This is to certify that the
dissertation entitled

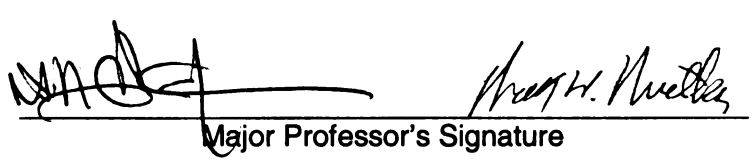
A SELF-MODERATING PEER-TO-PEER FILE SYSTEM

presented by

Boris Gelfand

has been accepted towards fulfillment
of the requirements for the

Ph.D. degree in Computer Science



Major Professor's Signature

5/2/03

Date

A SELF-MODERATING PEER-TO-PEER FILE SYSTEM

By

Boris Gelfand

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science and Engineering

2003

ABSTRACT

A SELF-MODERATING PEER-TO-PEER FILE SYSTEM

By

Boris Gelfand

Peer-to-peer systems have received much recent attention, starting with file sharing systems such as Napster and Gnutella, and developing into true file systems such as PAST and CFS. They demonstrate many benefits over central storage systems including fault tolerance and load balancing as well as using idle resources to provide service to others.

A breakthrough in more recent peer-to-peer file systems was the advancement of lookup systems from centralized or broadcast-based search to directed search in which the number of queries is logarithmic in the number of nodes in the system. Most of the issues necessary to allow these systems to scale to internet-wide levels have been addressed: efficient lookups, file security, throughput performance, fault tolerance and reliability. However, one issue that has not been resolved in a true peer-to-peer manner is that of resource allocation.

Early file sharing systems are notorious for their problems with free-riders, that is, users who consume but do not contribute resources. These systems have not provided any incentive for users to contribute other than by appealing to their sense of altruism. Later approaches involve central authorities, as in the case of PAST, with per-publisher quotas or, as in the case of CFS, simple per-internet address quotas. While these quota systems partially address denial-of-service attacks, they are still vulnerable to a user consuming far more than their "fair" share of resources. It is clearly desirable to develop a self-regulating peer-to-peer quota system that does not rely on any centralized trust or authority, yet still ensures a rigorous standard of fairness within the system.

Table of Contents

List of Figures

Chapter 1: Introduction

In this document, we will discuss the current state of peer-to-peer file sharing systems and experimental distributed file systems and we will present a novel approach to distributed data storage based on some of the features of these systems. This novel approach, called An *Agent-based Distributed File System* (A-DFS), rests on the introduction of agents which, despite a lack of trust between each other, cooperate to perform in the best interests of their owner's file storage needs. Each agent is responsible for the management of both a user's data and a portion of their personal hard disk space. These agents can then interact and trade storage and service between each other to increase the reliability and performance characteristics of the data with which they are entrusted.

This approach is interesting because it does not rely on altruistic behavior nor on monetary compensation on the part of the user and has the capacity to ensure fairness because each agent is interested in its own user's personal storage requirements. This self-moderating approach can be extended to other peer-to-peer and agent-based systems and should prove useful in other domains in addition to peer-to-peer file systems.

1.1 Personal Computing

Since their introduction in the 1970s, personal computers (PCs) have gained a strong foothold in our way of life, especially since the inception of the Internet and its accessibility via the World Wide Web. At first, only professionals used computers, then hobbyists, and now most people cannot get by without one. Nearly every home and office has some personal computer connected to the internet.

Computer manufacturers quickly caught on to the consumer nature of personal computers. Demand for faster, cheaper, higher capacity, smaller sized systems and

components have driven production from a specialized manufacturing process to a large-scale consumer industry. As with many consumer goods, quality and reliability sometimes suffer as production and consumer costs decrease.

Consumers have become accustomed to this component failure rate because PCs are treated as a disposable commodity; as faster processors, cheaper memory, and larger disk storage becomes available, older PCs are very quickly retired, so a PC is not expected to last very long. A key selling point with PCs and components is price-quantity, with little regard for expected product life.

