RESOURCE MANAGEMENT IN FEDERATED COMPUTING ENVIRONMENTS

Luis L. Cova
(Thesis)

CS-TR-282-90

October 1990

Resource Management in Federated Computing Environments

Luis L. Cova

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

October 1990

© Copyright by Luis Leopoldo Cova Franco 1990
All Rights Reserved

To Rosaura, my companion for life

Viejos leños para quemar.	*Old logs to burn.*
Viejos libros para leer.	*Old books to read.*
Viejos vinos para beber.	*Old wines to drink.*
Viejos amigos para confiar.	*Old friends to trust.*

D. Alipio Peréz Tabunero

Resource Management in Federated Computing Environments†

Luis L. Cova

ABSTRACT

This dissertation demonstrates that resource management mechanisms designed to support several degrees of autonomy allow the establishment of effective cooperation among autonomous computing sites. Using two distinct resources, storage and processing, this dissertation introduces two distinct notions for autonomy: *functional autonomy* and *service autonomy*. For functional autonomy, it is shown that is practical to implement a network file system that provides increased client independence from the file server while maintaining a useful level of functionality. For service autonomy, a load sharing mechanism has been implemented that provides sites' owners with ways to control how much performance degradation local jobs will perceive when remote jobs are serviced.

This dissertation claims that, although some very large distributed systems will be formed by the growth of integrated distributed systems, most will be created by joining together a number of separate, already existing, and independent distributed systems. In an environment composed of a multitude of cooperating sub-systems the autonomy of each entity must be respected in order to convince the owners of each of the separate components to join the federation. Consequently, any viable distributed system architecture must support the notion of autonomy if it is to scale at all in the real world.

† This dissertation was partially supported by New Jersey Governor's Commission Award No. 85-990660-6, IBM Research Initiation Grant, IBM Graduate Fellowship, and a grant from SRI's Sarnoff Laboratory.

Acknowledgments

There are three persons at Princeton that merit my highest gratitude. My advisor Rafael Alonso, for his constant support and advice in academic and personal matters, as well as for all the experiences we shared. He was always kind, understanding and friendly.

Daniel Barbará, my friend and **honorary** co-advisor, who collaborated in parts of this dissertation and with whom I shared many dreams, doubts and beers. As *un-official* reader of this dissertation, his comments were invaluable.

Hector Garcia-Molina, the first friendly face I saw here. He helped me feel welcome during my first year at Princeton. During my entire stay he was always ready to listen to my ideas, to provide helpful insights, and to support unusual endeavors. As a reader of this dissertation, he was influential in making it better. I will always remember his *fatherly* nature.

I would also like to thank Kai Li for being one of my dissertation's readers. His suggestions helped improved this presentation.

Many thanks to the friends I gained at Princeton, specially, Father Robert Ferrick, Rees and Carol Thomas, Jane and Fisher Brooks, Brother Bob Berger, Felix Velez, Astrid Arrarás, Linos Frantzeskakis, Karin Petersen, Steve Kugelmass, and Don Ferguson.

I want to make known my deepest appreciation for my parents, my brothers, my sister, and my grandmothers, for never doubting in me, always being proud of me, and never letting me forget it. Their love is one of the most important parts of myself.

I am most grateful to Rosaura, my wife, who endure my graduate studies from afar and from near. Her faith in me is my strength. Her love is my fortress. Her beauty is my delight.

Luis L. Cova
Princeton, New Jersey
August 24, 1990

Table of Contents

Abstract i

Acknowledgment ii

Table of Contents iii

Chapter 1. Federated Computing Environments 1

- 1.1 Introduction 1
- 1.2 Distributed Systems 2
- 1.3 Federations 3
- 1.4 Resource Management 5
- 1.5 Autonomy 6
- 1.6 Related Work 7
- 1.7 Organization of the Dissertation 10

Chapter 2. Stashing 13

- 2.1 Introduction 13
- 2.2 Design Issues 16
  - 2.2.1 File Selection 17
  - 2.2.2 Data Consistency 19

2.2.3 Data Integration 24

2.3 Quantifying Availability Gains 26

2.4 Related Work 33

2.4.1 Distributed Caching 33

2.4.2 Replication 35

2.5 Summary 38

Chapter 3. FACE: a file system architecture for Federated Computing Environments 39

3.1 Architecture 39

3.1.1 Network File System Interface 40

3.1.2 Bookkeeper Processes 44

3.1.3 Integration Modules 45

3.1.4 User-level Tools 51

3.2 Prototype 52

3.2.1 NFS Overview 54

3.2.2 New Data Structures 56

3.2.3 New and Modified Kernel Routines 57

3.2.4 New System Calls 57

3.2.5 The Bookkeeper Process 60

3.2.6 An Example 61

3.2.7 Performance 62

3.2.8 Prototype's Weakness 67

3.3 Summary 68

Chapter 4. Load Sharing 69

4.1 The Load Distribution Problem 70

4.1.1 Designing Load Distribution Schemes 71

4.1.2 Integrated Distributed Systems & Federated Computing Environments 71

4.2 The High-Low Scheme 73

4.2.1 Description 74

4.2.2 An Analytical Model for Setting High-mark and Low-mark 79

4.3 High-Low, lsh and no-ld 80

4.3.1 Experiments and Results 82

4.4 High-Low, All-or-Nothing, and Priority 89

4.4.1 Description of Emulated Acceptance Policies 90

4.4.2 Experiments and Results 91

4.5 Summary 99

Chapter 5. Conclusions & Future Directions 101

6.1 Lessons Learned 102

6.2 Future Work 103

6.2.1 Propagating Updates to Replicated Copies 103

6.2.2 Bootstrap Negotiation 106

6.2.3 Exchanging Data Among Heterogeneous Database Systems 107

References 110

Chapter 1. Federated Computing Environments

Very Large Distributed Systems (VLDS) will most commonly arise by joining together several independent and previously existing distributed systems in a **Federation**. This dissertation presents two resource management mechanisms that illustrate how cooperation among autonomous components of a federation is effectively established when these mechanisms support several degrees of local autonomy.

1.1. Introduction

The next generation of distributed systems will be formed by hundreds of thousands of autonomous computing sites. These VLDS will arise in either of two ways. One method is by smaller integrated distributed systems growing from within by the addition of new users and equipment. Alternatively, VLDS will come into existence by several independent distributed systems joining together to form a federation.

In federations of autonomous computing systems two main issues will have to be dealt with. One issue is the heterogeneity of hardware and software among sites. The other is the interplay between autonomy and cooperation among the federation's components.

This dissertation claims that administrators, user communities and sites' owners will not give away control over their local resources for even the most wonderful global system. Instead they will want to maintain control as if they were not part of the federation, but at the same time they would like to to be able to expand their site's capabilities by using other sites' resources. Hence, the focus of this dissertation is to design and evaluate mechanisms that manage shared resources maintaining local autonomy of the cooperating sites.

In this chapter we explain what a distributed system is and we define the particular type of distributed systems that we study: a *Federated Computing Environment*. Then we describe resource management in distributed systems, explaining the role of

autonomy in this area. We end this chapter describing related work and giving a brief description of following chapters.

1.2. Distributed Systems

Experimental distributed systems research began at the end of the 1960s with the establishment of Arpanet by the (Defense) Advance Research Projects Agency. The development of TCP/IP and networking capabilities for BSD UNIX† were direct results of this effort. Commercial companies have also been offering for a some time their own networking solutions to their customers, for example, International Business Machines' Systems Network Architecture (SNA). But it was the advent of the personal computers, the workstations and the widespread use of local area networks (LAN) in the eighties that launched distributed systems from research into everyday life.

The phrase *distributed systems* has many definitions, but a widely accepted one is: "a set of *loosely coupled* processing elements that interact among themselves." It is understood by *loosely coupled* that no shared physical memory exists among the processing elements, i.e., the interactions are done by sending and receiving messages through an underlying communication network.

A distributed system can also be characterized using Schroeder's list of "symptoms" [Mullender89a] :

1) *Multiple processing elements*, that function independently of each other.
2) *Interconnection hardware*, through which the processing elements communicate.
3) *Components fail independently*, i.e., portions of the system will keep providing service to its users in the face of a bounded number of failures.
4) *Shared distributed state*, that represents the status of the system at any given point in time.

---

† UNIX is a trademark of A.T.&T.

The most widespread instances of a distributed system found today are those built around LANs. These systems are characterized by having the processing elements (mostly minicomputers, workstations, and personal computers) connected by *carrier sense multiple access* (CSMA) networks. The best known example of CSMA is the Ethernet [Metcalfe76a]. Main characteristics of this type of systems are that the processing elements are within a couple of miles and the whole system is under a single administration domain.

The next most frequent encountered distributed systems are those that provide communication among branches of a company and its main office. Most of this type of systems use vendors' specific products, such as SNA.

Today, major efforts by government agencies, academic institutions, research laboratories, and private companies are in their way to create a national network built around a loose federation of smaller, private and public networks. NFSnet is a clear example of this type of effort. Is in this type of computing environment that we are most interested.

1.3. Federations

In this dissertation we will concentrate ourselves with federated computing environments. We will define such environments as follows:

> **Definition 1.1**: a *Federated Computing Environment* is a distributed system that crosses several administration domains and where a permanent central authority does not exist.○

Networks like Internet and Bitnet are the best known examples of federated environments. This type of environment can be characterized by three properties:

1) *heterogeneity*, of software and hardware components, as well as of level of experience among the user communities;

2) *scale,* which can be in the order of hundred of thousands of components.

3) *autonomy* , of the federation components from each other and from any centralized resource.

The heterogeneity of hardware and software among the sites is an important issue in distributed computing research. The ISO Open Systems Architecture [Zimmermann80a], several research projects ([Kruger88a], [Notkin88a], [Needham82a]) and several development efforts ([SUN88a], [Dineen87a]) directly address this issue. We can safely say that during the '80s the experimental distributed systems community has actively pursued solutions to many questions related to heterogeneity.

Scaling has lately become a popular topic of research. A distributed system protocol is said to be *scalable* if it can keep working efficiently as the number of sites in the system increases. To support interactions among large number of components, decentralization of resources and control becomes inevitable. Most of the algorithms traditionally used in experimental distributed systems have been designed to take advantage of LAN-base technology, therefore they do not necessary scale well. For example, the V kernel [Cheriton88a] and the Emerald system [Jul88a] both use broadcast messages to locate the address for a given name. Because of the large number sites in a VLDS, the extensive use of broadcast messages would quickly congest the communication network. Other systems, like Grapevine ([Birrell82a], [Schroeder84a]) and Amoeba [Tanenbaum89a], restrict the use of broadcast messages to small subsets of the system. There are dedicated name servers that handle the name space outside the smaller subsets. Another interesting research on scale is the work on the Global Name Service [Lampson85a]. It focuses on the design of a name service that supports a large naming space that changes very frequently. In [Neumann88a] there is a good survey on the topic of scale in distributed systems.

The issue of autonomy deals essentially with the interrelation between local control and cooperation among the federation's components. This issue is widely acknowledged by the research community, but its study still is in a preliminary stage. Its repercussions are best felt when managing share resources in a distributed system.

1.4. Resource Management

It has long been recognized that **network transparency** is the fundamental characteristic to be achieved by resource management mechanisms in distributed systems. A computer network achieves network transparency when users and user programs access any resource in the same manner, independent of the resource physical location or the user physical location. While it is easily achieved for networks under a single administrative control, through the enforcement of standards and homogeneous components (e.g., Cambridge Distributed System [Needham82a], V-kernel [Cheriton88a], Locus [Walker83a]) the concept is less feasible for federated environments where multiple administrations, each controlling a subset of the computing sites, exist.

Very often, a distributed system is structured so that a given service is transparently shared among the users (e.g., file storage). This usually is implemented by having a dedicated machine (a server), or group of servers, that perform the service for the users, who are unaware of their existence. Although this scheme has many advantages (e.g., centralized administration of the service), it results in a dependence of the users on the dedicated machines for this particular service. Frequently, this translates to an inability to obtain the service if the communication with the servers are not possible, as in the case of a network failure. This is true even in cases where the user has a powerful local machine that could be capable of performing the servers' function, although perhaps with a degraded quality. In federated environments this issue of autonomy among components is even more fundamental.

Thus we believe that what is important in managing resources in federated environments is how to balance the desire for local control of individual sites with the wish for cooperation among sites. Is the basis of our work that what is needed are mechanisms to allow different modes of cooperation depending on the level of autonomy of the sites interacting. That is, we believe autonomy is not a binary property, where either the sites have unrestricted sharing of resources, or have no sharing at all, depending whether they belong to the same administration domain or not. Instead, several types and degrees of site autonomy will exist.

1.5. Autonomy

We can distinguish two definitions for the concept of autonomy in federated computing environments. Given two distinct computing sites A and B we define the following types of autonomy:

> **Definition 1.2**: *Functional Autonomy*, A is *functionally autonomous* from B with respect to a function X, if A can perform X regardless of the state of B. ○
>
> **Definition 1.3**: *Service Autonomy*, A is *service autonomous* from B if A can unilaterally refuse to service a request issued by B at any point in time. In other words, A does not make any guarantee of service to requests coming from B. ○

In this dissertation we study examples of resources and present techniques that allow cooperation while guaranteeing the autonomy of each site. We propose that mechanisms to support cooperation in federated environments should be built to be flexible with respect to the autonomy of each component. In this way resource sharing can be done more extensively since it will be customized to the level of cooperation and autonomy that the interacting sites agrees upon.

We illustrate our ideas by focusing on mechanisms for two types of resources. First, we study network file service to show how clients can be less dependents on

the status of the servers, thus preserving their functional autonomy (Definition 1.2). Second, load sharing among autonomous computers is explored to show how a level of performance can be guaranteed to local processes while still allowing local execution of remote jobs. In this case we exemplify service autonomy (Definition 1.3).

1.6. Related Work

Several research projects have explored issues related to VLDS. Mainly, they have concentrated in how to make large, heterogeneous distributed systems work. In this section we survey several of these projects.

In [Sheltzer86a], the Locus distributed operating system is extended to operate transparently across long haul links in an internet environment. Their approach uses semantic-based protocols, exploits locality in the operating system's functions (using mostly caching), and selects execution sites (when necessary) on the basis of data location to minimize data movement across wide area networks. Most of this work is geared to demonstrate the feasibility of achieving network transparency in the Internet. They do not consider the issues of authorization, confidentiality of the data, fault-tolerance or autonomy in this type of environment.

In [Renesse88a], wide-area Amoeba is discussed. Amoeba is a distributed operating system based on the processor pool model which is completely network transparent to its users. The goal in this project is to achieve a level of performance of system functions as close as possible as to the performance of centralized systems. They achieve this level of performance by using special purpose remote procedure calls (RPC) to perform the traditional operating system functions. Amoeba was designed for LAN based systems. When Amoeba was extended to function across WANs, the slow and unreliable nature of the long haul links made their protocols too slow to satisfy their performance goal. Their solution to the problem was to establish Amoeba ''domains.'' Each of such domains represent an interconnected collection of LANs. The key characteristic of an Amoeba domain is that

broadcast messages issued within a domain are only received by all the machines in that domain.

Communication across Amoeba domains is done by specialized processes that sit on gateway machines and act as intermediaries. These processes, called agents, are responsible to convert RPC messages within a domain back and forth to messages in the appropriate communication protocol of the long haul link connecting the domains. These agents are under the control of each domain administrator. Therefore, local control can be exercise over which service are being provided from and to a given domain. Although this is an appropriate architecture for federated environments, the Amoeba researchers assume that all the domains are Amoeba systems. Clearly, this is a lot to expect in a VLDS.

Neumann [Neumann88a] presents several design issues that arise due to the scale of VLDS. He surveys the proposed and implemented solutions of several research projects. Neuman states that there are three dimensions to scaling in distributed systems: number of users and computers, distance between the farthest two sites, and organization of the system. In the latter dimension he includes issues of trust among components of the system as well as the number of distinct administrations that the system spans. His approach concentrates on ways to allow users to cope with the large scale of VLDS, since users are generally interested in a small part of the entire system.

In [Turnbull87a], resource sharing across many organizational boundaries in a VLDS is explored. He focus on solving the heterogeneity problem by providing a simple, coherent and consistent base on which to build distributed services. His approach is based on providing an homogeneous distributed operating system kernel to all the computers in the VLDS. This kernel provides two functions: process scheduling and interprocess communication. The rest of the functions are provided as user level services†. Turnbull acknowledges that for his approach to be accepted

† This approach is similar to the V-kernel [Cheriton88a] and Sprite [Douglis87a] approaches

and used by many organizations there must be a way to make the transition from their present system to his architecture. He states that this support must be provided locally and therefore local autonomy and local policies must be taken into account. He suggests that local operating systems could emulate his homogeneous kernel to make the transition easier.

Dash [Anderson88a] is a research project that takes into account projections of future technology on which VLDS will be built. Mainly, they expect that network bandwidth and processor speed will greatly increase while CPUs will become cheaper and multi-processors will be more prevalent. Dash also explores the problems that will arise as administration domains shift from a few large central entities (e.g., universities, corporations, etc.) to large number of domains formed by individuals and small groups. Their focus on autonomy centers on the naming mechanism. They state that this mechanism must support organizational autonomy with respect to delegation of authority for name assignment and with respect to the impossibility of having central trusted entities for name resolution.

The Desperanto research project [Mamrak85a] explores how to provide operating system support for distributed applications in networks of autonomous and heterogeneous computing systems. They acknowledge that linking independent computing sites, which are each under autonomous control, is a common event. Desperanto looks for solutions that do not require changes to the existing software at each site. When characterizing a single component of their distributed environment they state that its mode of participation should be constrained by its need to service its local users. Moreover, they acknowledge that individual computing sites will not accept significant performance degradation in local operations in order to share resources with other sites.

---

which are both tailored for LANs.

In [Flavin88a] a different approach to resource sharing in VLDS is presented. Instead of relying on a distributed operating system that spans the entire network, their approach is based on the careful separation of server, application and user interface function of distributed software; on the centralized maintenance of the application interface function; and on the careful design of the program interface. By focusing on the structure of each distributed software instead of the distributed environment where they will run, they achieve a local autonomy of those aspects of the software that merits it.

Most of the above mentioned projects concentrates in making VLDS feasible by studying different mechanisms to allow interaction among heterogeneous sites, and solving the problems that arise from scale in naming, authentication, access control, directory service, etc. Few of these projects have explored how the autonomy of the individual computing sites, that are part of the federated environment, will shape the software of these VLDS. The contribution of this dissertation is to show how cooperation among autonomous sites can occur while guaranteeing the autonomy of each site in an environment where no central authority exist and thus agreements cannot be enforced.

1.7. Organization of the Dissertation

In this dissertation we will proceed as follows. In the next two chapters we present our work on distributed file systems for a federated computing environment. In such environments we expect that servers will have the freedom to deny service to any user (*service autonomy*), thus we believe that the proper view of a server is that it is the best place to obtain a resource, rather than the notion of a server being the only place in which to do so. Consequently, a client has to be ready to take on a given service when the corresponding dedicated machines are not accessible. In the case of distributed file services, a possible approach consists of storing at the server the latest copy of all user files, while keeping at client sites copies of possibly

older versions of the most crucial information. Thus, even after a failure users may be able to proceed with their work (*functional autonomy*), although in a degraded manner.

The idea of keeping local copies of key information has been called **stashing** in the literature. We augment the usefulness of **stashing** by combining it with the idea of **quasi-copies** [Alonso90a] (replicas of a data item that are allowed to diverge from the primary data in a controlled, application-dependent fashion), which eases the cost of maintaining replicas. With quasi-copies, the notion of *controlled inconsistency* can be introduced in distributed systems. In Chapter 2 we present the main design issues that arise when incorporating stashing to a distributed file system.

In Chapter 3 we describe the design of a distributed file system architecture called FACE, that incorporates **stashing** and **quasi-copies**. We also discuss a prototype built by modifying Sun's Network File System. We show performance measurements of this prototype

In Chapter 4 we focus our attention to load sharing among autonomous computers. In some systems, load sharing has been accomplished in an all or nothing fashion, i.e., if a node is idle then it becomes a candidate for executing a remote workload, otherwise it is not. Other attempts have used priority schemes where remote jobs are run with lower scheduling priority than local jobs. These styles of sharing are too restrictive in an environment where most resources are underutilized. Also, they do not scale well as the system load increases. We present a scheme that replaces these sharing approaches with a gradual one, i.e., where each machine in the network determines the amount of sharing it is willing to do (i.e. the level of *service autonomy* it will preserve). The scheme, called **High-Low**, makes sure that the service provided to local jobs of a lightly loaded node does not deteriorate by more than a predefined amount. It simultaneously helps improve the service at heavily loaded nodes. In Chapter 4 we empirically compare different

approaches to load sharing and show that it can be effective in a federated computing environment.

Finally in Chapter 5, we give our conclusions and future directions of research. We present the interesting theoretical problem of deciding how to efficiently propagate updates to quasi-copies disperse throughout a federated environment. We then discuss the possibility of an automatic process that would allow entities of a VLDS to negotiate the type of interaction they will permit among themselves (e.g., decide the transmission protocol to be used.) We conclude by presenting some ideas for a layered architecture to allow cooperation among heterogeneous database systems.

Chapter 2. Stashing†

In federated computing environments one type of autonomy that has to be preserved is the independence of sites from other sites (*functional autonomy*). This chapter focuses on the design issues of a technique called **stashing** for distributed file system that provides increased client independence from file servers. The usefulness of stashing is increased by using a **quasi-copy** framework to manage the consistency between servers' files and the client's stashed copies. It is claimed that by allowing applications to specify the degree of inconsistency of the data they use, users at client sites can continue to access files even when a disengagement from the server occurs, without the necessary overhead that perfect consistency would require. It is shown, using probabilistic analysis, that stashing does increase the availability of data to applications.

2.1. Introduction

Leslie Lamport once described a distributed system as "*one that stops you from getting any work done when a machine you've never even heard of crashes.*" [Mullender89a]. It is often the case that by distributing functions among several machines, the fault-tolerance of the whole systems is reduced instead of increased.

This is a problem of the traditional *client-server* model, a model which is widely used in many distributed systems. In the client-server model, there are two types of machines: clients and servers. Servers are dedicated machines that provide a specific service (e.g., file storage). Clients are machines that users use and from where request for services are issued (e.g. reading and writing to a file). The client-

† A paper based on preliminary versions of parts of Chapters 2 and 3 will appear in the proceedings of the Ninth Symposium on Reliable Distributed Systems, IEEE/CS Press, Huntsville, Alabama, October 1990. A shorter description appeared in the proceedings of the second Workshop on Workstation Operating Systems, pp. 1-5, IEEE Computer Society Press, September 1989,

sever paradigm is useful to enhance the capabilities of any particular computing installation.

The functional dependency of client machines on servers to acquire a given resource can be more severe in VLDS, since remote services can be overloaded by many simultaneous requests. Moreover, in federated environments, since servers and clients may belong to different administration domains, changes to a server, or to a client, may produce service disruption for which certain sites were not prepared for. In this chapter we explore how to avoid the Lamport effect we just described, i.e., we study a mechanism to allow client machines to keep making progress when they secede from the federation to which they belong.

One commonly used approach is to use redundancy techniques to augment the availability of the essential information, i.e., to replicate servers to reduce the probability of a service not being available. (Locus [Walker83a] is an example of a distributed operating system based on this idea.) This solution is inherently expensive due to the maintenance of the copies' consistency and does not ultimately solve the problem. It just lowers the probability that the service will not be available.

Another approach that has been used in an ''ad-hoc'' manner for a long time, and was formally discussed and named at the 1988 SIGOPS European workshop [Birrell88a] is **stashing**. Stashing designates a class of techniques to store key information for use when a system disengages from the network it belongs to (either voluntarily or involuntarily).

Stashing does not mean that the concept of a server becomes useless. Rather, it indicates that the correct manner to characterize a server is as the best place to receive service and not as the only place. Consequently, clients can receive a given service although the best place (the server) is not available. For example, a file server may contain fast storage devices, the latest copy of all the information, archival storage, and guarantee frequent backups. If the file server cannot be used,

users may be happy having slow access to a local storage device containing hourly snapshots of some of their files (for example, the most frequently accessed ones).

The actual details of maintaining a stash have not been fully discussed in the literature. In particular, one of the issues that have not been addressed is that of how to maintain consistency between the stashed information and the remote copy. In this chapter we present the main design issues of implementing stashing to increase the autonomy of client machines in a distributed file system. These issues are:

(1) Selecting which files are vital for functioning when the file server is inaccessible.

(2) Deciding how consistent the data in these files is going to be. The more consistent the higher the performance cost that we will have to pay. For instance, if we demand perfect consistency, the local copies become replicated copies of the files in the remote server, and we have to preserve consistency by resorting to well known protocols (e.g., two-phase commit [Gray78a].) Since such protocols will be unacceptable in general, we propose to reduce the overhead by using the notion of **quasi-copies** [Alonso90a]. Quasi copies are replicated ''copies'' that may be somewhat out of date, but are guaranteed to meet a certain consistency predicate.

(3) Integrating versions after a failure. In general, the local users can make updates to the stashed data while the file server is unavailable. Since more than one user might be performing updates during periods of disengagement, we might be faced with diverging copies in the moment of reconnecting the users to the file server. A mechanism is required to produce a single, integrated copy of the file to store back in the file server.

We should stress here that stashing should not be confused with the concept of caching. Stashing differs from caching in that the data is pre-fetched and pre-

loaded to be used only when a failure makes the server unreachable. The difference between caching and stashing may be highlighted by pointing out that caching stores locally for performance while stashing saves locally for fault-tolerance. Both caching and stashing use remote stores to provide better storing service (e.g., ease of sharing, larger disk space, archival storage, etc.) during normal operation of the system, i.e., when failures are not present. It is worth noticing that stashing and caching are **not** competitors. Both can coexist in the same design of a distributed file system, contributing together to the availability and performance of file accesses. In fact, stashing can be implemented on top of any existing system, to enhance its availability. It is also possible to study how to make use of the caching space for stashing purposes. However, we feel that by studying a mechanism solely to increase clients' autonomy (and thus file availability), we can isolate the important issues and provide tailored solutions.

