8.8 shows that the simulated system without the disk bottleneck achieves a six-fold improvement over the measured system. At this point, the CPU becomes the bottleneck. Thus, Porcupine with replication performs comparatively better than on the real system. The high performance observed in 2- and 4-node clusters is due to the short-cutting of inter-node RPCs into local procedure calls, which occurs frequently in small clusters.

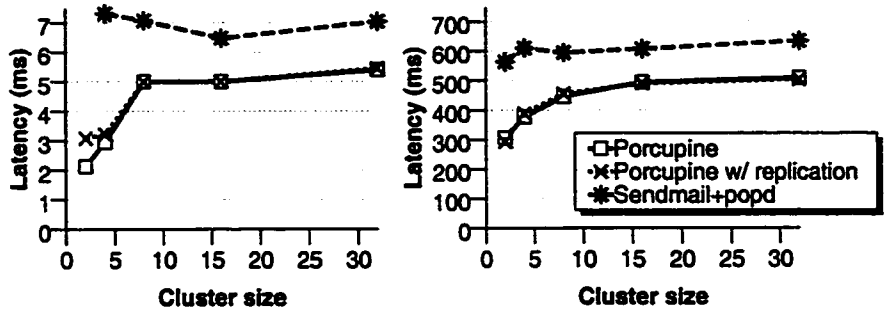
With balanced nodes, the network clearly becomes the bottleneck. In the non-replicated case, each message travels the network four times: (1) Internet to delivery agent (2) to mailbox manager (3) to retrieval agent (4) to Internet. At an average message size of 4.7KB, a 1Gb/second network can then handle about 6500 messages/second. With a single "disk loaded" node able to handle 105 messages/second, roughly 62 ($\approx 6500/105$) nodes will saturate the network as they process 562 million messages/day. With messages replicated on two nodes, the same network can handle about 20% fewer messages (as the message travels five times, because it must be copied one additional time to the replica); this is about 5200 messages/second, or about 450 million messages/day. Using the throughput numbers measured with the faster disks, this level of performance can be achieved with 108 NVRAM nodes, or about 137 nodes without NVRAM. More messages can be handled only by increasing the aggregate network bandwidth or by splitting the system into multiple sub-clusters and localizing traffic within each sub-cluster. We discuss this issue further in Section 9.2.

8.3.4 Email Session Latency

Figure 8.9 shows the average latency of POP sessions that read and delete mailboxes containing either one message (left) or 200 messages (right). We tried to obtain reproducible numbers by fully caching the entire mailbox in the kernel file buffer, keeping the system idle except for these POP sessions, and averaging the middle 40 percentile of 200 samples to exclude extremes.

Overall, the figure shows that session latency is largely unaffected by cluster size. This is reasonable, because latency is determined essentially by the number and speed of cross-node control transfers, which are independent of cluster size. The good performance observed in Porcupine with 2- and 4-node clusters is because of the short-cutting of inter-node RPCs into local procedure calls,

which occurs frequently in small clusters.



(a) One message per mailbox (b) 200 messages per mailbox

Figure 8.9. *Latency of reading and deleting a mailbox. The latency remains mostly constant regardless of cluster size.*

8.3.5 Scaling to a Large User Population

One of the potential downsides of FHC is the memory overhead by the soft state: the space occupied by the profile bank and mail maps grows as the per-node client population or the spread size grows. Using nodes with a less-than-adequate amount of memory thrashes the nodes and severely degrades the overall performance. Figure 8.10 shows Porcupine's memory requirement for various user populations and spread sizes. We obtained these numbers by filling disks with dummy email messages and measuring the virtual image size of the Porcupine process. Thus, the chart represents the worst-case memory overhead, with messages awaiting every user in the cluster.

The figure shows that a 128-node cluster can support as many as 82 million users using nodes with 165MB of memory with a spread of up to four, or with 265MB of memory with a spread of eight. Given the rapid decline of the memory price, we believe that this memory overhead is reasonable.

A spread of two and four incur the same memory overhead, because Porcupine pre-allocates four entries in mail maps (allocating fewer entries makes no difference due to the overhead in `malloc` itself). Replication also has no effect on memory overhead, because Porcupine pre-allocates the mail map to manage up to three replicas by default.

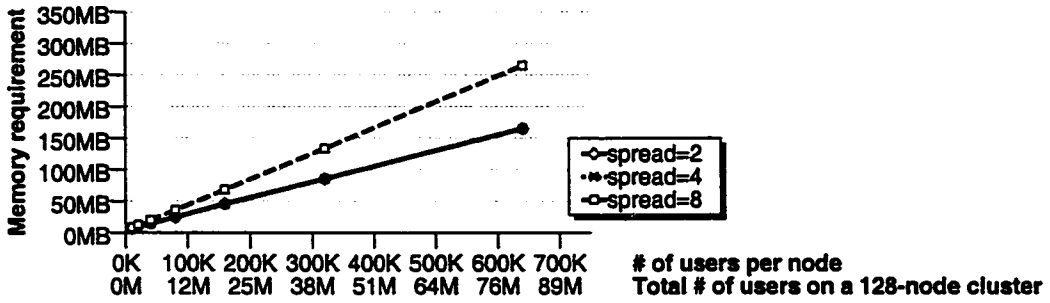


Figure 8.10. *Porcupine's memory requirement when managing various numbers of users. The first row of the X-axis shows the number of users supported per node, and the second row shows the total user population that could be supported in a 128-node cluster. The spread of two and four incur exactly the same memory overhead, because Porcupine preallocates four entries in mail maps.*

8.4 Adapting to Non-uniform Environments

The previous sections demonstrated Porcupine's performance assuming a uniform workload distribution and homogeneous node performance. In practice, though, workloads are not uniformly distributed, and the speed of nodes differs. This can create substantial management challenges for system administrators when they must reconfigure the system manually to adapt to the load and configuration imbalance. This section shows how Porcupine automatically handles workload skew and heterogeneous cluster configurations.

8.4.1 Adapting to Workload Skew

Figure 8.11 shows the impact of Porcupine's dynamic, spread-limiting, load-balancing strategy on throughput as a function of workload skew for our 30-node configuration (all with a single slow disk). Both the non-replicated and replicated cases are shown. Skew along the X-axis reflects the inherent degree of balance in the incoming workload. When the skew equals zero, recipients are chosen so that the hash distributes uniformly across all buckets. When the skew is one, the recipients are chosen so that they all hash into a single user map bucket, corresponding to a highly imbalanced workload.

The graphs compare random, static, and dynamic load balancing policies. The random policy, labeled R on the graph, simply selects a host at random to store each message received; it has the

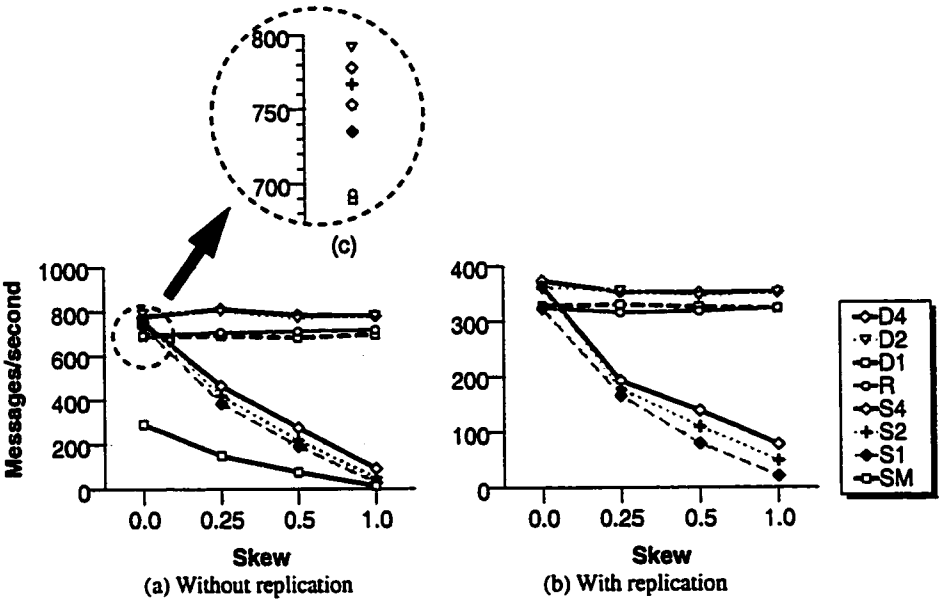


Figure 8.11. *Throughputs on a 30-node system with various degrees of workload skew. Graph (c) shows a close-up view of the non-replicated throughputs under a uniform workload. Lines labeled "D" use Porcupine's dynamic load balancing (numbers behind D show the spread size), whereas lines labeled "S" use load balancing based on static partitioning. R chooses nodes at random. SM is sendmail+popd.*

effect of smoothing out any non-uniformity in the distribution. The static spread policy, shown by the lines labeled S1, S2, and S4, selects the least-loaded node based on a hash of the user name spread over 1, 2 or 4 nodes, respectively. The dynamic spread policy – the one used in Porcupine – selects from those nodes already storing fragments for the recipient (Section 7.1.3). It is shown as D1, D2 and D4 on the graph. Again, the spread value (1, 2, 4) controls the maximum number of nodes (in the absence of failure) that store a single user's mailbox.

Static spread manages affinity well but can lead to an imbalanced load when activity is concentrated on just a few nodes. Indeed, a static spread of one (S1) corresponds to our conventional configuration in which users are statically partitioned to different machines. This effect is shown as well on the graph for the conventional sendmail+popd configuration (SM on Figure 8.11). In contrast, the dynamic spread policy continually monitors load and adjusts the distribution of mail over the available machines, even when spread is one. In this case, a new mailbox manager is chosen for a user each time his/her mailbox is emptied, allowing the system to repair affinity-driven imbalances as necessary.

The graphs show that random and dynamic policies are insensitive to workload skew, whereas static policies do poorly unless the workload is evenly distributed. Random performs worse than dynamic because of its inability to balance load and its tendency to spread a user's mail across many machines.

Among the static policies, those with larger spread sizes perform better under a skewed workload, since they can utilize a larger number of machines for mail storage. Under a uniform workload, however, the smaller spread sizes perform slightly better, since they respect affinity. The key exception is the difference between spread=1 and spread=2. At spread=1, the system is unable to balance load. At spread=2, load is balanced and throughput improves. Widening the spread beyond two improves balance slightly, but not substantially, as was discussed in Section 7.1.3.

The effect of the affinity loss with larger spread sizes is not pronounced in the Linux ext2 file system, because it creates or deletes files without synchronous directory modification [153]. On other operating systems (e.g., the traditional UNIX fast file system [102]), load-balancing policies with larger spread sizes will be penalized more by the increased frequency of directory operations.

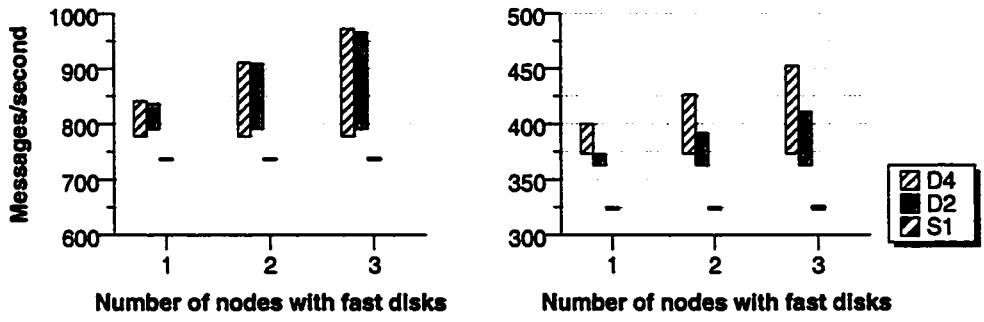
8.4.2 *Effect of the Spread Size on Load Balancing*

Figure 8.11 (c) shows system throughput under a uniform workload. It is interesting to see that FHC's load balancing service can improve system performance even when the workload is uniform. D4, D2, S4 and S2 all perform well with statistically insignificant differences. S1, which emulates a statically partitioned cluster, performs about 5 to 10% worse than the rest because of the lack of load balancing. Under a uniform workload, the load balancing service improves the performance mainly by avoiding nodes that are undergoing periodic buffer flush activities (`bdflush`), which stall all other disk I/O operations for a few seconds. R and D1 both perform about 15 to 20% worse, but for different reasons. R performs worse because it lacks load balancing and ignores message affinity. D1 performs worse because it lacks load balancing and tends to overload a few nodes that happen to host hyper-active users. On the other hand, D2 and D4 host hyper-active users on multiple nodes, and the load balancer can split the workload at a fine grain to keep the load on these nodes low.

8.4.3 *Adapting to Heterogeneous Configurations*

As mentioned in Section 8.3.3, the easiest way to improve throughput in our configuration is to increase the system's disk I/O capacity. This can be done by adding more machines or by adding more or faster disks to a few machines. In a statically partitioned system, it is necessary to upgrade the disks on all machines to ensure a balanced performance improvement. In contrast, because of Porcupine's functional homogeneity and automatic load balancing, we can improve the system's *overall* throughput for all users simply by improving the throughput on a few machines. The system will automatically find and exploit the new resources.

Figure 8.12 shows the absolute performance improvement of the 30-node configuration when adding two fast SCSI disks to each of one, two, and three of type "B" nodes (Table 8.1), with and without replication. The improvement for Porcupine shows that the dynamic load balancing mechanism can fully utilize the added capacity. Here, a spread of four (D4) slightly outperforms a spread of two (D2), because the former policy is more likely to include the faster nodes in the candidate set for each new fragment. This result confirms the discussion in Section 7.1.3: when a few nodes are much faster than the rest, as is the case with our setting, the spread size needs to be



(a) *Without replication* (b) *With replication*

Figure 8.12. *Performance improvement by the Porcupine load balancing mechanism. The bottom of each bar shows the performance of the baseline system with a particular load balancing mechanism. The height of the bar shows the relative improvement over the baseline system, when two fast SCSI disks are added to each of one, two, or three nodes.*

increased.

In contrast, the statically partitioned policy (S1) demonstrates little improvement with the additional disks. This is because their assignment improves performance for only a subset of the users.

8.5 Recovering from Configuration Changes

This section investigates how Porcupine reacts to node failures and recoveries. We first observe the throughput transition during node failures and recoveries. Next, we look closely at the events that unfold during reconfiguration and show that the recovery protocol will be able to scale to clusters far larger than the ones we built. Finally, we study, through simulation, the performance of our membership protocol for stabilizing the membership in large and unreliable clusters.

8.5.1 Throughput Transition during Configuration Changes

As described previously, Porcupine automatically reconfigures whenever nodes fail or restart. Figures 8.13 and 8.14 depict an annotated timeline of events that occur during the failure and recovery of 1, 3, and 6 nodes in a 30-node system without and with replication. Both figures exhibit the

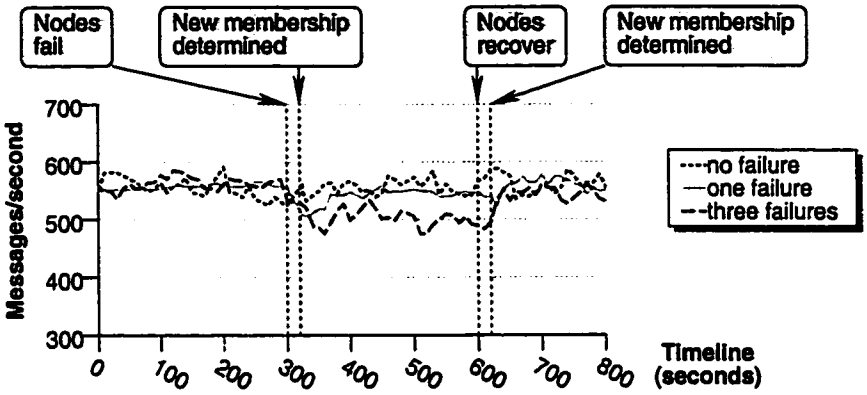


Figure 8.13. *Reconfiguration timeline without replication. In this experiment, one, three, or six nodes fail at time 300 seconds in a 30-node cluster. These nodes recover at time 600 seconds.*

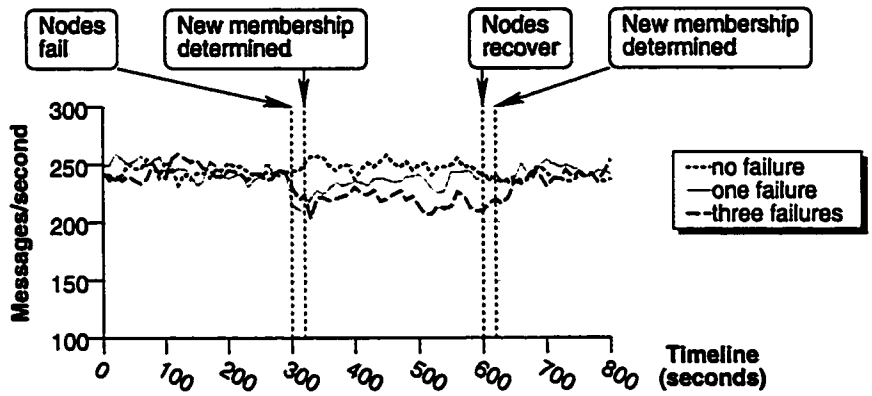
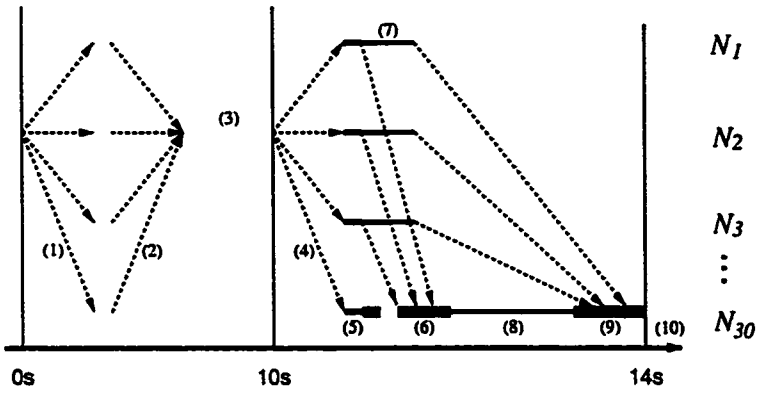


Figure 8.14. *Reconfiguration timeline with replication.*



(1) N_{30} recovers. N_2 starts TRM.
(2) Nodes reply to N_2.
(3) N_2 waits timeout.
(4) N_2 computes the new user map (4ms) and broadcasts the new membership.
(5) — User profile disk scan. 6ms/node.
(6) — User profile updates on N_{30}. 7ms/RPC.
(7) — Fragment scan. 20ms on existing nodes.
(8) — Fragment scan. 3300ms on N_{30}.
(9) — Mail map updates on N_{30}. 7ms/RPC.
(10) Soft state reconstruction completes. Replica reconciliation begins.