An example of this is exact trend is the growing popularity of 7200 RPM hard disk drives. 5400 RPM drives have a slower spin speed and thus have slightly slower seek and transfer times, but in general they have a considerably longer component life since their slower spin speed generates less heat and puts less stress on their mechanical components. This longer component life has had little effect on consumers' habits since they purchased 7200 RPM drives at nearly a 3:1 rate over 5400 RPM drives in the year 2000, opting for negligible performance gain over reliability.

Since production is limited by profitability, low capacity systems are not as profitable for the manufacturers to produce and market, so they push consumers to buy increasingly larger capacity or faster components. The manufacturing cost for an 10GB hard disk is nearly the same as an 80GB disk considering that the manufacturing processes are nearly identical, while the price-value of an 80GB disk to a consumer may be as high as eight times as much. For all these reasons, new PCs are currently sold with no less than a 30GB disk.

This problem of unreliable storage is compounded by the general unavailability of adequate backup solutions in the same price range as primary storage. Consumer backup systems tend to be less reliable than primary storage, and more importantly, they are too small to be practical given today's large disk sizes. These secondary storage systems have not kept up with the size of hard drives, and have little chance

at doing so, since consumers are much more interested in size and value rather than reliability and are not willing to pay for the costs of developing and producing quality backup systems. This consumer market has fundamentally changed the model of home computing.

The role of systems managers has shifted from taking care of servers to taking care of users and the infrastructure required by those users. More often than not, users now manage their own software and hardware. Even Unix, a very popular server-oriented operating system has evolved to compete as a single-user desktop operating system. This paradigm shift, while not necessarily efficient or beneficial in all respects, is inevitable. If at all possible, people prefer to do things (and be responsible for them) themselves, rather than relying on someone else to do them.

Examples of similar transitions from centralized systems to personalized systems include public transportation to the automobile, one telephone per household to every member of the family having their own mobile phone, and personal entertainment in the form of radio and television as opposed to going to live shows, to name a few. Exactly the same transition is happening with computing systems: centralized processing and storage is being replaced by local processing and storage, despite its many disadvantages. These disadvantages, of course, include security, reliability, and maintainability, but despite this, people prefer having their own resources.

Given this trend of increasing individual ownership of resources and other trends, predictions are that future computing systems will consist of interconnected autonomous agents, each with their own processing, storage and other resources [13] [53]. Efficient use of storage (as well as other resources) will become increasingly more important in the future as demands and raw (as compared to managed) availability grow [31] [28] [43].

1.2 Data and Its Storage

The terms "data storage" and "user data" need to be defined. We will define user data as the computer encoded information that a user generates, by any means, that cannot easily be reproduced in an automated fashion. This could include documents, photographs, spreadsheets, music, art, source code, practically anything that a user might consider important and worth keeping. This probably would not include the operating system, applications and miscellaneous temporary files. Data storage is the practice of storing this user data in a convenient and hopefully reliable manner.

In a server-based data storage system, there is a shared storage space that is under the control of an administrator. This might be the "root" account or some other user with enough privileges to perform maintenance and other administrative operations. In contrast, most personal computer users today prefer to be their own storage administrators. This would include managing disk allocation, upgrading disks, backups, and general system maintenance.

Special consideration must be given to address issues of data backups. Hard drives, being mechanical devices, are prone to failure, and are even more so in an uncontrolled desktop environment. For this reason backups are an important but often overlooked aspect of data storage.

Here are some possible data storage backup solutions:

- Get a second disk and use it to mirror files. A disadvantage is that backups must stay on-site, and multiple backups are difficult.
- Use a hardware or software RAID. Same disadvantages as above.
- Tape backups for off-site storage. Disadvantages include cost and convenience, since tape sizes do not scale well to disk sizes.
- Removable storage (Zip, Superdisk, etc). Even smaller sizes than tapes lead to

further inconvenience.

- Emergency disk recovery services, for when a disk does go bad. Obviously this is expensive and very inconvenient.

It is unfortunately true that most users today do not perform adequate data backups. The most common reasons for are cost and convenience – backing up a 40GB disk at regular intervals onto 5GB tapes is not very practical, particularly for a home user. In most cases, users back files up in a haphazard fashion onto removable media such as Zip disks or CD-Rs, if at all.