The format of this chapter is as follows. In the next section we explore in greater detail the issues presented above. In Section 2.3, we develop a probabilistic model to quantify the availability gains that stashing might deliver to an application. In Section 2.4, we compare our work to related approaches in the literature. Finally, in Section 2.5, we present a summary of our ideas.

2.2. Design Issues

In this section we present in greater detail the relevant issues of stashing, but before we discuss these issues, we briefly mention how we expect that stashing might be used.

Before the user starts working on a particular application, he or she selects the files that are essential to completing the task. Clearly, different users will require more or less sophisticated methods for selecting the files, and we will discuss some of them in Section 2.2.1. After the essential files are chosen, the system will make copies of them in, say, the user's local workstation. During normal operation of the

system all updates to the files are installed at the server (i.e., they are not performed on the stashes), so we have to address the issue of how to keep the stash consistent with the originals. Keeping the stash perfectly up to date might be expensive, and in Section 2.2.2 we present a technique for minimizing this cost. This technique involves letting the stashes diverge from the originals but not so much that they become useless. For example, for some applications, files that are guaranteed to be no more than ten minutes old might still be useful even though they are not perfectly up to date. If at some point there is a network failure, the user can now use the stash copies. The copies might be a little out of date (e.g., they might be ten minutes old in the example above), but they are still useful. If the user proceeds to modify the stashes, then there must be an integration phase (with the modified stashes of other users) when the network failure is repaired. This issue is discussed in Section 2.2.3.

2.2.1. File Selection

We now discuss how a user selects the files that will be stashed (or dropped from the stash). While considering how the selection process is done we must keep in mind that the user community of most systems is quite heterogeneous, and users may differ in their level of computer sophistication. The view of stashing that a sophisticated user may be expected to have is quite different from that of an inexperienced one. Moreover, the less a user understand the application being used, the more automated the selection process would have to be.

There are a number of ways in which an experienced user will be able to select the files that need to be stashed. There are both static and dynamic types of choices. The static approach is more suitable for files which the user almost always needs stashed (for example, the operating system files, a compiler, a favorite editor). The method for specifying such ''permanently stashed'' files is via a list of names kept in a well-known file in the user's home directory, e.g., a ''.stashrc'' file. While the

knowledgeable user will add and delete file names from the .stashrc file, less experienced users can simply use a default .stashrc provided by the system administrator.

Of course, we require a more dynamic way of specifying files to be stashed. In general, users will be involved in completing a certain task (or a collection of tasks) that is important enough that they do not want its completion jeopardized by a possible communication failure. Users who understand their application very well can simply use the stash command (i.e., ''stash filename'') to select the files to be stashed. Those involved in more complex projects will probably already have a mechanism equivalent to the UNIX *make* facility [Leffler84a] in order to keep track of the files in their application. Stashing can also be done by extending the stash command so that it can understand the format of the *make* file and then proceed to stash every file that is mentioned in the *make* file (either data files or commands).

Somewhat less sophisticated users may not be as inclined to think beforehand about their applications and the files that they will require. For them, another approach may be offered. Just before the user is about to start on his usual activity cycle (e.g., edit, compile, run, etc.) the user will invoke the ''record_stash'' command. From this point on, and until an ''end_record_stash'' command is given, every file (data or otherwise) that is used will be added to the collection of files to be stashed. Thus, once the user has gone through one application cycle he or she will be able to continue his or her work, at least for the near future. A way of implementing this facility in window systems is to associate a *stashing* attribute with each window. If the attribute is set, a ''record_stash'' is issued on the window, thus stashing every file touched by commands entered in the window.

Even this last approach may not be appropriate for less sophisticated users. For them, there are two choices. The first is that the application itself takes care of the stashing. In general, the designer of an application can determine the essential files required by the application to run. The application designer can embed stash-

ing calls into the code of the application so that these required files are stashed in the user's local system. For example, a spreadsheet program can, on its user's behalf, stash a copy of itself as well as one of the files being processed. This approach certainly requires very little input from the user. Alternatively, the system can save a "working set" of files, i.e., stash the files that are accessed in the last δ commands, where δ is a predefined, tunable parameter. In the latter case, the stashing facility will be performing some of the tasks of a file caching mechanism. If there is such a caching facility already in place, then the stashing mechanism could take advantage of the cached files.

A final point that must be addressed is that of how difficult it is for users to determine which files to stash. Our experience with our prototype (to be presented in the next chapter) suggests that it is straightforward for users to select the essential files for stashing. In hierarchical file systems such as UNIX-type, users tend to keep related files in the same directory (or sub-tree). Consequently, after ensuring that certain obvious tools (e.g., the editor) are available, stashing the files in the current directory seems sufficient in practice for most of our applications.

2.2.2. Data Consistency

Stashing files is somewhat similar to replicating them throughout the system. Thus, in stashing we must also consider the issue of consistency between the stashed copies and the original file at the server. Clearly, we have to keep in mind that the more consistent we demand the copies to be, the higher the performance overhead that we will incur in maintaining the consistency.

If one wants to have perfect consistency for the copies at all moments, i.e., to keep them identical to the file in the server, there is no alternative but to use a two-phase commit protocol (or an equivalent mechanism). That is, every time an update is produced at the server, all the copies should be locked and the update propagated to them. This is costly to perform. Certainly in federated environments (where the

possibility of thousand of copies exist) this is an impractical solution. Moreover, for many applications we would be paying a high price for an unnecessary service. For instance, if a user is working to meet a paper deadline and the network fails, he or she may be content with having a version that is several minutes old. Of course, if the application requires perfect data, we must be prepared to pay the price of using a protocol that maintains the stashed copies consistent with the server copy at all times.

Using information that might be stale, but still useful is not novel. For example, using old data is already common practice in databases (i.e., snapshots [Lindsay86a]), as well as in distributed caches [Terry87a], just to name two applications. However, in these applications the degree of inconsistency between the local version and the primary copy is left unspecified. Clearly, there is a wide spectrum of possibilities between fully consistent data and data that is simply known to be out of date. To resolve this issue, the notion of **quasi-copies** was developed in [Alonso90a]. Quasi-copies are replicated ''copies'' that may be somewhat out of date but are guaranteed to meet a certain consistency predicate. Although the idea of quasi-copies was developed in the context of Information Retrieval Systems, we feel that the notion of *controlled inconsistency* can be useful for many other distributed applications. In particular, we feel that this idea is appropriate in a file system services for federated environments.

With quasi-copies, it is assumed that a central location exists (e.g., a file server), where all the updates are processed, and several copies are spread throughout the network. A predicate is associated with each copy, establishing the degree of inconsistency which can be tolerated. For instance, the predicate can state that the copy must not be more than ten minutes old. The user is free to choose from a spectrum of consistency specifications. These may range from demanding a perfect, consistent copy, to settling for a stale snapshot. The system guarantees, in one

of two ways, that the specified predicate is not violated when updates occur. In the first one, the server constantly watches for updates and becomes responsible for propagating them when the predicate is about to be violated. Alternatively, the clients could be responsible for the consistency of their copies, requesting fresh ones periodically. (Notice that clients can only maintained age-dependent predicates.) We call the two ways of maintaining consistency *server maintained* and *client maintained*, respectively. The same file may be client maintained for some of the copies and server maintained for others.

There are several types of predicates that are useful in keeping quasi-copies of files. For the following definitions let x be a remote file at a file server and x' a stashed copy of x. Let $x(t)$ be the content of the file at time t. Let $v(x(t))$ be the version number of file x at time t.

(1) *Delay Condition.* It states how much time a stashed copy may lag behind the file server copy. For file x, an allowable delay of α is given by the condition

$$\text{for all times } t \geq 0 \, \exists k \text{ such that } 0 \leq k \leq \alpha \text{ and } x'(t) = x(t-k)$$

Since this defines a window of acceptable value, we use the notation $W(x) = \alpha$ to represent this condition.

(2) *Version Condition.* A user may want to specify a window of allowable values, not in terms of time, but of versions. For example, if a file represents a VLSI circuit, it may be useful to require a copy that is at most two versions old. We represent this condition as $V(x) = \beta$, where β is the maximum version difference. That is , $V(x) = \beta$ is the condition

$$\begin{aligned} &\text{for all times } t \geq 0 \, \exists k, t_0 \text{ such that } 0 \leq k \leq \beta \\ &\qquad \text{and } 0 \leq t_0 \leq t \\ &\qquad \text{and } v(x(t)) = v(x(t_0)) + k \\ &\qquad \text{and } x'(t) = x(t_0) \end{aligned}$$

(3) *Number of changes.* In this condition the maximum number of updates missing in the stashed copy is bounded by the user. We represent this condition as

$U(x) = \varepsilon$, where x is the file and ε is the maximum number of updates missing.

These predicates are only valid while the file server is accessible. Moreover, transmission delays should be taken into account when enforcing these conditions from the file server, i.e., in the case of sever maintained consistency. To illustrate, consider the condition $U(x) = \varepsilon$ and assume that an update is about to occur in the file server that would violate the condition, i.e., $\varepsilon + 1$ updates will be missing from some stashed copy. Hence the update to the stash must be performed ''at the same time'' as the remote file is changed. Strictly speaking, this is not possible. Since we want to avoid using a two-phase commit protocol (or similar strategy), we propose interpreting every condition $C(x)$ on file x at server j as

$$C(x) \quad \vee \quad W(x) = \delta \quad \vee \quad S_FAILED(j)$$

This means that all conditions have an implicit delay window of δ, where $\delta \geq T_D$, the maximum transmission delay, and also that conditions do not have to be enforced if the server is down ($S_FAILED(j) = TRUE$).

Of course, if the stashed copy is client maintained, none of these problems arise. The client machine will be responsible for asking for a fresh copy of the file when the delay condition is about to become false. The delay in the condition will have to take into account the maximum transmission time plus the maximum response time of the server to a service request (server's time-out). This strategy leaves the file server oblivious to the existence of the stashed copy, offloading the work to the client.

Some optimizations can be made to reduce the overhead of sending files over the network, and to improve the quality of the data stashed. For example, one can make use of cached data, if a file cache is maintained in the distributed file system for performance reasons. Very often, the cache contains the latest modifications made to the file in the local facility. We could copy the contents of the cache to the stash, whenever a local user has made a modification to the file, thereby improving

the quality of the data stashed. Note that integrating cached data with stashed data may be a complex process if the file in question is partially cached in blocks, instead of in its entirety.

Another optimization involves having the client save the version number of the stash files it has. When the client requests a fresh copy (in client maintained predicates) this version number is included in the request. The server will then compare this version number with the actual version of the file, thus determining whether sending the file is necessary or not. Alternatively, to avoid sending large files, one could break the file into segments and apply file comparison techniques [Barbara89a] to find out which segments have changed since the last time the copy was send, and send only those segments.

It is also possible to reduce the overhead imposed on the server to maintain the consistency of the stashed copies. For example, for server-maintained predicates, one could collapse several clients' predicates into a single one (e.g., delay predicates of 5, 10, and 15 minutes could be collapsed to a single one of 5 minutes). This would reduce the number of predicates that the server will have to keep track off at the expense of sending more messages through out the network (e.g., with the above collapsed predicate of 5 minutes three messages would be sent every time).

For client-maintained predicates, we could arrange a propagation scheme of updates according to the clients' predicates, i.e., updating first the clients with the tightest consistency requirements, and they would in turn propagate updates to other clients with less demanding requirements. In this manner the server is only directly contacted by a subset of the clients holding stashed copies.

As a final point, we should mention that so far we have only discussed constraints on a single file and its stash copy. It is also possible to have multi-file constraints. For instance, it might be important for an application to keep in the stash the same version number of two different files which might exist in different file

servers (e.g., the source and the object files of a program.) In this example, the user can impose a predicate covering both files that guarantees that the files will have the same version. For example, if f_s and f_o are the source and the object file respectively, the condition might read as follows.

$$v(f_s'(t)) = v(f_o'(t))$$

The system should now guarantee that the two stashed copies f_s' and f_o' obey that predicate. This implies that when one of the copies is refreshed, either because the server propagated it or the client requested it, the other should also be propagated or requested. The client should make sure that both copies are installed atomically, thereby ensuring that the condition is not violated.

2.2.3. Data Integration

Stashing is useful when the server is no longer available. At this point the client sites with stashed copies will begin to use them, possibly making updates. After the server is once again available, we need to worry about integration of the possibly diverging copies that coexist in the system.

One trivial way of solving the integration problem is by avoiding diverging copies altogether, i.e., only allowing updates to occur in a single group of sites while allowing all sites to read their local copies. The ''updating'' group can be composed of the file server and all the sites that kept connected with it, or if we truly want to operate after server failures, a single group of local sites that remained connected (perhaps making sure that this group always contains the owner(s) of the file). This strategy could be implemented using voting schemes [Gifford79a], but it would be far too restrictive for our purposes, since many local facilities could only read their stashed data. This strategy corresponds to a *pessimistic* approach to recovery.

Alternatively, we could use an *optimistic* recovery mechanism, allowing copies to diverge, and integrating them afterwards. The related problem in

distributed database systems has received a lot of attention in the past (e.g., see [Davidson85a]), and some of the solutions are applicable to our problem, with the difference that instead of transactions, we deal with individual actions to the involved files. One could construct dependency graphs similar to the ones used by Davidson [Davidson82a] and analyze them afterwards, rolling back some of the actions taken to break the conflicts. Or one could provide a patching mechanism [Garcia-Molina83a] that allows the divergent copies to be merged into a final integrated file. But whatever solution is used, it is clear that we need to keep a log at all sites involved which would register all the operations that have been performed to the file. The file server's log has to have all the updates that have been made to the files and are not reflected in some stash. At the clients, all operations done after losing contact with the server have to be registered.

Our plan is to provide application-dependent solutions for this problem. That is, knowing the semantics of the application involved, we can analyze the logs of the different sites making updates to their stashes and determine the set of conflicting operations. Once we do this, we will take the approach of [Garcia-Molina83a] in ''patching'' the file. Briefly stated, this method requires that the application designer (or the users of the application) define, *a priori*, a table of actions to be taken when two conflicting operations are performed to different stashed copies of the same remote file during a network partition. These actions are then applied until no more conflicts remain. Of course, the involved users would have to be notified of the decisions taken. Alternatively, the system could ask for human intervention, presenting the users with the conflicting actions and requesting a procedure to be follow to reintegrate. These requests can be made to the community of users that updated the file, or to a central authority.

We should note that application-dependent approaches have been studied by researchers with similar problems. For example, Horwitz *et al.* ([Horwitz88a]) have

developed a system to integrate diverging versions of a computer program. Also, as described in [Hardwick89a], researchers have implemented a system which generates scripts from diverging engineering designs. These scripts are then used to aid users in building a design that consolidates different features from the individual designs.

2.3. Quantifying Availability Gains

To evaluate the usefulness of stashing, we have implemented a prototype at Princeton that will be discussed in the next chapter. We have had a successful although limited experience with this prototype. It would be highly desirable to have a larger user community taking advantage of our stashing facility for different applications to qualitatively assess the benefits of the system. Since we do not have a large number of users at present, we will try to gauge the possible benefits of a stashing mechanism in two ways. First, we develop a simple probabilistic model, and examine its predicted performance under a variety of assumptions. Second, we outline a typical application and measure the benefits it can obtain under a stashing file system.

Let us define a discrete random variable X that represents the event of an application process accessing a particular block of data within the file space of that application. Formally, $p(x)$, is the probability mass function for the discrete random variable X, i.e., $P\{X = x\} = p(x)$ is the probability of the application accessing block x. For the following analysis we need to determine the cumulative distribution function $F(k)$ of X, which is computed by

$$F(k) = \sum_{\text{all } x_i \text{ such that } i \le k} p(x_i)$$

where x_i represents a data block within the applications file space and k represents the number of blocks that have been placed in the local stash. (Without loss of generality we assume that x_1 through x_k are the data blocks in the stash.) Notice then

that $F(k)$ represents the probability of an application process accessing any block of data within a local stash filled with k blocks.

To measure how much availability an application gains by using stashing we define the following two terms:

Definition 5.1- *Survival*: is the number of file accesses that are issued by an application before the first file access is issued to a file not stashed. ○

Survival can be modeled as a discrete random variable S, with a geometric distribution. Its probabilistic mass function is then given by

$$P\{S = s\} = (1-F(k))\ F(k)^{s-1}\ ,\ s = 1,2,3,...$$
$$E[S] = \frac{1}{1-F(k)}$$

Definition 5.2- *Length of Survival*: is the length of time that the application continues to run from the moment of a failure until the application issues a file access corresponding to a file not stashed. ○

Length of Survival can be modeled as a continuous random variable L. If we define T as the interarrival time of the application's file accesses, and we assume that the time between accesses is independent of the random variable S, the expected valued of L can be computed as follows:

$$E[L] = E[S/T] = E[S]\ E[T]$$

For this analysis we are interested in the mean survival, $E[S]$, and mean length of survival, $E[L]$, as a function of the cumulative distribution function of X, $F(k)$. Therefore, we need to determine $p(x)$, for all data blocks x in the application file space. For this, we assume three different probabilistic models for $p(x)$:

(1) Uniform distribution: a file access is equally likely to access any file in the application's file space. Let the random variable X have a probabilistic mass function given by an uniform distribution. Let N be the sum of all the blocks in the application's file space. The probability of accessing a stashed file is given by the

cumulative distribution function,

$$F(k) = \frac{k}{N}$$

Using $F(k)$ for X we can compute the probabilistic mass function and expectation for S (the survival of the application).

$$P\{S = s\} = \left(\frac{N-k}{N}\right)\left(\frac{k}{N}\right)^{s-1} \quad s = 1,2,3,...$$

$$E[S] = \frac{N}{N-k}$$

Figure 2.1 shows the cumulative distribution function for a total set of files occupying a maximum of 473 Kbytes†.

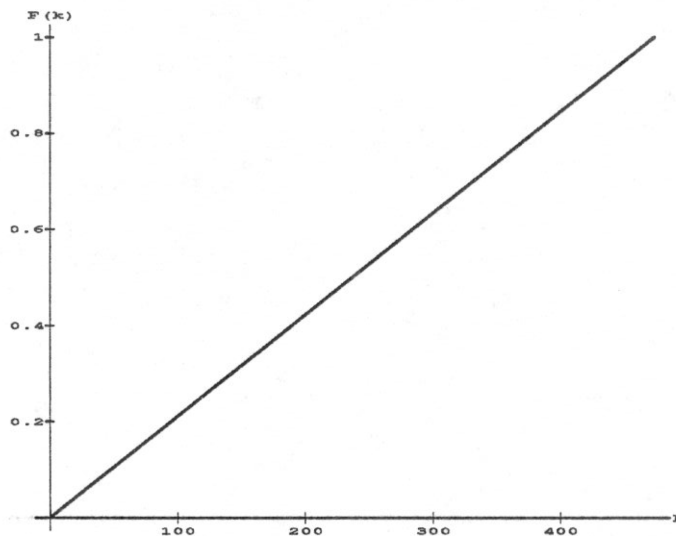

Figure 2.1: Cumulative distribution function for a uniform random variable

(2) Zipf distribution: several file access characterization studies ([Ousterhout85a], [Bozman89a], [Staelin88a], [Floyd86a], [Satyanarayanan81a], [Smith81a]) have observed that file access patterns are heavily skewed, i.e., most of the file accesses

† For the analysis that we are doing here we are using data from a document processing application. Details are given when explaining the "Empirical distribution" later in this section.

go to a small fraction of the files. These studies often quote a 90:10 ratio, i.e., 90% of the file accesses go to 10% of the files. This is true even if the files are divided by types [Floyd86a]. Assuming that stashing perfectly guesses the k file blocks that the application will access next, the probabilistic mass function $p(k)$ follows a Zipf distribution [Zipf49a]. Consequently, the cumulative distribution function $F(k)$ is

$$F(k) = \frac{1}{k^{\alpha+1}\zeta(\alpha+1)} \qquad k = 1,2,3,...$$

where $\zeta(i) = \sum_{j=1}^{\infty} \frac{1}{j^i}$ is the Riemann Zeta distribution [Olkin80a]. Using this $F(k)$ we can formulate the probability distribution function and expectation for S :

$$P\{S = s\} = \left[1 - \frac{1}{s^{\alpha+1}\zeta(\alpha+1)}\right]\left[\frac{1}{s^{\alpha+1}\zeta(\alpha+1)}\right]^{s+1}, s = 1,2,3, \cdots$$

$$E[S] = \frac{s^{\alpha+1}\ \zeta(\alpha+1)}{s^{\alpha+1}\ \zeta(\alpha+1) - 1}$$

In Figure 2.2 we show the cumulative distribution function for a Zipf random variable with a 90:10 ratio, using a value for $\alpha = 2.03895\mathrm{i}^{\dagger}$.

(3) Empirical distribution: assuming that we have the exact file access behavior for a given application, it could be possible to determine the application's Survival during a partition. It is difficult to come up with a probabilistic model that fits the behavior of a particular application, because it depends on the behavior of its user. For this model we derived the behavior from traces belonging to a particular session of a document processing application (vi, troff, and lpr [Leffler84a]). Although this is not statistically significant, it is indicative of what gains we may expect in a particular instance of an application. Table 2.1 presents the files accessed by the user to edit, format and print a particular file. The files are listed in the same sequential

† This value was computed from the average file size of the application measured, which is $\bar{x} = 16$ blocks of 1 Kbyte, and using the equation $E[X] = \frac{\zeta(\alpha)}{\zeta(\alpha + 1)}$

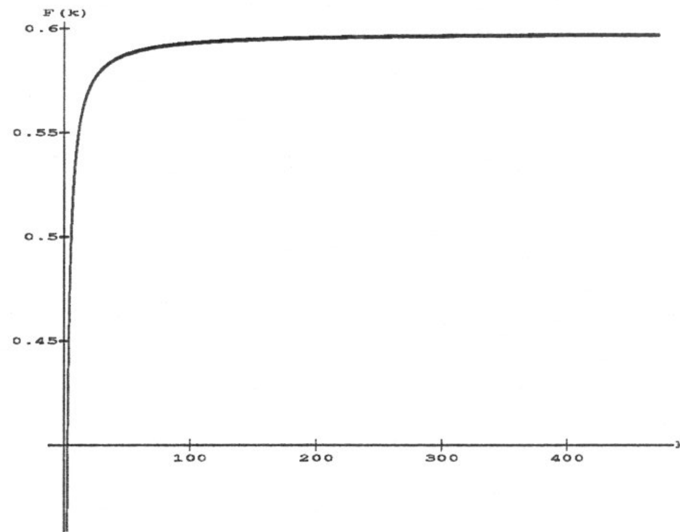

Figure 2.2: Cumulative distribution function for a Zipf random variable

order that they are accessed by the user. In Figure 2.3 we show the cumulative distribution function, $F(k)$, for this empirical measurements. The time that the file is in use (column ''*Time in use*'') is used as a measure of $F(k)$.

file name	*size (Kb)*	*Time in use (mins)*	*Purpose of file*
vi	124	128	editor
intro	13	39.58	user's document
sec2	10	30.58	user's document
model	3	9.58	user's document
sec3	11	33.58	user's document
sec4	13	39.58	user's document
concl	4	12.58	user's document
troff	2	0.5	formatter shell
ditroff	96	0.5	formatter
ms_macros	1	0.5	macro package
fonts_(RBI)	3	0.5	fonts
lpr	47	0.08	printer software

Table 2.1. Files accessed by a document processing application

Using our models, we plotted *length of survival* (L) with respect to the size of the filled stashed space in the local storage (k). Figure 2.4 presents on the ordinate the expected length of survival ($E[L]$), assuming that the expected value of the

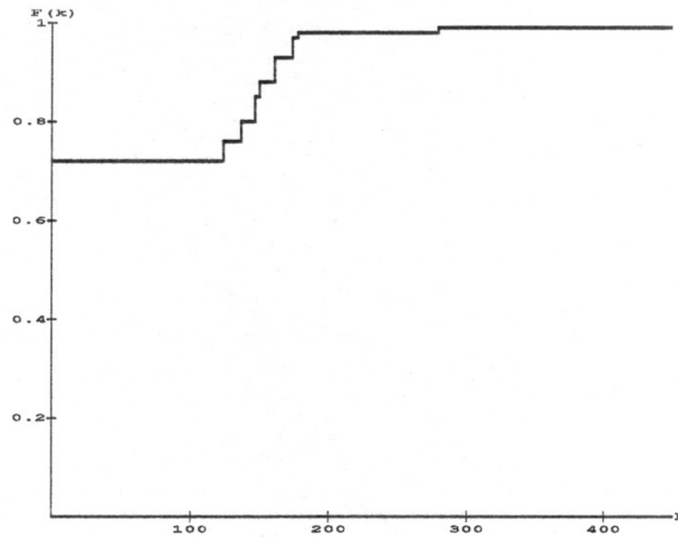

Figure 2.3: Cumulative distribution function from empirical measurements

interarrival time of the application's file accesses is $E[T] = 0.2$ *minutes*. In the abscissa we represent the size of the filled stash space in the local storage device (k).

From measurements made to Princeton University Computer Science department's main file server, we have observed that the average length of an unplanned disruption is approximately thirty minutes. An unplanned disruptions occurs when a given service is unavailable without previous notice to users (e.g. the server crashed or there was an unpredictable network partition). In Figure 2.4 we show with a horizontal dashed line this measured average length of disengagement. The purpose of showing this measure is that it provides a realistic point of reference to how much availability gain is needed by an application.