Figure 8.15. *The time breakdown of failure recovery procedure. The timeline is not to scale.*

same behavior. Nodes fail and throughput drops as two things occur. First, the system goes through its reconfiguration protocol, increasing its load. Next, during the reconfiguration, SMTP and POP sessions that involve the failed nodes abort. After ten seconds, the system determines the new membership, and throughput increases as the remaining nodes take over for the failed ones. The level of performance degradation during the crash period is proportional to the number of nodes that crashed. Thus, Porcupine achieves one of the recovery goals, graceful performance degradation.

The failed nodes recover 300 seconds later and rejoin the cluster, at which time throughput starts to rise. For the non-replicated case, throughput increases back to the pre-failure level almost immediately, demonstrating Porcupine's ability of graceful performance improvement. With replication, throughput rises slowly as the failed nodes reconcile while concurrently serving new requests.

8.5.2 Reconfiguration Timeline

Figure 8.15 shows the timing of events that unfold during the reintegration of one node (N_{30}) to a 29-node cluster. Overall, fourteen seconds are spent in reconfiguring the membership and recovering the soft state. The first ten seconds are spent in the membership protocol (steps 1 to 4). Ongoing

client sessions are not blocked during this period, and the computational and networking overheads of the membership protocol are minimal. The next four seconds are spent in recovering the soft state (steps 5 to 10). Again, ongoing client sessions on existing nodes are not affected during this period, because the soft state recovery affects nodes other than N_{30} only in a limited way — 6ms to scan the user profile and 20ms to scan fragments. On the other hand, N_{30} must scan its entire email spool to discover fragments and fill the mail maps managed on the other nodes (step 8). In addition, N_{30} must receive from the other nodes the portion of the profile bank and mail maps it manages (steps 6 and 9).

Notice that the cost of disk scan (i.e., steps 7 and 8) is orders of magnitude larger than that of all the other steps combined, and it depends only on the node's disk capacity and not on the number of nodes in the cluster. This confirms the analysis in Section 5.7, i.e., that the cost of Porcupine's failure recovery mechanisms is mostly independent of cluster size.

The crash of a node involves steps similar to the reconfiguration, except that the overhead will be distributed evenly among the nodes. This is because the bucket management change affects each node evenly (in the recovery case, node N_{30} spends a disproportionate amount of time discovering all the fragments in all the buckets).

8.5.3 Cost of Membership Reconfiguration at a Large Scale

One issue that can potentially impede FHC's scaling is the membership agreement, with its cost growing at $O(N^2)$ (Section 5.7.1). This section investigates the cost of membership agreement at a large scale using a discrete event simulator. In particular, we study how improvements to the original TRM protocol (Section 5.3.2) allow the system to stabilize the membership views in unreliable networks.

Figure 8.16 shows the cost of membership agreement per failure or recovery in a lossless network. This graph confirms that the cost per failure grows linearly with cluster size (Section 5.7.1)[3]. For a cluster of size N, Porcupine's membership protocol costs $12.9N$ messages per node for membership agreement. This translates to 2.28 milliseconds of CPU time per node, and $24100N + 12N^2$ bytes of network messages per node (the byte count grows quadratically, because the packet size

[3]The time-averaged cost still grows at $O(N^2)$, because failure frequency grows at $O(N)$.

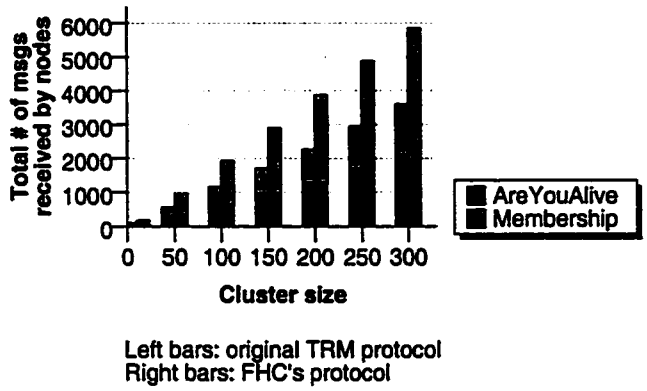


Figure 8.16. *The total number of packets received for membership agreement after a single node crash on a loss-less network. The gray bars represent packets for membership agreement, and the black bars represent packets for failure detection. (Failure detection packets are sent even when a new membership is being formed.)*

grows linearly with cluster size). In contrast, the original TRM costs $5.0N$ messages per node, because it sends each type of message only once. In the big picture, the cost of membership agreement is negligible in either of the protocols, as already demonstrated in Section 5.7.

In fact, the real measure of the effectiveness of membership protocols is not the cost, but the speed of agreement in unreliable networks. As discussed in Section 5.3.2, a badly designed membership protocol can itself destabilize a network by sending more packets than the network can handle. Figure 8.17 shows the cost of the original TRM and Porcupine's protocol in a network that randomly drops 0, 3, 10, or 30% of packets. In such loss-filled networks, the original TRM must rerun the protocol many times, escalating its cost quickly as the loss rate increases. We also tried to simulate a 300-node cluster, but the original TRM was not able to complete — even after one hour — with a non-zero loss rate. In contrast, FHC's packet retransmission mechanism effectively counters packet losses. It completes the membership agreement even on severely loss-filled networks with roughly a constant cost.

8.6 Summary

This chapter showed that Porcupine, our prototype implementation of FHC, accomplishes our three goals of scalability:

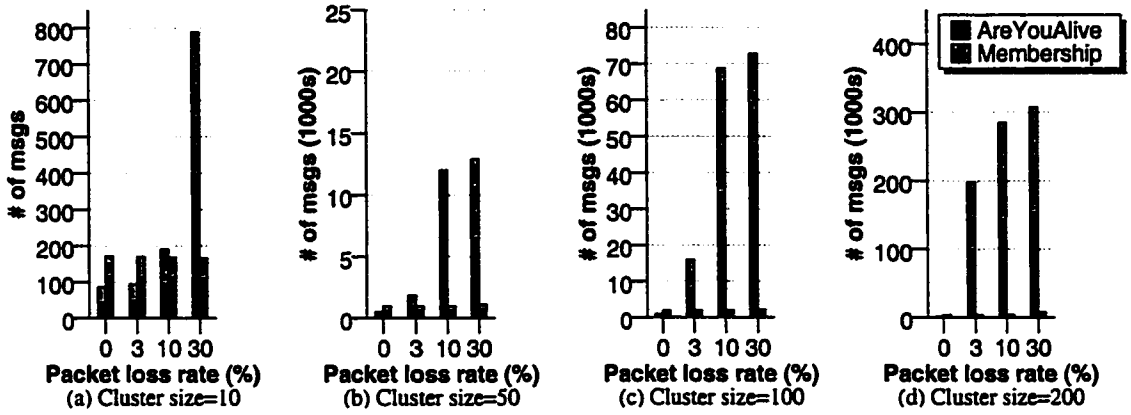


Figure 8.17. *The cost of membership agreement after a single node failure. The X-axis shows the packet loss rate, and the Y-axis shows the total number of membership packets received by the nodes. The left-hand bars show the performance of the original TRM, and the right-hand bars show that of FHC's improved protocol.*

- Porcupine's throughput scales linearly with cluster size, and session latency is constant regardless of cluster size. Porcupine also scales to a large user population with a reasonable memory overhead, allowing a cluster built of today's computers to support tens of millions of users easily.

- Porcupine's load balancing mechanism automatically tunes system behavior to changing environments and improves system manageability. In particular, it masks the skew in incoming workloads, yielding a consistent system throughput even when some users are far more active than others. It also masks the skew in the hardware configuration and improves system performance proportionally as disks are added to a few nodes.

- Porcupine can react to configuration changes quickly. System performance grows or shrinks linearly as the cluster's aggregate capacity changes. The cost of reconfiguration stays small even for large clusters. The overhead imposed by the replication service is modest: it slows the system due to increased disk activity, but otherwise it performs as expected. Its space and networking overheads are negligible.

Chapter 9 Discussions and Future Work

In the course of designing and implementing the FHC architecture, we found several interesting issues that elude satisfactory solutions in the current framework and warrant additional investigation. This chapter overviews some of them and presents opportunities for future work.

9.1 Limitations to Scaling

FHC's architecture and implementation have been designed to run well in very large clusters. There are, however, some aspects of its design and the environment in which it is deployed that may need to be rethought as the system grows to larger configurations.

First, FHC's communication patterns are flat, with every node as likely to talk to every other node. As discussed in Section 8.3.3, a 1Gb/second switched network should be able to serve about 6500 messages/second (or 560 million messages/day) without replication. With replication, the network can handle 5200 messages/second, or 450 million messages/day. Beyond that, faster networks or more network-topology-aware load balancing strategies will be required to continue scaling.

Our strategy for reconstructing the soft state may need to be revisited for large clusters. This dissertation assumes independent failures (Section 4.5.1), which may not hold in a truly large cluster. The frequency of perceived failures, especially network partitioning, may increase hyper-linearly as the number of routers and bridges increase. Our network-partition recovery process is relatively expensive, because it forces each partition to discard much of its soft state and recover from the hard state (e.g., when a cluster splits into two partitions and later merges, half of the entire soft state is transferred across nodes after both partitioning and merging). As another problem, when nodes join and leave the cluster incessantly, the membership protocol cannot let nodes agree on the membership view, destabilizing the entire soft-state management scheme. The next section presents a proposal, federated clusters, for solving these problems.

Our soft-state reconstruction scheme may also require change when a single user manager manages millions of users (many users, few machines). Rather than transferring the user profile soft state

in bulk, as we do now, we could modify the system to fetch profile entries on use and cache them. This would reduce node recovery time (possibly at the expense of making user lookups slower, however).

9.2 Geographical Distribution

This dissertation has assumed that nodes are connected by a fast network that rarely fails (Section 4.5.1). An interesting extension to FHC is to split a cluster into geographically separated sub-clusters. Geographical distribution is already popular in large Internet sites, such as Yahoo, AltaVista, and Google. It lets a service overcome catastrophic failures (e.g., power blackout) and utilize wide-area networking resources more efficiently by serving users from nearby regions. Geographical distribution has not been common in data-intensive services, however, because of the difficulty of transparently locating and routing data across regions. Large data-intensive Internet services today either centralize all resources in one place (e.g., AOL) or just build a logically isolated set of clusters in each region and statically assign each user to a region.

There is a natural affinity between FHC and geographical distribution, because FHC is already designed to tolerate most of the failure types that the Internet experiences (Section 4.5.1). The Internet, however, is slower and less reliable than what FHC assumes. For example, as discussed in the previous section, FHC's flat communication structure leads to a large volume of cross-region traffic if used naively. FHC's soft-state recovery mechanism will also become prohibitively expensive, because wide-area distribution dramatically increases the rate of perceived failures.

One way to solve these problems is to employ *federated clusters*, in which several small clusters form a federation to route users and messages across them. Figure 9.1 shows an example. Here, each user is assigned to a specific geographic region. The assignment is semi-static, e.g., based on the user's residence. We create a user database that holds the names and regions of *all* users hosted by the service. This database is split into buckets, like the profile database, but we place a few replicas of each bucket in every region to ensure that the user can be routed from any region to another region. Thus, to run a session for a particular user, a proxy first consults the global user database and identifies the region to which the user belongs. If the user is local, the proxy starts a local session as described in Section 4.4. Otherwise, it forwards the session to (any node in) the

user's region.

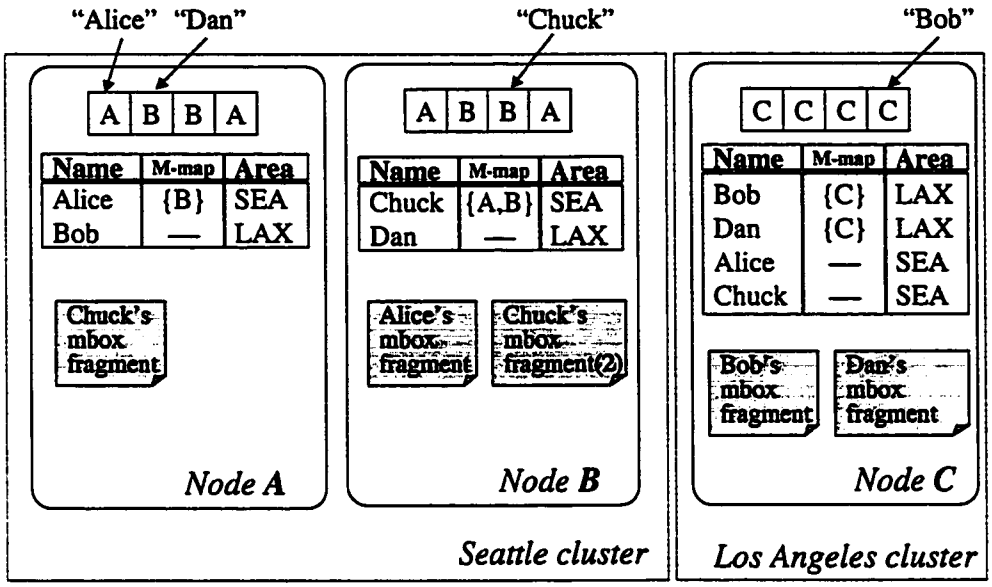


Figure 9.1. *Example of data distribution in federated clusters. One cluster, located in Seattle (SEA), contains two nodes, A and B, and the other cluster, located in Los Angeles (LAX), contains one node, C. Users Alice and Chuck are assigned to the Seattle cluster, and users Bob and Dan are assigned to the LA cluster (by the administrator). Each cluster in a geographical area knows the names and locations of all users, but it only manages those users in the area.*

This architecture raises many interesting issues. Replica placement is one such issue. The load balancing service needs to satisfy two contradictory requirements: for better performance, all replicas should be placed within a single region, but for better availability, they should be in different regions [19]. Placing a user's data in multiple regions complicates failure recovery, because cross-area links must be maintained in the user's mail map. Simply running the current version of the membership and soft-state reconstruction protocols globally in all regions destabilizes the system. Rather, we need to develop a lazier protocol that delays recovering soft state until the long-haul network becomes stable.

User migration is another open issue. When a user moves from one region to another, the system should detect the movement and migrate the users' profile and fragments to the new region. Detecting such a movement requires a light-weight mechanism that can detect the distance between

two end points. Few studies currently exist of this problem [54, 62]. Deciding when it is economical to move the user's data is a variation of file allocation problem (Section 3.4.3), but most previous work has made simplified assumptions about the environment — e.g., constant networking cost and reliable and timely communication.

9.3 Supporting a Wider Variety of Services

FHC takes advantage of the semantics of applications to scale without sacrificing its quality of service. The downside of this philosophy is that a part of the architecture must be changed for every new application. So far, Porcupine is the only service we developed on this architecture, with mature email support and experimental support for WWW, FTP, and BBS.

This section outlines the strategies for extending the range of applications for FHC. We first discuss data-intensive services that can naturally be supported by our current architecture. We then discuss services that are not amenable to the current FHC architecture and present several ideas for supporting them.

9.3.1 Data-intensive Services

The most application-specific component in FHC is its naming service. The naming service is a product of compromises between simplicity and memory overhead, and between common-case performance and recovery speed. The foremost challenge in designing a new service in the FHC framework is deciding what to place in the name database.

For most data-intensive Internet services, the design of the name database is straightforward, and many of the components in Porcupine can be reused. For example, in a personal calendar service, users and their schedule naturally correspond to users and their email messages. Thus, to build the calendar service, one needs to write a storage manager that reads and write events as well as proxies that interact with clients. Other components, such as the profile manager, the load balancer, and the replication manager, will be reused. As another example, in BBS services, newsgroups and articles correspond to users and email messages. To build such a service, one must write a storage service for reading and writing articles and proxies for protocols such as NNTP [81]. User profile management requires a small change; login names, passwords, etc., are replaced with information

pertinent to newsgroups, e.g., article sequence counters and expiration period.

9.3.2 *Large Name Spaces*

As demonstrated in Section 8.3.5, FHC's name service can manage about 1 million users per node using today's PCs. Supporting a much larger name space is difficult in the current framework. A general-purpose web server is an example of services that demand such a large, flat name space. HTTP allows clients to request pages (URLs) directly without any prior "data discovery," as required in POP or NNTP. An FHC-based web server, if constructed naively, needs to register every page it manages directly in the name database as the soft state; however, this solution bloats the soft state and slows the failure recovery process. Alternative designs, such as to group web pages by authors and register only the authors' names and the list of nodes that store their pages (our prototype web server is built this way), will work only for sites that group pages by authors; it will slow the system by forcing clients to discover pages every time a URL is requested.

One future extension to FHC is to provide a persistent name space that can manage an arbitrary number of objects [71]. This feature sacrifices some of the advantages of FHC, such as robust recovery from any number of failures, but it widens FHC's applicability. Highly available persistent naming mechanisms have been studied actively, but most of them offer single-copy consistency by assuming reliable hardware and do not offer dynamic replica-set changes as are required in our environment (Section 3.2). We believe that this problem can be solved by applying our replication algorithm to naming. Here, we treat the pair ⟨*item-name*, *item-location*⟩ as a "name object" and apply our replication algorithm to add, delete, or update object names and locations. Combining such a replicated name database with hash routing (Section 3.4.4) will provide a scalable and dynamic naming database suitable for large PC-based clusters.