Usually one of the major advantages of a centralized storage system is that expensive and otherwise inconvenient backups are performed, since the costs are distributed among all users.

1.3 File Sharing and Replication

In addition to data storage and backup needs of users, shared file systems facilitate sharing files by the use of user groups and access control lists, in addition to external methods of sharing, meaning computer-to-computer sharing. In an isolated personal computer model only external sharing is available since each user has his own file space under his own domain. We'll discuss these in further detail in the following chapter, but essentially, there are two characteristics of external sharing as illustrated in Figure 1.3.

	Replicated	Shared
Server	FTP, HTTP	NFS, Windows Shares
Peer-to-Peer	Gnutella, Napster	xFS, CFS, PAST

Figure 1: Characteristics of external sharing systems

The first is characteristic is either "server-based" or "peer-to-peer". In a server-based system, a user who wishes to share files runs a specialized program whose

purpose is to export or make files available to other computers, and a user who wishes to have access to those files runs a client program whose purpose is to access files on a server. Peer-to-Peer is a nascent paradigm in which an application can act as both a client and a server at the same time, sharing resources between the two roles.

The second characteristic is whether the files are truly shared or replicated. In a shared system, the exact same file (down to the bits on the disk) is accessed by all parties, while in a replicating system, files are copied. Permissions are fundamentally limited by file replication systems to upload and download privileges, while shared systems are more sophisticated and include user-based access control and file locking.

1.4 Problem Statement

Our goal is to maximize reliable disk space use for individual users who do not want to be part of a strict organization. We will extend ideas from peer-to-peer replication systems and peer-to-peer storage systems by creating decision-making agents whose goal it is to maximize reliability and usability of information for their owner.

Since each agent is responsible to a single user, there is no trust among agents, and we should assume that there are "rogue" agents in the system who are actively attempting to accumulate an unfair amount of resources for themselves. To this end, a strong security model must be integral to the system.

We will design a basic agent template with the underlying mechanisms to act as an intermediary between a user and his files, wherever they may be stored. This agent should be adaptive enough to allow variances in user preference as well as adaptable to changing external conditions and usage patterns.

Our basic line of discovery stems from the following observation: suppose there are only two hosts in a peer-to-peer system and each host has some of the other's

files. These two hosts can self-regulate and ensure that neither cheats the other out of disk space by simply reciprocating malicious behavior once it is detected. We can recognize the zero-sum nature of such storage systems this way: if one host consumes too much of a resource, other hosts are hindered and should cut off access to the abusive host; his gain is their loss, and vice-versa. Taking this line of reasoning to a large-scale peer-to-peer system leads us to a somewhat paranoid model of data storage, a model that we feel may be appropriate in an internet-wide distributed storage system: each storage hosts will act as an autonomous agent solely interested in preserving the storage space of its owner. These agents may interact to exchange storage space among each other to increase performance and reliability of the data with which they are entrusted, but there will be no implicit trust between any of the agents. There should also be an expectation of "rogue" agents or agents that are actively attempting to subvert storage space from other legitimate users.

Chapter 2: Literature Review

In this chapter we will review the current state of data storage as it relates to file systems, file sharing and peer-to-peer systems. We will first briefly discuss traditional systems, including local file systems, server-based file replication and file sharing systems. We will then proceed to the bulk of related work to our topic, peer-to-peer storage systems.

2.1 Local File Systems

A file system is an integral part of all computing systems as it is the interface between the operating system and the disk drives, see Figure 2.

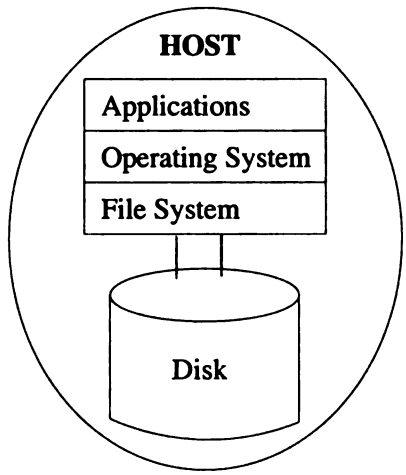


Figure 2: A local file system

Two important but relatively disjoint features of file systems are the user interface and file system implementation [57]. The user interface specifies how the user and his applications interact with the files: how files are named, organized and accessed for reading, writing and deletion from the user's perspective. Implementation refers to the methods used to optimize the allocation and deallocation of disk blocks, to keep track of which disk blocks are used by which files and support any disk quota system or security paradigm mandated by the operating system.