The curve representing the uniform model in Figure 2.4 shows that this model predicts hardly any gain in availability to applications using stashing. Almost all the files that the application may access have to be stashed so the application can survive any meaningful amount of time. If we now look at the curve representing the Zipf distribution, we can see that if this assumption were true, then with little

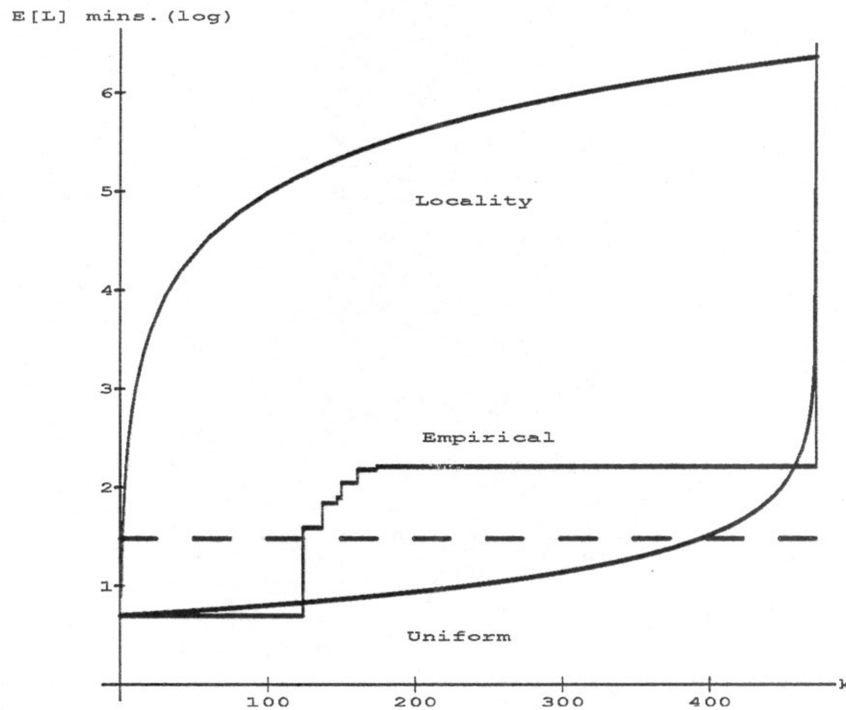

Figure 2.4: Expected Length of Survival for different probabilistic models

stash space filled, the system can keep serving application's file accesses for a relative long time. Of course, we are assuming that the files being stashed are the best choices. In practice, those files may be identified automatically (say, based on previous access patterns), or users may indicate them to the stash mechanism.

Finally, in the empirical case we can distinguish three types of files used by the document processing application:

(1) Those files which are essential for the application to run at all (e.g., the editor);

(2) Those files which have the user data, which are relative small in size and are used extensively (e.g., document's text);

(3) Those files which are needed by the application to complete its task (e.g., fonts, macros, printing command, etc.). These files are often used once and for

a short period of time.

This classification of files is also true for other applications, like software development. The empirical curve shows that the measured application can benefit from a stashing facility to survive for a relative long time compared to the average length of an unplanned disengagement.

2.4. Related Work

Our approach to augmenting file availability can be compared with two different areas of research: distributed caching and replication. These two areas have received much attention from different researchers. For this discussion we will focus on a few representative work done on each of these areas. Table 2.2 shows the systems we surveyed, the area of research they fall under and the type of availability they offer.

Distributed caching	*Replication*	*Availability during disengagement*
Sprite		none
Andrew		partially
Coda		always
V-kernel		partially
	Locus	always
	Deceit	user defined
	Echo†	partially

Table 2.2: Type of availability provided by surveyed systems

2.4.1. Distributed Caching

The purpose of distributed caching is to improve the performance of file access operations (e.g., read and write). For this, the fetch and replacement mechanisms of the caching strategy are designed to store locally the latest version of the data needed by the running application at the moment it is requested.

† Echo is the only system surveyed that explicitly refers to the term stashing. We discuss Echo at the end of Section 2.4.2.

The Sprite file system [Nelson88a] makes extensive use of statefull servers to support high performance main-memory cache at diskless clients. A problem with statefull servers is that recovering from server crashes can be complex and time consuming. Welch et al. [Welch89a] present a mechanism to rebuild the state of a crashed Sprite server with the server's replicated state at its clients. Even with this mechanism, client applications are blocked while their server is not reachable, i.e., modifications are not allow to be done at a client's cache while the server is inaccessible (the same as in NFS). They do this to maintain the consistency of the files.

In Andrew [Howard88a], caching is a fundamental part of the architecture. Andrew caches entire files on demand at client's disks when an open operation is issued. After a file is opened, individual read and write operations are directed to the cache, without any involvement with the server or any client holding the same file in its cache. On a close operation, if the file has been modified, it is copied back to the server's storage. A "callback" mechanism is used by the servers to invalidate clients' cache holding a copy of the modified file, i.e., servers will notify clients when to invalidate their cache, instead of clients periodically checking if their cache are valid.

All of this works fine while there is no network partition or server crash. If at any moment a client cannot reach its server, the client applications can only use the currently opened files. Any other un-opened file that happens to be in the client cache is not usable, since open validation can not be performed, i.e., the client can not check if it has the latest version of these files.

An on-going project at CMU, called Coda [Satyanarayanan89a] is trying to build a successor to Andrew that provides a high degree of file availability when faced with failures (network partitions and server crashes). The goal is to preserve reasonable usability and performance. As in Andrew, they rely on entire file caching on client disks.

Coda's servers are responsible for file replication and operation during partition. Coda also allows disconnected operations of clients, i.e., clients can access all of their cached files even if the corresponding servers are not reachable. Our work overlaps in many areas with that of Coda, in particular on the problem of disconnected operation. The main difference between Coda and the stashing approach is that Coda is extending the Andrew caching facility to provide increased availability of files. As we have mentioned before, the objectives of cache fetch and replacement policies may contradict those of policies to increase availability. Although it is too early to know if Coda's enhanced caching mechanism is successful increasing availability, the close design of Andrew and Coda makes their approach inappropriate for a federated environment. (We have to note that Coda researchers are exploring this issue.) We feel that by studying mechanisms to solely increase client autonomy (and thus file availability), we can isolate the important issues and provide tailored solutions.

Gray and Cheriton [Gray89a] have proposed a time-based consistency mechanism to access cache data in distributed systems. They present the concept of a *lease* which is a token - with an expiration time - that grants the client holding it the right to access the corresponding data item located in its cache. Leases are just a particular example of quasi-copies, using delay predicates, applied to caching. If a client holding a lease can not communicate with the corresponding server, then it can still access its cached files until the lease expires. We see this work as the application of quasi-copy framework to distributed caching. This approach may provide a way of merging caching and stashing under the umbrella of quasi-copies. Further study will have to be undertaken to validate this idea.

2.4.2. Replication

One of the main objective of replicating data has always been to augment availability when failures occur in the distributed system.

In Locus [Walker83a] replication of files is implemented to increase availability, reliability and performance. We agree with the Locus researchers in that allowing updates in all partitions, when a failure occurs, will not in most cases lead to conflicting updates. They present automatic conflict resolution schemes for directories and mailboxes. They use a conflict detection mechanism based on version vectors [Parker83a]. Locus supports replication at the granularity of entire file directories (although the replicated directories do not have to contain all the files). The main difference between stashing and Locus' replication is the quasi-copy predicates that stashing uses to control the inconsistency of the replicas. Locus presents to an user within a partition the ''latest'' version of a file. The ''lateness'' of a version depends on the synchronization facility and update propagation mechanism used by Locus and in no way reflects what the user considers ''good enough'' to use.

Deceit [Siegel89a] is a distributed file system that allows users to adjust systems semantics on a per file basis. The users are able to set parameters for different levels of file availability, performance, and one-copy serializability. The Deceit first prototype uses the NFS client/server protocol. A set of Deceit servers can appear to be a single, highly available and reliable NFS server. Deceit concurrency control is based on a dynamic primary copy scheme using tokens. If a file is going to be modified in a given server, this server obtains a token from the last server that modified the file. Once the token is acquire modifications are allowed. All file operations from other servers and clients are forwarded to the token holder. Deceit allows different methods of replicated access during network partitions, depending on the policy used to regenerate tokens. In particular, Deceit allows updates to be made to replicas across partitions. As in the work presented in this dissertation, they also acknowledge the need to resolve diverging versions of a file using semantics of the corresponding application.

The ideas presented in this chapter overlaps with Deceit's design of client agents. These agents are software modules that reside at client machines. They provide the interface between user processes and the NFS protocol to provide caching and "failover." Failover is the mechanism by which clients access a different server containing a replica of the desire file, when the current server is no longer reachable. We believe that the approach taken in this dissertation to augment availability (i.e. stashing) could well be integrated to the functions of Deceit's client agents.

Echo [Hisgen89a], is a distributed file system that employs replication to increase file availability. The Echo hierarchical name space is divided into two parts: a lower level that holds the logical files (i.e., identifiers to data repositories) and an upper level that holds the global name service (i.e., directories). Each of these parts allow replication, but is implemented with a different consistency scheme. The lower level uses a dynamic primary site and majority voting consensus to allow updates to a replicated file. Clients cache files and servers use a "callback" mechanism to invalidate caches. They chose to do this on the belief that "... a strong guarantee is less confusing to average users than a weaker guarantee would be."

The upper level allows a weaker consistency among replicas of a file. It does not have the notion of a primary site and updates are allowed to any replica during a partition. Diverging copies are resolved using time-stamps: the most recent one is preserved. Clients also cache data from the global name service, but in read-only mode. These caches get invalidated by expiration times. The client cache provides increase availability for the global name service by being loaded with data that is important to the user application and only allowing these entries to expire if the client is connected to its corresponding server. The Echo researchers have used the term *stashing* to refer to this behavior of the global name service cache. We feel

that Echo is just exploiting one aspect of stashing and are limiting its use by not allowing users (or applications) to access partially inconsistent data that they are willing to use.

2.5. Summary

In this chapter, we have presented the relevant design issues for a distributed file systems that allows users to keep local copies of important files, decreasing their dependency over file servers. Using the notions of *stashing* and *quasi-copies*, the system allows users to specify the quality of the service they want to receive when the file server is not reachable. We feel that one of the key points of this work is our focus on the tradeoff between availability and degradation of service. This design is ideally suited to federated environments, because it provides users of a component with greater autonomy from other components and tolerance of network partitions and server failures.

We want to stress that the use of stashing does not preclude the use of other performance-enhancing or fault-tolerant techniques. Stashing may be used in conjunction with, say, caching or file replication, and it will serve to increase the availability of distributed file systems.

In this chapter we also presented a probabilistic model to evaluate how much availability a stashing facility would provide to a particular application. This model shows that applications can benefit from a stashing facility to survive for a relative long time compared to the average length of an unplanned disengagement or failure of the server.

Chapter 3. FACE: a file system architecture for Federated Computing Environments

This chapter presents the general architecture of a network file system that incorporates the notions of **stashing** and **quasi-copy** presented in Chapter 2. A prototype of the FACE architecture using Sun†'s NFS is described and it is used to asses the difficulty of adding stashing to an existing network file system, as well as to measure the performance overhead that stashing incurs. It is shown that stashing can be provided with relative few changes to an exiting implementation and with negligible overhead to user processes.

3.1. Architecture

In Figure 3.1 we show a diagram of FACE's general design. There are four main components in this architecture: the network file system interfaces, the bookkeeper processes, the integration modules, and the user-level tools (on the client side only). The network file system interface directs user file accesses to the appropriate underlying file system (i.e., local or remote). It also implements the system support for stashing. The client and server bookkeeper processes provide the runtime support for keeping the stashed copies of remote files within the user consistency constraints (recall the discussion in section 2.2.2 about client maintained and server maintained predicates). The integration modules implements the functions presented in section 2.2.3, i.e., an optimistic recovery mechanism to allow the merging of divergent stashed copies. Finally, the user-level tools provide the facilities which allow users to select which files to stash. In the following subsections we discuss in more detail the network file system interface and the bookkeepers. We also sketch an example of how application specific integration modules should be implemented, as well as discuss what facilities should the user-

† SUN is a trademark of Sun Microsystems, Inc.

level tools provide.

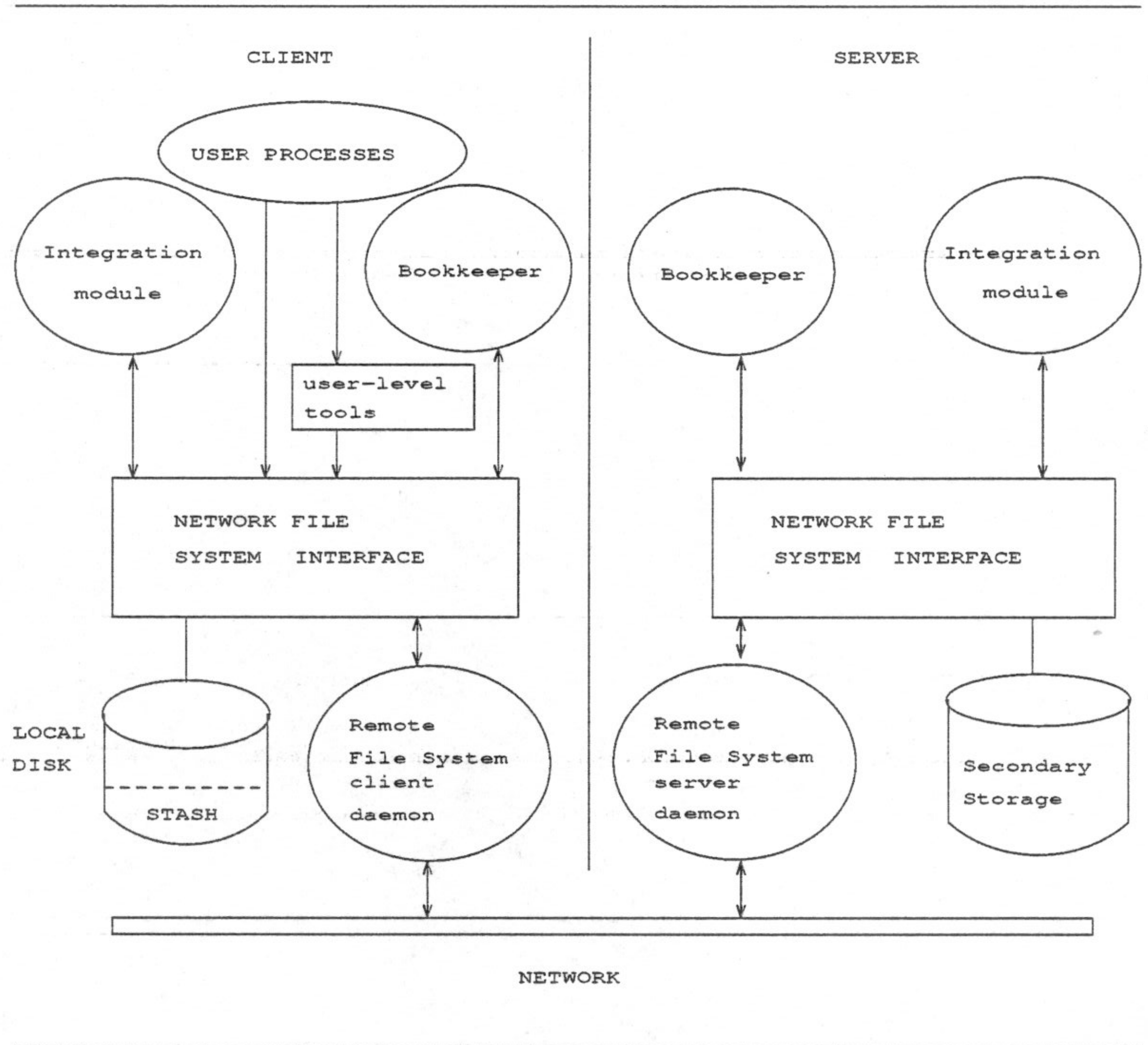

Figure 3.1: FACE general design

3.1.1. Network File System Interface

When enhancing a network file system to support stashing, one decision that has to be made is where in the architecture should the added functionality be implemented. To make this decision we have to take into account our design goals, which follow from the federated environment we are considering: large scale and local autonomy, which implies minimal modifications to be enforced to each federation component, as well as transparency to user level processes, portability to a

variety of computer systems, and minimal performance degradation.

Clearly, stashing must affect the file name translation process. Depending on the status of the connection between client and server, the file name use by a client to refer to a stashed file has to ultimately translate to either the remote file or to the stashed copy. Figure 3.2 shows the different layers in the file name translation process. User processes use file names that are translated to Internal File IDentifiers (IFID) used by the kernel. These IFDI's are translated to blocks in main memory managed by the cache system. If the file has not been brought into the main memory, the content of the file is brought into the cache space from its physical location.

For the first layer, *file name*, there are different naming conventions among operating systems. Therefore translation mechanisms from file names to IFID, are different for each type of system. Implementing stashing at this level would require keeping two disjoint file spaces together by maintaining a correspondence between file names on the remote file system and file names on the local file system. A user-level mechanism would use this correspondence to decide what file name to use whenever a user process tries to access a stashed file. Although this could be provided by ''enhanced'' library functions, and it would be relatively easy to implement, it would have to be done on a per language basis. Clearly, our goals of portability and transparency at the user level would not be met. Moreover, the overhead cost to maintain the name correspondence, and the monitoring of the status of the connection between client and server, at the user level would be too time consuming and redundant, since in many operating systems it is already in the kernel.

The *cache* layer also differs from implementation to implementation. Although in some cases (CFS [Schroeder85a], Andrew [Howard88a], Amoeba [Renesse89a]) there can be a file oriented cache instead of the more traditional block cache, this is not yet the general case nor we foresee it being a *de-facto* standard. In the cases

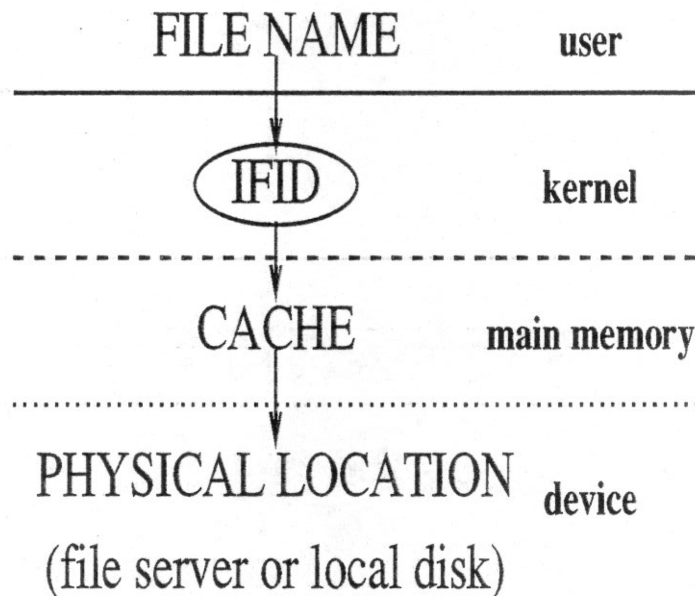

Figure 3.2: Name translation process

where clients have a block oriented cache, the cache system of the clients would have to be significantly modified to allow it to cache entire files. In the introductory paragraphs of Chapter 2 we discussed why we belief that caching and stashing should be treated separately. Forcing a client caching system to also handle stashing may result in a negative impact on the cache performance since decisions made by the stashing fetching and replacement policies may worsen the caching hit ratio. For example, take a large font file that has been stashed together with a data file to display a picture. Since this font file is used infrequently, it would quickly become a good candidate for caching replacement. When it becomes necessary to replace a file in the cache, it is not obvious which replacement policy should prevail. If the caching one wins, then the purpose of stashing the data file would have been defeated (i.e., to produce the picture even if the file server is no longer reachable). If the stashing policy prevails, then the caching system might start to ''thrash.'' However, one common implementation that could handle both caching and stashing would provide many advantages. At the end of section 2.4.1 we suggested that the

quasi-copy framework could be this common implementation, but more study is needed to evaluate this idea.

The IFID layer provides a way for the operating system to identify the content of a file independent of the file's name. In this way, several different file names can be mapped to the same physical location. By slightly modifying the implementation of this layer we can achieve the inverse functionality: the same file name mapped to different physical locations. Therefore, a file name can be translated either to one physical location in a local disk (the stashed copy) or one in a file server (the remote file), depending on the status of the connection between client and server. Figure 3.3 shows the name translation process of Figure 3.2 when a stashing facility is implemented at the IFID layer and the remote file has a stashed copy.

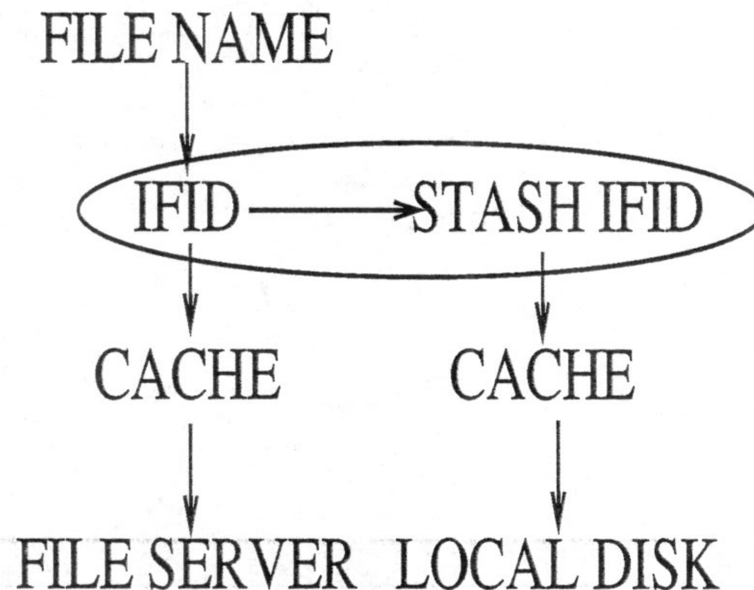

Figure 3.3: Modified name translation process to support stashing

In the Sun's Network File System (NFS) [Sandberg85a], this layer is implemented by the use of *vnodes*, an extension of the UNIX *inodes* that corresponds to a triplet: *(computer-id, file store number, inode number)*.

To support stashed copies of remote files we have extended the data structure that NFS has representing remote files' IFID (called rnodes). Two new data fields are associated to this data structure. The first is a pointer to the IFID corresponding to the stashed copy. This IFID belongs to the local file system that the client uses as the stash space. The second field saves the quasi-copy predicate associated with the stashed file.

The routines in the network file system interfaces have to be able to select, depending on the status of the network and the file server, which IFID to use for a given file name. Each file operation has to be implemented following the structure of Figure 3.4.

```
<file operation> :
    IF (IFID represents a remote file) AND
        (stash IFID is not NULL) AND
        (there is no connection to the file server)
    THEN
        (use the file operation corresponding to the stash IFID )
    ELSE
        (use the file operation corresponding to the remote file IFID)
```

Figure 3.4: General structure for file operations

There are also specific stashing routines in the network file system interface to take care of adding and removing stashed copies' IFID to the corresponding IFID of the remote files. Also, there are routines to install and delete the internal data structures for the stashed copies' IFID in the local stash space.

3.1.2. Bookkeeper Processes

Client and server bookkeepers are processes that work with the network file system interface to provide the quasi-copy support for the stashing facility. The network file system interface, at the client or server side, informs the corresponding bookkeeper of what

files the user has stashed and provides the consistency constraint associated with each file.

The management of a quasi-copy predicate will be performed in general by the server bookkeeper, although in the case of delay conditions, there is the option of unloading this task to the client bookkeeper. Regardless of which side maintains the consistency, the actions taken are the same. When the predicate condition is about to be violated, the bookkeeper acts and updates the stashed copy from the remote file's content.

At the server side, if the coherence is other than a delay condition, in principle, each update to the remote file has to be intercepted to check whether the condition will be violated or not, a possibly expensive task. However this requirement can be made less expensive if the server bookkeeper can be built as an internal module of the network file system interface and not as an external (user-level) process.

At the client side, the bookkeeper only needs to maintain expiration times for each of the remote files being stashed by it with client-maintained predicates. In contrast to the server bookkeeper, the client bookkeeper can be implemented as an user-level process.

In the case that the system includes predicates among different objects, like those discussed in Section 2.2.2, the bookkeeper process should make sure that the multi-file constraints are obeyed. For instance, if two files must have the same version number at all times, the bookkeeper must guarantee that their installation is atomic. In this case it would be enough to restrict the access to a new version of one of the files (by raising a condition flag) until the corresponding version of the other is in place.

3.1.3. Integration Modules

Several file usage characterizations studies in a diverse of computing environments ([Floyd86a] in academic and research UNIX environments, [Ousterhout85a] in academic

UNIX environment, [Staelin88a] in data processing environments using CMS, [Bozman89a] in research and development environments using CMS, [Satyanarayanan81a] in an academic distributed environment, and [Smith81a] in a production center running IBM's MVT operating system) agree that files can be separated in two large groups: *private* and *shareable* files. Moreover, for sharable files three classes can be distinguished: *read-only* (e.g., executables), *append-only* (e.g., logs), and *read-write*. It should be clear that the only type of files that is a real problem to integrate is sharable read-write files. Private files, by definition, are only accessed by one user, therefore after a disengagement is over, the integrating mechanism just have to installed at the server the last modified version of the stashed file. For sharable read-only files no integration is needed since diverging copies will never occur. Sharable append-only files can be integrated in the following way: during a disengagement, each client keeps all their new additions to the sharable append-only file in a local file. When the disengagement is over, these local files would be used one after another (in any predefine sequence) to perform the additions to the file at the server.

A second aspect of file usage that the above mentioned studies agree on is that in the computing environments they studied, the probability of a shareable read-write file being used at the same time by more than one user is extremely low. For example, in [Floyd86a] they report that in their traces, only 1.3% of the events collected accounted for two or more users simultaneously reading the same file, 0.84% of the events collected accounted for two or more users simultaneously writing to the same file, and 2.7% of the events collected accounted for two or more users simultaneously reading and writing to the same file. Therefore, we have taken an optimistic approach to integrate diverging stash copies after disengagement periods are over. Our belief is that, for those few cases where concurrent use of files is a common case, time and effort can be spent in building integration tools that are specific to such cases. We build these tools by using an approach based on the data-patching mechanism proposed in [Garcia-Molina83a] for dis-

tributed databases.