9.3.3 *Retrofitting FHC to Legacy Applications*

Writing a new FHC service admittedly requires a large amount of engineering effort, because one must re-factor service components into the functionally homogeneous framework. This involves deciding the type of information managed in the name database and the format of on-disk data, splitting the service into proxies and on-disk data managers, and connecting them by remote procedure calls.

Thus, writing a service in FHC often means writing most of the service from scratch, with little reuse of existing software. For example, in Porcupine, the only component adopted from existing software is the IMAP request parser from UW IMAPD [116], with a reuse rate of 15% (Appendix A.4). Even the reused components required extensive modifications to make them thread-safe and to use remote procedure calls for calling other modules.

One interesting research topic is to study how FHC can be retrofitted into existing services. Although it is the synergy of FHC's mechanisms that enables FHC high scalability, each technique we developed in the dissertation could be applied separately to improve existing systems. For example, the membership service could be added to any cluster-based services for scalable system monitoring. The replication service could be used in a statically partitioned web storage back-end to improve its availability without expensive hardware investments.

9.4 Running Multiple Services in a Cluster

This dissertation assumes that the entire cluster is dedicated to a single service, e.g., email in Porcupine. Often in practice, however, an organization needs to run multiple services simultaneously — e.g., email, WWW, News, and advertisement management — each of which requires a substantial amount of computational resources. Using the current FHC framework naively, the administrator is forced to partition nodes among these services manually, re-introducing the problems with statically partitioned clustering discussed in Section 1.3.2. The following sections discuss several issues that need to be resolved to run multiple services in a single FHC cluster.

9.4.1 Load Balancing in Mixed-resource Environments

FHC's load balancing service currently uses the number of pending disk-related RPC requests as the measure of a node's load, assuming that the disk is the sole performance bottleneck in the system (Section 7.1.1). While we believe that this assumption is mostly valid for data-intensive Internet services, it is not when the cluster runs multiple types of services, some of which are CPU-, memory-, or network-bound. In fact, the problem may potentially arise even in Porcupine. For example, with remote DNS caching, a few nodes in the cluster can be chosen to handle all the client sessions (that are CPU bound) from a large remote site. The current load balancing service may

assign many disk I/Os to these nodes and slow down all client sessions.

A new load balancing policy is needed for workloads that stress all of the CPU, the network, memory, and disks. This area is still in an infantile stage, and prior work handles only the simple cases in which the resource requirements and speedup characteristics of tasks are known beforehand [126, 26, 135]. Such an approach will work poorly for Internet workloads characterized by many, often unknown, performance parameters and short-lived sessions. The system needs to measure workload characteristics automatically and assign the best mix of tasks to each node for better throughput.

Earlier in this project, we studied a scheme that converts the resource requirement of a task (e.g., storing a fragment on disk or interacting with an SMTP client) into the "time" needed to complete the task and balances load using the total estimated time the node needs to complete pending tasks. The "time" here is not just virtual CPU time. We tried to account for the time spent in interrupt handlers or in I/O wait (i.e., when the workload is disk-intensive, the CPU unavoidably waits for I/O completion) by charging each active task an even slice of the idle and interrupt time. We eventually abandoned this scheme, because it failed to make a significant improvement over the simpler scheme (Section 7.1), but we believe that this line of study warrants future investigation.

9.4.2 Distributed Scheduling

This dissertation did not delve deeply into the issue of task scheduling. Porcupine currently deploys ad-hoc techniques (or just prays) to keep resource contention from disrupting client sessions. For example, the rebalancer runs during the night when the level of user activity is generally low (Section 7.2.3), but this may not always be true. In fact, anecdotal evidence shows that truly large sites, such as AOL and Hotmail, are only slightly less busy during the night. As another example, Porcupine currently propagates an update to an on-disk object immediately to all live replicas to bring them up-to-date as quickly as possible (Section 6.6.4), but it should propagate in the background when the system is busy. Finally, data-retrieval sessions should be given priority over other types of sessions, because the former require real-time responses, but we currently do nothing in this regard.

One potential solution to this problem is to use a priority-based scheduler, already standard in most operating systems, to give client sessions a higher priority than others. However, because most

schedulers rely on CPU usage to guide their decisions, this solution will not work well in disk-bound workloads. Moreover, activity in the cluster — e.g., a client's session or the rebalancer — usually involves multiple nodes. A single-node priority scheduling mechanism cannot ensure the quick completion of an important activity.

Another interesting approach is to apply the end-to-end argument to scheduling [138, 37]. Network protocols, such as TCP, and distributed scheduling have something in common: both try to utilize a distributed set of resources (i.e., routers and network links or CPUs and disks) without knowing the internal structure of the resources. TCP's feedback-based progress control could be applied to distributed scheduling, as well. Here, instead of scheduling tasks independently on each node, this approach would let high-level activities (e.g., the rebalancer or a mail delivery session) monitor their own progress and regulate their speed so as not to disrupt more important tasks. This idea has been applied to the scheduling of long-running tasks within a node [50], but we believe that its application to distributed environments is a promising research topic.

9.5 Improving Programming Productivity

The FHC architecture is prone to deadlocks and starvations, because it makes every node a consumer of resources on other nodes (Appendix A.1). These problems are among the most difficult to diagnose, because they happen only occasionally in a large cluster under a heavy load. The problem is in some part fundamental to the functionally homogeneous architecture: every resource that may cause a shortage requires a solution similar to segregated thread pools (discussed in Appendix A.1). Unfortunately, it is not easy even to identify such resources (e.g., in-kernel process control blocks are not usually considered a resource in textbooks), much less solve the problem.

A part of the problem, however, is attributable to the lack of tools for distributed systems programming. For example, for deadlock analysis, we had to manually sort through post-mortem log files spread over thirty nodes to learn what caused the system lockup. Automated tools for detecting deadlocks and other anomalies in clusters would have made our life far easier. Distributed monitoring and debugging is actually an active area of study, but it has so far failed to show practical impact, because it was either too theoretical or confined in uncommon programming environments. We believe that recent advances in software diagnosing techniques, such as binary rewriting and dynamic

kernel extension, could be combined to build practical tools for distributed system monitoring and debugging.

9.6 Handling Uncommon Failure Modes

Past studies of distributed systems usually made a simplified assumption about failures: nodes always fail by stopping (fail-stop failure model). This assumption has been popular, because it is mostly true in practice, is convenient for designers, and is amenable to formal analysis.

One of our goals in FHC is to provide practical solutions for failures that fall outside of the fail-stop model. Long-term failure is one problem we handle (Section 6.4.5). Disk fill-ups and sector crashes are other failure modes that are common in practice, are a major source of administrative overhead, and yet are addressed by few systems. FHC provides a framework for solving these problems by allowing data to be dynamically migrated to other nodes without sacrificing data consistency.

However, we are only partially satisfied with our solutions. First, our solutions are not end-to-end. That is, while they provide the mechanisms for rectifying the anomalies, they do not cover issues such as detecting disk failures, discovering the extent of damages, and deciding on the fate of the failed disk (e.g., discard the entire drive, reformat and reuse, etc.). Second, they are ad-hoc. We still do not know if long-term failures and disk crashes cover all the failures that practical systems face. We hear anecdotes about the types of failures that happen in real systems, but we lack a comprehensive taxonomy of failures, much less the formalism of failures. We need a simple and comprehensive mechanism for handling all practical failure modes automatically.

Notice that a universal solution to handling any type of failures does exist in the form of Byzantine consensus protocols [88, 27, 28]. These protocols, however, have not been (and are unlikely to be) accepted in practice because of their high cost (Section 4.5.1).

Chapter 10 Conclusions

This thesis presented a new software architecture, functionally homogeneous clustering (FHC), for building large-scale, data-intensive services that scale in three dimensions: manageability, availability, and performance. This architecture achieves our goals by dynamic and automatic distribution of data and functions among cluster nodes.

We developed three key techniques for realizing this architecture. The automatic recovery mechanism recovers the name service that keeps track of the locations of users and their data after configuration changes. The replication mechanism keeps multiple replicas of on-disk objects consistent with little computational and space overhead. Finally, the load balancing mechanism distributes workloads dynamically among nodes and hides skew in the incoming workload and the system configuration.

Our architecture scales without sacrificing service quality by taking advantage of the semantics of applications. This strategy exhibits itself in various aspects of the key mechanisms. All the state in the name service is stored only in memory and is recomputed after failure from on-disk data. While this design makes the contents and operations of the name database application-specific, it makes the system simple and robust. Our replication algorithm also takes advantage of application semantics and ensures data consistency only in the steady state. In return, it makes the system extremely resilient against failures.

We developed the Porcupine email server as proof of the concept of functionally homogeneous clustering. Porcupine distributes the user management and storage of email messages dynamically to maximize system throughput and to ensure that users are never denied the service. It replicates the user profile and email messages on multiple nodes to ensure continued service to users. We evaluated the manageability, availability, and performance of Porcupine on a 30-node PC cluster. Through the evaluation, we showed that Porcupine's performance indeed scales perfectly and that it reacts to configuration changes gracefully and quickly. We also showed that Porcupine's load balancing service efficiently utilizes a heterogeneous set of hardware resources by automatically

discovering idle resources in the cluster.

The three techniques we developed can all be traced back to the existing body of studies, especially in the areas of distributed coordination, optimistic replication, and load balancing. It is the synergy of these mechanisms that solves the problems that existing clustering architectures could not avoid. We believe that FHC has made important contributions to the construction of scalable and self-managing systems, but, as in any valid research, our work raises as many questions as it answers. Studying these questions will further advance us toward the ultimate goal of understanding large-scale distributed systems.

Bibliography

[1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *17th Symp. on Operating Systems Principles (SOSP)*, pages 186–201, Kiawah Island, SC, December 1999. 3.5.2

[2] Noha Adly. *Management of Replicated Data in Large Scale Systems*. PhD thesis, Corpus Cristi College, University of Cambridge, August 1995. 3.6.3

[3] Divyakant Agrawal, Amr El Abbadi, and R. C. Steike. Epidemic algorithms in replicated databases. In *16th ACM Symp. on Princ. of Database Systems (PODS)*, pages 161–172, Tucson, AZ, May 1997. 3.6.3

[4] Yair Amir. *Replication using group communication over a partitioned network*. PhD thesis, Hebrew University of Jerusalem, 1995. 3.5.1, 3.6.1

[5] Anthony Anderberg. History of the Internet and Web. http://www.anderbergfamily.net/ant-/history/, December 1999. 1.1

[6] Darrell C. Anderson, Jeffrey S. Chase, and Amin M. Vahdat. Interposed Request Routing for Scalable Network Storage. In *4th Symp. on Operating System Design and Implementation (OSDI)*, pages 259–272, San Diego, CA, October 2000. 1.3.5

[7] Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randy Wang. Serverless Network File Systems. In *15th Symp. on Operating Systems Principles (SOSP)*, Copper Mountain, CO, December 1995. 1.3.5, 4, 3.2, 3.4.4

[8] Mohit Aron, Darren Sanders, Peter Druschel, and Willy Zwaenepoel. Scalable Content-aware Request Distribution in Cluster-based Network Servers. In *USENIX Annual Technical Conference*, 2000. 1.2, 3.4.2

[9] Baruch Awerbuch, Yair Bartal, and Amos Fiat. Competitive Distributed File Allocation. In *25th ACM Symp. on the Theory of Computing (STOC)*, October 1993. 3.4.3

[10] Gaurav Banga, Fred Douglis, and Michael Rabinovich. Optimistic deltas for WWW latency reduction. In *USENIX Annual Technical Conference*, Anaheim, CA, 1997. 6.6.3

[11] A. Barak and O. La'adan. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Journal of Future Generation Computer Systems*, 13(4–5):361–372, March 1998. 3.4.1

[12] Philip Bernstein and Eric Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, 1997. 1.2, 3.6.1, 5.5.5, 6.6.5

[13] Phillip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987. 1.2, 3.3.2, 3.5.3

[14] A.K. Bhide, E.N. Elnozahy, and S.P. Morgan. A highly available network file server. In *USENIX Winter Technical Conference*, pages 199–205, January 1991. 3.2

[15] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Trans. on Computer Systems (TOCS)*, 5(1):272–314, February 1987. 3.6.1

[16] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. on Computer Systems (TOCS)*, 9(3):47–76, August 1991. 3.6.1

[17] Andrew D. Birrell, Andy Hisgen, Chuck Jerian, Timothy Mann, and Garret Swart. The Echo Distributed File System. Technical Report 111, DEC Systems Research Center, Palo Alto, CA, September 1993. 3.2

[18] Bluetail mail robustifier. http://www.blutail.com/bmr, 2000. 3.1

[19] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. In *ACM SIGMETRICS*, pages 34–43, Santa Clara, CA, June 2000. 1, 7.1.4, 9.2

[20] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf distribution: evidence and implications. In *IEEE INFOCOM*, pages 126–134, New York, NY, March 1999. 1.2

[21] Eric A. Brewer. Lessons from Internet Services: ACID vs. BASE. In *NSF Workshop on Industrial/Academic Cooperation in Database Systems*, Los Gatos, CA, November 1998. 1.2, 3.3.2

[22] Thomas P. Brisco. RFC1794: DNS support for load balancing. http://info.internet.isi.edu/in-notes/rfc/files/rfc1794.txt, April 1995. 4.2

[23] Michael Burrows et al. Personal communication, about the architecture of altavista., August 1998. 3.3.2

[24] Rajkumar Buyya. *High performance cluster computing: Volume 1, Architectures and systems*. Prentice Hall, 1999. 3.4.1

[25] Todd Campbell. The first email message. http://www.pretext.com/mar98/features/story2.htm, March 1998. 1.1

[26] Michael Carey and Hongjun Lu. Load balancing in a locally distributed database system. In *ACM SIGMOD International Conference on Management of Data*, pages 108–119, 1986. 9.4.1

[27] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *3rd Symposium on Operating System Design and Implementation (OSDI)*, New Orleans, LA, February 1999. 7, 9.6

[28] Miguel Castro and Barbara Liskov. Proactive Recovery in a Byzantine-Fault-Tolerant system. In *Symp. on Operating System Design and Implementation (OSDI)*, pages 273–288, San Diego, CA, October 2000. 7, 9.6

[29] B. Chandra, M. Dahlin, L. Gao, and A. Nayate. End-to-end WAN service availability. In *3rd USENIX Symp. on Internet Technologies and Systems (USITS)*, March 2001. 1.2, 5.6

[30] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685–722, July 1996. 3.2, 3.6.1

[31] Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg, and Bernadette Charron-Bost. On the impossibility of group membership. Technical Report TR95-1548, Cornell University, October 1995. 3.5.1

[32] Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, March 1996. 3.2, 3.6.1

[33] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A Hierarchical Internet Object Cache. In *USENIX Winter Technical Conference*, January 1996. 1, 1.2, 3.6.1

[34] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, June 1994. 1.3.1

[35] Nick Christenson, Tim Bosserman, and David Beckemeyer. A Highly Scalable Electronic Mail Service Using Open Systems. In *USENIX Symp. on Internet Technologies and Systems (USITS)*, Monterey, CA, December 1997. 2.1

[36] Cisco Systems. Local director. http://www.cisco.com/warp/public/751/lodir/index.html, 1999. 1.2, 3.4.2, 4.2

[37] David D. Clark. The design philosophy of the DARPA Internet protocols. In *ACM SIGCOMM*, pages 106–114, Stanford, CA, August 1988. 2.4.2, 9.4.2

[38] Internet Software Consortium. INN: InterNetNews. http://www.isc.org/products/INN/, January 2001. 1.3.2

[39] Marc Crispin. RFC2060: Internet message access protocol version 4 rev 1. http://info-.internet.isi.edu/in-notes/rfc/files/rfc2060.txt, December 1996. 2.5.1, 4.4.1, 6.6.3

[40] Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, pages 642–657, June 1999. 3.5.1, 4.5.1

[41] Flaviu Cristian and Frank Schmuck. Agreeing on processor group membership in asynchronous distributed systems. Technical Report CSE95-428, UC San Diego, 1995. 3.5.1, 5.3, 5.3.2, E.1

[42] Michael Dahlin. Interpreting Stale Load Information. In *The 19th International Conference on Distributed Computing Systems (ICDCS)*, Austin, TX, May 1999. IEEE. 3.4.1, 7.1.1, 20

[43] Om P. Damani, P. Emerald Chung, Yennun Huang, Chandra Kintala, , and Yi-Min Wang. ONE-IP: Techniques for hosting a service on a cluster of machines. In *Proc. of the Sixth Int. World Wide Web Conference*, April 1997. 3.3.2, 3.4.2

[44] Dean Daniels, Lip Boon Doo, Alan Downing, Curtis Elsbernd, Gary Hallmark, Sandeep Jain, Bob Jenkins, Peter Lim, Gordon Smith, Benny Souder, and Jim Stamos. Oracle's symmetric replication technology and implications for application design. In *ACM SIGMOD International Conference on Management of Data*, page 467, Minneapolis, MN, May 1994. 3.6.1

[45] Delphi.com. http://www.delphi.com, December 2000. 3.1

[46] Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, and John Larson. Epidemic algorithms for replicated database maintenance. In *6th ACM Symp. on Princ. of Distr. Computing (PODC)*, pages 1–12, Vancouver, BC, August 1987. 2, 3.6.3, 13, 6.1, 6.2

[47] James Deroest. Clusters help allocate computing resources. http://www.washington.edu-/tech_home/windows/issue18/clusters.html, 1996. 2.1, 3.1

[48] Daniel J. Dietterich. DEC data distributor: for data replication and data warehousing. In *ACM SIGMOD International Conference on Management of Data*, page 468, Minneapolis, MN, May 1994. ACM. 3.6.1

[49] D. Dolev, D. Malki, and R. Strong. An Asynchronous Membership Protocol that Tolerates Partitions. Technical report, Institute of Computer Science, The Hebrew University of Jerusalem, March 1994. 3.5.1

[50] John R. Douceur and William J. Bolosky. Progress-based Regulation of Low Importance Processes. In *17th Symp. on Operating Systems Principles (SOSP)*, pages 247–258, Kiawah Island, SC, December 1999. 9.4.2

[51] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. on Software Engineering*, 12(5):662–675, May 1986. 3.4.1, 7.1.1, 20