The most immediate type of file system is a local file system. In such file systems, files are stored on the local storage medium. This allows a computer to operate as a standalone system, without accessing remote systems for necessary operations. Local file systems are by far the most prevalent among personal computer systems; they are the norm in Windows, Macintosh, and personal Unix systems. Some examples of such local file systems include FAT and its relatives, FFS [1] [5], and ext2fs [40]. More advanced systems include ReiserFS and Linux's ext3fs which perform journalling to further improve non-catastrophic failure recovery.

2.2 Server-Based Systems

Server-based file replication systems are systems in which client programs on a remote machine explicitly request a file stored on a machine that is running a server program. A client is a program that establishes connections to servers for the purpose of sending requests and receiving their results. A server waits for client requests and processes them as they come in. Both client and server run on top of their local file systems as illustrated in Figure 3.

FTP, or File Transfer Protocol, is a very simple client-server system for copying files across networks. It consists of a standalone server program that runs on a machine with access to the local file system which contains the files to be replicated.

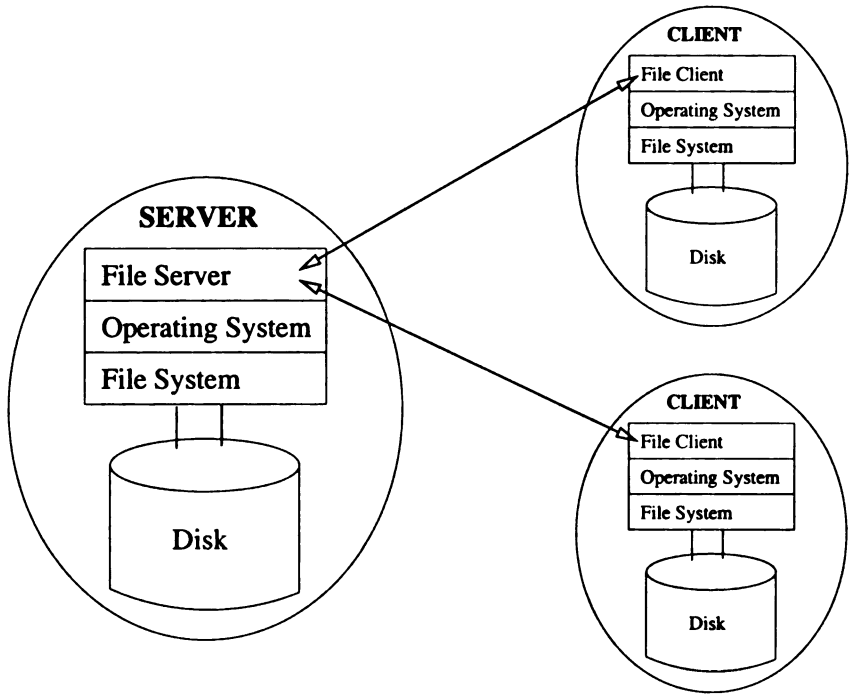


Figure 3: A server-based file replication system

The server's access to files is limited by the user running the server and the underlying local file permission scheme [45]. The client program connects explicitly to a specific server. The client user must either have authentication on the server machine or may sometimes connect with a special "anonymous" account which may have limited file access.

FTP is designed to transfer data reliably between computers on an as-needed, per-file basis, in which the user running the client must have a good idea about where the files are located on the server. FTP neither provides a way to easily search within a server for specific files nor to identify addresses of potential servers. FTP was first proposed in 1971 and is still in in wide-spread use for transferring files manually between computers.

HTTP (Hypertext Transfer Protocol) is a more sophisticated client-server system

designed as a general-purpose file transfer system in which meta-information about the data transferred is included in both client requests and server responses [24].