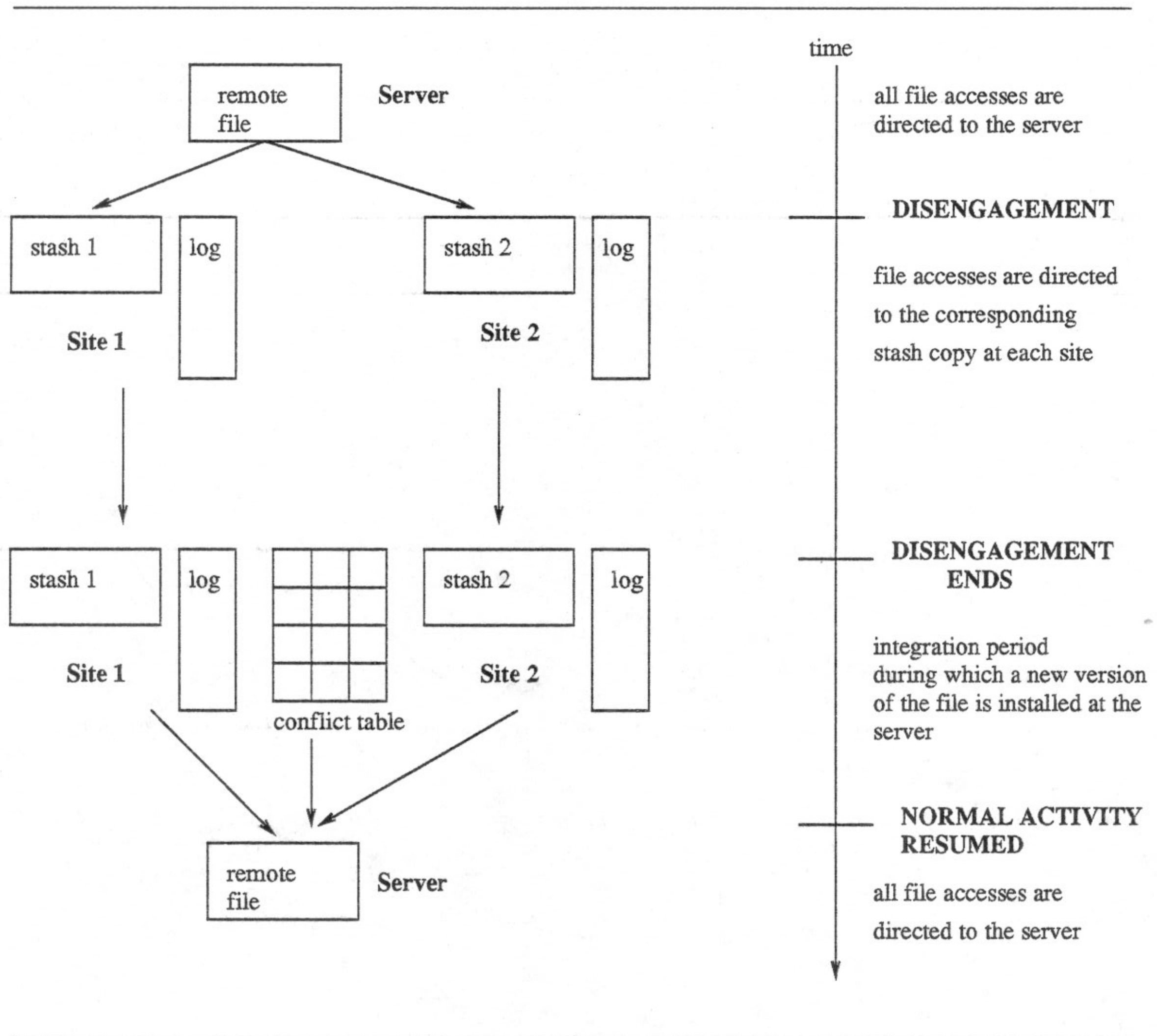

Figure 3.5: General integration process

In general, the data integration process is application driven. A **conflict table** is defined per application and it specifies what are the possible conflicting actions that can occur during a disengagement period to different stash copies†. (It is assumed, without lost of generality, that conflicting actions can be paired.) For each pair of conflicting actions, a **resolution** action is defined. A possible resolution action could be informing the appropriate users that a conflict has occurred, or invoking an interactive tool to help

† Later in this section we show an example in the context of replicated directory.

users come up with a version of the file that reflects the actions they performed during the disengagement period.

The general process of integration is shown in Figure 3.5. While there are no partitions or disengaged clients, all file accesses to the remote file are directed to the corresponding file server. When there is a disengagement from the server, users at sites with stashed copies direct their file accesses to their quasi-copies. The file operations to these stashed copies are recorded as intended actions in log files at each site. These intended actions will become permanent once they are applied to the server's copy of the file, after the disengagement is over.

In Figure 3.5 we show the canonical situation of two sites with stashed copies of the same remote file (stash 1 and stash 2) disengaging from the same file server. Note that a similar situation could arise if only one site with a stash copy disengages from the file server. In this case, the integration process would occur, if during a disengagement period, both the file at the sever and the stashed copy at the client are modified.

Once communication to the server is reestablished, an integration period begins. During this period, the integration process uses the conflict table and the logs of intended actions of each site to reconcile the individual actions done to each stash copy. After all the conflicting actions are resolved a new integrated version of the file exists. It is installed at the server and user file accesses are once again directed to this server.

To illustrate the data-patching approach, let us show an example for the particular case of replicated directories. In an abstract sense, a directory is a file that contains a collection of entries, each of which contains a (*key*, *value*) pair, with a unique *key*. Operations on directories can be listed as follows: lookup(*key*), delete(*key*), insert(*key*,*value*), and update(*key*,*value*). The lookup operation returns true if the *key* requested exists in the directory, and false otherwise. The other operations are self-evident.

In Figure 3.6 we show an initial replicated directory configuration for a two site system (site 1 and site 2). During a disengagement period, users at these two sites issue the

key	value
1	X
2	Y

Site 1

key	value
1	X
2	Y

Site 2

Figure 3.6: Initial configuration of a replicated directory

following commands on the directory:

Site 1: insert(3,Z), lookup(2), update(2,U)

Site 2: lookup(3), insert(3,V), delete(2)

There are ten possible pairing of directory operations that can occur during a disengagement†. Only four of these pairs are in conflict as is indicated in the conflict table (Table 3.1). Using Table 3.1 the integration process would allow the following operations to be installed at the server's directory file:

operation 1	*operation 2*	*resolution action*
insert(A,α)	insert(A,β)	user intervention
update(A,α)	update(A,β)	user intervention
delete(A)	update(A,α)	ignore the delete
delete(A)	lookup(A)	ignore the delete

Table 3.1: A conflict table defined for the replicated directory example

Intended operations issued at site 1: lookup(2), update(2,U)

† There are six pairs combining different operations plus four pairs of identical operations.

Intended operations issued at site 2: lookup(3)

The integration process would require user intervention to determine which insert operation (insert(3,Z) or insert(3,V)) would be installed. The integration process will also ignore the delete(2) operation issued at site 2. Figure 3.7 shows the final result of the integration process for this particular case.

key	value
1	X
2	U

Server's directory

Pending:

key	value	user
3	**Z**	**Site 1**
3	**V**	**Site 2**

Figure 3.7: Result of the integration process on a replicated directory

It is important to recognize that specific algorithms to manage replicated directories during partition exist ([Daniels86a], [Jia90a]) which do not require user intervention. These approaches could be implemented to allow automatic integration for this particular case.

In the case of replicated directories we have just showed, we have assumed that both sites have an identical quasi-copy at the moment the disengagement began. In general, this is not always required. Take for example a banking application; the stashed files may represent information on bank accounts. In this case, since each account has a unique identifier and operations on bank accounts can be stated in a *stateless* form (e.g.,

''increment by $100 account 13045''), the stash copies can be completely different, and still an integration mechanism would be able to reconcile the operations done to accounts during a disengagement period.

Anyway, most of the time the system will have to decide (most probably with user intervention) what version of the file (the one at the server or the stash copy) to leave as the remote file. In our present prototype, the last modified file is the one installed at the server after a disengagement is over.

With respect to maintaining logs, it is desirable to have as little logging activity as possible. For some cases, always maintaining logs at the server and at the clients with stashed copies will be unavoidable. In particular, in those cases where there is a need of a common origin for the integration process (e.g., the replicated directory example). In other cases, just maintaining logs at clients during disengagement periods is sufficient (e.g., applications for which operations can be specified in a stateless way).

Finally, in our modified data-patching approach, during disengagement, stashed copies are used as a ''reference'' for the users to issue operations. These operations are entered into the logs as intended operations which will be permanently installed at the remote file once the disengagement is over and if the intended operation does not create a conflict. The actions are also installed at the local stashed copy, but the stashed copy is only use by the application (during disengagement) as a ''hint'' so that the application can decide what other actions to perform.

3.1.4. User-level Tools

User-level tools aid users in selecting the files needed to keep a given application running when disengagement from the server occur. These tools provide the facilities described in Section 2.2.1. We can classify these tools in two types:

1) *Static*: where users explicitly name the files their applications require. These tools just function as front-ends to make the task of file selection easier. Examples of this

type of tools are a list of file names edited directly by the user that specify those files that always have to be stashed, as well as tools that can interpreted specification files used by software maintenance programs (e.g. *make* or SCCS [Leffler84a].)

2) *Dynamic*, where the tools, on behalf of the users, select the files required by the application. The more sophisticated these user tools are the more they will be able to capture those files indirectly required by the application, e.g., font files in document processing applications or libraries used in software development ones. An examples of this type of tools is a ''record-stash'' command that is used to capture the files used during an user activity cycle (e.g., using something similar to *script* [Leffler84a].) Also, the application itself can issue stashing calls, but in this case the application developer has to add these calls into the application's code.

All these tools use the system calls provided by the network file system interface to stash and unstash files. For example, in our prototype (see Section 3.2.4) the tools must use the system calls **stash** and **unstash**, supplying to them the pathnames and the corresponding quasi-copy predicates of the files needed by the application.

3.2. Prototype

In this section we present the description of a prototype based on the architecture previously described. The prototype is an extension of Sun's NFS software [Walsh85], and has been implemented to test the stashing concepts that were presented in Section 2.2.2 (Data Consistency). The prototype has been implemented on the facilities of the Princeton Distributed Computing Laboratory, and was developed on a Sun 3/50 computer running Sun's UNIX 4.2 release 3.3 (SunOS 3.3).

In the prototype, all file accesses are directed to the file server during normal operation, and to the local stash if there is a local copy and the connection to the server is down. This access decision is made at the vnode layer for the reasons already discussed.

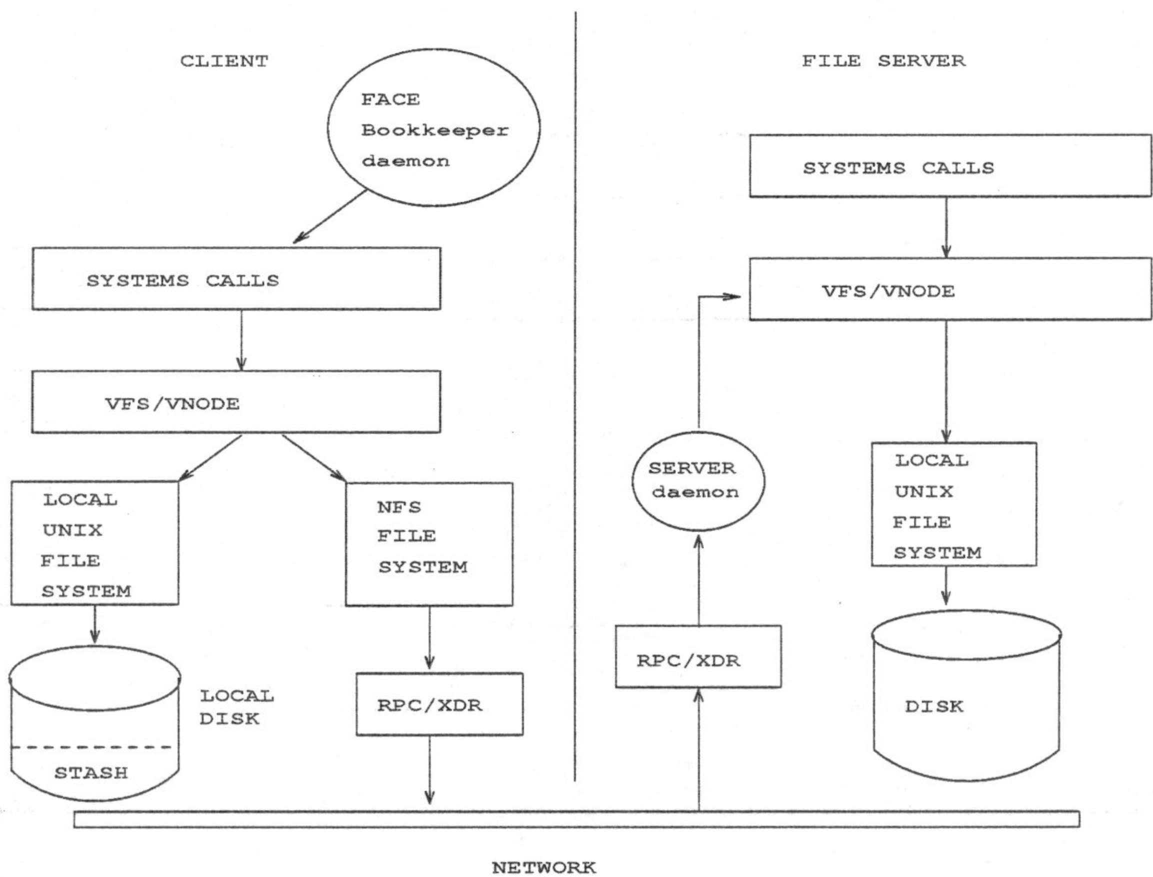

Figure 3.8: Block diagram of the FACE prototype based on NFS

The initial implementation utilizes a client-maintained strategy to guard the stash consistency, and thus only supports delay conditions.

Figure 3.8 shows a block diagram of the FACE prototype. The server side does not change at all from the NFS implementation since, as mentioned before, this stashing facility is client maintained. On the client side, the VFS/vnode component has been modified. Extra data structures were added to support the stashing facility. Several kernel routines were modified and new ones were added to perform stashing operations (e.g., initiating and maintaining the necessary data structures, redirecting users file accesses to the stashed copies when the client machine disengages from the file server, etc.). New system calls were introduced to **stash** files, and to **unstash** them. These sys-

tem calls are privileged and are utilized by users through the library routines *mkstash* and *rmstash*, respectively. Also, we implemented a bookkeeper process, which runs in user space. The bookkeeper is the runtime support system that maintains the consistency constraints (i.e., the quasi-copy predicates) between the remote files and their stashed copies.

In the next subsections we present a detail description of the prototype's implementation. For completeness, we begin by presenting a brief description of NFS.

3.2.1. NFS Overview

NFS is a system by which different computers share a file space. Its user interface is similar to the UNIX file system one [Leffler89a]. Currently, several operating systems support the NFS protocols. Among them we find SunOS, System V release 4, Mt. Xinu's BSD UNIX implementation, CMU Mach [Spector87a], and MSDOS [SUN89a].

In NFS all file activities are centered around the **vnode**, a data structure whose role is equivalent to the *inode* in traditional UNIX file systems (for an extended description of vnodes see [Kleiman89a]). NFS splits the kernel file system functionality into a file-system-dependent and a file-system-independent parts. For example, in SunOS, the former is represented by the *inode* and the latter by the *vnode*

In Figure 3.9 we present the architecture of NFS. User processes perform file operations by using system calls. These system calls operate on the Virtual File System (VFS). VFSs are logical storage units that contain files and consist of data structures and operations for each file system "mounted" on the computer. This is similar to the mount table information in standard UNIX. Each file in a VFS is represented by a vnode, and any file operations on it are translated to the specific routines of the particular VFS to which the file belongs. For example, if the vnode represents a local UNIX file, then a read system call is translated to the appropriate routines to handle inodes (e.g., iget, bread, etc.) [Leffler89a]. If instead the vnode represents a remote file the appropriate Sun's RPC and XDR routines are invoked [SUN88a].

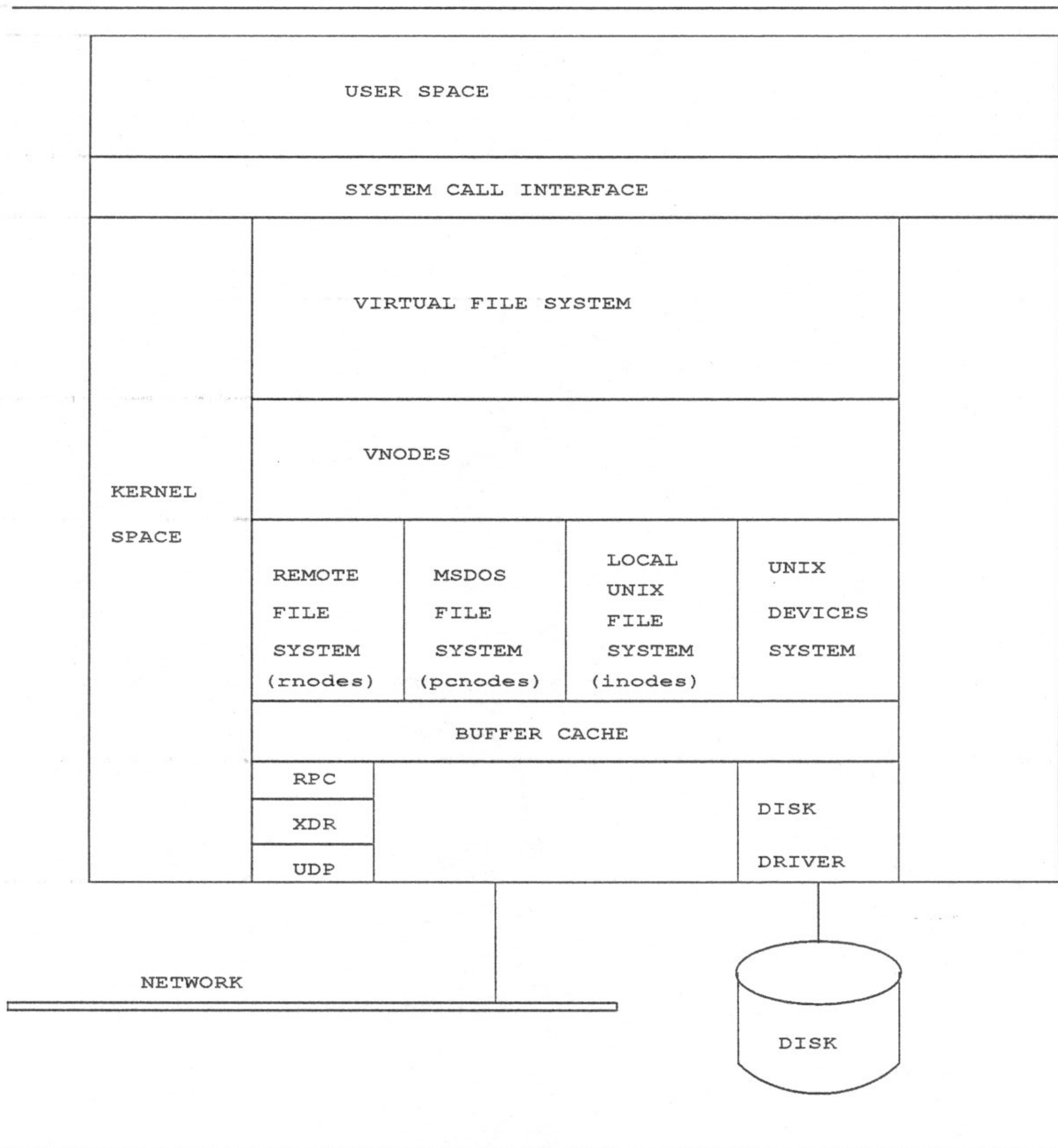

Figure 3.9: NFS architecture

Figure 3.10 shows the contents of the VFS data structure and the vnode data structure. Each structure contains a pointer to an array of function entry points for the specific routines of each particular VFS or vnode. The private data field also points to file-system-dependent information.

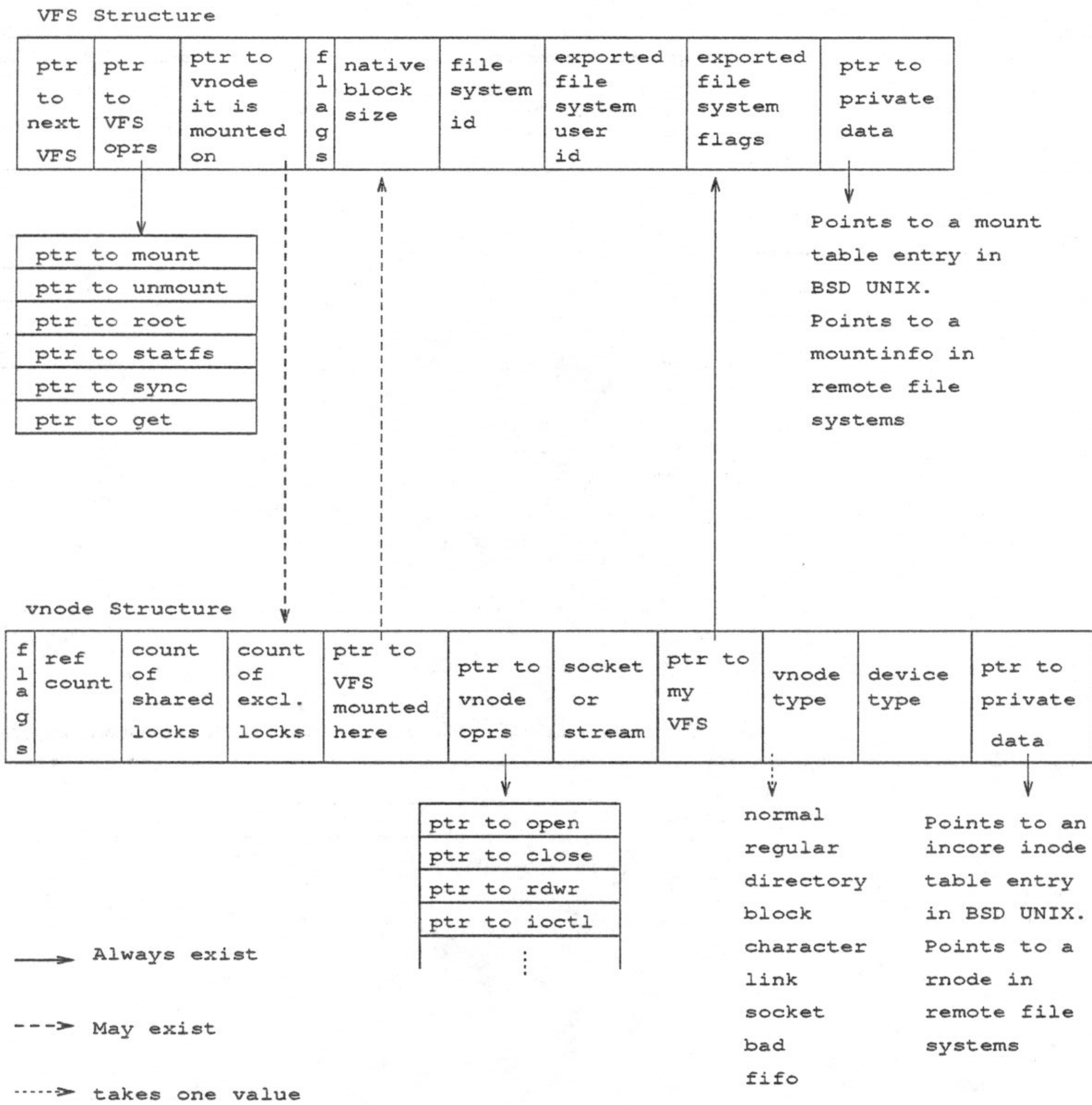

Figure 3.10: VFS and vnode structures

3.2.2. New Data Structures

Figure 3.11 shows the data structures added to the NFS' data structures to support stashing. The data structures enclosed in solid boxes are provided by NFS. We have added the fields inside dashed boxed to support stashing. These four new data fields have been added to the rnode data structure used by clients to represent remote files. The first added is a pointer to the vnode corresponding to the stashed copy (*stash vp*). The vnode

pointed by *stash vp* belongs to the VFS of the client local UNIX file system. The second field, *predicate* , contains the user consistency constraint (the quasi-copy predicate). For this prototype, only time related predicates are allowed, i.e, the length of time a stashed copy is valid from the time it is copied from the file server. The time of last modification is already maintained in the vnode structure. The third field, *strategy*, indicates the user's preference toward directing all file accesses to the stashed space during normal activity, i.e., using the stash copy as either a backup facility or a file cache. The latter option is only useful for read-only files. Finally, a *valid* field is added to indicate if the stashed copy conforms or not with the user predicate. This field is only added for convenience of user application. During partitions, applications could check if the stashed copy still conforms to the user predicate or not.

3.2.3. New and Modified Kernel Routines

In NFS, kernel code for all file related operations consists of macro definitions. These macros translate vnode operations to actions on the underlying file system†. In our prototype we redefined these macros to allow the use of either the remote files or the stashed copies. In pseudo-code, these macro definitions are presented in Figure 3.12.

There are a number of new routines in the kernel. The first two handle the linking and unlinking of remote files' rnodes with the corresponding vnodes of stashed copies. Two others are used for allocating and de-allocating the new vnodes of the stashed copies in the local file system's stash partition. There are also routines for saving and deleting values from the rnode stash fields.

3.2.4. New System Calls

The new system calls added to the NFS system calls interface to support stashing are the following:

---

† See Figure 3.9 and the discussion in Section 3.2.1.