[52] EMC^2. Celerra file server. http://www.emc.com/techlib/, December 2000. 1.3.3, 3.3.1

[53] Michael M. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, Henry M. Levy, and Chandramohan A. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *15th Symp. on Operating Systems Principles (SOSP)*, pages 130–146, Copper Mountain, CO, December 1995. 3.4.4

[54] Zongming Fei, Samrat Bhattacharjee, Ellen W. Zegura, and Mostafa H. Ammar. A novel server selection technique for improving the response time of a replicated service. In *IEEE INFOCOM*, San Francisco, CA, March 1998. 9.2

[55] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC2616: Hypertext transfer protocol – HTTP/1.1. http://info.internet.isi.edu/in-notes/rfc/-files/rfc2616.txt, June 1999. 1.2, 3.5.2

[56] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985. 2, 3.2, 3.6.1

[57] Sally Floyd, Van Jacobsen, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Journal on Networking*, pages 784–803, December 1997. 5.3.2

[58] Foundry Networks. ServerIron Switch. http://www.foundrynet.com/serverironfspec.html, 1999. 1.2, 3.4.2, 4.2

[59] Armando Fox. *A Framework for Separating Server Scalability and Availability from Internet Application Functionality*. PhD thesis, UC Berkeley, 1998. 1

[60] Armando Fox and Eric A. Brewer. Harvest, Yield, and Scalable Tolerant Systems. In *Workshop on Hot Topics in Operating Systems (HOTOS)*, Rio Rico, AZ, March 1999. 1.2, 2.4.1

[61] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-Based Scalable Network Services. In *16th Symp. on Operating Systems Principles (SOSP)*, pages 78–91, St. Malo, France, October 1997. 1, 1.2, 2.4, 2.4.2, 3.3.2, 3.4.2, 3.5.1, 3.5.2

[62] Paul Francis, Sugih Jamin, Vern Paxson, Lixia Zhang, Daniel F. Gryniewicz, and Yixin Jin. An Architecture for a Global Internet Host Distance Estimation Service. In *IEEE INFOCOM*, New York, NY, March 1999. 9.2

[63] David K. Gifford. Weighted voting for replicated data. In *13th Symp. on Operating Systems Principles (SOSP)*, pages 150–162, Pacific Grove, CA, 1979. 3.6.1

[64] Richard A. Golding. *Weak-consistency group communication and membership*. PhD thesis, UC Santa Cruz, December 1992. 3.6.2

[65] Richard A. Golding. Modeling replica divergence in a weak-consistency protocol for global-scale distributed data bases. Technical Report UCSC-CRL-93-09, UC Santa Cruz, February 1993. 2

[66] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986. 7.2.1

[67] Jim Gray. A census of Tandem system availability between 1985 and 1990. *IEEE Trans. on Reliability*, 39(4):409–418, October 1990. 7.2.1

[68] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan-Kaufmann, 1993. 1.2, 3.3.2, 3.5.3, 6.6.5, A.3.3

[69] Steven D. Gribble. *A Design Framework for a Scalable Storage Platform to Simplify Internet Service Construction*. PhD thesis, UC Berkeley, 2000. 1

[70] Steven D. Gribble and Eric A. Brewer. System Design Issues for Internet Middleware Services: Deduction from a Large Client Trace. In *USENIX Symp. on Internet Technologies and Systems (USITS)*, Monterey, CA, December 1997. 1.2

[71] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, distributed data structures for Internet service construction. In *4th Symp. on Operating System Design and Implementation (OSDI)*, pages 319–332, San Diego, CA, October 2000. 1.2, 2.4, 2.4.1, 4, 3.2, 9.3.2, A.1

[72] Mor Harchol-Balter, Tom Leighton, and Daniel Lewin. Resource Discovery in Distributed Networks. In *18th ACM Symp. on Princ. of Distr. Computing (PODC)*, pages 229–237, April 1999. 6.4.4

[73] Maurice Herlihy. A quorum-consensus replication method for abstract data types. *ACM Trans. on Computer Systems (TOCS)*, 4(1):32–53, February 1986. 3.6.1

[74] Hewlett-Packard. HP-UX high availability. http://www.unix.hp.com/highavailability/, November 2000. 1.3.3, 3.3.1

[75] IBM. *High Availability Cluster Multi-Processing for AIX*, 1998. Available at http://www-.rs6000.ibm.com/doc_link/en_US/a_doc_lib/aixgen/hacmp_index.html. 1.3.3, 3.3.1

[76] IBM. Disk storage systems. http://www.storage.ibm.com/hardsoft/disk/disk.htm, December 2000. 1.3.3, 3.3.1

[77] Jupiter Communications Inc. E-mail marketing to soar to $7.3 billion in 2005, cannibalizing 13 percent of direct mail revenues, May 2000. http://www.jup.com/company/pressrelease-.jsp?doc=pr000508. 1

[78] Gregory Johnson and Ambuj Singh. Stable and fault-tolerant object replication. In *19th ACM Symp. on Princ. of Distr. Computing (PODC)*, Portland, OR, June 2000. 3.4.3

[79] Steven Jones. Personal communication. Some of the information is available at http://staff-.washington.edu/noyd/projects/network-news/news.htm, December 2000. 2.1, 3.1

[80] Jakob Kaivo. GNU POP3 daemon. http://www.nodomainname.net/software/mailutils/, 1998. 3.1, 8.1

[81] Brian Kantor and Phil Rapsey. RFC977: Network news transfer protocol. http://info.internet-.isi.edu/in-notes/rfc/files/rfc977.txt, February 1986. 1.2, 2.5.1, 3.6.3, 9.3.1

[82] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *ACM Symp. on the Theory of Computing (STOC)*, pages 654–663, El Paso, TX, 1997. 3.4.4, 5.4

[83] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. on Computer Systems (TOCS)*, 10(5), February 1992. 3.6.2

[84] J. Klensin, N. Freed, M. Rose, E. Stefferud, and D. Crocker. RFC1869: SMTP service extensions. http://info.internet.isi.edu/in-notes/rfc/files/rfc1869.txt, November 1995. 1.2, 4.4.1

[85] Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker. VAXclusters: A closely-coupled distributed system. *ACM Trans. on Computer Systems (TOCS)*, 4(4):130–146, 1986. 1.3.3, 3.3.1, 3.5.1, 3.6.1

[86] Leslie Lampont. The part-time parliament. *ACM Trans. on Computer Systems (TOCS)*, 16(2):133–169, May 1998. 3.5.3

[87] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978. 1, 16

[88] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine generals problem. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, July 1982. 7, 9.6

[89] Butler W. Lampson. How to build a highly available system using consensus. *Distributed Algorithms, LNCS 1151*, pages 1–17, 1996. http://www.research.microsoft.com/lampson/58-Consensus%5CAbstract.html. 3.5.3

[90] Edward K. Lee and Chandramohan Thekkath. Petal: Distributed virtual disks. In *7th International Conf. on Architectural Support for Prog. Lang. and Operating Systems (ASPLOS)*, pages 84–92, Cambridge, MA, October 1996. 3.2, 3.5.3

[91] Y.W. Lee, K.S. Leung, and M. Satyanarayanan. Operation-based update propagation in a mobile file system. In *USENIX Annual Technical Conference*, Monterey, CA, June 1999. 3.6.2

[92] Kurt Lidl, Josh Osborne, and Joseph Malcolm. Drinking from the firehose: Multicast USENET news. In *USENIX Winter Technical Conference*, 1994. ftp://ftp.uu.net/networking/news/muse/usenix-muse.ps.gz. 3.6.3

[93] Linux HA, 2000. http://linux-ha.org. 3.5.1

[94] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp File System. In *13th Symp. on Operating Systems Principles (SOSP)*, pages 226–238, Pacific Grove, CA, October 1991. 3.2, 16

[95] Barbara Liskov, Liuba Shrira, and John Wroclawski. Efficient at-most-once messages based on synchronized clocks. *ACM Trans. on Computer Systems (TOCS)*, 9(2):125–142, 1991. 6.2, 6.4.3

[96] Witold Litwin, Marie-Anne Neimat, and Donovan A. Schneider. RP*: A family of order preserving scalable distributed data structures. In *International Conf. on Very Large Data Bases (VLDB)*, pages 342–353, Santiago, Chile, 1994. 3.2

[97] Witold Litwin, Marie-Anne Neimat, and Donovan A. Schneider. LH* — a scalable, distributed data store. *ACM Trans. on Database Systems (TODS)*, 21(4):480–525, December 1996. 3.2

[98] Witold Litwin and Thomas Schwarz. A high-availability scalable distributed data structure using Reed-Solomon codes. In *ACM SIGMOD International Conference on Management of Data*, pages 237–248, Dallas, TX, May 2000. 3.2

[99] David Lomet. Replicated indexes for distributed data. In *Proceedings of Conference on Parallel and Distributed Information Systems*, pages 108–119, 1996. 3.2

[100] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996. 2, 1

[101] Mail.com. Email market. Web page, March 2000. http://corp.mail.com/mediakit/market-.html. 1.1, 2.1

[102] Marshal Kirk McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Trans. on Computer Systems (TOCS)*, 2(3):181–197, August 1984. 1, 8.4.1

[103] Microsoft. *Windows 2000 Server Resource Kit*. Microsoft Press, 2000. 4, 3.6.3, 6.1, 6.2

[104] Sun Microsystems. Sun Internet mail server. http://www.iplanet.com/products/infrastructure-/messaging/sims, November 2000. 3.3.1

[105] David L. Mills. RFC1305: Network time protocol (version 3). http://info.internet.isi.edu/in-notes/rfc/files/rfc1305.txt, March 1992. 16, 6.9

[106] David L. Mills. Improved algorithms for synchronizing computer network clocks. In *ACM SIGCOMM*, pages 317–327, London, UK, September 1994. 16, 6.9, D.1

[107] Michael Mitzenmacher. How useful is old information? In *16th ACM Symp. on Princ. of Distr. Computing (PODC)*, Santa Barbara, CA, August 1997. 3.4.1, 19, 7.1.1, 20

[108] Michael Mitzenmacher. On the Analysis of Randomized Load Balancing Schemes. In *The 9th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 292–301, 1997. 3.4.1, 20

[109] P. V. Mockapetris. RFC1035: Domain names — implementation and specification. http://info.internet.isi.edu/in-notes/rfc/files/rfc1035.txt, November 1987. 4, 6.1

[110] P. V. Mockapetris and K. Dunlap. Development of the domain name system. In *ACM SIGCOMM*, Stanford, CA, August 1988. 4, 6.1

[111] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996. 3.6.1

[112] Lily B. Mummert, Maria R. Ebling, and M. Satyanarayanan. Exploiting Weak Connectivity for Mobile File Access. In *15th Symp. on Operating Systems Principles (SOSP)*, pages 143–155, Copper Mountain, CO, December 1995. 3.6.2

[113] John G. Myers and Marshall T. Rose. RFC1939: Post office protocol version 3. http://info-.internet.isi.edu/in-notes/rfc/files/rfc1939.txt, May 1996. 2.5.1, 4.4.1

[114] NetworkAppliance. Filer storage appliances. http://www.netapp.com/products/filer/, December 2000. 1.3.3, 3.3.1

[115] F5 Networks. Big IP. http://www.f5.com, March 2001. 3.4.2

[116] University of Washington. The IMAP connection. http://www.imap.org, December 2000. 3.1, 4.4.1, 8.1, 9.3.3, A.4

[117] Oracle. *Oracle7 Server Distributed Systems Manual, Vol. 2*, 1996. 1.2, 1.3.4, 3.6.1

[118] M. Tamer Özsu and Patric Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1999. 1.3.4

[119] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-aware Request Distribution in Cluster-based Network Servers. In *8th International Conf. on Architectural Support for Prog. Lang. and Operating Systems (ASPLOS)*, pages 206–216, San Jose, CA, October 1998. 1, 1.2, 3, 3.3.2, 3.4.2, 3.4.4

[120] M. Patiño-Martinez, R. Jímenez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *14th International Conference on Distributed Computing (DISC)*, pages 315–329, Toledo, Spain, October 2000. 3.6.1

[121] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *16th Symp. on Operating Systems Principles (SOSP)*, pages 288–301, St. Malo, France, October 1997. 2, 3.6.2

[122] Gregory F. Pfister. *In Search of Clusters, 2nd Ed.* Prentice Hall, January 1998. 1.3.2, 1.3.3, 2.1, 3.3.1

[123] Brian Pinkerton. *WebCrawler: Finding what people want.* PhD thesis, University of Washington, November 2000. Available at the author's home page. 3.3.2

[124] Jonathan Postel. RFC821: Simple mail transfer protocol. http://info.internet.isi.edu/in-notes/-rfc/files/rfc821.txt, August 1982. 1.2, 4.4.1

[125] M. Rabinovich, I. Rabinovich, and R. Rajaraman. Dynamic replication on the Internet. Technical Report HA6177000-980305-01-TM, AT&T Labs, March 1998. 3.4.3

[126] Erhard Rahm and Robert Marek. Dynamic multi-resource load balancing in parallel DB systems. In *International Conf. on Very Large Data Bases (VLDB)*, 1995. 9.4.1

[127] S. Raman and S. McCanne. A model, analysis, and protocol framework for soft state-based communication. In *ACM SIGCOMM*, Cambridge, MA, September 1999. 2.4, 3.5.2

[128] David H. Ratner. *Roam: A Scalable Replication System for Mobile and Distributed Computing*. PhD thesis, UC Los Angeles, 1998. UCLA-CSD-970044. 2, 3.6.2

[129] Keith Regan. Four web sites control half of surfing time. E-Commerce Times, June 2001. http://www.ecommercetimes.com/perl/story/10222.html. 1

[130] Resonate, Inc. Central Dispatch. http://www.resonate.com/products/central_dispatch/, 1998. 3.4.2

[131] A. M. Riccardi and K. P. Birman. Using process group to implement failure detection in asynchronous environments. In *10th ACM Symp. on Princ. of Distr. Computing (PODC)*, pages 341–352, Montreal, Quebec, August 1991. 3.5.1

[132] Thomas L. Rodeheffer and Michael D. Schroeder. Automatic Reconfiguration in Autonet. In *13th Symp. on Operating Systems Principles (SOSP)*, pages 183–187, Pacific Grove, CA, October 1991. 3.5.4

[133] Thomas L. Rodeheffer, Chandramohan A. Thekkath, and Darrell Anderson. Smartbridge: A scalable bridge architecture. In *International Conf. on Architectural Support for Prog. Lang. and Operating Systems (ASPLOS)*, 2000. 3.5.4

[134] Keith W. Ross. Hash routing for collections of shared web caches. *IEEE Network*, November/December 1997. 3

[135] Emilia Rosti, Guiseppe Serazzi, Evgenia Smirni, and Mark Squillante. The impact of I/O on program behavior and parallel scheduling. In *ACM SIGMETRICS*, Madison, WI, June 1998. 9.4.1

[136] Yasushi Saito. Optimistic replication algorithms. General examination report. Available at the author's web page, May 2000. 2.4.3

[137] Yasushi Saito, Jeffrey Mogul, and Ben Verghese. A Usenet performance study. http://www-.research.digital.com/wrl/projects/newsbench/, September 1998. 1.1, 1.2, 3.6.3, 6.2, 7.2.1

[138] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Trans. on Computer Systems (TOCS)*, 2(4):277–288, November 1984. 9.4.2

[139] Russel Sandberg, David Goldberg, Steve Tleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network File System. In *USENIX Summer Technical Conference*, pages 119–130, 1985. 1.3.2, 3.1, 3.5.2

[140] Michael D. Schroeder, Andrew D. Birrell, and Roger M. Needham. Experience with Grapevine: The growth of a distributed system. *ACM Trans. on Computer Systems (TOCS)*, 2(1):3–23, February 1984. 3.1

[141] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *USENIX Annual Technical Conference*, June 2000. 1

[142] Sendmail. home page. http://www.sendmail.org, 1998. 1.3.2, 3.1, 8.1

[143] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. RFC3010: NFS version 4 protocol. http://info.internet.isi.edu/in-notes/rfc/files/rfc3010.txt, December 2000. 1.3.2, 3.1, 3.5.2

[144] Ashish Singhai, Swee Lim, and Sanjay R. Radia. The SCALR framework for Internet servers. In *The 28th International Symposium on Fault-Tolerant Computing (FTCS)*, Munich, Germany, June 1998. IEEE. 3.3.2, 3.4.2, 3.5.1

[145] William E. Snaman and David W. Thiel. The VAX/VMS distributed lock manager. *Digital Technical Journal*, (5), 1987. 3.2, 3.4.4, 3.5.1

[146] Sleepycat Software. The Berkeley database. http://sleepycat.com. A.3.2

[147] Sun Microsystems. Sun directory services 3.1 administration guide. http://www.sun.com-/sims/Docs-4.0/html/sunds/admin/, July 1998. 6.1

[148] Sun Microsystems. *Sun Cluster Architecture*, 1999. Available at http://www.sun.com/clusters/wp-clusters-arch.pdf. 1.3.3, 3.3.1

[149] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *15th Symp. on Operating Systems Principles (SOSP)*, pages 172–183, Copper Mountain, CO, December 1995. 2

[150] Chandramohan Thekkath, Timothy Mann, and Edward Lee. Frangipani: A Scalable Distributed File System. In *16th Symp. on Operating Systems Principles (SOSP)*, pages 224–237, St. Malo, France, October 1997. 1.3.5, 3.2

[151] Robert Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Systems (TODS)*, 4(2):180–209, June 1979. 3.6.1, 5.5.5, 6.2

[152] Jon William Toigo. *The Holy Grail of Data Storage Management*. Prentice Hall, 2000. 1.3.3, 3.3.1

[153] Theodore Ts'o. Ext2 home page. http://web.mit.edu/tytso/www/linux/ext2.html. 1, 8.1, 8.4.1, A.3.1

[154] Vinod Valloppillil and Keith W. Ross. Cache array routing protocol v1.0. Internet draft, February 1998. http://www.ircache.net/Cache/ICP/carp.txt. 3, 3.4.4

[155] R. van Renesse, Y. Minsky, and M. Hayden. A Gossip-Style Failure Detection Service. In *Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, 1998. 3.5.1

[156] Paul Vixie et al. RFC2136: Dynamic updates in the domain name system (DNS UPDATE). http://info.internet.isi.edu/in-notes/rfc/files/rfc2136.txt, April 1997. 4.2

[157] Werner Vogels, Dan Dumitriu, Ken Birman, Rod Gamache, Mike Massa, Rob Short, John Vert, Joe Barrera, and Jim Gray. The Design and Architecture of the Microsoft Cluster Service — A Practical Approach to High-availability and Scalability. In *28th International Symposium on Fault-Tolerant Computing*, pages 422–431, Munich, Germany, June 1998. IEEE. 1.3.3, 3.3.1, 3.5.1

[158] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. In *15th Symp. on Operating Systems Principles (SOSP)*, pages 96–108, Copper Mountain, CO, December 1995. 3.5.4

[159] Mark Wistrom. Microsoft Exchange 2000 Clustering. http://msdn.microsoft.com/library/-techart/clustering.htm, May 2000. 3.3.1

[160] O. Wolfson and S. Jajodia. Distributed algorithms for dynamic replication of data. In *11th ACM Symp. on Princ. of Database Systems (PODS)*, pages 149–163, San Diego, CA, 1992. 3.4.3

[161] Alastair Wolman, Geoff Voelker, Nitin Sharma, Neil Cardwell, Anna Karlin, and Henry Levy. On the scale and performance of cooperative Web proxy caching. In *17th Symp. on Operating Systems Principles (SOSP)*, Kiawah Island, SC, December 1999. 1, 1.2

[162] Gary Write and Richard Stevens. *TCP/IP Illustrated, Volume 2*. Addison Wesley, 1993. A.2.1

[163] Gene T. J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *3rd ACM Symp. on Princ. of Distr. Computing (PODC)*, pages 233–242, Vancouver, Canada, August 1984. 3.6.3

[164] Haifeng Yu and Amin Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *4th Symp. on Operating System Design and Implementation (OSDI)*, pages 305–318, San Diego, CA, October 2000. 2.4.1

[165] Y. Zhang, V. Paxon, and S. Shenkar. The stationarity of Internet path properties: routing, loss and throughput. Technical report, ACIRI, May 2000. 1.2, 5.6

Appendix A Implementation of Porcupine

This Appendix describes implementation aspects of the Porcupine email server. In particular, it focuses on three issues that heavily influence the system's structure and performance: multi-threading and its implications for functionally homogeneous clustering, inter-node communication, and storage management.