Although its use is not limited to hypertext transfers, the primary use of HTTP is for world-wide web hyper media information pages. In order to display web pages, web browsers such as Opera or Netscape Navigator send requests using URL (uniform resource location) identifiers which contain the location of the server and the location of the requested resource within that server. Additionally, the client may include extra information about itself or about the requested resource. The server replies with the requested resource and possibly more information.

Server-based sharing systems differ from replication systems in that files accessed by remote clients are accessed directly and are not copied over the network. The server and client provide an application-transparent mechanism for the client machine to work with the files remotely. Applications and users are not aware that the files they use are not stored locally; file-related system calls (such as open, read, write, etc) are interpreted differently by the operating system and translate into remote procedure calls. Figure 4 illustrates a generic server-based file sharing scheme.

Sun Microsystems was among the first to provide a network shared file system. In the 1985, Sun introduced NFS (Network Filesystem) to provide transparent remote access to shared files [55] [49]. Since this was the first such system, it became the de facto standard.

An NFS server generally runs on a dedicated host whose task is to store user files. These files can then be accessed by any machine that is trusted as a client. NFS is a convenient way of centralizing large scale storage for the purposes of user space management and backups. This is particularly useful in environments in which multiple users use multiple machines, for example in corporate or university settings.

The security scheme in NFS is fairly weak; NFS servers trust clients on a per-host basis, and thus a single compromised client could compromise the entire file system

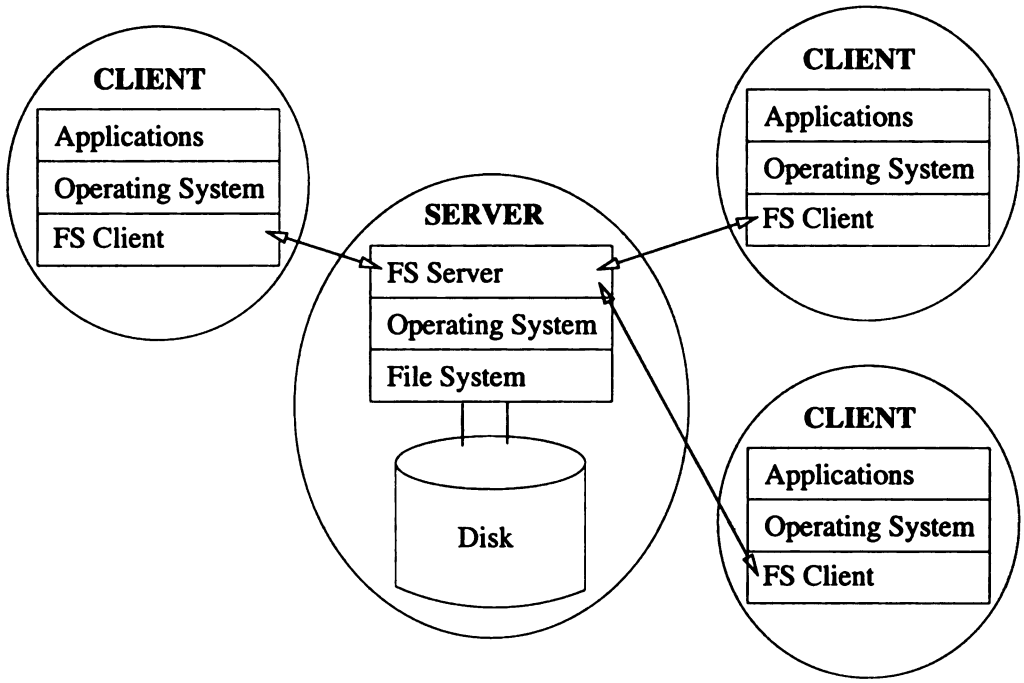


Figure 4: A server-based network file system

that it has access to. NFS is generally used on Unix clients and servers (although there are clients available for Microsoft operating systems).

AFS (Andrew File System) was designed at Carnegie Mellon University to address the shortcomings of NFS in a very large, complex, heterogeneous environment. AFS is more secure and reliable than NFS, but requires even more infrastructure to operate [50] [30].

Features of AFS that go beyond NFS include a good security model, cooperating servers, and intelligent client-side caching. An interesting difference between AFS and NFS is that AFS does not work on top of a local file system, like NFS does. An AFS "cell" takes up a physical disk device and can only by accessed by a client-server interaction, even from the same machine.