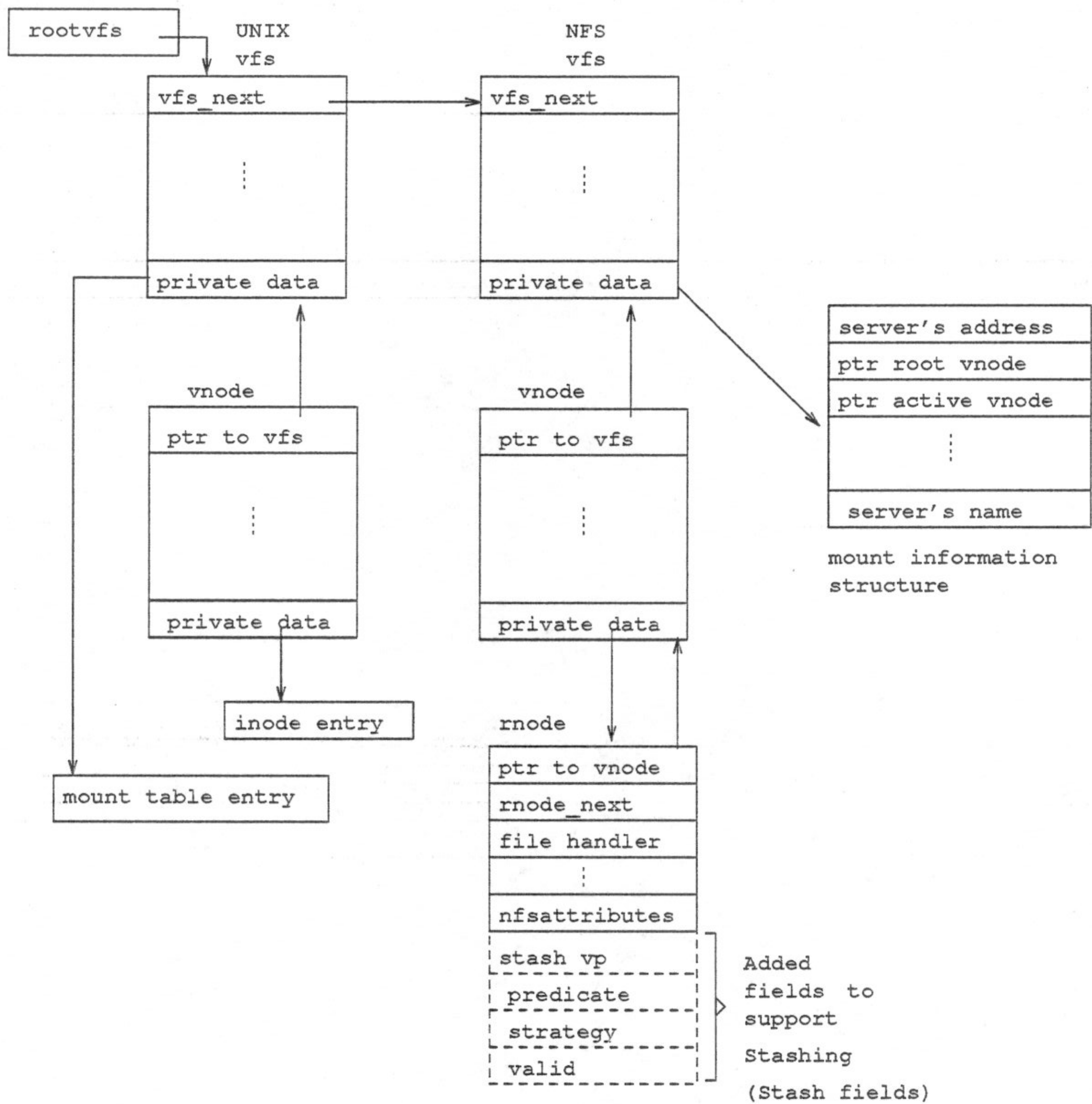

Figure 3.11: New NFS data structures to allow stashing

handle = **stash**(*pathname*,*predicate*,*strategy*):

the **stash** call assigns the values provided by the user (*predicate* and *strategy*) to the corresponding rnode fields of the remote file to be stashed. *Predicate* is a positive integer that indicates the amount of time in minutes before a ''refresh'' of the stashed copy is attempted. The remote file is identified by the *pathname* given by the user. The call also creates a new vnode in the stash partition, with its

```
DEFINE <file operation>
    IF (vnode represents a remote file) AND
        (stash vp is not NULL) AND
        (strategy is local or there is no connection to the file server)
    THEN
        (use <file operation> pointed by the rnode's stash vp pointer)
    ELSE
        (use <file operation> pointed by the current vnode)
```

Figure 3.12: New NFS macro definitions (in pseudo-code)

corresponding inode entry. If the data structures for the stashed copy are successfully allocated, *handle* has the value of an identifier to the remote file being stashed. This *handle* is an ''opaque'' identifier,i.e., it is not understood by the receiving process, just by the kernel. In the current implementation, *handle* is a vnode identifier. The value of *handle* is sent as an interprocess message to the bookkeeper so that it can keep track of the files being stashed. If the call is not successful, *handle* takes the value -1 and an error number is assigned to the variable *errno* indicating the type of error that occurred.

handle = **unstash**(*pathname*):

the **unstash** call deletes from the local file system the vnode of the stashed copy (and its inode entry) corresponding to the remote file represented by *pathname*. It also clears the stash fields in the remote file's rnode. *Handle* behaves as in the **stash** system call. If the call is successful, it sends a message to the bookkeeper indicating that the file represented by *handle* is no longer stashed.

status = **strategy**(*pathname*,*strategy*):

the **strategy** call directs all subsequent file accesses, either to the remote file represented by *pathname* , or to its stashed copy, depending on the value of the variable *strategy*. The default is to direct access to the file server. *Status* has the return value indicating the success or not of the call.

status = **stashopen**(*handle*,*rfd*,*sfd*):

Stashopen is a system call used only by the bookkeeper process to obtain file descriptors for the remote file and its stashed copy corresponding to the value in *handle*. If the call is successful, *rfd* and *sfd* contain file descriptors to the remote file and its stashed copy, respectively. The file descriptors are equivalent to the one returned by a successful open system call [Leffler84a].

status = **setvalidity**(*handle*):

setvalidity toggles the content of the field *valid* in the stash data structure of the stashed copy identified by *handle*, to indicate if the stashed copy conforms or not with the user predicate. This system call is used by the bookkeeper.

3.2.5. The Bookkeeper Process

The bookkeeper process is an user level program that provides quasi-copy support for the stashing facility. The bookkeeper keeps information about all the files that have been stashed, i.e., opaque handles to the remote files that have a stashed copy. It also keeps the expiration times for each stashed copy. Thus, the bookkeeper can keep track of predicates, determining the validity of the contents of the stashed copies, and obtain fresh information from the file server. The information is kept in an ascending time ordered linked list of remote file handles (Figure 3.13).

The bookkeeper sends itself an alarm signal with the value of the earliest expiration time. When the signal arrives, the bookkeeper checks all those remote files which do not conform with the user consistency constraint, marks them invalid (using the system call **setvalidity**), and tries to back them up from the file server (using the system call **stashopen** to get file descriptors to use in read and write calls). For each successful backup the bookkeeper rearranges the ordered linked list of handles and marks the stashed copy as valid. When the bookkeeper finishes with this procedure it sends itself a new alarm signal with the next expiration time.

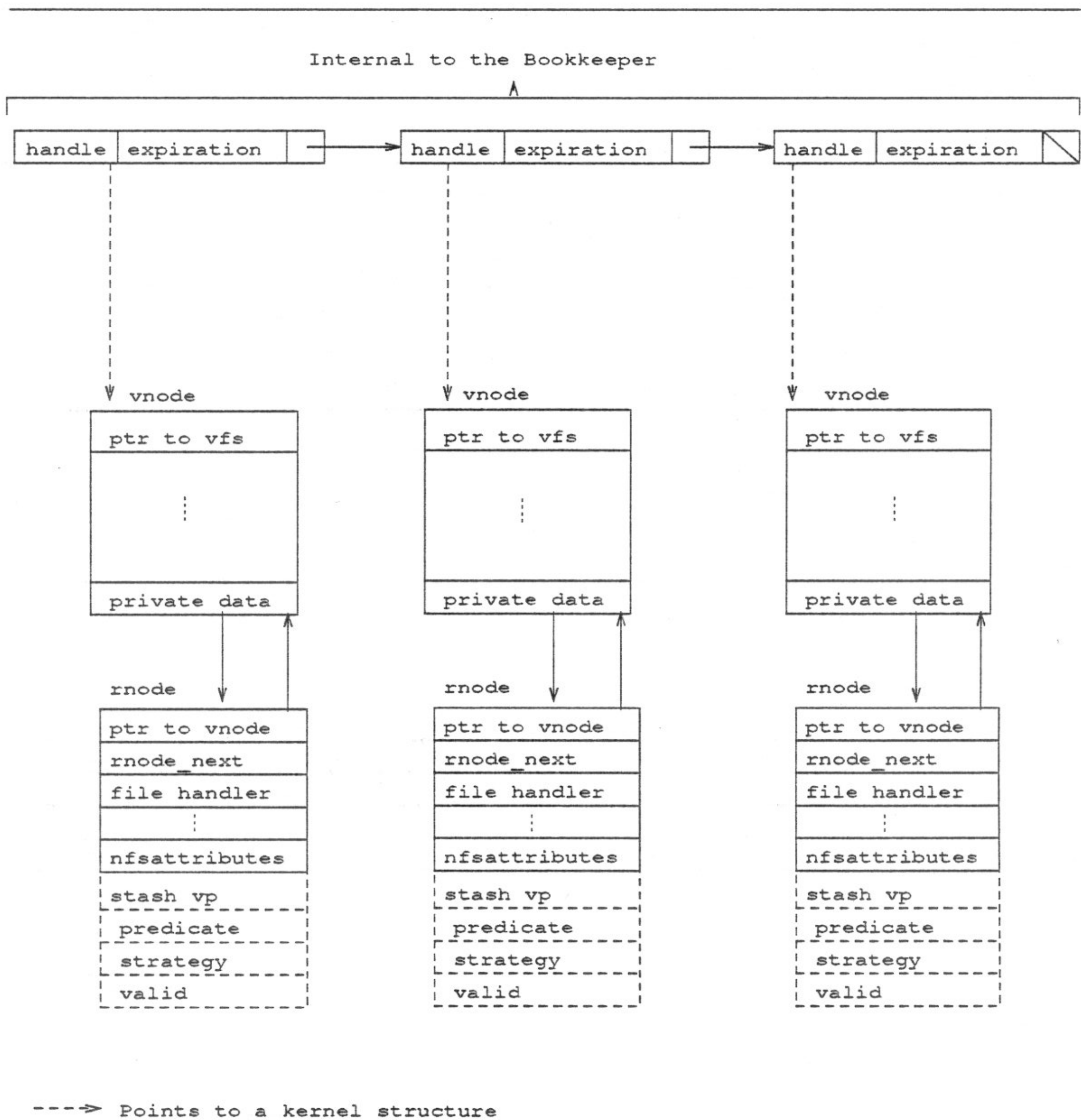

Figure 3.13: Data structures used by the Bookkeeper

3.2.6. An Example

To clarify the design presented in this section we present an example of how the data structures are used to support stashing. Figure 3.14 shows the link between the data structures of a remote file and its stashed copy. Figure 3.14c shows the commands being issued by an user of our FACE prototype. The user changes directory to ''/usr'', edits a remote file (''a'') and then requests a stash copy of it. Figure 3.14a shows the vnode and

VFS data structures for the remote file. There are three file systems mounted: root ("/") from which the machine boots, a remote file system ("/usr") and the stash partition ("/stash"). "/" and "/stash" are UNIX 4.3 BSD file systems and reside in the client local disk. "/usr" is a NFS file system representing a UNIX 4.2 file server. The vnode labeled "/root" is the vnode for the root directory of the user mounted file system. The vnode labeled "/usr" is the vnode corresponding to the mount point for the remote file system "/usr." The "/usr/a" vnode represents the remote file that has been edited by the user. Once the user issues the command - *stash a 1hour remote* - the added data structures in the rnode of "/usr/a" (*stash vp*, *predicate* and *strategy*) are filled by the system call **stash**. This system call allocates a vnode in the "/stash" file system for the stash file "/stash/usr/a" (Figure 3.14b). It also sends an interprocess message to the bookkeeper process with the information that a new stash copy has been created. The bookkeeper stores the needed information to backup the content of the stashed copy with the content of the remote file. It uses the value in the predicate field of the rnode of "/usr/a" to send itself an alarm signal for when the stash copy "/stash/usr/a" not longer conforms with the user consistency constraint.

Since the strategy field is set to remote, all file accesses to "/usr/a" are done exactly as in NFS, i.e., to the file server. In NFS, the client periodically checks for the status of the server, toggling a flag when communication cannot be established. At the moment that the file server is not longer reachable the user is informed of such an event, with a message to the user console, and subsequent accesses are done to the stash copy "/stash/usr/a" through the *stash vp* field. The stash copy is used until the server is again reachable.

3.2.7. Performance

To evaluate the overhead caused by our code, we have measured the time to perform file system calls - open-close, read, write ([Leffler84a]) - in three different situations: first with a file stored at the server (remote file), then with a file stored locally (not

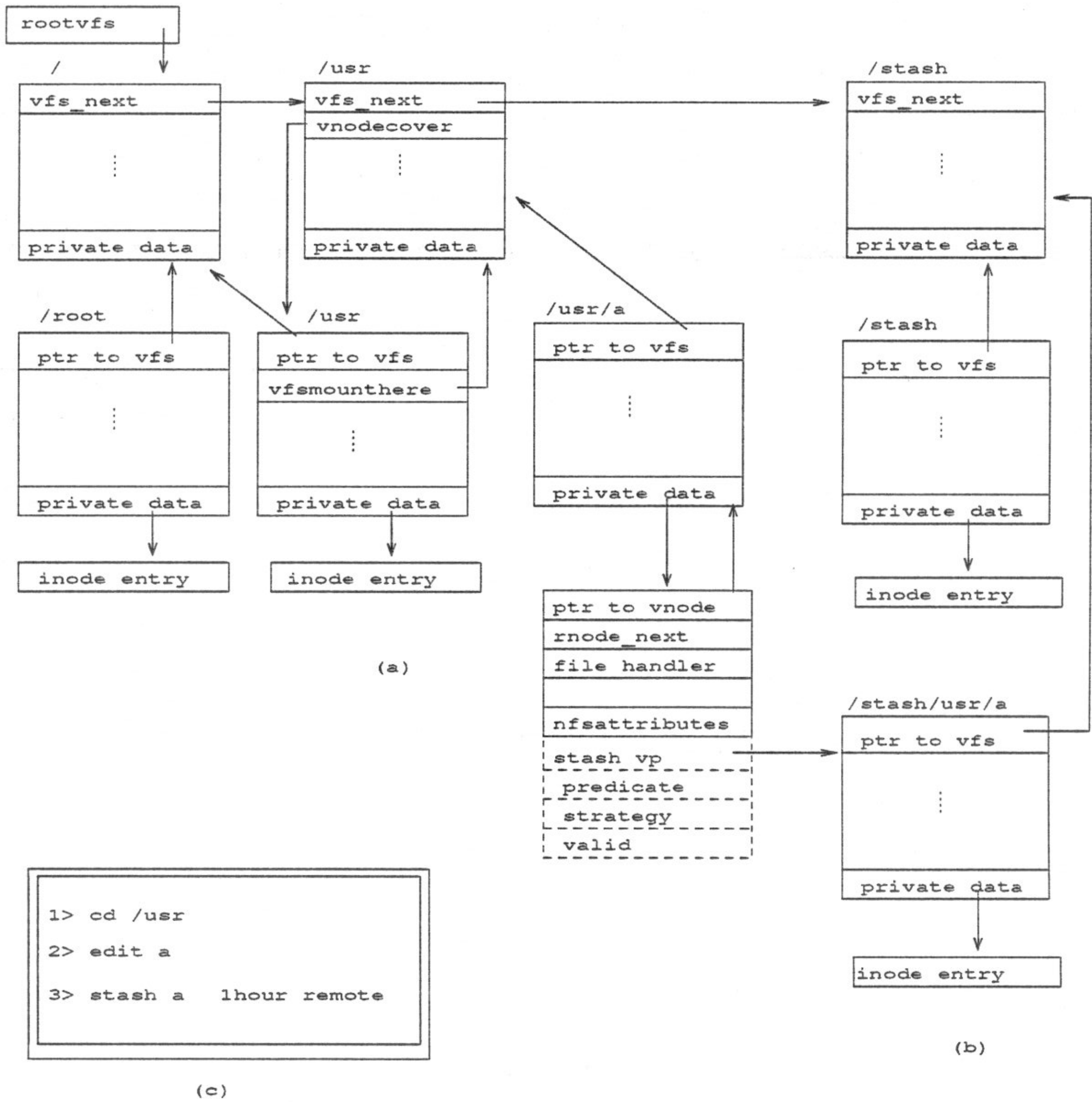

Figure 3.14: An example

a stashed copy), and finally with a stashed copy of a remote file. The tests were made with the FACE kernel and the SunOS 3.3 one (without the FACE modifications). The tests consisted of 100 samples of 10,000 iterations of calling each of the system calls for a 2.5 Mbytes file. For the read and write system calls 256 Kbytes blocks were used.

Table 3.2 shows the times (in milliseconds) that we measured. These results show that the added overhead introduced by FACE to support stashing is negligible compared

file type:		remote		local		stashed copy	
kernel	operation	mean	st.dev	mean	st.dev	mean	st.dev
FACE	write	4.354	0.394	3.249	0.064	3.251	0.202
	read	1.664	0.122	1.401	0.018	1.401	0.021
	open-close	11.092	0.343	2.223	0.004	2.116	0.004
SunOS 3.3	write	4.334	0.353	3.256	0.027	not supported	
	read	1.698	0.058	1.406	0.018		
	open-close	11.503	0.370	2.153	0.047		

Table 3.2: Comparison of system calls overhead (in milliseconds)

to the base kernel (SunOS 3.3). The low cost of our stashing service means that there is no degradation of normal service when partitions are not present or stashing is not being used by an application.

Figure 3.15 and 3.16 show the response time of executing a *mkstash* library routine and a *rmstash* library routine, respectively. These routines allow users to obtain a stashed copy of a remote file and to remove it. *Mkstash* invokes the **stash** system call to allocate the kernel structures corresponding to the remote file (vnode and rnode). It associates a local vnode with the remote vnode, and allocates the local inode for the stashed copy. It then copies the remote file's content into the stashed copy†, and passes a handle (representing the remote file) to the client bookkeeper. *Rmstash* invokes the **unstash** system call to remove the association between the vnode representing the remote file and the stashed copy. It then sends a message to the bookkeeper indicating the remote file is no longer stashed. Finally, it removes the stashed copy from the local disk.

The performance measurements were gathered by executing a thousand times *mkstash* followed by *rmstash*, in an otherwise empty Ethernet between a Sun 3/50 acting as client and a Sun 3/180 acting as file server. In the figures we used a horizontal line to show the average response time of the stashing routines. We also used a vertical line, intersecting the horizontal one, to show the standard deviation.

There are two modes for each of the stashing routines: synchronous or asynchro-

† Note that the NFS protocol is using a block size of 2 Kbytes.

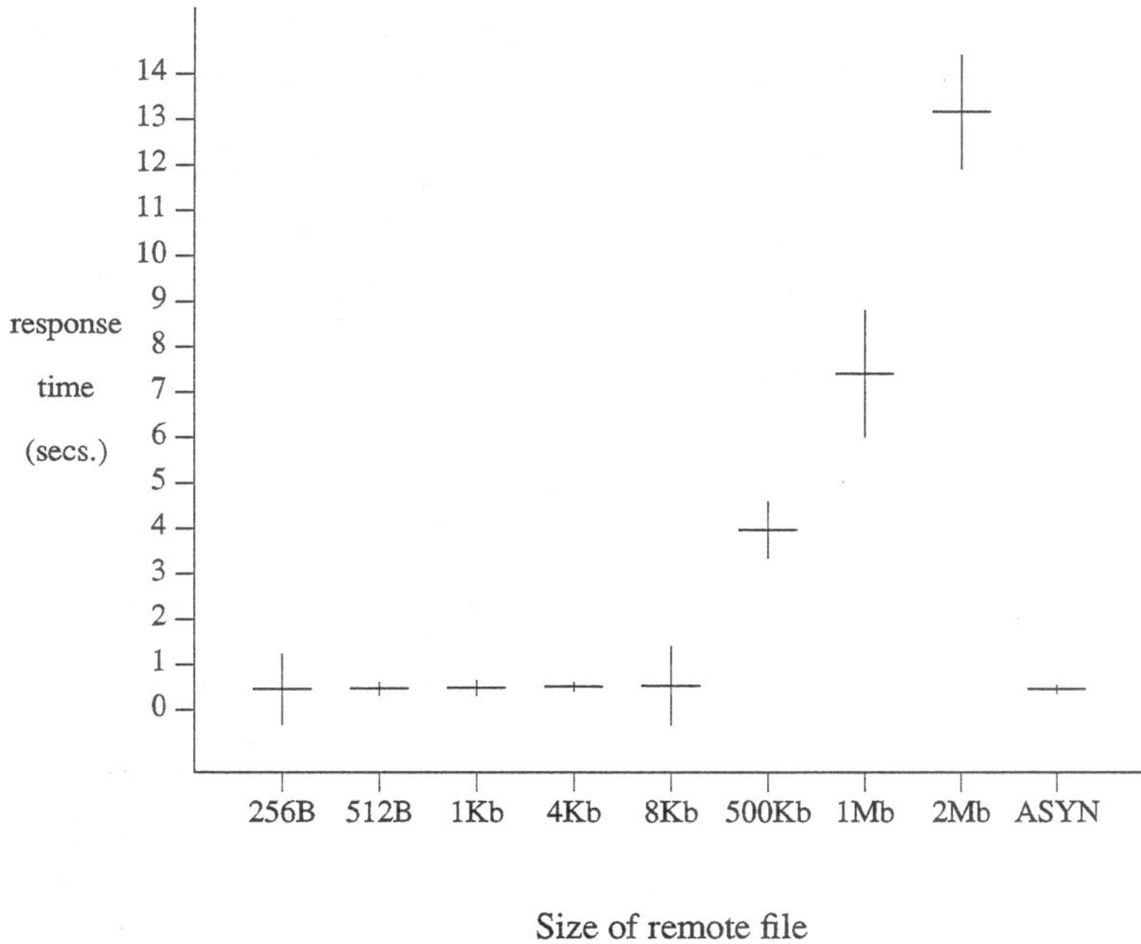

Figure 3.15: Mkstash routine's response time depending on the remote file's size

nous to the calling process. As Figure 3.15 shows, the response time of the *mkstash* routine in the synchronous case depends on the size of the remote file. The main part of this cost is attributed to the copying of the remote file's content into the stashed copy. In the asynchronous case (labeled ''ASYN'' in Figures 3.15 and 3.16) the *mkstash* routine allows the user to do this first backup asynchronously from the requesting user process. The response time of the *rmstash* routine in the synchronous mode (Figure 3.16) also

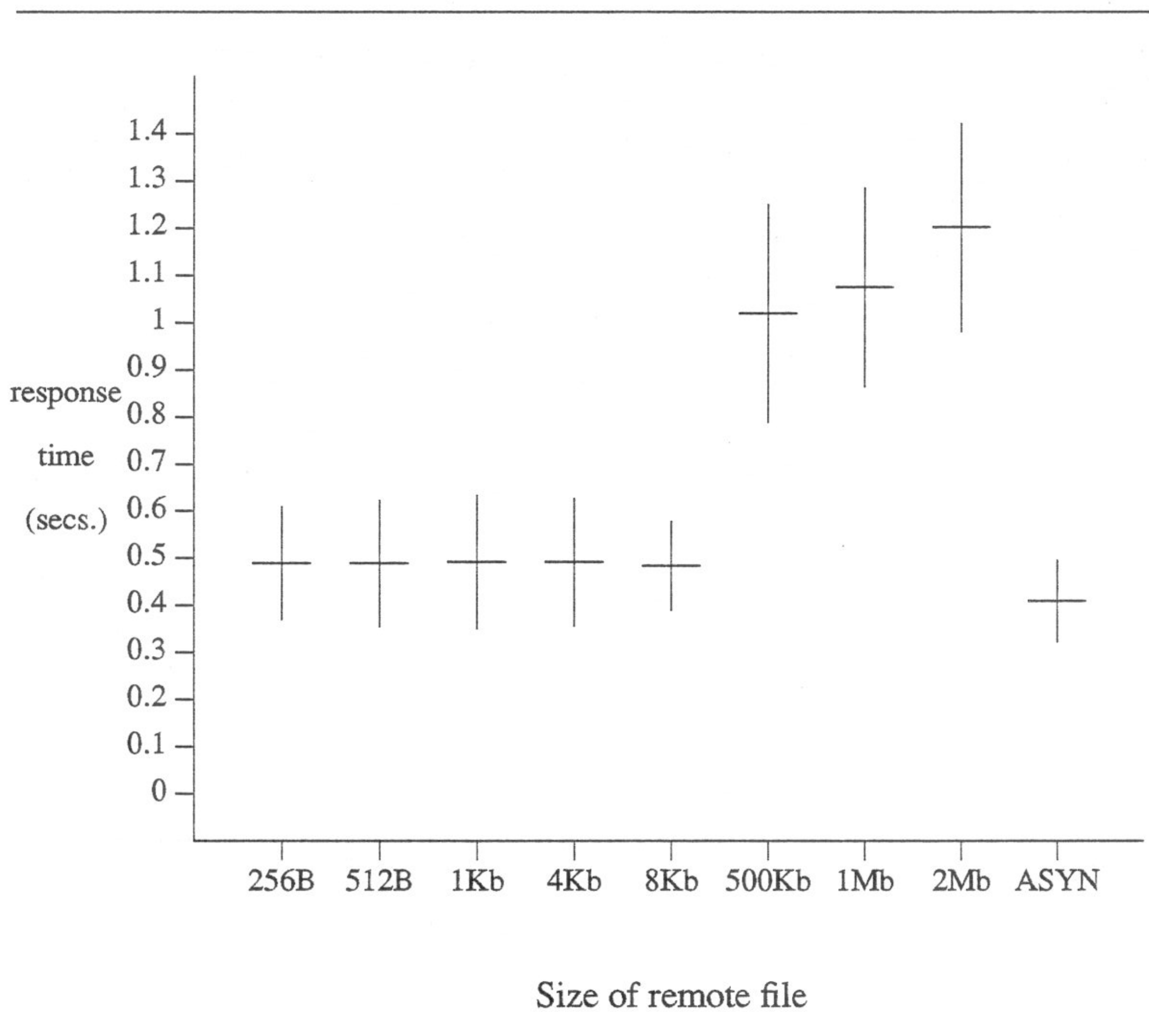

Figure 3.16: Rmstash routine's response time depending on the remote file's size

depends on the size of the stashed copy, but to a lesser degree. Most of this cost is due to the deallocation of the data blocks pointed by the stashed copy's inode. As with the *mkstash* routine, in the asynchronous option the deallocation of the data blocks is done asynchronously from the requesting process.