Figure 4.2 shows the major components that comprise a Porcupine node. Most of the components are linked into a single process that handles client interaction, load balancing, user management, email message storage and user profile storage. The rebalancer (Section 7.2) is implemented as a separate process that accesses the node's data via a local-domain socket. We chose this design because the rebalancer runs infrequently, and it needs to keep its own persistent state, e.g., the number of days a node has remained failed and the group of users it has scanned the last time it ran (Section 7.2.2). Several other functions, including status monitoring and the watchdog[1], also run as separate processes.

A.1 Multi-threading Architecture

Porcupine is a multi-threaded server. One kernel thread is assigned for each email session or remote procedure call request. Several threads additionally handle background maintenance tasks, such as soft state delivery (Section 5.5), retirement notification propagation (Section 6.4.3), timer scheduling, and node status monitoring. We chose the multi-threaded architecture over the event-driven architecture [71] mainly for its ease of programming. We also believe that multi-threading is faster on UNIX for heavily disk-bound workloads, because it can hide the latency of synchronous operations such as `open(2)`, `fsync(2)`, and buffer cache misses.

The functionally homogeneous architecture affects multi-threading in a subtle way: it makes every node a potential consumer of resources — particularly threads — on other nodes, thereby

[1]The watchdog continuously monitors the cluster and restarts dead Porcupine processes.

rendering the system vulnerable to deadlocks. For example, consider the following scenario: 300 SMTP proxy threads on node *A* remotely call node *B* to store messages and consume 300 RPC threads on *B*; at the same time, 300 POP proxy threads on node *B* remotely call node *A* to read messages and consume 300 RPC threads on *A*. If the operating system supports only up to 512 threads, which is the default with Linux-2.2, the system deadlocks. We solve this problem by dividing threads statically into three classes and ensuring that a thread in a more important class is never starved by those in less important classes. Specifically:

- Class-0 threads interact with clients directly. For example, SMTP proxies run in class 0.
- Class-1 threads handle RPC requests from remote class-0 threads. For example, RPCs such as "authenticate user" and "store email message" run in class 1.
- Class-2 threads handle RPC requests from remote class-0 or -1 threads. For example, the RPC "update a replica" runs in class-2 because it is called by a class-1 thread handling a "store email message" RPC.

This class hierarchy can be extended to an arbitrary depth, but three turns out to be enough for Porcupine. For each class, a thread pool is created with a static capacity. Furthermore, the sum of the sizes of the pools is set not to exceed the operating system limit. Thus, this scheme breaks the resource dependency cycle among nodes by partitioning the thread resource statically. This "static partitioning", however, is free from the managerial problems discussed in Section 1.3.2, because devising a proper pool size is easy. Multi-threading is known to yield diminishing returns beyond a certain limit [71]. On Linux, we observed that system throughput reaches a plateau with five threads with IDE disks and with forty threads with fast SCSI disks. Setting the capacity of each pool to a value larger than 40 (say, 100) will avoid deadlocks and yield good performance under any workload.

A similar solution would be required for every type of resource that may run out in a cluster (e.g., memory and file descriptors); however, in our experience, threads were the only resource that actually caused a shortage. Improving the programming productivity in FHC is our future work, as we discussed in Section 9.5.

A.2 Inter-node Communication

Nodes communicate in two ways in FHC. First, they communicate through the membership management module (Section 5.3) to exchange the node's status and to converge the membership views within the cluster. The membership protocol uses UDP because it uses broadcasts to discover nodes. The use of UDP in the membership service is straightforward (see Appendix C.1 for a description of the algorithm) and is not discussed further in this section.

The second type of communication is remote procedure calls (RPCs) that happen during the course of an email session. RPC messages are carried over TCP channels because they require reliable point-to-point communication. The remainder of this section first discusses the low-level networking mechanism. It then presents the high-level specification of the RPC interface.

A.2.1 Low-level Communication

Porcupine opens a single TCP channel for each class of RPCs (Appendix A.1) for each pair of nodes. The channel is long-lived, created when the nodes boot and lasting until they crash. Threads wait in queues on both sides of the channel to multiplex remote procedure calls over a single channel. On the server side, the thread at the head of the queue reads a request, detaches itself from the queue (and lets the next thread in the queue fetch the next request), processes the request, sends the reply on the same channel, and adds itself to the queue. On the client side, a thread issuing a request waits in the queue to allow others to issue requests while it awaits the reply.

The use of a single TCP channel for each class and pair of nodes offers several advantages over alternatives, such as the use of a proprietary transport protocol or the creation of a single TCP channel for each RPC request. First, this design is simple, because TCP already offers reliable and fast communication over a wide variety of media. This approach also detects process crashes quickly by catching TCP shutdown events (membership-based crash detection is still needed, because TCP cannot quickly detect node crashes). The only downside is that this scheme limits cluster size to 2^{16}, which is the size of the TCP port space [162]. This limitation is practically a non-issue, though, because Porcupine can handle even the largest conceivable workload with just a few hundred nodes (Chapter 8).

A.2.2 Remote Procedure Calls

Remote procedure calls between nodes are described using an interface definition language (IDL) developed for Porcupine. The IDL supports a CORBA-like syntax for remote procedure calls, but no object orientation features. The IDL compiler generates efficient code that tightly integrates with the RPC manager.

Figure A.1 shows the procedures for user management. Figure A.2 shows the procedures for fragment storage management, and Figure A.3 shows the procedures for replica consistency management.

A.3 Storage Management

A.3.1 Email Spool Management

Porcupine uses the standard UNIX file system (ext2 [153] on Linux) for storing email messages. Mailboxes are stored in two types of files. A *contents file* stores the actual message contents, and it is created for each fragment (i.e., the group of messages for a user sharing the same replica set). An *index file* contains the digest of messages (file offset, size, date, sender, message ID, etc.). Only one index file is created for all the user's messages on a node, because the replication manager requires a message to be located from its ID without the replica-set information (Section 6.3 and Figure A.3). On the other hand, one content files is created per fragment, because proxies usually read or modify data at the unit of fragments (Appendix A.2.2).

The contents file and the corresponding index file are updated non-atomically to reduce the number of synchronous disk operations. When a fragment is to be updated, only the contents file is forced to the disk (using `fsync(2)`) before acknowledging to the client. The index file is modified only in the buffer cache and is written to disk lazily. Thus, a node crash may render the index file inconsistent with the contents file. This inconsistency is detected by embedding a "fingerprint" in the index file, which is the combination of the last modification time and the size of the contents file. When the fingerprint in the index file differs from that of the contents file, the index file is recreated by scanning all the contents files for the same user.

As described in Section 5.5.2, the contents and the index files are grouped into directories cor-

```
struct fragment_name {
    user_name user;
    mbox_name mbox;
    replica_set peers;
};
struct profile_update {
    fragment_name frag;
    memdb_modify_operation_t op; // ADD or DELETE
    timestamp ts; // Used to serialize updates (Section 5.5.5).
};

struct mbox_info {
    mbox_name name;
    mail_map map;
};

// Proxy -> User manager:
// First, authenticate the USER by PASSWORD. If successful, find
// the mailboxes for USER and return their names and mail maps.
proc memdb_lookup(in user_name user, in string passwd,
         out sequence<mbox_info> profile);

// Proxy -> User manager: add or delete node NID from the mail
maps in U.
proc memdb_modify(in node_id nid, in sequence<memdb_update> u);

// Recovery module -> profile manager:
// This is used to push mail maps to a new profile manager.
proc memdb_mbox_recovered(in timestamp ts, in long bucket,
         in sequence<fragment_name> list);

struct diskdb_entry {
    user_type_t type;
    user_name name;
    string passwd;
    ...
};

// Recovery module -> profile manager:
// Used to push the profile bank to a new profile manager.
proc diskdb_recovered(in timestamp ts, in long bucket,
         in sequence<diskdb_entry> info);
```

Figure A.1. *Remote procedure definitions for accessing user profile and mail maps. Procedures memdb_{lookup, update, modify} are used by proxies to either find or change user profile or mail maps. The last two procedures are called during soft state recovery.*

```
// Get the list of messages stored in the MBOX_NAME.
proc mbox_list(in fragment_name frag, out rpc_envelopes messages,
                 in long flag);
// Get the body of the message ID.
proc mbox_retrieve(in fragment_name mbox, in message_id id,
    in mail_location_hint loc, in long flag, out mail_body body);
// Write back the FRAG's status info. If EXPUNGE is true and
// some msgs are marked for deletion, they are really deleted.
proc mbox_write(in fragment_name frag, in rpc_envelopes mails,
                 in boolean expunge);
// Append the list of MSGS to the FRAG.
proc mbox_append(in fragment_name frag, in new_message_list msgs);
```

Figure A.2. *Remote procedure definitions for accessing fragments. The IMAP and POP proxies first retrieve the digest of a fragment by calling `mbox_list`, and they then obtain the message body by calling `mbox_retrieve`. When messages are to be deleted or updated, the proxies call `mbox_write`. Procedure mbox_append is used by the SMTP proxy to deliver a message to the node. Procedures `mbox_write` and mbox_append may in turn call the replication manager to update the objects, which in turn call procedure `replica_update`.*

```
struct rpc_update_t {
    replica_oid oid;
    replica_set peers; // nodes to replicate the object
    replica_set targets; // nodes that should receive the update.
    replica_set remaining; // nodes that have not sent acks.
    timestamp ts; // identifies the update.
    long type; // UPDATE or RETIRE
    short manager_id; // Identifies the object type.
    char need_logging_p; // Some hints to the replication manager.
    char creating_new_object_p;
    body_chunk data; // Actual object contents.
};
struct rpc_update_reply_t {
    long result; // Ok or not.
    replica_set targets; // The set of nodes to be added.
};

// Used to send updates between different nodes.
proc update_replica(in node_id src, in sequence<rpc_update_t> u,
                    out sequence<rpc_update_reply_t> r);
```

Figure A.3. *A single procedure, `replica_update`, is called by the replication manager to invoke the same manager on another node.*

responding to the user's hash values, allowing fragments in a bucket to be discovered by a quick directory scan.

Figure A.4 summarizes how user "Ann"'s mailbox might be stored. Here, we assume that her hash value is 5 and that she owns two fragments: one fragment, replicated on nodes *A* and *B* and containing messages *X* and *Y*, and the other, replicated on nodes *A* and *C* and containing messages *Z* and *W*. Notice that node *A* stores two contents files, but only one index file containing entries for all of Ann's messages.

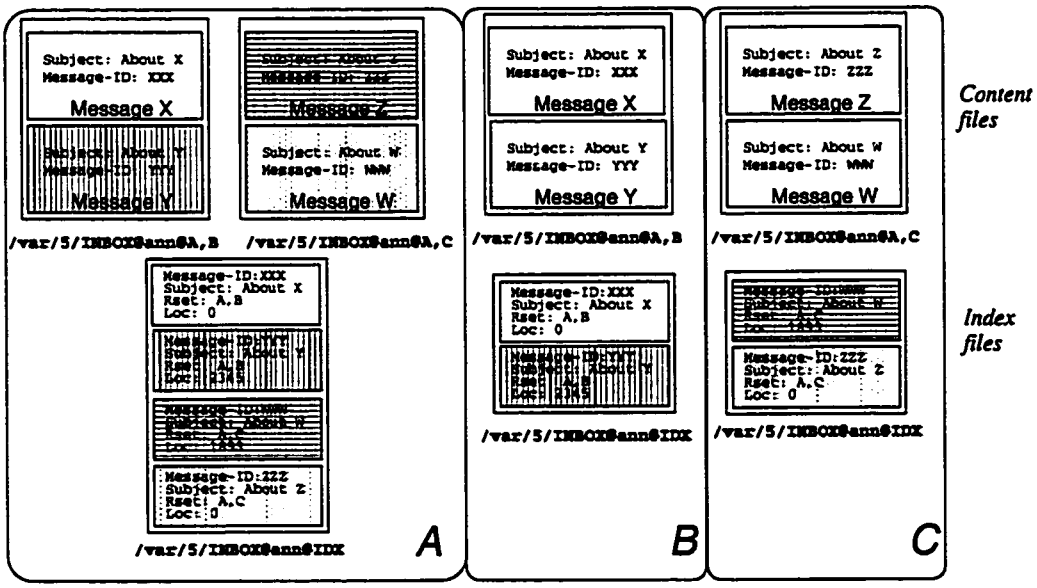


Figure A.4. *This picture shows how Ann's mailbox may be stored on files. We assume that the string "Ann" hashes to 5 and she owns two fragments: one, replicated on nodes A and B, containing messages X and Y, and the other, replicated on nodes A and C, containing messages Z and W. All the files are placed under the directory corresponding to Ann's hash value (5). One contents file is created for each replica set ({A,B} and {A,C}) on each node, and one index file is created for all Ann's messages on each node.*

A.3.2 Profile Database Management

As already discussed in Sections 2.5.1 and 7.2, the profile database is partitioned into buckets, which are (sometimes) partitioned into groups. Each node manages a set of groups designated by

the administrator (which is often a random designation).

Just like fragments, each profile group is composed of two types of files: the contents and the index files. The contents file contains the profile of all users in the group in a flat text. Deleting a user from the file simply creates a space-filled hole, which is currently compacted off-line. The index file, implemented using the Berkeley DB package [146], hash-maps user names to their locations in the contents files on the node. To update a user's profile, the user database manager first consults the index file to obtain the location, and it then seeks into the location in the contents file. For example, suppose bucket number 15 is split into two groups, one replicated on nodes A and B, and the other replicated on nodes A and C. Then, node A will store contents files "`/var/spool/prof/15.A-B`" (replicated on A and B) and "`/var/spool/prof/15.A-C`" (replicated on A and C), and index file "`/var/spool/prof/15.idx`" that locates all users in both the contents files.

This design facilitates efficient profile recovery (Section 5.5.1) by letting the node simply send the contents file to the profile manager verbatim, while still supporting random accesses for quick profile update.

A.3.3 Replication

The update record table (Sections 6.4.2 and 6.6.1) is implemented in an in-memory table backed up by an on-disk redo log [68]. Before an update is applied to an object, it is written to the in-memory table and to the log, which is flushed to disk using the `fsync(2)` system call. When a node recovers from a crash, it scans the log and recreates updates in the table. To keep the log size small, a snapshot of the table is periodically flushed to disk, after which the log is truncated.

Currently, only update records are written to the log, and the objects themselves (e.g., email messages) are flushed separately. This design creates several, albeit small, windows in which the replicas may be corrupted. First is the use of the `fsync(2)` system call to flush the contents to disk. This system call does not ensure atomic file update; in particular, a node crash in the middle of an inode update may corrupt the file permanently. Second, the two separate `fsync(2)` calls to flush the update record and the replica contents may leave the two files inconsistent with each other when the node crashes in between the two calls. The first window of vulnerability can be closed

by using a transactional storage service, such as a journaling file system or a relational database management system. The second window can be closed by flushing the object changes as well as update records to the log, as is usually done in database systems.

A.4 Module Size Breakdown

The Porcupine implementation consists of seventeen major components written in C++. Most of the code was written from scratch, except for the IMAP request parser, which was borrowed from the UW imapd [116]. Table A.1 shows the breakdown of the modules. The total system size is about 42,300 lines of code, yielding a 1MB executable.

Table A.1. Module size breakdown of Porcupine.

Module	Number of lines
Membership and user map management	2000
Replication	3100
RPC support	2000
Mail map management	1200
IDL description	200
IDL compiler	1000
User profile database storage	1500
Rebalancer	800
Load balancer	400
Others	13000
Generic service total	25200
IMAP support	5300
POP support	700
SMTP support	1100
Email storage management	4500
External mail delivery	1600
Clients (diagnostics, spool repair, etc.)	4000
Email service total	17200
Total	42400

Appendix B Coding convention

This chapter presents the coding style used to describe FHC's various protocols.