There are a number of other server-based file systems including Echo [10], Sprite

[42] [41], Coda, and Amoeba [59] that make many advances in areas such as reliability, performance, logging, clustering [52], and disconnected operation [34] [51], usually at the expense of complexity [50]. It is clear that performance in network file systems has been a very active research topic for the past several decades [8].

2.3 Peer-to-Peer File Systems

By the late 1990s, personal computers had become ubiquitous; new systems shipped with at least 10GB disk drives and most consumers had Internet access of some sort (AOL and other dial-up Internet service providers, cable modems, DSL). When MP3 technology [1] was unleashed, many consumers began MP3-encoding their music collections since they had enough storage capacity on their local file systems to store their own music. With so many MP3s stored on so many computers that were connected to the internet, the situation was ripe for a convenient way of sharing them. Naturally, the first sharing of all of these MP3s was done by adventurous individuals who ran large FTP or HTTP servers that distributed MP3 files to anonymous or semi-anonymous users. The ensuing legal repercussions to the individuals running such copyright-infringing servers caused people to rethink their distribution model and peer-to-peer file replication was born.

This type of peer-to-peer file replication relies on altruism from a portion of its users. To function, such systems need their clients to act as servers as well; this is generally accomplished by a default setting in the client configuration that forces the client to, in turn, serve any files it has downloaded. The basic design is illustrated in Figure 5

[1] mpeg layer 3 (MP3) is a very efficient audio storage format. There are programs that automatically "rip" a music compact disk into MP3s. MP3s can be played back on a computer or on a standalone MP3-player.

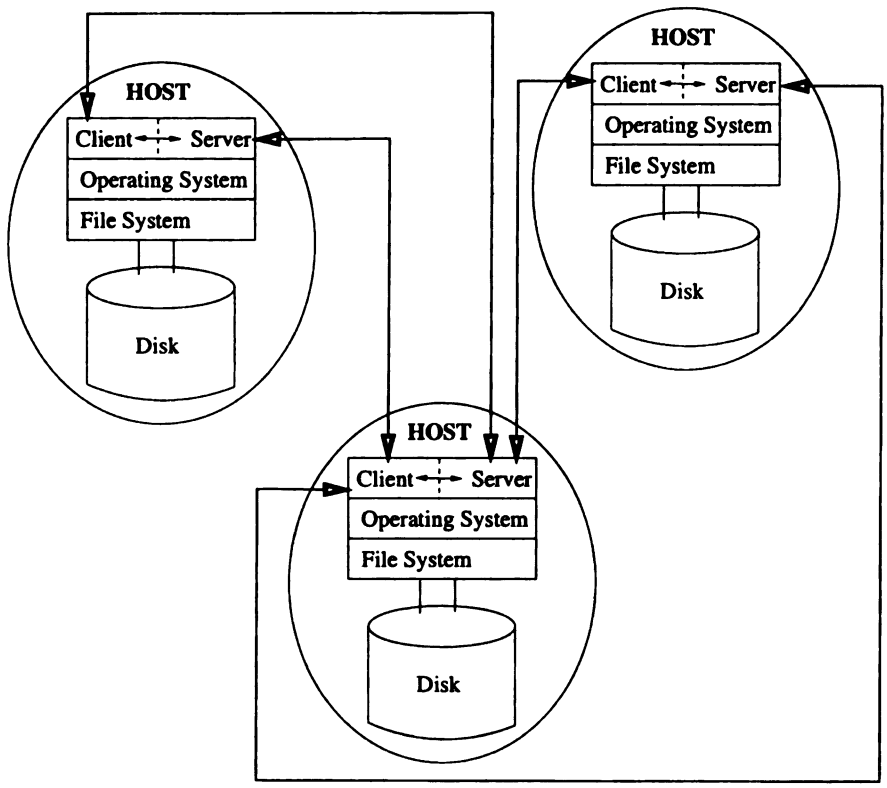


Figure 5: A peer-to-peer file replication system

2.3.1 Napster

Napster was one of the first peer-to-peer file replication systems developed. It was specifically developed with the intent of allowing users to exchange MP3 files with ease. A Napster program has both a client and a server rolled into one: it connects to other servers and downloads files while at the same time provides those files to others.