3.2.8. Prototype's Weakness

One problem we encountered implementing the prototype was how to stash the pathname of remote files at client machines. In NFS, the client name resolution procedure takes each component of a pathname and asks the server for the corresponding vnode. To improve the response time of this procedure a directory cache is often implemented. The directory cache maintains a set of triples of the form (*parent vnode, component name, component vnode*), that are consulted by the name resolution procedure before asking the server. The problem we encountered was how to stash all the triples corresponding to the pathname of a remote file.

One solution would be to provide another pathname resolution procedure to be used when the client disengages from the server. The client could use the name given to the stash copy in the client local space to construct the adequate entries in the directory cache. Alternatively, when stashing a file, as part of the *mkstash* call, every component in the remote file's pathname would be mark ''special'' in the directory cache. These specially marked tuple would not be considered for replacement by the directory cache replacement policy.

Another problem we encountered had to do with the names we were using for the stashed copies in the clients' local space. For our prototype we had the goal of making the local names of stashed copies unimportant to the user. The reason was that in this way the client and the server could use different naming convention, and a translation mechanism for these conventions would not be necessary.

The problem is that when a client crashes the cache directory content and the incore vnode structures get cleared. If after the client recovers from the crash, the server is unavailable, then the client cannot access the stashed copies through the normal mechanism provided by the FACE architecture (using the rnode's stash vnode pointer). We realized that we could provide a recovery mechanism for client crashes if the names used for stashed copies had some meaning. Consequently, we could use the stashed copies local

names to reconstruct the incore vnode structures and the directory cache corresponding to the stahed copies, thus providing more fault-tolerance and autonomy to client sites.

3.3. Summary

We have presented in this chapter the design of FACE, a network file system that lessen the dependency of clients to servers. This system provides file stashing to increase the availability of important information when file servers are not reachable. This feature enhances both the autonomy of the local nodes as well as the degree of fault tolerance of the overall system.

We deal with the issue of stashed copy consistency by incorporating into our design quasi-copy techniques. Thus, users may decide the level of consistency that stashed copies must have in order to meet the demands of their particular applications. We feel that this is an important feature with broad usefulness. It also liberates the system from the burden of keeping perfectly consistent copies in cases where perfect consistency is not required.

We also presented the first prototype implementation of FACE. The goals that governed our design were fast prototyping, transparency at the user level and minimal modifications to the underlying implementation. These considerations led us to implement the FACE prototype by modifying NFS because the latter has become a *de facto* standard for sharing files in distributed environments. Performance figures are reported herein. These figures show that the overhead of providing the service is negligible.

Chapter 4. Load Sharing[†]

One of the reasons for the existence of distributed systems is to allow resources to be shared across a computer network in the same way they are shared in a centralized system, i.e., in a way that is transparent to users. One network resource which has received much attention has been the processing capability of the sites. Because each site usually has its own user community and CPU scheduler, an imbalance of the system workload throughout the network can be a common situation. One solution to this imbalance is to allow users at one site to run processes on other sites in the network. The usual mechanisms used are remote sessions (e.g. *rlogin* [Leffler84a] and *telnet* [Postel80a]), or explicit remote process executions (e.g. *rsh* [Leffler84a]), but for these mechanisms the selection of the execution site and the control of the remote execution is completely up to the users. It is more desirable to dedicate a system program to the task of sharing the processors in much the same way that memory management software allocates the use of memory. Users can then rely on it to automatically handle remote execution of their jobs in order to take advantage of less loaded processors, thus possibly achieving better average response time.

In federated computing environment, load balancing schemes are not appropriate since one cannot consider the whole network as a single unit and thus cannot try to optimize average response time or system throughput. In any load balancing scheme, heavily loaded sites will obtain all the benefits while lightly loaded machines will suffer poorer response time than in a stand-alone configuration. Users of a frequently heavily loaded machine will cheer for a load balancing scheme while users of mostly underutilized ones will strongly oppose participation in such a scheme. What is desirable is a fair strategy that will improve response time to the former without unduly affecting the latter. In this chapter we present an approach to load distribution in federated environments that

† A paper based on a preliminary version of this chapter appeared in the Proceedings of the 8th. International Conference on Distributed Computing Systems, San Jose, California, pp. 282-288, June 1988.

guarantees a level of performance to local site's users. We empirically compare our approach to other approaches used to preserve ownership of local resources.

In the following section we give an introduction to load distribution in distributed systems. Then Section 4.2 presents the definition and description of the **High-Low** scheme, a way of allowing load sharing in federated environments. In Section 4.3 we describe a load balancing algorithm based on running jobs at the least loaded machine in the network. We empirically compare this scheme to High-Low and to the case when no load distribution takes place. In Section 4.4 we present other load sharing schemes and we empirically compare them to High-Low. We conclude by summarizing our findings.

4.1. The Load Distribution Problem

The **load distribution** problem consist of reallocating the workload in a computer network to achieve better service performance. There are two ways of distributing load. One is **load balancing**, which has the objective of equalizing the workload at each site in the network. In this way the same level of service is provided to all the users in the network. Alternatively, **load sharing** reassigns jobs from heavily loaded sites to other ones, chosen through some specific criteria (e.g., ''idleness''). With load sharing the objective is to improve the service rendered at heavily loaded nodes by taking advantage of underutilized resources in the network.

Most of the literature on load distribution has concentrated on load balancing. This trend can be seen in the many load balancing schemes that have appeared in the published literature: see [Wang85a] for a proposed taxonomy and a review of the various approaches that have been pursued, or [Zhou88a] for a comparative performance study of several load balancing policies. Load balancing schemes can be divided in two types: static policies, as in [Ni85a], which ignore the current system state when making decisions and which usually follow average system behavior, and dynamic policies, as in [Eager86a], which rely on system state information.

4.1.1. Designing Load Distribution Schemes

A load distribution scheme has traditionally been composed of three parts: a transfer policy, a load information policy, and a location policy. The transfer policy determines if a locally invoked job should be served locally or remotely. The information policy is given by the load metric that is used to determine the load in any given machine. For example, a possible load metric could compute the average number of running processes during a certain period of time. If the transfer policy decides to service a job remotely, then the location policy (using the information policy) determines where in the network that job is going to be executed. For example, a possible location policy would run jobs on the machine which has the lowest load. Clearly, the selection of a particular load metric depends in the type of jobs submitted to the machine as well as the site's resources and capabilities. Ferrari and Zhou [Ferrari87a] suggest that an appropriate load metric is a linear combination of resources queue lengths. We define the load[†] of a machine as the value of a given load metric for that machine, and the load of a distributed system as a function of the load of all the sites in the network.

Implementors of load distribution systems have to take into account the type of distributed environment where their systems will be implemented. In the next subsection we present why different types of distributed environments merit different types of load distribution systems.

4.1.2. Integrated Distributed Systems & Federated Computing Environments

Distributed systems have some typical configurations. For example, an *integrated distributed system* comprises a group of interconnected computers dedicated to evenly serve a group of users. The goal is to provide the user community with improved service. Most of the work described in the load balancing literature has dealt (explicitly or

---

† Throughout this chapter, whenever we refer to the **load** of a machine we mean a consistent load metric that characterizes the usage of that machine.

implicitly) with this type of environment ([Stankovic84a], [Alonso86a], [Eager86a], [Zhou88a]). In this environment all the sites in the network functionally belong to one organization. The appropriate load balancing scheme for this environment should focus on enhancing the overall system performance which is the system's goal. Examples of this type of system are most of the computer centers in industries and universities.

Another type of distributed system is the *federated computing environment*. In this environment each site of the network functionally belongs to a different user, whether that is a single person or a group. Instances of this environment are networks of inter-departmental machines, and wide area networks like Internet. Although a load balancing scheme can be used in this type of system, it cannot be treated with the same techniques used in an integrated distributed system. For this type of environment, *load sharing* is more appropriate.

In some systems the issue of maintaining the autonomy of each site has been resolved by implementing an ''all or nothing'' strategy. If a machine is completely idle then it becomes a candidate for executing a remote workload. If a machine is being used, even if it is underutilized, then no remote workload is allowed. Basically, any machine can take over an idle one in a master-slave relation, but as soon as the owner of the idle machine uses it (even slightly) all the remote jobs are either put in the background (run with low priority) [Hagmann86a], moved back to their originating site [Theimer85a], moved to another idle machine [Litzkow87a] or just killed (abnormally terminated) [Nichols87a].) While all of these techniques guarantee the ownership of resources to the owner of an idle machine, they do not assure any performance improvement to the remote jobs the idle site may be servicing. It is desirable to have a more gradual style of sharing that would attempt to guarantee certain level of performance to site owners as well as offer some help to the remote jobs that may have been submitted to a site.

Our purpose is to adapt the results obtained for the load balancing problem to load sharing in federated computing environments. To do this we suggest the use of a fourth

policy: the **acceptance** policy. The acceptance policy reflects the disposition of a site owner to accept a certain level of remote jobs to be serviced by his or her machine. In other words, our interest is to provide the owners with control over their machines independently of the load distribution scheme being used.

In this work we are only concerned with the initial placement problem, i.e., where in the network a job should be run. We will not consider the migration of jobs once they have started running at a site†.

4.2. The High-Low Scheme

Computing sites participating in a system where load sharing takes place may be viewed as being at any one time either sources of jobs or servers of jobs. When a machine is viewed as a source of jobs, a machine should only try to execute remote jobs if transferring some of them to another site will greatly improve the performance of the rest of its local jobs. This observation parallels the usual banking practice of borrowing only when necessary. On the other hand, when a computer is viewed as a server of jobs it should only accept remote jobs if its load is such that the added workload of processing these incoming jobs does not significantly affect the service to the local ones. This approach mirrors the sound banking practice of only lending excess funds‡. These two notions can be adapted to a load sharing environment via two policies that we denote by **High-mark** and **Low-mark**.

The High-mark policy behaves as follows: each time the execution of a new job is requested at a site the load of the machine is compared against its High-mark value. If the former is greater than the latter then the load sharing mechanism tries to execute the job in a remote host. Otherwise the job is processed locally. Thus, High-mark sets a

† Load balancing in distributed systems with process migration capabilities was studied in [Kyrimis90a].

‡ It should be pointed out that the banking analogy does not hold completely. In contrast to the banking situation, borrower sites need not return their borrowed cycles and lender sites may not receive back their lent cycles.

lower level on the load a machine must have before it begins to transfer jobs to other hosts. Its purpose is to try to reduce processing overhead by load sharing only when the workload of the machine degrades its service dramatically.

The Low-mark policy sets a ceiling on the load a computer may have and still accept incoming remote jobs for service. Its purpose is to be able to handle these incoming jobs while the service to the local ones is not significantly affected. Low-mark works as follows: whenever a request to execute a remote job arrives at a machine, the system checks if its load is less than its Low-mark value. If so, the request is accepted and the job is processed locally. Otherwise the request is rejected.

Clearly, High-mark and Low-mark may be implemented together. We refer to this combined policy as High-Low. At this point we should contrast this work with that of Eager et al. [Eager86a], who also proposed threshold policies for load sharing. Their analysis used the same threshold value for deciding when to offload work to other sites (transfer policy) and for deciding where to run a remote job (location policy). In our scheme, the High-mark plays the role of the transfer policy, while the location policy can be chosen depending on the situation. Low-mark represents the acceptance policy. High-mark and Low-mark would be used in addition to location policies to control when a remote interaction should take place.

4.2.1. Description

Figure 4.1 presents the algorithm for the High-Low scheme. In our first implementation we chose a random allocation of jobs as the location policy, although it is not a particularly good choice. Under this location policy, the executing site for a job is selected at random and the job is transferred there. No exchange of information is done between the machines in the network. It is simple to implement and no system state has to be gathered by the sites. In [Eager86a] and [Zhou88a] it was shown that randomly selecting where to run a job reduces the system's average response time when compared to the case when no load distribution takes place. These two studies also showed that the

random location policy is very unstable, i.e., its improvements depends on the system's workload. As will be seen later, our scheme reduces the instability of the random allocation policy. This is so because the acceptance policy allows a given site to reject requests for remote execution even when the location policy selects such site. We also implemented another version of High-Low using a least loaded site selection as location policy.

```
At each node:
    When a local job is invoked:
        IF local_load > High-mark THEN
        BEGIN
            executing_node = location_policy();
            <request execution at executing_node>;
            IF <request accepted> THEN
                <transfer job to executing_node>;
        END
        ELSE
            <execute job locally>;

    When an executing request arrives to a node:
        IF local_load < Low-mark THEN
        BEGIN
            <accept request>;
            <receive remote job>;
            <execute remote job>;
        END
        ELSE
            <reject request>;
```

Figure 4.1: The High-Low algorithm

The salient feature of the High-Low scheme is that two different thresholds are used to decide if a job is to be run remotely. This allows a computer to play multiple roles depending on the values that High-mark and Low-mark take. For example, Figure 4.2a shows that if the High-mark value is greater than the Low-mark value then the space of possible load values that a machine can have is divided into three regions:

1) **overloaded** (above the High-mark and Low-mark values);

2) **normal** (above the Low-mark value and below the High-mark value);

3) **underloaded** (below both values).

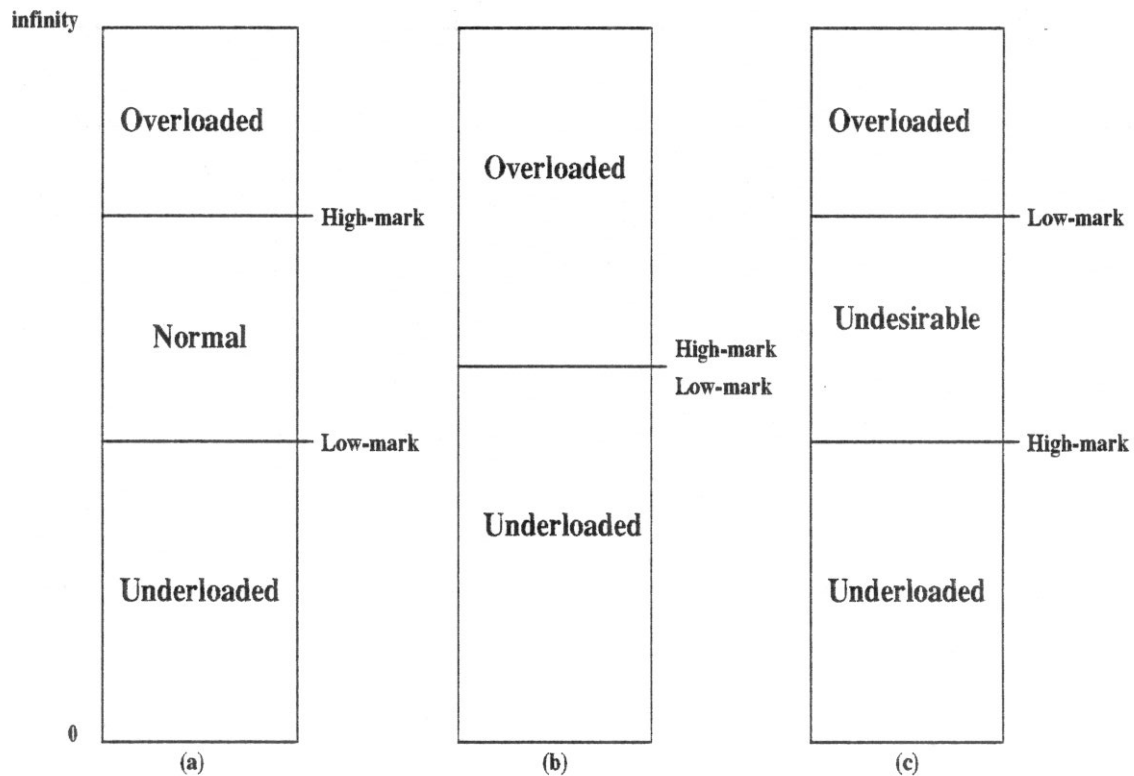

Figure 4.2: Load regions

When the load of a machine is in the overloaded region, new local jobs are sent to be run remotely and remote execution requests are rejected. In the normal region, new local jobs run locally and remote execution requests are rejected. In the underloaded region, new local jobs run locally and remote execution requests are accepted.

Most load sharing algorithms use a single threshold (typically the ''average'' load among all the sites in the network), and thus only have overloaded and underloaded regions (Figure 4.2b). High-Low defines a third region (normal) by its use of two different thresholds. This normal region guarantees a predefined level of performance to the

site owners. It may account for the overhead that the load sharing scheme incurs in transferring and receiving a remote job, or for the level of service that the owner expects. Consequently, a job will not be transferred to another site unless it is worthwhile and a remote job will not be accepted unless there is enough excess capacity to handle it.

Notice in Figure 4.2c that if the Low-mark value is allowed to be greater than the High-mark value, then a fourth region could be recognized: the **undesirable** (above the High-mark value and below the Low-mark value). In the undesirable region the machine would send its new local jobs to remote sites while accepting remote jobs to be executed locally. This is clearly a wasteful use of the resources. To avoid this anomaly, the High-mark value should always be greater than or equal to the Low-mark value. However, if High-mark and Low-mark have the same value (Figure 4.2b) then the implementation of High-Low behaves as a load distribution algorithm that uses a single threshold.

Figure 4.3 shows how particular settings of the High-mark and Low-mark parameters correspond to known modes of operations of a computer. For example, by setting both the High-mark and the Low-mark to 0 (or the lowest possible value), a computer acts as a job dispatcher (Figure 4.3a). The computer behaves as if it were always overloaded, thus placing all of its new local jobs in any available remote machine. This setting could be used to distribute jobs to a set of machines acting as a pool of processors. If instead, both parameters are set to the maximum possible load value (Figure 4.3b), then the machine would act as if it were always underloaded, i.e., it would accept any remote process for execution. Thus, the computer is behaving as a compute server.

It could be possible to vary the High-mark and Low-mark parameters dynamically to allow the computers to change their role in the load sharing scheme. A system process could set these parameters depending on the number of users, or user processes, in the computer. For example, when the last user signs off, the High-mark and Low-mark values could be set to leave the machine in the compute server mode (Figure 4.3b). As soon as a user signs in, these parameters could be set back to leave the computer in its

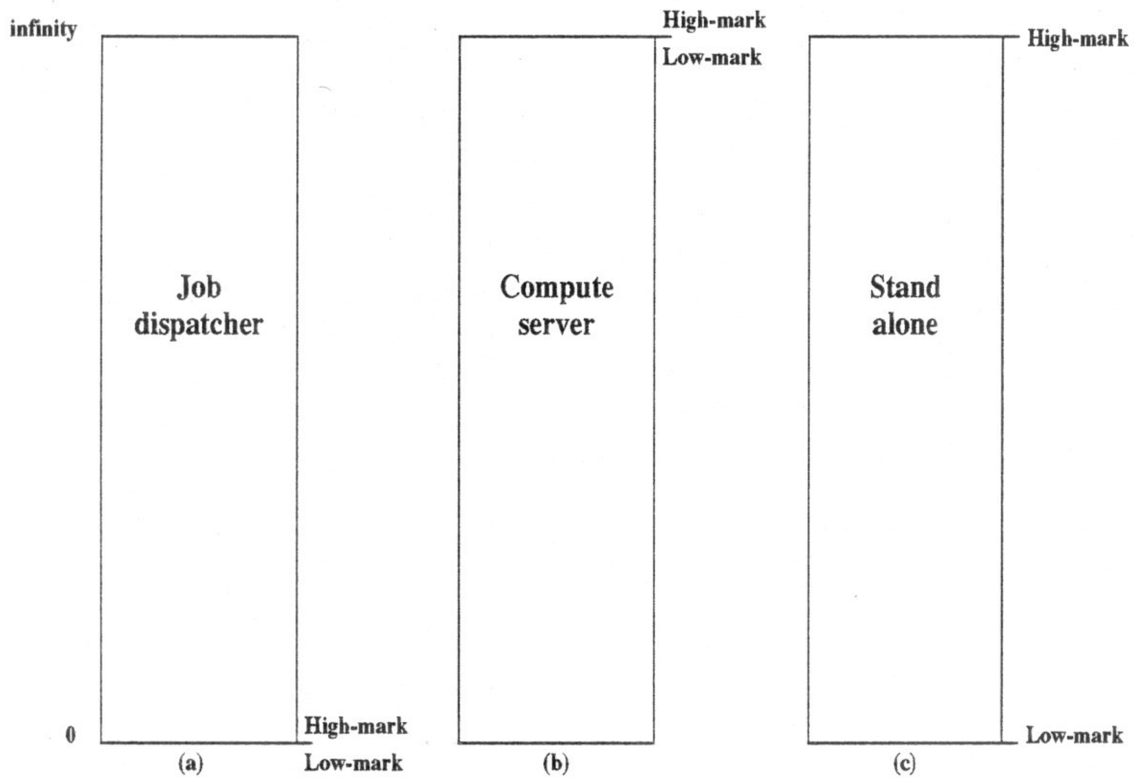

Figure 4.3: Known modes of operations of a computer

normal operational mode. If a computer is needed exclusively by its owner then the High-mark and Low-mark parameters could be set to the stand-alone mode (Figure 4.3c). The attraction of this scheme is that different modes of operation can be easily implemented by dynamically setting the High-mark and Low-mark parameters.

An important observation is that the degree of participation of each computer in this load sharing scheme is completely distributed. It does not depend on any global information or central controller, just on the site's local use and purpose. This is an important characteristic for federated environments.

Finally, since the load information of a machine represents the available resources in that machine and Low-mark represents the amount of resources that the site's owner is willing to lend, in schemes that require sites to make known their load, the Low-mark value could be included with the machine's load information. In this way other sites

would know in advance the available resources in the network and could plan to use them.

4.2.2. An Analytical Model for setting High-mark and Low-mark

Choosing appropriate values for High-mark and Low-mark is not a simple task. An automatic fine tuning mechanism together with specifications submitted by machine owners could be used to obtain the best results. For example, a user could specify that he will allow his machine to process remote jobs if the average response time for his jobs does not deteriorate by more than 10% of the stand-alone time. Below, we use a simple analytical model to estimate the High-mark and Low-mark values.

We will model a site as a M/M/1 queue [Kleinrock75a], with λ being the number of jobs arriving at the site per time unit, and $\frac{1}{\mu}$ being the number of jobs that can be processed by the site per time unit. Then we know that the average response time (T) of a job in that system will be $\frac{\frac{1}{\mu}}{1-\rho}$ (where ρ is the CPU utilization, $\rho = \frac{\lambda}{\mu}$) and on average there will be $\overline{N} = \frac{\rho}{1-\rho}$ jobs at the site.

Suppose that the users of that site request that the range of response time for local jobs be $T \pm \Delta$. Thus, the High-mark should be set to a load average of N_{high}, such that with N_{high} jobs in the system, a job obtains a response time of $T + \Delta$, and the Low-mark should be set to a load average N_{low}, such that having N_{low} jobs running concurrently will lead to a response time of $T - \Delta$.

Thus, the new T (T') will be :

$$T' = T \pm \Delta = \frac{\frac{1}{\mu}}{1-\rho'}$$

which implies,

$$\rho' = 1 - \frac{\frac{1}{\mu}}{T \pm \Delta} = 1 - \frac{1-\rho}{1 \pm (1-\rho)\ \mu\ \Delta}$$

and hence

$$N_{high} = \frac{\rho'}{1-\rho'} = \frac{1+(1-\rho)\ \mu\ \Delta - 1 + \rho}{1-\rho} = \frac{(1-\rho)\ \mu\ \Delta + \rho}{1-\rho} = \bar{N} + \mu\Delta$$

and similarly,

$$N_{low} = \bar{N} - \mu\Delta$$

For example, consider a system with an arrival rate of 8 jobs/second with a CPU that can process at most 10 jobs/second. Then, $\rho = 0.8$ and on average $\bar{N} = 4$ jobs, and $T = 0.5$ second. If $\Delta = 10\%$, then High-mark is $4 + 10(0.1) = 5$, and the Low-mark is $4 - 10(0.1) = 3$. This means that the site, to guarantee the specify response time ($T \pm \Delta$), should try to execute some of its jobs remotely when its workload is greater than 5 jobs ($load\ average > N_{high}$) and that it could serve jobs from other sites when its workload is less than 3 jobs ($load\ average < N_{low}$).

4.3. High-Low, lsh and no-ld

In [Alonso86a] a load balancing scheme (called **lsh**) was developed based on broadcasting local system state to all the sites in a local-area network (LAN) and on transferring jobs to the least loaded site. Each site reaches load balancing decisions in a decentralized fashion, i.e., without the existence of a central controller. The purpose of the prototype was to demonstrate that sizable overall system performance gains could be achieved using a simple load balancing mechanism on top of an existing system with small overhead and making very few changes in the underlying software. It was noticed that having accurate information about the entire system was expensive because processing broadcast messages from other sites takes a substantial amount of CPU cycles. There is a tradeoff between the broadcasting interval and the processing overhead which directly affects the accuracy of the information on which a machine has to base its locality decision. In a follow-up study [Alonso86b], this tradeoff was discussed and the issues involved in evaluating load metrics and decision policies were described.