- The code describes what runs on a single node. Each node stores its own copy of global variables. Procedures on different nodes execute in parallel.

- Types are written in **bold**, variables are written in *oblique*, and procedures are written in the normal typeface.

- Type *NodeID* stands for the names of a node, which is usually the node's IP address. Constant *Me* represents the name of the node itself.

- Type **Set**⟨**Foo**⟩ stands for the collection of type **Foo**. Duplicate values are removed.

- Code blocks are marked by indentation (cf., Occam and Python).

- Each procedure executes non-preemptively until completion.

- Procedures marked "**periodic** *PERIOD*" are called periodically every *PERIOD* seconds.

- Procedure Send(*node*, *procedure*, *args*...) sends a message to *node* and requests calling the *procedure* with *args*.... If *node* is "BROADCAST", the message is sent to all nodes in the cluster (e.g., using Ethernet broadcast). Send does not wait for *proc* to finish; it merely queues the message. Moreover, the message may be delayed, reordered, lost, or duplicated.

- Procedure SendReliably(*node*, *proc*, *args*...) is similar to Send(), but SendReliably() sends a message semi-reliably, at least once. SendReliably has semantics similar to TCP (in fact, it is implemented using TCP). This procedure sends the message to *node* repeatedly (in the

background) until it receives an acknowledgement from *node*. When it fails to receive an acknowledgement after a certain period, it aborts the transmission and initiates the membership protocol.

- Procedure SetTimer(*delay, proc*) schedules *proc* to be called *delay* seconds later. Procedure CancelTimer(*proc*) cancels the timer set by SetTimer.

- Procedure Now() returns the node's current wall-clock value. The clock resolution is assumed to be fine enough to distinguish between any two events that happen on a node. This can be achieved by using a combination of the system clock (e.g., `time(2)`) and the cycle-counting register found in most modern CPUs (e.g., `rdtsc` instruction in Pentium).

- Texts after '// ' are comments.

- $\langle val_1, val_2 \rangle$ is a tuple of two values.

- The variables marked "**persistent**" are stored on stable storage and survive node crashes. Persistent variables are used only in the replication protocol described in Chapter 6 (in fact, all global variables used by the algorithm are persistent).

- Procedures marked "**public**" run as transactions that are non-preemptive and update global variables atomically.

Appendix C Description of the Recovery Protocols

This chapter describes the four protocols that recover FHC's soft state: membership agreement, user map recomputation, mail-map recovery, and profile-bank recovery. These protocols were overviewed in Chapter 5.

C.1 Membership Protocol

Figure C.1 show the types, variables, and callback functions used by the membership manager on a node. Lines marked with "†" are absent in the original TRM and are added in FHC. The callback functions let other modules piggyback any information on membership messages. We demonstrate their use when we discuss the recomputation and distribution of the user map in Appendix C.2.

The membership manager exports only one procedure, ChangeDetected, to be called by other modules (Figure C.2). This procedure tells the manager that something unusual, e.g., a non-responsive node, has been found. Calling this procedure is optional, since the membership manager detects node failures and recoveries by itself.

Figures C.3 and C.4 show the core of the protocol that converges the membership views of nodes (Section 5.3.1).

The service also runs failure and recovery detection protocols separately from the core protocol, as shown in Figures C.5 and C.6.

C.2 User Map Recomputation

Figure C.8 shows the user map recomputation algorithm (Section 5.4). This algorithm runs as an embedded step of the membership protocol, using the callback procedures defined in Figure C.1.

```
const
   MAX_RESEND = 3† // Number of times each type of packets is sent. Section 5.3.2.
   RESEND_PERIOD = 500ms† // Resend packets at this interval.
   TIMEOUT = 5 s // Estimated steady-state, one-way messaging latency.

// Type GroupID identifies a particular membership reconfiguration attempt.
type GroupID = record
   ts: integer // Local counter (logical clock)
   nid: NodeID // Node that started the group creation.

// Per-node global variables. They are in the node's memory.
var
   gid: GroupID // The current group the node is in.
   members: Set⟨NodeID⟩ // Group members.

   // Variables below are used only internally.
   proposedGID: GroupID // ID of the group being created by Me, if any.
   maxGID: GroupID // ID of the newest group ID encountered, i.e., Lamport clock.
   acceptors: Set⟨NodeID⟩ // The set of nodes that responded to NEWGROUP.
   nrMsgsSent: [0 .. MAX_RESEND]† // Controls retransmissions.

// Callback procedures:
// AcceptSendCallback is called just before ACCEPT is sent.
// This procedure may return any data, which are sent to the coordinator.
proc AcceptSendCallback(newGID)

// AcceptReceiveCallback is called after the coodinator receives an ACCEPT message.
// Parameter data is whatever AcceptSendCallback produced on
// node sender.
proc AcceptReceiveCallback(newGID, sender, data)

// JoinSendCallback is called on the coordinator node just before
// sending the new membership.
proc JoinSendCallback(newGID, acceptors)

// JoinReceiveCallback is called on each member node after receiving a JOIN
// message. Parameter data is whatever AcceptSendCallback produced.
proc JoinReceiveCallback(newGID, acceptors, data)
```

Figure C.1. *Global variables used by the FHC's membership service. These variables are kept in each node's memory. Their initial values are set by procedure* Initialize, *described later.*

```
// This procedure is called when someone suspects a crash or recovery of another node.
proc ChangeDetected()
  if maxGID > gid then // Membership reconfiguration is already in progress.
    return

  maxGID.ts ← maxGID.ts + 1 // New local event; increment the Lamport clock.
  maxGID.nid ← Me
  proposedGID ← maxGID
  acceptors ← {Me}
  SetTimer(TIMEOUT*2, MissingAcceptTimeout)
  nrMsgsSent ← 0†
  BroadcastNewgroup()

proc BroadcastNewGroup()†
  if nrMsgsSent < MAX_RESEND ∧ maxGID = proposedGID then
    // I'm still coordinating the newest group creation
    nrMsgsSent ← nrMsgsSent + 1
    Send(BROADCAST, NewGroup, Me, proposedGid, acceptors)
    SetTimer(RESEND_PERIOD, BroadcastNewgroup)
```

Figure C.2. *The start of the membership protocol. Any higher-level managers (e.g., proxies) that detect a change call* ChangeDetected() *to initiate the membership protocol and become the coordinator themselves.* ChangeDetected *may also be called internally, as discussed later.*

```
// Called in response to a NEWGROUP message from the coordinator.
proc NewGroup(sender, newGID, acceptors)
  if maxGID ≤ newGID ∧ Me ∉ acceptors then
    // This is the newest group. I should join it.
    maxGID ← newGID
    piggy ← AcceptSendCallback(newGID)†
    Send(newGID.nid, Accept, Me, newGID, piggy)
    // If I don't receive JOIN, start the TRM.
    SetTimer(TIMEOUT*3, ChangeDetected)
  elif newGID < maxGID
    // Old group found. Make the sender abort the protocol.
    Send(sender, NewGroup, Me, maxGID, ϕ)

// Called in response to an ACCEPT message from a node.
proc Accept(sender, newGID, piggy)
  if newGID = proposedGID then
    acceptors ← acceptors ∪ {sender}
    AcceptReceiveCallback(newGID, piggy)†
  else
    // A new group is being formed. Ignore this reply.
```

Figure C.3. *Procedure NewGroup is called when a node receives a NEW_GROUP message from the coordinator. The node replies by sending an ACCEPT message to the coordinator.*

```
// Executed by the coordinator after NEWGROUP times out.
proc MissingAcceptTimeout()
  if proposedGID ≠ gid then
    return
  // Update my own membership status first.
  piggy ← JoinSendCallback(proposedGID, acceptors)
  Join(proposedGID, acceptors, piggy)
  // Inform others of my status.
  nrMsgsSent ← 0†
  BroadcastJoin()

proc BroadcastJoin()†
  if nrMsgsSent < MAX_RESEND ∧ proposedGID = gid then
    nrMsgsSent ← nrMsgsSent + 1
    piggy ← JoinSendCallback(gid, members)
    Send(BROADCAST, Join, gid, members, piggy)
    SetTimer(RESEND_PERIOD, BroadcastJoin)

proc Join(newGID, acceptors, piggy)
  if newGID ≠ maxGID then // Stale join message.
    return
  elif gid = newGID then † // Rebroadcast of previous join message.
    return
  elif Me ∉ acceptors then † // I received Join for which I'm not a participant.
    return
  members ← acceptors
  gid ← newGID
  CancelTimer(MissingJoinTimeout)
  if IAmLeader() then
    SetTimer(PROBE_PERIOD, SendProbeTimeout)
  JoinReceiveCallback(newGID, acceptors, piggy)

proc IAmLeader()
  return Me = gid.nid
```

Figure C.4. *MissingAcceptTimeout() is called on the coordinating node after it waits for ACCEPTs to come back from live nodes. The coordinator sends the JOIN messages in BroadcastJoin(), and nodes process these messages in Join().*

```
const
  PING_PERIOD = 10 s // Interval for sending PING messages.
var
  nrPingsSent: [0 .. MAX_RESEND]† // Controls retransmissions.

proc PingTimeout() periodic PING_PERIOD
  if ||members|| > 1 then
    nrPingsSend ← 0
    SendPing()
    SetTimer(TIMEOUT*2, ChangeDetected)

proc SendPing()
  nextNode ← NextNodeInRing(members, Me)
  if nrPingsSent < MAX_RESEND ∧ nextNode ≠ Me then
    nrPingsSent ← nrPingsSent + 1
    Send(nextNode, AreYouAlive, Me)
    SetTimer(RESEND_PERIOD, SendPing)

proc AreYouAlive(sender)
  Send(sender, IAmAlive)

proc IAmAlive()
  CancelTimer(ChangeDetected)
  CancelTimer(SendPing)
```

Figure C.5. *Crash detection by periodic pinging along a virtual ring formed by nodes in the membership.*

```
proc SendProbeTimeout() periodic PROBE_PERIOD
  Send(BROADCAST, Probe, Me)

proc Probe(sender)
  if IAmLeader() ∧ sender ∉ members then
    ChangeDetected()
```

Figure C.6. *Probing facilitates merging of partitions and newly booted nodes.*

```
proc Initialize()
  proposedGID.ts = 0
  proposedGID.nid = Me
  Join(proposedGID, {Me}, JoinSendCallback(proposedGID, {Me}))
```

Figure C.7. *Procedure Init is called when the node starts up. It puts the node in a singleton group consisting of itself and starts the probing timer.*

```
type UserMap = array [0 .. USER_MAP_SIZE-1] of record
  manager: NodeID;
  epoch: GroupID;

var
  userMap: UserMap // Current user map.
  // Remembers what other nodes manage during reconfiguration.
  oldEpochs: Map⟨NodeID,Map⟨integer,GroupID⟩⟩

// Called just before sending ACCEPT.
proc AcceptSendCallback(): Map⟨integer,GroupID⟩
    map = {}
    foreach i such that userMap[i].manager = Me
      map[i] ← userMap[i].epoch
    return map;

// Called on the coordinator when ACCEPT is received.
proc AcceptReceiveCallback(newGID, caller, map)
    oldEpochs[caller] ← map

// Called just before JOIN is sent. It computes the new user map from the current one.
proc JoinSendCallback(newGID, acceptors): UserMap
    newMap ← userMap
    foreach i such that newMap[i].manager ∉ acceptors
      ReassignBucket(newMap, i, acceptors)
    loop
      busyNode ← PickMostLoadedNode(newMap, acceptors)
      nrManagedByBusyNode ← # of buckets managed by busyNode in newMap.
      idleNode ← PickLeastLoadedNode(newMap, acceptors)
      nrManagedByIdleNode ← # of buckets managed by idleNode in newMap.
      if nrManagedByBusyNode − nrManagedByIdleNode ≤ 2 then
        break
      idx = pick an arbitrary bucket managed by busyNode in newMap
      ReassignBucket(newMap, idx, acceptors)
    return newMap

proc JoinReceiveCallback(newGID, members, newMap)
    // StartRecoverSoftState pushes the soft state to new profile managers. (Figure C.9).
    StartRecoverSoftState(userMap, newMap)
    userMap = newMap

proc ReassignBucket(newMap, idx, acceptors)
    idleNode ← PickLeastLoadedNode(newMap, membership)
    newMap[idx].manager = idleNode
    if defined(oldEpochs[idleNode][idx]) then
      newMap[idx].epoch ← oldEpochs[idleNode][idx]
    else
      newMap[idx].epoch ← proposedGID
```

Figure C.8. *The user map recomputation algorithm. Functions with the names XXXCallback are called by the membership manager. Procedure PickMostLoadedNode picks a node from the second argument that manages the most buckets in the user map (the first argument). Procedure PickLeastLoadedNode similarly picks the node that manages the fewest buckets.*

C.3 Mail Map Recovery

Figures C.9 and C.10 show the pseudo-code of the mail map reconstruction algorithm (Section 5.5.2). Reconstruction of the user profile is omitted, because it is exactly the same as in Figure C.9 except that the user profile is pushed instead of mail maps.

```
// Updates to mail maps are applied in batch. This type represents a queued update.
type MailMapUpdate = record
   operation: {ADD, DELETE}
   user: string
   ts: Wallclock

var
   mailMapUpdates: Set⟨MailMapUpdate⟩ // Queued updates to mail maps.
   // needPush[i] is true if the assignment of i'th bucket's manager has changed.
   needPush: array [0 .. USERMAPSIZE-1] of bool

// Add entry to a mail map. Called by the frag. manager after a fragment is created.
proc AddNodeToMailMap(user)
   mailMapUpdates ← mailMapUpdates ∪ {operation = ADD, user = user, ts = Now()}

// Delete entry from a mail map. Called by the frag. manager after a fragment is deleted.
proc DeleteNodeFromMailMap(user)
   mailMapUpdates ← mailMapUpdates ∪ {operation = DELETE, user = user, ts = Now()}

proc MailMapUpdatePusher() periodic
   // Variable nodes holds all the buckets that need to be contaccted
   idxes = { Hash(u.user) || u ∈ mailMapUpdates }
   foreach i ∈ idxes
      updates ← { u || u ∈ mailMapUpdates ∧ Hash(u.user) = i }
      SendReliably(userMap[i].manager, UpdateMailMap_Request, Me, gid, updates) // ⟨3⟩

proc UpdateMailMap_Reply(updates, ok)
   if ok then // ⟨4⟩
      mailMapUpdates ← mailMapUpdates \ updates

// This procedure is called from user map recomputation procedure. See also Figure C.8.
proc StartRecoverSoftState(oldUserMap, newUserMap)
   foreach i such that oldUserMap[i] ≠ newUserMap[i]
      needPush[i] ← true
   // Cancel updates that are queued to dead nodes.
   foreach u ∈ mailMapUpdates such that Hash(u.user) = i
      mailMapUpdates ← mailMapUpdates \ {u}

proc FillMailMaps_Reply(senderGid, ok)
   if ok ∧ senderGid = gid then
      needPush[i] ← false

// Scan the directories for buckets and push the fragment names to new profile managers.
proc RecoverMailMapsForBucket() periodic 10 s
   foreach i such that needPush[i] = true
      Wait random seconds.
      users = ScanSpoolForBucket(i)
      Send(usermap[i].manager, FillMailMaps_Request, Me, gid, Now(), fragments)
```

Figure C.9. *Mail map reconstruction. The mailbox manager on the node calls AddNodeToMailMap(user) or DeleteNodeFromMailMap(user) after a local fragment for the user is created or deleted. Procedure* StartRecoverSoftState*(oldUserMap,newUserMap) is called when a new membership is determined. It only schedules the delivery of mail map entries to remote nodes, and the disks are actually scanned by procedure* RecoverMailMapsForBucket *in the background.*

```
// Mail map partially maps the node to the timestamp. See also Section 5.5.5.
type MailMap = Map⟨NodeID,integer⟩

var mailMaps: Map⟨string,MailMap⟩ // Set of users that a node manages.

proc UpdateMailMap(caller, callerGID, updates)
  ok ← true
  if callerGid ≠ gid then
    // Got the update before I got membership info. See Section 5.5.4.
    ok ← false
  else
    foreach u ∈ updates
      if u.operation = ADD then
        AddToMailMap(caller, u.ts, u.user)
      else
        RemoveFromMailMap(caller, u.ts, u.user)
  Send(caller, UpdateMailMap_Reply, u, ok)

// Update mail map using the Thomas write rule. See Section 5.5.5.
proc AddToMailMap(caller, ts, user)
  map ← mailMaps[user]
  if defined(map[caller]) then
    map[caller] ← max(ts, map[caller])
  else
    map[caller] ← ts

proc RemoveFromMailMap(caller, ts, user)
  map ← mailMaps[user]
  if defined(map[caller]) ∧ map[caller] < ts then
    Delete caller from map.

proc FillMailMaps_Request(caller, callerGID, ts, users)
  ok ← true
  if callerGid ≠ gid then
    // Got the update before receiving the membership info. See Section 5.5.4.
    ok ← false
  foreach user ∈ users
    AddToMailMap(caller, ts, user)
  Send(caller, FillMailMaps_Reply, callerGID, ok)

proc JoinReceiveCallback(newGID, members, newMap)
  // Remove the profile of all users not managed by me.
  foreach ⟨user,map⟩ ∈ mailMaps
    idx = Hash(user)
    if newMap[idx].manager ≠ Me then
      mailMaps = mailMaps \ ⟨user,map⟩
```

Figure C.10. *User map reconfiguration. The code for the profile manager.*

Appendix D Description of the Replication Algorithm

This appendix formally describes FHC's replication algorithm. An overview of the algorithm appears in Chapter 6.

D.1 Data Structures

This section describes the data structures used by the replication algorithm. It is actually a rehash of those introduced in Section 6.4.2, with variables and fields added to handle corner cases.

Figure D.1 shows the types used in the algorithm. **Timestamp** uniquely identifies an update and defines total ordering among updates. Its *time* field records a loosely synchronized wall-clock [106] that can order any two events on a node (Appendix B). The *nid* field records the name (IP address) of the node that generated the timestamp. It breaks ties between wall-clocks on two different nodes.