Napster uses a centralized index server to allow users to search for files by keyword. The index server responds with the addresses of servers which have copies of a given file. The server part of the Napster program periodically contacts the index server and provides it with a list of files that it is serving.

This centralized index server allows users to easily find the locations of files while still off-loading the heavy work of actually storing and transferring files to peers in

the system. This allows for a scalable, searchable system, but also leaves the system vulnerable since it would not function at all without a centralized server. This is in fact how Napster was shut down; legal suits were filed against the Napster index server operators, even though they were not distributing illegal files themselves.

2.3.2 gnutella

In order to avoid the legal issues of running anything at all in a centralized manner, gnutella [2] was developed to perform distributed searching as well as distributed file serving.

To this end, each node in a gnutella network talks to a set of its rather arbitrary "neighbors" and broadcasts search queries to its neighbors who, if they have a match, respond. If not, they pass the query on to their neighbors, and so on. File transfers themselves are still conducted in a point-to-point manner, but since there is no centralized indexing system, searching for files is less vulnerable to complete failure as was the case with Napster. On the other hand, this broadcast-based searching scheme carries a considerable amount of overhead and, in the long run, prevents gnutella from scaling to anywhere near the levels Napster does.

2.3.3 Other Peer-to-Peer File Replication Systems

A number of peer-to-peer file replication systems have been developed, each with a different emphasis, either on security, anonymity, meta-information, throughput, fairness and a number of other factors.

Some of these systems include:

- Morpheus: addresses the scalability issues of gnutella by using dynamic "supernodes" – nodes that have higher than average bandwidth act as Napster-like indexing servers, but since these servers are dynamic and may come and go as

they please, are not bound by the legal issues that Napster index servers were eventually shut down for[3].

- Ohaha: a system that combines other file sharing systems into one, including Napster and gnutella [4].

- edonkey2000: a system similar to Morpheus but has the additional feature of being able to download the same file from multiple sources for increased download speed.

- directconnect: a system designed to allow small- to mid-sized ad-hoc hubs for sharing files and chatting.

- filetopia: a system with goals of privacy and anonymity. All files in this system are encrypted.

- FreeNet: an advanced system outwardly similar to gnutella but with the property that every file inserted into the system has a set of keywords associated with it instead of a filename. Strong emphasis on privacy, security, and efficiency [14] [15].

2.4 Peer-to-Peer Virtual RAID Disk Systems

The observation that server-based storage was expensive while local disk storage as well as network bandwidth was abundant and cheap prompted the development of serverless file sharing systems. This development is analogous to the development of RAID storage systems [2] in light of the abundance cheap small disks as an alternative to a single large disk.

[2]RAID (Redundant Array of Inexpensive Disks) is a common term for several schemes of software and hardware disk apportionment that allow many small, inexpensive and possibly unreliable disks to function as one large and reliable disk. Example include striping, mirroring, and bit-wise error checking RAIDs known as RAID0, RAID1 and RAID5 respectively.

A serverless network file system distributes storage, cache and control functions across a number of peer-related workstations in order to provide server-like services to each other or to other clients. Early peer-to-peer file sharing systems that will be described in this section can be thought of as sophisticated network RAIDs since their general goal is to emulate a single efficient and reliable network file server or virtual disk as illustrated in Figure 6 [27].