Lsh was revised and improved to take care of obvious flaws that a simple ''least loaded'' scheme has. These flaws are the swamping and drought effects. These effects are produced by the same factor: outdated system state information due to update interval and communication delays.

In the swamping effect many jobs are sent to one machine (the least loaded at that moment) before it can broadcast its new load. This occurs because several machines may choose to transfer jobs to the least loaded site within the same small interval of time, before new state information from the least loaded machine is broadcasted. Therefore, the response time of these transferred jobs may even be greater than if they had been processed in their originating sites.

In the drought effect, truly least loaded sites do not receive remote jobs. This happens because the moment a machine gets less loaded or even idle is not synchronized with its broadcast interval. A site may be the least loaded site in the network, but until it broadcasts its new state, no other site will know it.

To correct the above described anomalies two policies were incorporated into lsh: required load difference and implied load. The required load difference limits a machine to send jobs to a remote site only if the difference between its load and the load of the remote machine is greater than some specific amount. This would reduce remote execution overhead. With implied load the system load information kept at a machine is updated each time a local job is migrated to another site. For example, when a machine transfers a job to another site, it reflects this event in its local information about the receiving site. This is done to compensate for the added workload at the remote site. In this way the sending machine has more accurate information when making the next load balancing decision.

As stated in Section 4.1, the objective of load balancing schemes is to equally distribute the workload among all the participating sites. We will show empirical measurements of lsh to compare it against different implementations of High-Low. Also for

comparison purposes, we will show performance measurements for the case when no load distribution is done.

4.3.1. Experiments and Results

The system we used for our experiments uses a network of workstations. Our environment consists of four identical machines (SUN 2) connected by an Ethernet [Metcalfe76a] and using a fifth machine as a file server. The load metric we used for our experiments is the 4.2 BSD UNIX "load average" metric provided by the *uptime* command [Leffler84a] and defined as the exponentially smoothed average number of jobs in the run queue over the last 1, 5 and 15 minutes. In our experiments we use the average over the last minute. This is the load metric we used for all the experiments in this study. Our implementation also uses numbers related to this "load average" metric as High-mark and Low-mark values.

Using this facility we first implemented the High-Low scheme using a random allocation of jobs as location policy. We ran several tests with our implementation and compared the obtained results against the situation when there is no load distribution and with the lsh implementation (labeled respectively **no-ld** and **lsh** in our figures). In our experiments, each user is simulated by a script and the time it takes to complete is what we define as response time. The script consist of repeated cycles of editing, compiling and running a C program. The C program performs several arithmetic operations.

Emphasis was not only on the average response time of the entire system, but also on the average response time of the jobs at each individual machine. This last measurement gives an idea of the changes in local service time when a machine participates in a load sharing scheme. Therefore, indirectly the local average reflects how much service autonomy each site has.

Figures 4.4 through 4.6 show the average response time of the High-Low scheme with a fixed High-mark value (1.75) and several Low-mark values. The High-mark value

was selected after running several experiments with just the High-mark parameter [Cova88a]†. Each figure represents a system with a different system load. Each machine in the network has a particular number of users. For example, the distribution ''5,1,1,1'' represents the number of users in the system, i.e., five users in one computer and a single user in each of the other machines. This distribution, ''5,1,1,1'', represents a low system load, ''5,5,1,1'' represents a medium one and ''5,5,5,1'' represents a high system load.

The abscissa depicts a range of increasing Low-mark values and two special values one for no-ld and one for lsh. These values represent the amount of CPU cycles that each particular machine dedicates to service remote jobs: from no sharing to full sharing. That is, it represents the level of acceptance that the local machine has agreed to have. It is the degree of service autonomy that each machine has. The labels of the ordinate denote the average response time of jobs submitted by the local users at each site. In each figure there are three bars labeled ''5'',''avg'', and ''1''. The ''5'' and ''1'' bars represents the average response time perceived by the users submitting jobs at machines with five users and a single user, respectively. The ''avg'' bar is the average response time of the entire system.

The first noticeable result from all these figures is that no user perceives an average response time for its jobs close to the average response time of the system. This last measurement is what is usually reported in load balancing studies.

Figure 4.4 also shows that even maintaining a high degree of service autonomy (i.e., a small willingness to share represented by a Low-mark value of 0.4), there is a substantial improvement in the performance of heavily loaded sites while the lightly loaded ones are not significantly penalized. Also, as the Low-mark value increases more sharing is allowed and thus less service autonomy the site has. In this case, the response times of all

---

† In [Cova88a], after testing High-mark values ranging from 0.5 up to 3.25, we concluded that using a High-mark value that represents the average load of a user is a good threshold for deciding if a local invoked job should be run locally or remotely.

the sites tend to show a balanced effect, i.e., High-Low's performance is similar to lsh's.

Figures 4.5 and 4.6 show that even with increased system workload it is still possible to obtain performance improvements at the heavily loaded sites without significantly affecting the service at the lightly loaded ones. This is not true when there is complete sharing, as is the case of load balancing schemes. As you will note in the same figures the response time of the lsh scheme does not scale well as the load of the system increases (a low system's load in figure 4.4, a medium system's load in Figure 4.5, and a high system's load in Figure 4.6). Also, notice that in these figures the average response time (avg.) does not uniformly decrease as the Low-mark increases (Low-mark 1.4 in Figure 4.5 and Low-mark 1.2 in Figure 4.6). These "bumps" are produced by the random nature of the location policy. As it can be seen, even with this anomalous behavior the response time of all the heavily loaded sites is still lower than in the no-ld case and, in some cases, lower that the lsh scheme.

Figure 4.7 presents a summary of the results for a High-Low implementation that uses a "least-loaded" allocation of jobs as location policy. In this figure the abscissa represents the system workload, as explained before. For each abscissa label, there are three columns: one for each type of machine in the experimental system (heavily loaded ones, with five users, and lightly loaded ones, with a single user), and one for the average response time of the system. In each column the response time for each Low-mark value, no-ld and lsh is represented. The High-mark value is fixed to the same value as before (1.75).

From Figure 4.7 we can see the incremental changes in performance as we change the Low-mark value. It is clear from this figure that even when there is a lot of activity in the system (user distribution "5,5,5,1"), some improvement in response is achieved by sharing resources with High-Low. Also, the higher the value of the Low-mark parameter, the more the response time of heavily loaded machines improves, but the more the response time of lightly loaded machines degrades. Notice the excessive penalty that

lightly loaded sites are paying when there is complete sharing (lsh case). This behavior gave us the insight that load distribution algorithms in order to support autonomy should not behave in a binary fashion (share all or share nothing). Instead they should be gradual as is High-Low.

Also note that, as with the first implementation, there are some values for the High-Low parameters, under medium and high system load, that have lower average response time than lsh. In the next section we discuss this observation.

For the rest of the figures in this chapter we use the formats just described for the figures in this section.

4.4. High-Low, All-or-Nothing, and Priority

Informally, there have been other acceptance policies discussed and used in the load sharing literature. First, is the extensively used **All-or-Nothing** scheme where idle sites are used to service remote jobs until they are claimed by their owners ([Nichols87a], [Litzkow87a], [Theimer85a], [Agrawal87a], [Mutka87a], etc.) Second, is the **Priority** acceptance policy where remote jobs are accepted at a site with lower scheduling priority than local jobs ([Hagmann86a], [Leland86a], etc.). These methods clearly guarantee ownership of resources to the machine's owners and allow different degrees of service autonomy. We will empirically compare them against our High-Low scheme by using a synthetic workload.

We decided that to fairly compare the different acceptance policies we had to use the same schemes for the other policies involved in load distribution algorithms, i.e., the information, transfer and location policies. For this, we chose to emulate the different acceptance policies by using our High-Low implementation. Also, to have more general results, we used two different sets of information, transfer and location policies.

One set of policies consisted of no information exchange among the sites, a threshold transfer policy and a random selection of execution site as the location policy. The

other set consisted of a periodic exchange of load average information among the sites as the information policy, running each job in the least loaded site as the location policy and using a required load difference between the least loaded site and the originating site as the transfer policy [Alonso86a]. We denote the implementation of the former set by **random** and the implementation of the latter set by **lsh**† (the lsh name is used because the corresponding set of policies is based on the lsh scheme described in Section 4.3).

4.4.1. Description of Emulated Acceptance Policies

As explained in Section 4.2.1, High-Low can be used to emulate different modes of site operations. By slightly modifying our High-Low implementation we were able to emulate the All-or-Nothing policy and the Priority policy.

The All-or-Nothing policy was emulated by using High-Low with a High-mark set to the lowest possible value (0.01) for the lsh implementation. This setting forces High-Low to always consider the least loaded site in the network to offload all locally initiated jobs. For the random implementation the High-mark was set to a a median value (1.75), thus acting as the threshold for the transfer policy. The Low-mark was set to the lowest possible value (0.01) for both sets. In this way remote jobs are only accepted for execution at a given site if the load average of that site is almost zero, i.e., if the machine is idle.

The Priority policy was emulated by running the process that handles remote executions (the lshd daemon [Alonso86a]), with the lowest scheduling priority possible (a nice(1) value of +20 [Leffler84a]). This process behaves as follows: when a remote job is accepted for execution at a site, it will instantiate (fork) a child process that will take care of getting the environment information of the job and running a local instance of it. This child process will have the same scheduling priority as its father , thus the remote

---

† There are many parameters involved in each set of policies. A limited sensitivity analysis on these parameters was done for the lsh set in [Alonso86a] and in [Cova88a].

job will run with low scheduling priority, too. For both sets, High-mark was set to the same value than for the All-or-Nothing emulation. The Low-mark was set to a large value (100) for both implementations to assure that every remote request would be accepted.

4.4.2. Experiments and Results

The experiments we ran were similar to the ones described in Section 4.3. Along with the two previously mentioned acceptance policies, we gathered results from three different instances of the High-Low algorithm:

- A configuration where all the sites have their High-mark set to the average load of the network (1.75), and the Low-mark set to a very small value (0.4), thus providing a high level of service autonomy (i.e, allowing little remote workload service). This configuration is labeled "0.4/1.75" in our figures.
- A configuration where heavily used sites, i.e. sites with 5 users, are completely service autonomous. Consequently they do not allow remote workload service (Low-mark = 0.01). Their High-mark was set to the average system's load (1.75). The lightly used sites, i.e. sites with a single user, allow some remote workload service (low-mark = 0.6), and their High-mark is set higher than the system's average (2.0). In this way these sites will not try to offload their own workload during transient conditions due to remote workload service. This configuration is labeled "0.01/1.5-0.6/2" in our figures.
- A configuration where heavily used sites have a high level of service autonomy, thus they allow very little remote workload service (Low-mark = 0.4) and have their High-mark set to a much higher value than the system's average (2.4). In this way they will only try to offload their workload when their local service has greatly degraded. The lightly used sites allow a considerable amount of remote workload service (a small degree of service autonomy reflected by a Low-mark = 0.8) and

only will try to offload their own workload when their load is well above the average (High-mark = 3.0). This configuration is labeled "0.4/2.4-0.8/3" in our figures.

We also gathered results for the cases where no acceptance policy is used, i.e, the random and lsh sets of policies by themselves, as well as for the case when there is no load sharing (no-ld). Thus, a total of seven load sharing situations were tested per set of policies.

In Figures 4.8 through 4.10 we present the response time of the different acceptance policies using the lsh set of policies† and under different system loads (user distributions). The first noticeable result is that for all the system loads the performance of the All-or-Nothing policy can always be improved, i.e., the All-or-Nothing policy is only better than the no load distribution case (labeled no-ld in the figures), but in some cases, when the system load is extremely high (Figure 4.10), it can even be worse than this case. As can be seen in Figures 4.8 through 4.10, there are other policies that have the same characteristics than the All-or-Nothing policy, but do not have this observed anomaly. The other policies behave well under increasing system load and even do better under low system load. The only observation in favor of this policy is that its standard deviation (Figure 4.11) tends to be lower than other policies, in particular for medium and high system load. This happens because less interaction among sites occurs as load increases.

In brief, our results suggest that the All-or-Nothing acceptance policy can always be improved, no matter the system load. It does not have any feature of merit except that it is the obvious way (and simplest to implement) to guarantee ownership of resources. The All-or-Nothing policy has worked well for many researchers because their systems load is constantly low [Mutka87b]. There is always a high probability of remote processing availability due to idle sites in the network.

---

† Although we have collected results using the random set, we will not present the corresponding figures in this work. We do this to limit the number of figures in this presentation, even though they support the conclusions drawn from the lsh set.

Examining now the results from the Priority scheme, we notice that it achieves considerable performance improvement for the heavily loaded machines, but it also penalizes more the lightly loaded sites in comparison to other acceptance schemes. As the system load increases the average response time of the lightly loaded sites degrades. Also, a higher variability (Figure 4.11) is present for both type of sites. This anomalous behavior has to do with the load information policy used for lsh. The load average measure we use in these experiments does not distinguish between local jobs and remote jobs, i.e., it expresses the actual load of a site regardless of what type of job is producing it. Thus a lightly loaded site may get a number of remote jobs to service (placed in the background) which increases its load average value. The next local job that arrives will see a high load average value which will induce the load sharing software to place such a local job in a remote site with **low** scheduling priority This is reflected in a poorer response time than if it was run locally. Therefore, the lightly loaded sites are effectively being penalized.

One way to correct this anomaly in the Priority policy is to be able to distinguish at each site the load generated by local jobs from its load average. In other words, the load sharing mechanism could use two load measurements: one for local jobs and one for the entire workload. Clearly, this improved Priority policy relies on the use of more information to based its load balancing decisions, i.e., a different information policy that the ones we have been using.

The Priority scheme has the largest standard deviation for all system loads (Figure 4.11). The variability increases in direct response to the increase in system load. This is an undesirable behavior because users of the load sharing software cannot estimate the response time for their jobs.

Researchers using the Priority policy explicitly allocate jobs throughout the network ([Hagmann86a], [Mutka87a].) Users of lightly loaded sites do not use the load sharing mechanisms, while users of heavily loaded sites explicitly request that part of their work-

load be placed at remote machines. This approach to using the load sharing software goes against our belief that such mechanisms should transparently take care of remote executions on behalf of inexperienced users (as stated in the introductory paragraphs of this chapter).

With respect to the load sharing algorithms not using an acceptance policy (lsh), we notice that the lightly loaded sites are heavily penalized. The system improvement comes from equalizing the system workload among all the sites. The degradation is more noticeable when the system load is high. As we have already noted in Subsection 4.1.2, this scenario is acceptable if the entire network belongs to the whole user community, i.e., the system is being shared equally by all its users. This situation is not appropriate when each site is part of a federated environment because the components loose control over their resources. Again, local autonomy to guarantee resource ownership is the motivation behind our work, and thus behind the acceptance policy.

Let us now consider the High-Low results. Since High-Low allows a gradual sharing of resources, it offers a marked improvement over the All-or-Nothing policy and the no-ld case. It also provides better control of the local resources than the Priority scheme, i.e., a lightly loaded site participates in the load sharing mechanism only to the extent that its owner allows. Also, because each site pledges a portion of its resources to service remote jobs, any remote job once accepted for execution, has some assurances over the quality of service it is going to receive. This property is not present in any of the other acceptance policies.

A final observation is that the standard deviation of High-Low tends to decrease as the system load increases, in contrast to Priority and similarly to All-or-Nothing. As with All-or-Nothing this is due to the fact that less remote processing occurs as the system load increases.

4.5. Summary

We have discussed the federated computing environment, a type of distributed system where sharing the processing capabilities among the sites improves performance. We have argued that load balancing algorithms are not appropriate for this type of environment because of its characteristics of autonomy and local ownership of resources at each site.

The sharing scheme we have presented, called High-Low, replaces the notion of stealing CPU cycles with the notions of lending and borrowing CPU cycles. Users in federated environments do not have to be concern with their resources being abused. The degree of participation of each site in this load sharing scheme is completely distributed. It does not depend on any global information or central controller, just on the site's local use and purpose.

The All-or-Nothing and Priority acceptance schemes of load sharing are too restrictive in an environment where most resources are underutilized. Also, they do not scale well as the system load increases. Instead, by allowing a moderated sharing among the sites, good performance is guaranteed to the site local community and to jobs originated at other components

Our results suggest that the All-or-Nothing acceptance policy can always be improved, no matter the system load. It does not have any feature of merit except that it is the obvious way (and simplest to implement) to guarantee ownership of resources.

High-Low has the desirable characteristic that an All-or-Nothing scheme has, i.e., it guarantees to the owners of lightly loaded machines that their local resources will not be abused by remote jobs. This is achieved by fixing the maximum degradation that a user of a lightly loaded site might perceive (which reflects the level of service autonomy desired). At the same time, it avoids the anomaly of having poorer performance for the heavily loaded machines as the load of the network increases (when it should have an opposite behavior since it is in this situation that improvement is most needed). The

High-Low scheme improves the performance of heavily loaded sites in a greater amount than All-or-Nothing or Priority, and is close to the performance of the load balancing schemes tested (lsh and random).

We realize that the performance of a computer does not depend solely on its CPU utilization. We have just used this resource to illustrate our ideas. The notions behind the acceptance policy and service autonomy could also be applied to whatever local resource is being shared in the federation, such as physical memory or disk space.

Chapter 5. Conclusions and Future Work

This dissertation has empirically shown the practicality and desirability of resource management mechanisms that support local autonomy in distributed systems. It has been demonstrated that by using resource management mechanisms that support several degrees of autonomy, cooperation among autonomous sites is effectively established.

This dissertation claims that the next generation of distributed systems will arise by joining together several administratively autonomous computing sites into a *federated computing environment*. Consequently, cooperation among sites will occur only with the express consent of the participating sites, and only as long as the service obtained from this cooperation does not interfere with the service rendered to local users at each site.

The approach taken in this dissertation was to study the management of two resources, storage and processing, to explore two distinct notions for autonomy: *functional autonomy* and *service autonomy*, respectively. For functional autonomy, we studied the concept of *stashing* files. By implementing stashing using a quasi-copy framework, the resulting network file system allows users to fine-tune the quality of the service they require when the file server is not reachable. The key point of this particular work is the trade-off between data availability to applications and degradation of service due to ''imperfect'' information. For service autonomy, we studied load sharing in federated environments. We proposed the use of an *acceptance* policy to provide local control to each site participating in the load sharing system independently of the location, information, and transfer policies used. With the acceptance policy sites' owners control how much local processing they are willing to offer to remote jobs, depending on the performance degradation that local jobs will perceive. It was shown that even when some sites are only willing to accept few remote jobs, substantial improvement is obtained at heavily loaded sites while service at lightly loaded ones does not deteriorate significantly.

5.1. Lessons Learned

Many of the issues related to autonomy in distributed systems are also connected with fault-tolerance. While we were working on our scheme to make a client site more independent of server machines (i.e., *stashing*, Chapters 2 and 3), we concerned ourselves with problems of logging, checkpointing, transaction execution, time-out periods, etc. It is our belief that in a fault-tolerant distributed system local autonomy can be achieved almost at no cost, since many of the protocols used to achieve fault-tolerance can also be used to achieve autonomy. For example, when a client detects that a server is unavailable, it makes no difference to the client whether the server crashed, or there was a network partition, or that the server decided to exercise its *service autonomy*.

One issue that often arose during the work on stashing was the relationship between caching and stashing. This dissertation maintains that because of their distinct goals, caching and stashing have to be kept separate, but they do not compete with each other. If we had to design an entirely new distributed file system for a VLDS, we would use caching between clients and servers to improve performance, stashing between clients and servers to increase service availability when disengagement occurs, and replication among servers to increase the fault-tolerance of the service. Another idea that arose from the work on stashing was the possibility of using the quasi-copy framework to build a single mechanism that would manage caching and stashing at the same time. The operating system would be responsible for selecting the quasi-copy predicates for files being "cached" (as indirectly proposed in [Gray89a] with the notion of *leases*†), while user application would be responsible for the "stashed" files.

We have learned from our implementations that support for autonomy is easy to introduce to existing resource management mechanisms in distributed systems. The real problem is deciding what is the right policy to use, since autonomy is intrinsically related

---

† See discussion in Section 2.4.1.

to user applications and the level of service that users expect. For example, among my colleagues at Princeton, we have had many interesting conversations trying to decide what would be the best set of values for our load sharing scheme (Section 4.2). All our conversations have ended with no consensus, since what is fair for someone, is not fair for someone else. We believe that having flexibility to adapt to different levels of autonomy is the key to sharing resources in VLDS.

It is interesting to note that most interactions among autonomous organizations cannot be enforced, but have to be agreed upon, and will terminate as soon as any of the involved parties so wishes (e.g., The United Nations, The Open Software Foundation, or the Arab League). It is this organizational dynamic that we have tried to capture in our resource management mechanisms for VLDS.

5.2. Future Work

There are several open issues that were generated from the work of this dissertation. We now present three of these problems which we have started studying.

5.2.1. Propagating Updates to Replicated Copies

One particular problem we are exploring is how to keep sufficiently consistent quasi-copies in the presence of updates to the central data. Maintaining quasi-consistency implies deciding how to efficiently propagate updates to the quasi-copies throughout the system. We denote it as the **quasi-copy update problem**

In the basic quasi-copy scheme, each client that wants to have a quasi-copy does so directly to its local disk, thus there is a one-to-one correspondence between quasi-copy requests and the number of quasi-copies created. Our objective is to minimize the number of messages needed to maintain the consistency requirement defined for each quasi-copy when updates occur at the server.

Formally, the problem can be stated as follows. Given a graph $G = (V,E)$ that represents a network, where V is the set of computer nodes, E is the set of all

communication links between nodes, for all $v \in V$, p(v) is a consistency requirement of quasi-copies at v, and for all $e \in E$, c(e) is the cost of sending an update through e.

The question we would like to answer is what is the minimum cost propagation scheme to update all $v \in V$ so that the consistency requirements, p(v), are satisfied at all times.

In general, the quasi-copy update problem seems to be an NP-complete problem since we believe it is possible to show that the ''Steiner tree'' problem [Garey79a] can be reduced to an instance of the quasi-copy update problem†. The approach we are taking to deal with the quasi-copy update problem is in two stages. First, we would like to restrict the problem and develop algorithms that provides spanning trees for update distribution under the assumption that no dynamic changes will occur to the network configuration. Second, we would like to relax this assumption, and solve the quasi-copy update problem adapting the algorithm proposed in [Eppstein90a] for maintaining a minimum spanning forest subject to dynamic changes in the costs of sending updates, c(e), as well as insertions and deletions of nodes and communications links, V and E.

For the first stage of this research, although the general problem most probably is NP-complete, we can define restricted versions of the quasi-copy update problem and find solutions to them. In Table 5.1, we show a progression of these simplified problems and the techniques for solving them. Clearly, if we consider only the normal replicated copy case (i.e., perfect consistency is required), then each time an update occurs at the server, all the copies have to be locked and the updates also propagated to them. Since all the copies have to received the update ''at the same time'' the least expensive way to propagate updates is using the minimum cost spanning tree (MSP) connecting all sites

† An instance of the Steiner tree problem can be reduced to an instance of the quasi-copy update problem by adding to the statement of the former a ''perfect consistency'' quasi-copy predicate for all the vertices in the specified subset of the Steiner tree problem statement. If there were a polynomial algorithm to solve the quasi-copy update problem, it would find the minimum cost spanning tree to the problem, which would also be a solution to the Steiner tree problem.

with copies. Several algorithm for finding the MSP of a graph have been proposed in the literature. For our purpose it is sufficient to use Reingold's algorithm [Reingold77a] of order $|E| \log \log |V|$.

Problem	*Technique*	*Complexity*
Perfect copies	minimum-cost spanning tree	$O(\|E\| \log \log \|V\|)$ *(Reingold)*
Fixed consistency requirements	algorithm based on depth first search	$O(\max(\|V\|^2, \|E\| \log \|E\|))$ *(Cova)*
Flexible consistency requirements and triangle inequality holds	1-server with excursion problem	3-competitive *(Black and Sleator)*
Flexible consistency requirements and triangle inequality **does not** hold	OPEN	

Table 5.1: Progression of problems to propagate quasi-copy updates

If we now consider relaxing the perfect consistency assumption, and assume that the system cannot modify the user consistency requirements, then we have developed an $|V|^2$ algorithm that constructs the particular spanning tree satisfying such requirements. Our algorithm constructs a particular spanning tree that satisfies a "less restricted than" relation among the consistency predicates of the quasi-copies in the network.

Finally, we would allow the system to maintain quasi-copies that are more consistent than what the users requested. This gives the system greater flexibility to send update messages and thus could allow the use of a cheaper spanning tree than what our previous algorithm would find. Finding the minimum spanning tree in this case becomes more complex, and its solution depends on whether the triangle inequality holds†. If the triangle inequality holds, then we can reduce the quasi-copy update problem to an instance of the k-server with excursion problem, studied by Black and Sleator [Manasse88a]. They show that a 1-server problem on a tree has a competitive factor of three (which has also been proven to be the best possible factor). A competitive factor of three means that there is an on-line algorithm that can produce an answer to the problem

within three times of the "performance" of the optimal off-line algorithm that solves the same problem.

To conclude, we have outlined a problem of theoretical interest that arose from our implementation of a network file system using quasi-copies to support file stashing (Chapter 3). We believe that the study of algorithms to manage *controlled inconsistencies* in replicated data is important, since achieving perfect consistency in VLDS is an unrealistic goal.

5.2.2. Bootstrap Negotiation

Since in a VLDS there are potentially tens of thousands of machines, it is **very** difficult to have complete homogeneity across all the components, neither it is desire in many cases. What is required is to establish the role of homogeneity in this type of environment, i.e., set the **lowest common denominator** that allows the components to interact among themselves. There are others researchers also looking for this common denominator.