An update to the object is represented by the **Update** record. Its *state* field indicates the update's status. A new update starts as being ACTIVE. An update is RETIRING at the coordinator node while retirement notifications are being sent, and it is RETIRED after the node receives a retirement notice. An update is SUSPENDED when it is found to conflict with a newer update, and it stays dormant until the newer update arrives and supersedes it. Fields *targets*, *done*, and *peers* specify the set of nodes that should receive the update, have acknowledged the update, and should replicate the object, respectively. Thus, *done* and *peers* are always subsets of *targets*. The update propagation

```
type Timestamp = record
  time: Wallclock  // wall-clock time.
  nid: NodeID  // a tie-breaker.

type Update = record
  state: {ACTIVE, RETIRING, RETIRED, SUSPENDED}
  ts: Timestamp
  targets, done, peers: Set(NodeID)
```

Figure D.1. *Data structures used by the replication algorithm.*

```
persistent var
  gData: Content ← NIL
  gPeers: Set⟨NodeID⟩ ← ϕ
  gU: Update ← NIL
  gRetireTime: Clock
  gSavedData: Content
```

Figure D.2. *Per-node, per-object global variables used by the replication algorithm.*

```
public proc UpdateObject(peers, data)
  u←Update(ts←Timestamp(Now(), Me), state←ACTIVE, done←ϕ,
           peers←peers, targets←peers)
  ApplyUpdate(u, data)
```

Figure D.3. UpdateObject *is called by the application to create a new object, modify the object's contents, add or remove the replica set, or delete the object.*

finishes when *done* = *targets*.

We simply use an opaque type **Content** to represent the object contents here, because the replication service leaves the object format up to the application (Section 6.3).

Figure D.2 shows the global, persistent variables used by the algorithm. This appendix presents the algorithm only in the context of a single object. Multiple objects can be supported with a straightforward extension to the algorithm, as discussed in Section 6.6.1.

Five persistent variables are stored per replica. Two variables, *gData* and *gPeers*, are visible to the application, and the rest are used internally by the replication algorithm. Variable *gU* remembers the newest update, if any, issued for the object. Notice the absence of the new object contents in *gU* — the contents are propagated to other nodes by reading from *gData* directly most of the time. The exception is when the replica itself (i.e., *gData* and *gPeers*) is deleted by an update, but the object contents still need to be propagated to other nodes (this happens, for example, when the object is moved from one node to another). Variable *gSavedData* is used to save new object contents in such a case, and it is otherwise null. That *gSavedData* is usually null contributes to reducing the space overhead of the algorithm, because all other persistent variables used by the algorithm are of small and fixed size. Variable *gRetireTime* is used when deleting a retired update (Section 6.4.3).

```
proc ApplyUpdate(u, data): bool
  // Expand knowledge. ⟨5⟩
  u.targets ← u.targets ∪ gPeers
  if gU ≠ NIL ∧ gU.state ∉ {RETIRING,RETIRED}
    u.targets ← u.targets ∪ gU.targets
    gU.targets ← u.targets
  // Reject if u is stale. ⟨6⟩
  if gU ≠ NIL ∧ gU.ts > u.ts
    return false
  // Log the update. ⟨7⟩
  gU ← u
  gSavedData ← NIL
  // Modify the replica. ⟨8⟩
  if Me ∈ gU.peers then
    ⟨gData, gPeers⟩ ← ⟨data, gU.peers⟩
  else
    ⟨gData, gPeers⟩ ← ⟨NIL, ϕ⟩
    if gU.peers ≠ ϕ
      // Save data; not needed when peers is ϕ since everyone deletes the replica.
      gSavedData ← data
  gU.done ← gU.done ∪ {Me}
  return true
```

Figure D.4. *Local update application. This procedure logs and applies the update to the local replica. It returns whether the update was successfully applied or not.*

D.2 Application Programming Interface

Figure D.3 shows the only procedure called by the application (the API with the support for optimistic deltas will be described in Appendix D.6). Procedure UpdateObject takes two parameters, the new replica set (*peers*) and the new object contents (*data*). Passing an empty set to *peers* will delete the object entirely from the system. The caller of this procedure must ensure that the node stores a replica already, except when the object is newly created. This restriction is to prevent creating an orphan replica that is disconnected from others and is not found by the node discovery process (Section 6.4.4 and Appendix F.3).

D.3 Update Application

Procedure ApplyUpdate (Figure D.4) is called both from the local application and from remote nodes to apply and log the update to the replica.

This procedure first merges the target sets of both the new update (u) and the current update

(gU)⟨5⟩ as a part of the node discovery protocol (markers such as ⟨1⟩ refer to lines in the program listings). The two sets must be merged even when u is to be discarded ⟨6⟩ to ensure that the participants of both updates can eventually receive the newer update. This procedure then logs the update on persistent variables 7 and finally modifies the local replica ⟨8⟩.

The logged update is propagated later by the procedure described in the next section.

D.4 Update Propagation

```
public proc PushApply() periodic
  if gU≠NIL ∧ gU.state = ACTIVE
    if gU.done=gU.targets // Update propagation finished. ⟨9⟩
      gU.state ← RETIRING
      gU.done ← {Me}
      gSavedData ← NIL
    elif IAmCoordinator(gU)
      foreach node ∈ (gU.targets \ gU.done)
        if node ∉ gU.peers
          data ← NIL // Node will delete the replica.
        elif gData ≠ NIL
          data ← gData // I store valid contents.
        else
          data ← gSavedData
        Send(node, Apply_Request, Me, gU, data)

public proc Apply_Request(caller, u, data)
  ok ← ApplyUpdate(u, data)
  Send(caller, Apply_Reply, u.ts, ok, gU.done, gU.targets)

public proc Apply_Reply(ts, ok, done, targets)
  if ¬UpdateOverwritten(ts) // ⟨10⟩
    if ¬ ok
      gU.state ← SUSPENDED // ⟨11⟩
    else
      gU.done ← gU.done ∪ done
      gU.targets ← gU.targets ∪ targets // ⟨12⟩

proc UpdateOverwritten(ts): bool
  return gU = NIL  // The update retired.
       ∨ gU.ts > ts // A new update arrived.

proc IAmCoordinator(u): bool // See Section 6.6.2 for an implementation.
```

Figure D.5. *Update propagation.* PushApply *is called periodically to push the newest update to participants.* UpdateRequest *is executed on remote nodes in response to* PushApply. UpdateReply *is called on the coordinator to handle replies from* UpdateRequest.

An update is pushed to other nodes periodically by procedure PushApply (Figure D.5). The target node set expands as replies come back from the target nodes ⟨12⟩. The propagation finishes once all target nodes reply ⟨9⟩. The procedure IAmCoordinator tells whether the node is designated to coordinate a particular update (Section 6.6.2).

D.5 Update Retirement

The replication algorithm deletes update records from disk in two steps. The first step, *retirement*, is performed periodically by procedure PushRetire (Figure D.6). Here, the coordinator informs the target nodes that update propagation is complete.

The second step, *removal*, happens independently on each node (procedure RemoveUpdate in Figure D.6). Here, the node simply waits for *WAIT* seconds before deleting a retired update. *WAIT* is the sum of the maximum clock skew among nodes and the message lifetime (Section 6.4.3). This update removal scheme additionally requires each node to discard stale incoming network messages: each network message is stamped with the sender's clock value and is accepted by the receiver only when its timestamp is no older than *WAIT* on the receiver's clock (Figure D.6, MessageArrived).

D.6 Supporting Optimistic Deltas

Figure D.7 shows changes to the original algorithm to support optimistic deltas. Instead of the new object contents, procedure UpdateObject now takes *diff*, an application-specific representation of the change to the object. Procedure PushApply is called periodically to push the diff to peer replicas. On the receiving node, if the diff cannot be applied because the fingerprint mismatches (procedure DiffUpdateRequest), the node requests a full contents transfer via RequestContentsTransfer. This code uses two callback procedures supplied by the application: Fingerprint(), to compute the fingerprint of the replica, and Patch(), to apply a diff to the replica.

```
public proc PushRetire() periodic
  if gU ≠ NIL ∧ gU.state = RETIRING
    if gU.done=gU.targets
      gRetireTime ← Now()
      gU.state ← RETIRED
    elif IAmCoordinator(gU)
      foreach node ∈ (gU.targets \ gU.done)
        Send(node, Retire_Request, Me, gU.ts)

public proc Retire_Request(caller, ts)
  if ¬UpdateOverwritten(ts)
    gRetireTime ← Now()
    gU.state ← RETIRED
    gSavedData ← NIL
  Send(caller, Retire_Reply, Me, ts)

public proc Retire_Reply(node, ts)
  if ¬UpdateOverwritten(ts)
    gU.done ← gU.done ∪ {node}

public proc RemoveUpdate() periodic
  if gU ≠ NIL ∧ gU.state = RETIRED ∧ Now() >gRetireTime+WAIT
    gU ← NIL // Delete the update. ⟨13⟩
public proc MessageArrived(msg)
  if msg.ts < Now() - WAIT
    return // Message too old. Just ignore it.
  dispatch msg
```

Figure D.6. *Update retirement. Procedure* PushRetire *is called periodically to push retirement notices to participants.* RetireRequest *is executed on a remote node in response to* PushRetire. *Procedure* RetireReply *is called on the coordinator to handle replies from* RetireRequest. *Procedure* RemoveRequest *is called periodically to remove retired updates. Procedure* MessageArrived *is called for every incoming message to discard messages that are too old.*

```
type Update = record
  state: {ACTIVE, RETIRING, RETIRED, SUSPENDED}
  ts: Timestamp
  targets, done, peers: Set⟨NodeID⟩
  diff: Contents
  fingerprint: Fingerprint

public proc UpdateObject(peers, diff)
  u←Update(ts← Timestamp(Now(), Me), state←ACTIVE,done←ϕ, diff←diff,
           fingerprint←Fingerprint(gContent), peers←peers, targets←peers)
  ApplyDiff(u)
proc ApplyDiff(u): {OK, SUPERCEDED, RETRY}
  if Fingerprint(gContent)≠u.fingerprint then
    return RETRY
  elif ApplyUpdate(u, Patch(gU, u.diff)) then
    return OK
  return SUPERCEDED
public proc PushApply()
  if gU≠NIL ∧ gU.state = ACTIVE
    if gU.done = gU.targets then // Update done
      gU.state ← RETIRING
      gU.done ← {Me}
      gSavedData ← NIL
    elif IAmCoordinator(gU) then
      foreach node ∈ (gU.targets \ gU.done)
        Send(node, DiffUpdateRequest, Me, gU)

public proc DiffUpdateRequest(caller, u)
  s←ApplyDiff(u)
  if s = OK ∨ s = SUPERCEDED then
    Send(caller, UpdateReply, u.ts, (s=OK), gU.done, gU.targets)
  else // needs full contents transfer
    Send(caller, RequestContentsTransfer, Me, u.ts)
public proc RequestContentsTransfer(caller, ts)
  if ¬UpdateOverwritten(ts)
    if caller ∉ gU.peers then
      data ← NIL
    elif gData ≠ NIL then
      data ← gData
    else
      data ← gSavedData
    Send(node, UpdateRequest, Me, gU, data)
```

Figure D.7. *Changes to the algorithm to support optimistic deltas. The public procedure,* UpdateObject*(), is changed to take a diff instead of the new object contents. Procedure* PushApply *is called periodically to push the newest diff to participants. The revised algorithm uses two callbacks supplied by the application: function* Fingerprint *computes the fingerprint of contents, and procedure* Patch *applies a diff to the contents.*

Appendix E Correctness of the Soft-state Recovery Algorithm

This chapter proves the correctness of FHC's recovery mechanism. Specifically, we argue that the combination of the membership protocol, the user map recomputation algorithm, and the user profile and mail map recovery mechanisms ensures the consistency of the data structures managed in the cluster.

E.1 Correctness of the Membership Protocol

The correctness of the membership protocol is proved in [41], and it is not repeated here. The proof shows that protocol converges the membership views of all live nodes under the conditions defined in Section 4.5.1. The maximum time J needed for view convergence is bounded as below (the definition of the constants appears in Appendix C):

$$J = \max(PING\_PERIOD, PROBE\_PERIOD) + 9 * TIMEOUT.$$

E.2 Correctness of the Soft State Recovery Protocols

We prove two theorems, which together ensure that all fragments on all live nodes will eventually become reachable through the user map and mail maps.

Theorem 1 *For every bucket b, every live node agrees on the unique profile manager for b if no new failure happens for a long enough period.*

Theorem 2 *The mail map for every user is consistent if no new failure happens for a long enough period. A mail map is consistent when it contains a link to every live node that stores a fragment for the user.*

Let us first define the terms used in the ensuing proof. UM_g is the user map for the membership group with ID g. (A user map is uniquely identified by g, because it is centrally and atomically computed by the coordinator of group g.) $UM_g[b].manager$ is the profile manager of the b'th bucket of UM_g, and $UM_g[b].epoch$ is the epoch ID of the b'th bucket of UM_g.

Proof of Theorem 1. The membership protocol ensures that all nodes eventually receive an identical membership list (Appendix E.1). Because the user map is computed centrally by the membership coordinator, all nodes also eventually accept an identical user map. That the user map defines a many-to-one mapping from buckets to nodes is obvious from its structure. ■

Proof of Theorem 2. We enumerate all the possible ways a system configuration can change and show that each of them keeps mail maps consistent.

1. A node crashes after it creates a fragment on its disk, but before it sends the corresponding mail-map update request to the user manager.

   This case causes no problem, because this fragment becomes non-retrievable after the node failure. The replication service ensures that the fragment is still available on other nodes (Chapter 6).

2. A node crashes just after it deletes a fragment, but before it sends the corresponding mail-map update request to the user manager.

   This scenario creates a mail map entry that points to the failed node. It creates no consistency problem, however, because the theorem requires only that live fragments be reachable through a mail map, but not vice versa.

   In practice, a dangling mail map entry is removed in two ways: (1) when a proxy tries to read or update the user's mail map, the profile manager automatically removes the dead entry, and (2) the rebalancer periodically scans all mail maps on each node and removes dead entries (Section 7.2).

3. Node p creates a new fragment for user u. Later, p receives a new membership list and a new user map as the result of a configuration change.

From the assumption of the theorem, the system eventually reaches the steady state in which no failure happens for a long enough period. Let $g_0, g_1, \ldots, g_n$ be the series of membership groups that p joins since it creates the fragment, with g_n being the group that p joins during the steady period.

Node p tries to add itself to u's mail map after creating the fragment; let g_a be the group node p belongs to when p sends the mail-map update for which it receives a positive acknowledgement (an acknowledgement is positive when *ok* = **true** in Figure C.9 ⟨4⟩). Notice that $0 \leq a \leq n$, because in the steady state, any profile manager sends an acknowledgement promptly (which is the definition of the steady period).

Let $b = \text{Hash}(u)$, $m_a = UM_{g_a}[b].manager$, and $e_a = UM_{g_a}[b].epoch$.

- Suppose the bucket assignment for b does not change after g_a; i.e., $a < \forall i \leq n$, $UM_{g_i}[b] = UM_{g_a}[b]$.

  In this case, node m must have continuously managed bucket b ever since m sent out the acknowledgment, for the following reasons:

  - Node m did manage bucket b when it added p to u's mail map and acknowledged p, because node m_a must have checked if nodes p and m_a belonged to the same group before acknowledging positively (Figure C.9 ⟨4⟩ and Section 5.5.4).
  - Node m_a must have continuously managed bucket b until the end of the steady period. This is proved by contradiction, exemplified by the scenario depicted in Figure E.1. Here, node m_a had quit managing the bucket when it joined a group, say, g_y (notice that $g_y > g_a$, because group IDs are really Lamport clocks). Let $e_y = UM_{g_y}[b].epoch$. Then, $e_y > e_a$ because the epoch numbers are also Lamport clocks (Section 5.4). Later, when node m_a joins g_n, $UM_{g_n}[b].epoch = e_n > e_y > e_a$, contradicting our original assumption that bucket b's assignment has not changed since P received g_a.

  Thus, node m_a still has node p in user u's mail map after g_n, because node n has continuously been managing bucket b since it added node p to the mail map.

- Suppose otherwise; i.e., $a < \exists i \leq n$ such that $UM_{g_i}[b] \neq UM_{g_a}[b]$.

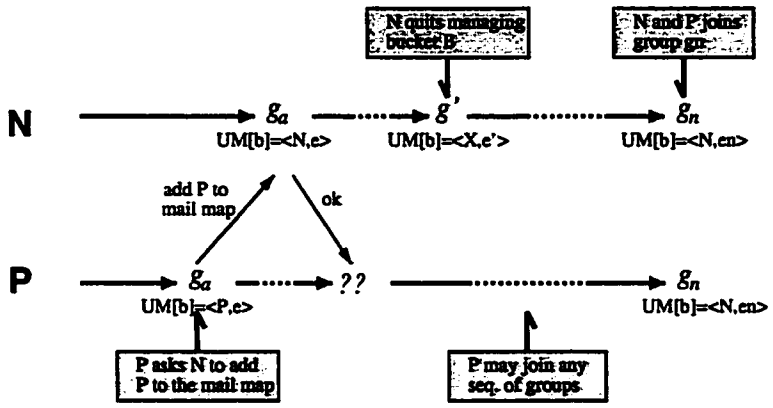


Figure E.1. *The sequence of events that demonstrates how configuration changes may affect the contents of a mail map. Here, node p adds itself to a mail map in bucket b, managed at node m in the beginning, and receives an acknowledgement, but node m quits managing bucket b later. When nodes m and p later agree on the common membership g_n, p will always be able to discover that m had quit the management of b in the past, regardless of the membership groups that nodes p and m may have joined.*

Let g_x be the first group (on node p) in which the b'th entry of the user map settles on the value $UM_{g_n}[b]$; i.e., let g_x be a group such that:

$$UM_{g_{x-1}}[b] \neq UM_{g_n}[b] \text{ and } x \leq \forall y < n,\ UM_{g_y}[b] = UM_{g_n}[b].$$

Let $m_x = UM_{g_x}[b].manager$. When node p joins g_x, p scans bucket b, discovers the fragment for user u, and pushes the fragment name to node m_x. It keeps retrying pushing until it receives an acknowledgement from node m_x. Now, we can apply the same argument as above and prove that node m_x has managed the bucket b between g_x and g_n. Thus, node m_x must have the entry for node p in user u's mail map when both nodes join group g_n.