Several researchers have proposed sets of protocols to be used by different vendors and implementors to develop their distributed applications, and therefore allow interoperability across machines ([Zimmermann80a], [SUN88a].) Others have proposed homogeneous kernels to support distributed operations ([Cheriton88a], [Turnbull87a], [Schmidtke82a].) A third group of researchers, represented by the HCS project [Notkin88a], works on a basic set of network services (mail, filing, printing, naming, authentication and remote computation) that adapt to the demands of a heterogeneous environment, mainly through dynamic binding.

Our idea is to take this search for a common denominator one step lower to what we call a **bootstrap** negotiation process. This process must be based on a simple language

† The triangle inequality property of a graph states that the total cost of a path between two vertices is proportional to the number of edges in that path. Thus, the shorter the path the less expensive it is.

to express basic operations and ways to iterate to higher levels of interactions. The common language should be very simple, which means a restrictive type of interaction. This common language should be use to agree on some higher level of cooperation among the nodes. In this way, implementation of this language for new members would not take much effort. Even more, there could be more than one common language (more that one standard) and nodes that know several standards could help other nodes to "learn" new common languages.

An example where the bootstrap negotiation process will solve many questions is in the discussion between stateless and statefull interactions. This is another aspect of an interaction that should be negotiated. Stateless interactions are easier to implement (no crash recovery mechanism necessary), but cost more since each message has to be self-contained. In many cases, using statefull protocols will reduce the costs of communication. Again the nodes can agree on what style of interaction they are going to use and what type of state they are going to save. The protocol should also allow each node to check with the other nodes to make sure that they are all in the same state.

An aspect of federated environments is that it is very difficult for an entity of the system to know what happens when something goes wrong in another entity, e.g., maybe a new operating system was brought up in another machine or a new RPC protocol is being used. By way of the bootstrap negotiation process, the interaction between these entities could again be resumed from the lowest common level, iterating to the appropriate level of cooperation. Therefore allowing new configurations to take place.

5.2.3. Exchanging Data Among Heterogeneous Database Systems

In this dissertation we have focused in operating system's resources (storage and processing), but many organizations want to share information that they already keep in databases. However, these organizations are not willing to surrender local control of their database management systems (DBMS) as the price for cooperation. Moreover, the choice of software and transaction operation should be a local decision. Heterogeneity in

hardware and software will prevail across the different organizations. Consequently, we are in the presence of a federated computing environment.

To share databases in federated environments, we propose an architecture that preserves the autonomy of each site. Figure 5.1 shows the software layers in the proposed architecture†. The architecture would establish a set of ''standard'' protocols to be implemented independently by different DBMS vendors.

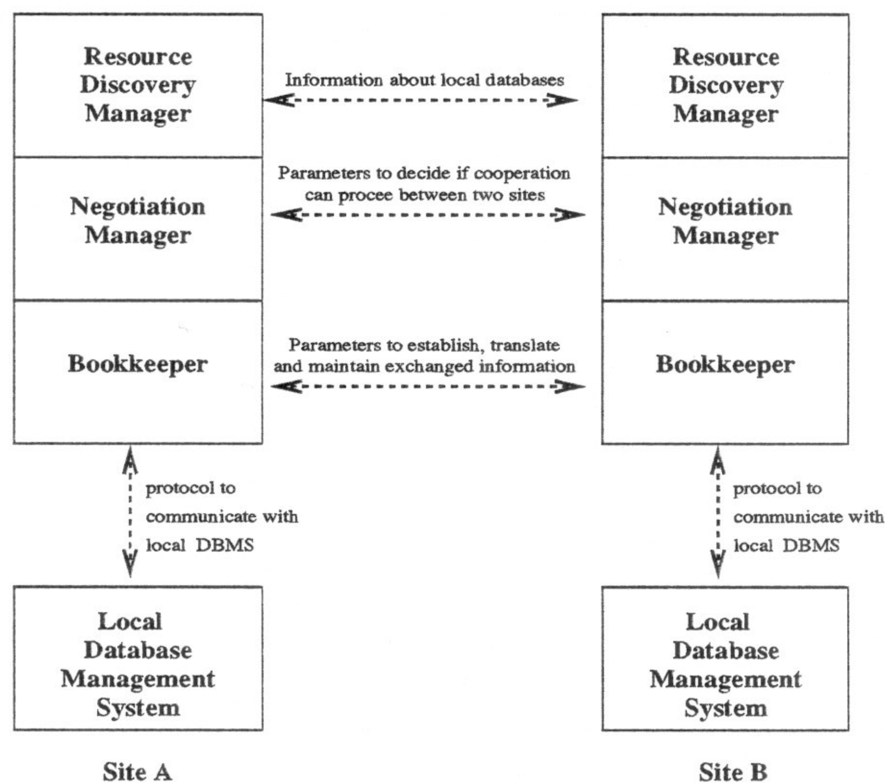

Figure 5.1: Layer architecture for data exchange among heterogeneous DBMS

The top layer is the Resource Discovery Manager (RDM). This layer would implement a protocol to inform other nodes in the network what information it has available for sharing in its local database. The protocol would also allow each node to collect such information about other node's local databases.

† Note that the architecture shown in Figure 5.1 was inspired in the NFS architecture [SUN88a] for network file systems (see Section 3.2.1 for an overview).

The middle layer, the Negotiation Manager (NM), would implement a protocol to exchange a particular information (''view'') between two sites that desire to cooperate. It is at this layer that local control would be enforced. It is the NM layer that decides, based on user provided parameters and the present ''load'' of the system, if a new interaction among two nodes will be establish. This layer receives from the RDM layer parameters indicating the site that holds the information desired and a specific way of identifying such information. It uses these parameters to talk to the NM layer of the remote site.

Finally, the Bookkeeper layer takes care of the actual transfer, translation and maintenance of the information being exchanged. This could be done by implementing quasi-copies of the remote information or by allowing remote queries to the site holding the desired information. This layer would perform equivalent functions to the Bookkeeper process of the stashing implementation described in Section 3.2.5. The Bookkeeper would receive from the NM layer the identifier for the required information as well as the parameters agreed on by the cooperating sites to maintain the desired level of autonomy (e.g., the rate of remote queries to the exchanged data). This layer will communicate with the underlying DBMS to extract and install the exchanged information, using a protocol specifically designed for the local DBMS.

The proposed architecture is just a general description of an approach to the multidatabase problem in federated environments. The design, development and evaluation of particular protocols for each layer is still an open research field.

References

Agrawal87a.

Rakesh Agrawal and Ahmed K. Ezzat, ''Location Independent Remote Execution in NEST,'' *Transactions on Software Engineering*, vol. SE-13, no. 8, pp. 905-912, IEEE, August 1987.

Alonso86b.

Rafael Alonso, ''The Design of Load Balancing Strategies for Distributed Systems,'' *Proceedings of the U.S. Army Research Office Future Directions in Computer Architecture and Software Workshop*, May 5-7, 1986.

Alonso86a.

Rafael Alonso, Phillip Goldman, and Peter Potrebic, ''A Load Balancing Implementation for a Local Area Network of Workstations,'' *Proceedings of the IEEE Workstation Technology and Systems Conference*, March 18-20, 1986.

Alonso90a.

Rafael Alonso, Daniel Barbara, and Hector Garcia-Molina, ''Quasi-Copies: Efficient Data Sharing for Information Retrieval Systems,'' (to appear) Transactions on Database Systems, ACM, 1990.

Anderson88a.

David P. Anderson and Domenico Ferrari, ''The DASH Project: An Overview,'' Technical Report 88/405, UC Berkeley CS Division, FEB 1988.

Barbara89a.

Daniel Barbara and Richard J. Lipton, ''Randomized Technique for Remote File Comparison,'' *Proceedings of the 9th. International Conference on Distributed Computing Systems*, Computer Society Press, Newport Beach, California, June 1989.

Birrell82a.

Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder, ''Grapevine: An Exercise in Distributed Computing,'' *Communications of the ACM*, vol. 25, no. 4, pp. 260-274, April, 1982.

Birrell88a.

Andrew D. Birrell, ''Position Paper ,'' *Proceedings of the 1988 ACM SIGOPS European Workshop*, Cambridge, England, September, 1988.

Bozman89a.

G. P. Bozman, H. H. Ghannad, and E. D. Weinberger, ''A Trace Driven Study of CMS File References,'' technical report, IBM Hawthorne Research Laboratory, Yorktown Heights, NY, July, 1989.

Cheriton88a.

David R. Cheriton, ''The V Distributed System,'' *Communications of the ACM*, vol. 31, no. 3, pp. 314-333, Association for Computing Machinery, March 1988.

Cova88a.

Luis L. Cova, ''Load Balancing in Two Types of Computing Environments,'' Tech. Report #CS-TR-165-88, Princeton University Computer Science Department, June 1988.

Daniels86a.

Dean Daniels and Alfred Z. Spector, ''An Algorithm for Replicated Directories,'' *Operating Systems Review*, vol. 20, no. 1, pp. 24-43, SIGOPS, January, 1986.

Davidson82a.

Susan B. Davidson, *An Optimistic Protocol for Partioned Distributed Database Systems,* Ph.D. Thesis, Princeton University, Princeton, N.J., October, 1982.

Davidson85a.

Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen, ''Consistency in Partitioned Networks,'' *Computing Surveys*, vol. 17, no. 3, pp. 341-370, ACM, Sep-

tember, 1985.

Dineen87a.

Terence H. Dineen, Paul J. Leach, Nathaniel W. Mishkin, Joseph N. Pato, and Geoffrey L. Wyant, "The Network Computing Architecture and Systems: An Environment for Developing Distributed Applications," *Proceedings of the USENIX Technical Conference*, pp. 385-398, Summer, 1987.

Douglis87a.

Fred Douglis and John Ousterhout, "Process Migration in the Sprite Operating System," *Proceedings of the 7th. International Conference on Distributed Computing Systems*, pp. 18-25, Computer Society Press, Berlin, West Germany, September 1987.

Eager86a.

Derek L. Eager, Edward D. Lazowska, and John Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 5, pp. 662 - 675, May 1986.

Eppstein90a.

David Eppstein, Giuseppe F. Italiano, Roberto Tamassia, Robert E. Tarjan, Jeffery Westbrook, and Moti Yung,, "Maintenance of a Minimum Spanning Forest in a Dynamic Planar Graph," technical report CS-TR-243-90, Princeton University Computer Science Department, Princeton, N.J., January 1990.

Ferrari87a.

Domenico Ferrari and Songnian Zhou, "An Empirical Investigation of Load Indices for Load Balancing Applications," Tech. Report #UCB/CSD 87/353, University of California at Berkeley Computer Science Division, May 1987.

Flavin88a.

Robert A. Flavin and John D. Williford, "Management of Distributed Applications in Large Networks," *Proceedings of the 21st Hawaii International Conference on*

System Sciences, pp. 232-241, IEEE/CS Press, Kailu-Kona, Hawaii, January, 1988.

Floyd86a.

Rick Floyd, ''Short-Term File Reference Patterns in a UNIX Environment,'' TR-177, Computer Science Department, University of Rochester, Rochester, NY 14627, March 1986.

Garcia-Molina83a.

Hector Garcia-Molina, Tim Allen, Barbara Blaustein, R. Mark Chilenskas, and Daniel R. Ries, ''Data-patch: Integrating Inconsistent Copies of a Database After a Partition,'' *Proc. of the 3rd IEEE Symposium on Reliability in Distributed Software and Database Systems*, pp. 38-48, IEEE, October, 1983.

Garey79a.

Michael R. Garey and David S. Johnson, *Computers and Intractability: A guide to the theory of NP-Completeness,* p. 208, W.H. Freeman & Co., first edition, 1979.

Gifford79a.

D. K. Gifford, ''Violet: An Experimental Decentralized System,'' *Computer Networks*, vol. 5, no. 6, pp. 423-433, December 1981. also in

Gray89a.

Cary G. Gray and David R. Cheriton, ''Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency,'' *Operating Systems Review*, vol. 23, no. 5, pp. 202-210, SIGOPS, December, 1989.

Gray78a.

Jim Gray, ''Notes on Data Base Operating Systems,'' in *Operating Systems: An Advanced Course*, ed. R. Bayer, R. M. Graham and G. Seegmuller, pp. 393-481, Springer-Verlag, 1978.

Hagmann86a.

Robert Hagmann, *Process Server: Sharing Processing Power in a Workstation*

Environment Proceeding of the 6th International Conference on Distributed Computing Systems., pp. 19-23, IEEE Society, Cambridge, May 1986.

Hardwick89a.

Martin Hardwick, David Spooner, Ebba Hvannberg, Blair Downie, Alyce Faulstich, David Loffredo, Alok Mehta, Don Sanderson, Rick Harris, Ghassan Abou-Ezzi, Jie Gong, and Jeffrey Young, ''ROSE: A Database System for Concurrent Engineering Applications,'' Rensselaer Design Research Center technical report No. 89062, Rensselaer Polytechnic Institute , 1989.

Hisgen89a.

Andy Hisgen, Andrew Birrell, Timothy Mann, Michael Schroeder, and Garret Swart, ''Availability and Consistency Tradeoffs in the Echo Distributed File System,'' in *Proceedings of the 2nd. Workshop on Workstation Operating Systems*, pp. 49-54, IEEE/CS Press, Pacific Grove, California, September 1989.

Horwitz88a.

Susan Horwitz, Jan Prins, and Thomas Reps, ''Integrating Non-Interfering Versions of Programs,'' in *Proceedings of the 15th. Annual ACM Symposium on Principles of Programming Languages*, pp. 133-134, ACM, January 1988.

Howard88a.

John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayan, Robert N. Sidebottom, and M. West, ''Scale and Performance in a Distributed File System,'' *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 51-81, February 1988.

Jia90a.

Xiaohua Jia, Hirohiko Nakano, Kentaro Shimizu, and Mamoru Maekawa, ''Highly Concurrent Directory Management in the Galaxy Distributed Systems,'' *Proceedings of the 10th International Conference on Distributed Computing Systems*, pp. 416-423, IEEE/CS Press, Paris, France, May 1990.

Jul88a.

Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black, ''Fine-grained Mobility in the Emerald System,'' *Transactions on Computer Systems*, vol. 6, no. 1, pp. 109-133, ACM Press, February, 1988.

Kleiman89a.

S.R. Kleiman, ''Vnodes: An Architecture for Multiple File System Types in Sun UNIX,'' *Tutorial T6: Open Network Computing and NFS*, USENIX, San Diego, California, February 1989.

Kleinrock75a.

Leonard Kleinrock, ''Queueing Systems, Vol. 1: Theory,'' *Wiley-Interscience*, John Wiley & Sons, New York, 1975.

Kruger88a.

G. Kruger and G. Muller, *Hector: Heterogeneous Computers Together. A Joint of IBM and the University of Karlsruhe. Volume II: Basic Projects,* Springer-Verlag, first edition, 1988.

Kyrimis90a.

Kriton Kyrimis, *Placement of Processes and Files in Distributed Systems,* Ph.D. thesis, Princeton University, Princeton, N.J., June 1990.

Lampson85a.

Butler W. Lampson, ''Designing a Global Name Service,'' *Proceedings of the ACM Symposium on Principles of Distributed Computing*, ACM Press, 1985.

Leffler84a.

S. Leffler, W. Joy, and K. McKusick, *4.2 BSD System Manual,* Computer Systems Research Group, University of California, Berkeley, 1984.

Leffler89a.

Samuel J. Leffler, Marshall K. McKusick, Michael J. Karels, and John S. Quarter-

man, *The Design and Implementation of the 4.3BSD UNIX Operating System,* Addison-Wesley Publishing Company, 1989.

Leland86a.

Will E. Leland and Teunis J. Ott, ''Load-balancing Heuristics and Process Behavior,'' *Proceedings of PERFORMANCE '86 and ACM SIGMETRICS 1986*, pp. 54-69, May 1986. ata

Lindsay86a.

B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms, ''A Snapshot Differential Refresh Algorithm,'' *Proceedings of the ACM SIGMOD International Conference on Management of D*, pp. 53-60, Washington, D.C., May 1986.

Litzkow87a.

Michael J. Litzkow, ''Remote UNIX: Turning Idle Workstations into Cycle Servers,'' *Proceedings of the 1986 Summer USENIX Technical Conference and Exhibition*, pp. 381-384, Phoenix, Arizona, June 1987.

Mamrak85a.

Sandra A. Mamrak, Dennis W. Leinbaugh, and Tony S. Berk, ''Software Support for Distributed Resource Sharing,'' *Computer Networks and ISDN Systems*, vol. 9, pp. 91-107, North-Holland, 1985.

Manasse88a.

Mark Manasse, Lyle McGeoch, and Daniel Sleator,, ''Competitive Algorithms for Server Problems,'' technical report CMU-CS-88-197, Carnegie-Mellon University Computer Science Department, Pittsburgh, Pennsylvania, December 1988.

Metcalfe76a.

R. M. Metcalfe and D. R. Boggs, ''Ethernet: Distributed Packet Switching for Local Computer Networks,'' *CACM*, vol. 19,7, pp. 395-404, July 1976.

Mullender89a.

Sape J. Mullender, ''Chapter 1: Introduction,'' in *Distributed Systems*, ed. Sape Mullender, p. 5, ACM Press, first edition, 1989.

Mutka87b.

Matt W. Mutka and Miron Livny, ''Profiling Workstation's Available Capacity for Remote Execution,'' Tech. Report #697, University of Wisconsin-Madison Computer Science Department, May 1987.

Mutka87a.

Matt W. Mutka and Miron Livny, ''Scheduling Remote Processing Capacity in a Workstation-Processor Bank Network,'' *Proceedings of the 7th. International Conference on Distributed Computing Systems*, pp. 2-9, Computer Society Press, Berlin, West Germany, September 1987.

Needham82a.

R. M. Needham and A. J. Herbert, in *The Cambridge Distributed Computing System*, Addison-Wesley Publishers Limited, 1982.

Nelson88a.

Michail N. Nelson, Brent B. Welch, and John K. Ousterhout, ''Caching in the Sprite Network File System,'' *ACM Transactions on Computer Systems*, vol. 6, no. 1, February, 1988.

Neumann88a.

Barry Clifford Neumann, ''Issues of Scale in Large Distributed Operating Systems,'' FR-35, University of Washington Computer Science Department, Seattle, Washington, May, 1988.

Ni85a.

Lionel M. Ni and Kai Hwang, ''Optimal Load Balancing in a Multiple Processor System with Many Job Classes,'' *IEEE Transactions on Software Engineering*, vol. SE-11, no. 5, pp. 491-496, May 1985.

Nichols87a.

David A. Nichols, ''Using Idle Workstations in a Shared Computing Environment,'' *Operating System Review*, vol. 21, no. 5, pp. 5-12, ACM Press, November 1987.

Notkin88a.

David Notkin, Andrew P. Blank, Edward D. Lazowska, Henry M. Levy, Jan Sanislo, and John Zahorjan, ''Interconnecting Heterogeneous Computer Systems,'' *Communications of the ACM*, vol. 31, no. 3, pp. 258-273, Association for Computing Machinery, March 1988.

Olkin80a.

Ingram Olkin, Leon J. Gleser, and Cyrus Derman, *Probability Models and Applications,* pp. 209-211, Macmillan Publishing Co., Inc., New York, first edition, 1980.

Ousterhout85a.

John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson, ''A Trace-Driven Analysis of the UNIX 4.2 BSD File System,'' in *Proceedings of the 10th ACM Symposium on Operating System Principles*, pp. 15-24, ACM, New York, New York, 1985.

Parker83a.

D. Stott Parker, Gerald J. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline, ''Detection of Mutual Inconsistency in Distributed Systems,'' *Transactions of Software Engineering*, pp. 240-247, IEEE, May, 1983.

Postel80a.

J. Postel, ''DOD Standard Internet Protocol,'' RFC 760, Information Sciences Institute, January 1980.

Reingold77a.

Edward Reingold, Jurg Nievergelt, and Narsingh Deo, *Combinatorial Algorithms - Theory and Practice,* p. 325, Prentice-Hall, first edition, 1977.

Renesse88a.

R. van Renesse, J. M. van Staveren, J. Hall, M. Turnbull, A. A. Janssen, A. J. Jansen, S. J. Mullender, D. B. Holden, A. Bastable, T. Fallmyr, D. Johansen, K. S. Mullender, and W. Zimmer, ''MANDIS/Amoeba: A Widely Dispersed Object-Oriented Operating System,'' *Proc. of the EUTECO 88 Conf.*, pp. 823-831, North-Holland, Vienna, Austria, April 1988.

Renesse89a.

R. van Renesse, A. S. Tanenbaum, and A. Wilschut, ''The Design of a High-Performance File Server,'' *Proc. of the 9th Int. Conf. on Distr. Computing Systems*, Newport Beach, CA, June 1989.

SUN88a.

Sun Microsystems, Inc., SUN, ''ONC/NFS Protocol Specifications and Service Manual,'' Part No. 800-3084-10, Revision A, of 26 August 1988.

SUN89a.

Sun Microsystems, Inc.SUN, *PC-NFS User's Manual*, 1989.

Sandberg85a.

R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, ''Design and Implementation of the Sun Network File System,'' *Proceedings of the Summer 1985 USENIX Technical Conference*, Portland, Oregon, 1985.

Satyanarayanan81a.

M. Satyanarayanan, ''A Study of File Sizes and Functional Lifetimes,'' in *Proceedings 8th Symposium on Operating System Principles*, ACM, December 1981.

Satyanarayanan89a.

M. Satyanarayanan, James Kistler, Puneet Kumar, Ellen Siegel, and David Steere, ''CODA: A Highly Available File System for a Distributed Workstation Environment,'' Technical Report No. CMU-CS-89-165, Carnegie Mellon University Department of Computer Science, July 1989.

Schmidtke82a.

F. E. Schmidtke, ''A Communication Oriented Operating System Kernel for a Fully Distributed Architecture,'' *Pathways to the Information Society. Proceedings of the 6th International Conference on Computer Communication*, pp. 757 - 762, North-Holland, Amsterdam, Netherlands, 1982.

Schroeder84a.

Michael D. Schroeder, Andrew D. Birrell, and Roger N. Needham, ''Experience with Grapevine: The Growth of a Distributed System,'' *Transactions on Computer Systems*, vol. 2, no. 1, pp. 3-23, ACM Press, February, 1984.

Schroeder85a.

Michael D. Schroeder, David K. Gifford, and Roger M. Needham, ''A Caching File System for a Programmer's Workstation,'' *Operating Systems Review*, vol. 19, no. 5, December 1985.

Sheltzer86a.

Alan B. Sheltzer and Gerald J. Popek, ''Internet Locus: Extending Transparency to an Internet Environment,'' *Transactions on Software Engineering*, vol. SE-12, no. 11, pp. 1067-1075, IEEE, November, 1986.

Siegel89a.

Alex Siegel, Kenneth Birman, and Keith Marzullo, ''Deceit: A Flexible Distributed File System,'' Technical Report No. TR 89-1042, Cornell University Department of Computer Science, November 1989.

Smith81a.

Alan J. Smith, ''Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms,'' *Transactions on Software Engineering*, vol. SE-7, no. 4, IEEE, July 1981.

Spector87a.

et al, Alfred Z. Spector, ''Camelot: A Distributed Transaction Facility for Mach and

the Internet - An Interim Report,'' CMU-CS-87-129, Computer Science Dept., Carnegie Mellon Univ., Pittsburgh, PA, June, 1987.

Staelin88a.

Carl Staelin, ''File Access Patterns,'' CS-TR-179-88, Department of Computer Science, Princeton University, Princeton, NJ 08540, September 1988.

Stankovic84a.

John A. Stankovic, ''Simulations of Three Adaptive, Decentralized Controlled, Job Scheduling Algorithms,'' *Computer Networks*, vol. 8, no. 3, pp. 199-217, North-Holland, June 1984.

Tanenbaum89a.

Andrew Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum, ''Experiences with the Amoeba Distributed Operating System,'' technical report, Vrije University Department of Mathematics and Computer Science, Amsterdam, The Netherlands, 1989.

Terry87a.

Douglas B. Terry, ''Caching Hints in Distributed Systems,'' *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 48-54, January 1987.

Theimer85a.

Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton, ''Preemptable Remote Execution Facilities for the V-System,'' *ACM*, vol. 1, pp. 2-12, 1985.

Turnbull87a.

Martin Turnbull, ''Support for Heterogeneity in the Global Distributed Operating System,'' *Operating System Review*, vol. 21, no. 2, pp. 11-21, SIGOPS, April 1987.

Walker83a.

Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel, ''The LOCUS Distributed Operating System,'' in *Proceedings 9th ACM Symposium on*

Operating System Principles, ACM, 1983.

Wang85a.

Yung-Terng Wang and Robert J. T. Morris, ''A Survey and Comparison of Load Sharing Strategies in Distributed Computer Systems,'' *New World of the Information Society, Proceedings of the 7th International Conference on Computer Communication*, pp. 392 - 393, North-Holland, Amsterdam, Netherlands, 1985.

Welch89a.

Brent Welch, Mary Baker, Fred Douglis, John Hartman, Mendel Rosenblum, and John Ousterhout, ''Sprite Position Statement: Use Distributed State for Failure Recovery,'' *Proceedings of the 2nd. Workshop on Workstation Operating Systems*, pp. 130-133, IEEE/CS Press, Pacific Grove, CA, September, 1989.

Zhou88a.

Sognian Zhou, ''A Trace-Driven Simulation Study of Dynamic Load Balancing,'' *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1327-1341, IEEE Computer Society Press, September 1988.

Zimmermann80a.

H. Zimmermann, ''OSI Reference Model: The ISO Model of Architecture for Open Systems Interconnection,'' *IEEE Transactions on Communications*, vol. COM-28, no. 4, pp. 425-432, April 1980.

Zipf49a.

George Kingsley Zipf, *Human Behaviour and the Principle of Least Effort,* Addison-Wesley Press, Cambridge, MA, 1949.