4. Node p deletes a new fragment for user u. Later, p receives a new membership list and a user map as a result of a configuration change.

   We can use the same arguments used in case 3 to prove that dead references are eventually deleted from the new profile manager. Technically, we need not even prove this case, because the theorem allows dead entries to be in a mail map.

5. A node p boots.

   Let us focus on a particular fragment in arbitrary bucket b. We assume that the fragment is left untouched after node recovery for a long enough period. Otherwise, the situation is handled by case 4. Below, we use the same argument as in case 3 to prove that the recovery protocol adds a mail map entry for the fragment.

   From the assumption of the theorem, the system eventually reaches the steady state in which no failure happens for a long enough period. Let $g_0, g_1, \ldots, g_n$ be the series of membership groups that p joins before reaching the steady state. Let g_x be the first group in which the b'th entry of the user map settles on the value $UM_{g_n}[b]$. Let $m_x = UM_{g_x}[b].manager$. Node p scans bucket b when it joins g_x and pushes the soft state to node m_x (which is identical to $UM_{g_n}[b].manager$). Furthermore, node m_x continuously manages bucket b after it receives g_x. Thus, node m_x will have the entry p in the mail map.

■

Appendix F Correctness of the Replication Algorithm

While being simple, our replication algorithm contains several subtleties, especially regarding replica additions and deletions. For example, how does it guarantee that all replicas receive an update when another update is adding replicas concurrently? In the following sections, we prove two main safety properties of the algorithm: that all nodes receive the newest update at the end of propagation, and that no stale updates are accepted by nodes regardless of concurrent updates. An overview of the proof appeared in Section 6.7. In Appendix F.6, we also argue the liveness of the algorithm, i.e., that all replicas will receive the newest update.

F.1 Knowledge Graphs

The state of the system can be viewed as a directed graph, called the *knowledge graph*, in which the vertices represent nodes and the edges represent the nodes' knowledge of others through *gPeer* and *gU.targets*. The graph is usually complete (i.e., no update is outstanding and the value of *gPeer* is identical on all the replicas), but it becomes incomplete during replica addition or deletion. At a high level, our proof shows that the algorithm ripples the newest update through the graph, adding edges to the graph along the way to cover all nodes and to restore the completeness of the graph eventually.

Following are notations used in the ensuing proof. Other symbols are summarized in Table F.1.

- A node "*stores a replica*" when $gData \neq NIL$.
- A node "*has an update*" when its $gU \neq NIL$ and $gU.state \in \{ACTIVE, RETIRING\}$.
- A node "*retires an update*" when it sets *gU.state* to RETIRED.
- A node n_1 "*knows*" another node n_2 when either n_1 has an update and $n_2 \in n_1$:*gU.target*, or $n_2 \in n_1$:*gPeer*.
- $G_T \equiv \langle V_{G_T}, E_{G_T} \rangle$, the *knowledge graph* for the object at time T[1], is defined as follows:

[1] All times mentioned in the proof are hypothetical global times observed by an external agent.

Table F.1. Notational conventions used in the proof of correctness for replication.

Symbol	Meaning
n:*var*	The value of variable *var* on node n.
n:*Utargets*,T	The value of *Utarget* on node n just before time T.
n:*Udone*,T	The value of *Udone* on node n just before time T.
$n_1 \xrightarrow{G_T} n_2$	n_1 knows n_2 in G_T, i.e., $(n_1 \rightarrow n_2) \in E_{G_T}$.
$n_1 \overset{G_T}{\rightsquigarrow} n_2$	A path from n_1 to n_2 exists, i.e., $n_1 \xrightarrow{G_T} n_2 \vee \exists n', n_1 \xrightarrow{G_T} n' \wedge n' \overset{G_T}{\rightsquigarrow} n_2$.

V_{G_T} = Nodes that store a replica or have an update.

$E_{G_T} = \{v_1 \rightarrow v_2 \mid \{v_1, v_2\} \subseteq V_{G_T} \wedge v_1 \text{ knows } v_2\}$

.

- $S_T \equiv \langle V_{S_T}, E_{S_T} \rangle$, an *induced subgraph* of G_T, excludes from G_T the vertices that correspond to failed nodes and the associated edges. S_T shows the knowledge graph in the presence of failure.

F.2 Correctness Criteria

Ideally, we want to prove that the algorithm keeps all the live replicas consistent regardless of the type or number of failures. Such a guarantee, however, is impossible when nodes or links fail in a way that makes the corresponding induced knowledge subgraph disconnected. For example, suppose two nodes fail simultaneously after both have created two new replicas, as illustrated in Figure 6.7 (page 89). After such a failure, an update issued on one of the new replicas will not reach the other, and the new replicas will remain inconsistent until the original two nodes recover. Therefore, we define the correctness only under the condition that a knowledge subgraph is at least weakly connected.

Correctness criteria: *Suppose S_{T_s} is weakly connected, no node or link newly fails, and no new update is issued during a long enough period (T_s, T_e). Let U be the newest update generated before T_e. The algorithm is correct if the following conditions hold.*

(1) Every node $n \in U.peers$ applies U before T_e.

(2) No node $n \notin U.peers$ stores a replica at T_e.

(3) No update U' older than U (i.e., $U'.ts < U.ts$) is applied on any node after U is applied.

Notice these criteria demand, in case of a graph disconnection, a replica consistency within each partition, and that as soon as the partitions re-integrate, all replicas converge onto the globally newest state. Indeed, this set of criteria is as strong as any non-blocking replication algorithm can guarantee. Moreover, as argued in Section 6.7.3, we believe that it is extremely unlikely for a knowledge graph to remain disconnected for a prolonged period.

In the discussions preceding Appendix F.6, the concept of "failure" is defined somewhat non-intuitively. A failed node may be live in reality and may execute the protocol in a timely manner. Likewise, a non-failed node may be dead in reality. The failure of a node really means "don't care" here. The replication algorithm needs to ensure the consistency of replicas when they eventually act according to the algorithm. The failed nodes may or may not cooperate, but the algorithm cannot assume anything about their behavior (except, of course, that they are not Byzantine). We can use this arbitrary definition because we are interested only in the safety of the algorithm. When we discuss liveness in Appendix F.6, we adopt a more conventional definition of failures.

F.3 Graph Invariants

Theorem 3 *If S_T is strongly connected, then $\forall T' > T$, $S_{T'}$ is also strongly connected if no node or link newly fails during the period (T, T').*

Theorem 4 *If S_T is weakly connected, then $\forall T' > T$, $S_{T'}$ is weakly connected if no node or link newly fails during the period (T, T').*

Proof. We show by induction that no transition to the subgraph can disconnect the subgraph. We omit the proof of Theorem 4 because it is identical to that of Theorem 3.

(1) An update is issued.

A new update keeps the subgraph strongly connected because it can only add edges (if any)

to the graph. Notice that a node does not appear in the graph until a replica is created on the node.

(2) A new replica n is created by coordinator c.
In this case, nodes n and c both know each other (i.e., two-way links are created), because both nodes are in the update's *target* field.

(3) A replica is deleted.
The graph does not change, because an update record is still kept on the node.

(4) Update U, coordinated by node c, retires on a non-coordinator node n at time T.
Notice that $n{:}Utargets_{,T} \subseteq c{:}Utargets_{,T}$, because node c merged n's target set $\langle 12 \rangle$, and because n applied no new update since it received U (otherwise, node n cannot retire U). Therefore, all the nodes that disconnect from n are still connected to the graph via c.

(5) Update U retires on coordinator c at time T.
The retirement substitutes the set of edges $E_n = \{c \rightarrow n \mid n \in U.peers\}$ for $E_o = \{c \rightarrow n \mid n \in c{:}Utargets_{,T}\}$. Usually, $E_o = E_n$, because the nodes not in U_{peers} have deleted replicas and they retire U before the coordinator does. Now, if there exists a vertex $n \in E_o - E_n$, n is always connected to the graph via a node different from c for the following reasons. First, for such an n to exist, n must have deleted its replica at time T' when it received U and later applied a newer update, U_2, that created a replica on n at time T'' ($T' < T'' < T$). At T'', n was strongly connected to the rest of the graph (cf., case (1)). Thus, it is also so after the edge $c \rightarrow n$ is removed at time T. To see why, suppose the edge $c \rightarrow n$ was removed at time T'; then the graph is clearly connected at time T. Deleting $c \rightarrow n$ at time T just delays the edge's[q removal with no activity on c in between.■

Theorem 5 *G_T is strongly connected for all T.*

Proof. The graph is clearly connected in the base case in which no replica exists. Thus, G_T is connected for all T from Theorem 3. ■

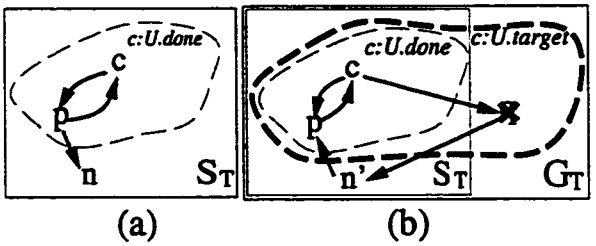


Figure F.1. *A coordinator c may fail to contact a node n in two situations.*

F.4 All Replicas Receive the Newest Update

Now, we prove that all nodes receive the newest update by distinguishing two cases. Theorem 6 proves that if an update retires, all nodes must have received the update. Theorem 7 proves that when an update is unable to retire because some of its targets are dead, all remaining nodes still receive the update. These two theorems together prove correctness criteria (1) and (2).

Theorem 6 *Suppose S_{T_s} is weakly connected, no node or link newly fails, and no new update is issued during a long enough period (T_s, T_e). Let U be the newest update generated before T_e. If a coordinator c begins the retirement of U at time T ($T < T_e$), then $V_{S_{T_e}} \subseteq c{:}Udone,T$; that is, all nodes in $V_{S_{T_e}}$ have received and applied U.*

Proof. We need only to prove that $V_{S_T} \subseteq c{:}Udone,T$; if $V_{S_T} \subseteq c{:}Udone,T$, then $V_{S_{T_e}} \subseteq V_{S_T}$, because no newer update is generated after T and no node possesses a stale update after T.

For the sake of contradiction, suppose $V_{S_T} \not\subseteq c{:}Udone,T$.

Because S_T is weakly connected from Theorem 4, we can pick a node $p \in c{:}Udone,T$ such that $\exists n \in V_{S_T} - c{:}Udone,T$ and that either $n \xrightarrow{S_T} p$ or $p \xrightarrow{S_T} n$. Below, we show that no such pair of p and n can exist.

First, suppose a pair (p,n) with an edge $p \xrightarrow{S_T} n$ exists (Figure F.1 (a)). Let T_p be the time c propagated U to p ($T_p < T$). First, the edge $p \xrightarrow{S_T} n$ must have been created at or before T_p, because U is the newest update and any other update that could have created $p \rightarrow n$ would have been rejected by p after T_p. On the other hand, $p \xrightarrow{S_T} n$ must have been created after T_p; otherwise, the edge $c \xrightarrow{S_T} n$ must be in the graph (Figure D.5 ⟨12⟩). Because of this contradiction, this pair (p,n) cannot exist. Second, suppose only a pair (p,n') with an edge $n' \xrightarrow{S_T} p$ exists (Figure F.1 (b)). From Theorem 5, a

path $c \overset{G_T}{\rightsquigarrow} n'$ exists in the full graph G_T (remember, G_T may include dead nodes). Therefore, there exists a dead or uncommunicative node $q \in c{:}U.targets_{,T}$ along the path $c \overset{G_T}{\rightsquigarrow} n'$, and q makes U unable to retire in the first place. Therefore, this pair of nodes (p,n') cannot exist as well. Therefore, for U to retire, $V_{S_T} \subseteq c{:}U.done_{,T}$. ■

Theorem 7 *Suppose S_{T_s} is weakly connected, no node or link newly fails, and no new update is issued during a long enough period (T_s,T_e). Let U be the newest update generated before T_e. If $(c{:}U.targets_{,T_e} - c{:}U.done_{,T_e}) \cap V_{S_{T_e}} = \phi$ on a coordinator node c, then $V_{S_{T_e}} \subseteq c{:}U.done_{,T_e}$.*

Proof. For the sake of contradiction, suppose $V_{S_{T_e}} \not\subseteq c{:}U.done_{,T_e}$. Using the argument that appeared in the previous proof, we can pick a node $p \in c{:}U.done_{,T_e}$ such that $\exists n \in V_{S_{T_e}} - c{:}U.done_{,T}$ and either $n \overset{S_{T_e}}{\rightarrow} p$ or $p \overset{S_{T_e}}{\rightarrow} n$, and we can show that a pair (p,n) with an edge $p \overset{S_T}{\rightarrow} n$ cannot exist.

Now, suppose only a pair (p,n') with an edge $n' \overset{S_T}{\rightarrow} p$ exists. For the moment, let us assume the following lemma holds for any pair of nodes, n_1 and n_2.

Lemma 1 *If $n_1 \overset{S_T}{\rightarrow} n_2$ but not $n_2 \overset{S_T}{\rightarrow} n_1$, then n_1 has an update.*

From this lemma, n' has an update, say, U'. Thus, n' contacts p, which in turn causes n' to discover c (Figure D.5 ⟨12⟩), which in turn cause n' to propagate U' to c, thereby letting c discover n' before T_e (Figure D.4 ⟨5⟩). Therefore, a pair (p,n) with an edge $n \overset{S_T}{\rightarrow} p$ cannot exist as well. Thus, $V_{S_{T_e}} \subseteq c{:}U.done_{,T_e}$. ■

Proof of Lemma 1. Intuitively, the lemma holds, because edges disappear only after an update retires; when an update retires, edges must have spanned any pair of update's targets.

The formal proof is by contradiction: Suppose that $n_1 \overset{S_T}{\rightarrow} n_2$ but not $n_2 \overset{S_T}{\rightarrow} n_1$, and n_1 does not have an update. Let U be the last[2] update that retired on n_1. First, $n_2 \in U.peers$ because $n_1 \overset{S_T}{\rightarrow} n_2$. Because U retired already and $n_2 \in U.peers$, n_2 must have applied U at some time T', and thus $n_2 \overset{S_{T'}}{\rightarrow} n_1$. Now, for the edge $n_2 \overset{S_{T'}}{\rightarrow} n_1$ to disappear, a newer update U' that deletes n_1 (i.e., $n_1 \notin U'.peers$) must have been applied and retired on n_2. However, the retirement of U' is impossible without n_1 applying U' and deleting its replica, thereby violating the assumption that U is the last update retired on n_1 or the assumption that n_1 has no update. ■

[2]Here, "last" means "chronologically last", and it does not necessarily imply "newest".

F.5 No Node Receives a Stale Update

Theorem 8 *Suppose no update is generated for a long period ending at T_e. Let U be the newest update issued before T_e. After U retires, no update older than U is applied on any node.*

Proof. A node may receive an older update after it retires U for two potential reasons: (1) another node that has not received U propagates a stale update, or (2) a network delay causes a message containing a stale update to be received after U retires. Theorem 6 prevents case (1). The use of synchronized clocks, described in Section 6.4.3, prevents case (2), as follows: suppose a node n received a retirement notice of U at time T from node c, then it received an older update U' at time T' from node n'. Let us study how late T' can be. From Theorem 6, n' must have received the update U at time T_3 $(T_3 < T)$. Notice that n' could not have sent U' to n after T_3, because any older update was superseded by U at time T_3. Therefore, $T' \leq T_3 + WAIT < T + WAIT$. Because n keeps U until at least $T + WAIT$ ⟨13⟩, node n rejects U'. ■

F.6 Liveness

So far, we have proved only the safety of the algorithm, i.e., that if an update retires or is unable to proceed because of dead target nodes, all peer replicas must have received the update. This section argues the liveness of the algorithm, i.e., that the algorithm indeed is able to retire the update, or that it is able to push the update to all live replicas when some targets are dead.

For the moment, lets assume there is at least one live coordinator for any update. Then the algorithm terminates trivially for the following reasons. The coordinator sends an update or retire messages to the nodes in *gU.target* until it receives replies from all the nodes. Although *gU.target* may grow as replies arrive (Figure D.4 ⟨5⟩), its size is finite, and the coordinator will eventually push the update to all live nodes in the target set.

Thus, the liveness of the algorithm boils down to proving the existence of at least one coordinator for each update in the steady state. We take the coordinator selection procedure in Section 6.6.2 and show, on a case-by-case analysis, that the procedure leaves at least one coordinator in the system in the steady state.

1. The issuer of the update is live.

The live issuer always becomes the coordinator $\langle 1 \rangle$.

2. The issuer is dead, and no live node has yet received the update from the issuer.

   This case is handled trivially, because the update technically does not exist. In other words, all live replicas agree on the old contents. This behavior may not be what users want, but it is the unavoidable cost of the algorithm's optimism.

3. The issuer is dead, and some live nodes have received the update.

   First, $\forall n, \forall U, \forall T$, if node $n' \in n{:}U\mathit{done}{,}T$ then n' has update U. This is because a node is added to $gU.\mathit{done}$ causally after the node has logged the update. Second, $\forall n, \forall U, \forall T$, $n \in n{:}U\mathit{done}{,}T$ if n has update U, because update application on a node happens transactionally.

   Thus, on any node n, all nodes in $\mathit{members} \cap n{:}gU.\mathit{done}$ always have update u (Figure 6.5). Notice that we use the fact that the membership protocol converges the membership views of nodes in the steady state (Appendix E.1).

   Furthermore, $\mathit{members} \cap n{:}gU.\mathit{done} \neq \phi$, because the node itself is always in $gU.\mathit{done}$. Thus, at least one live node elects itself as the coordinator for any update.

Vita

Yasushi Saito was born in the village of Takahama, Kagawa, Japan. He followed his nomadic parents and lived in Kyoto, Dallas, Osaka, and Tokyo before moving to Seattle. He earned Bachelor of Science and Master of Science degrees in Information Science at the University of Tokyo. In 2001, he earned a Doctor of Philosophy in Computer Science at the University of Washington.