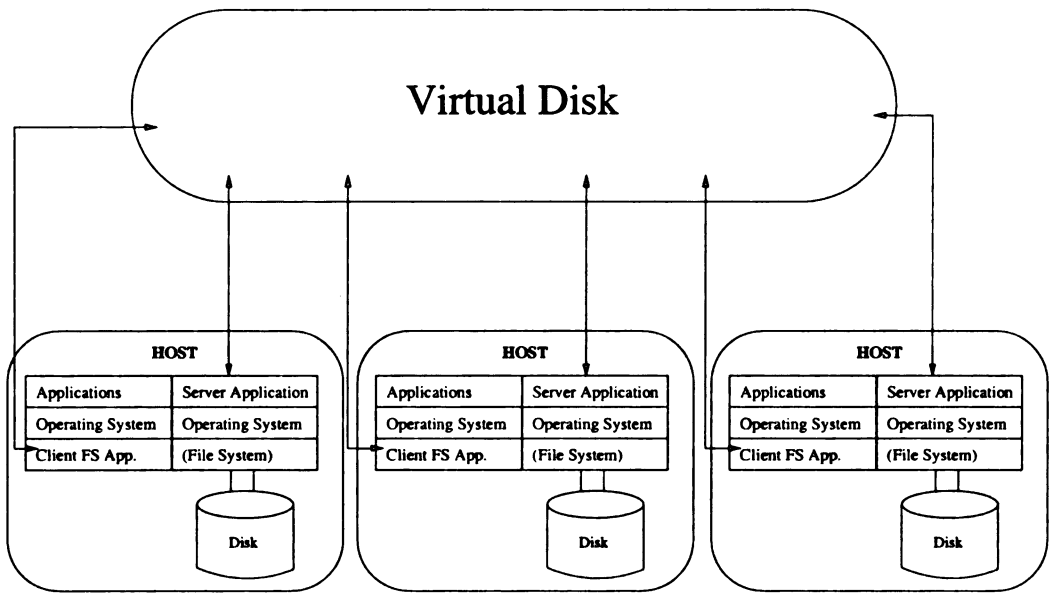


Figure 6: A peer-to-peer virtual file system

Advantages of these systems usually include efficient use of space, and performance and reliability gains over local and server-based networked file systems. These systems are intended to be used in a restricted environment requiring both fast network connections between machines and kernel-level trust between participating hosts.

2.4.1 xFS and NOW (Network of Workstation)

xFS is among the first serverless network file systems and uses many aspects of advanced network file systems such as cache consistency, cooperative caching and log structure [42] in addition to network disk striping [27].

Every participant in an xFS system runs a client, a storage server, a file manager, a logger and a log cleaner [7]. Together, these subsystems form highly efficient and reliable network RAID. Each host is responsible for a portion of a file, thus allowing for higher throughput as more hosts are added to serve file requests.

Work on xFS was incorporated into the NOW (Network Of Workstations) project. NOW was a massively parallel super computing architecture consisting of hundreds of workstations [6].

2.4.2 Jetfile

Jetfile is a light-weight serverless peer-to-peer file system that relies on peer-to-peer communication over multicast channels to perform all high cost operations such as file reads and writes [26].

Jetfile extensively uses hoarding and pre-fetching techniques. Clients hoard files that are frequently used by their own system [37] and retrieve remote files that are predicted to be used in the near future ahead of time. Each file managed by Jetfile is associated with a multicast address that is used by other clients to request the file; this makes network operations considerably more efficient than point-to-point interactions. There is a centralized versioning server that is used for every file update and is necessary due to the optimistic nature of Jetfile's scalable reliable multicast [25] paradigm.

2.4.3 Frangipani

Frangipani is a dual-layer approach in which the file system is built on top of a virtual RAID [58]. Although similar to xFS in behavior and design, Frangipani has a well-developed system recovery and reconfiguration scheme. The file system is meant to be implemented within a shared environment with a unified user and resource administration structure.

2.4.4 Farsite

Farsite is Microsoft's peer-to-peer distributed file system that employs full file replication rather than a virtual RAID. The research on Farsite is mostly focused on mathematically optimizing the file replica placement in order to optimize the efficiency and reliability of the whole system [22] [12].

Work on Farsite includes very detailed real-world statistics on PC uptime and reliability in the Microsoft corporate environment [21]. Farsite, while claiming to not trust any of the local clients, still attempts to create a single large virtual disk that is shared among all clients.

2.5 Next Generation Peer-to-Peer Systems

The next three systems which we will describe are the latest developments in peer-to-peer distributed file systems. Unlike previous systems, they separate the functions of finding or locating files (lookups) from other file system functions. Each of these file systems is built on top of a standalone peer-to-peer object-location system. They are designed to be deployed at the Internet scale, and are therefore intended to scale and perform well with a very large number of nodes.

The primary success of these systems is derived from their object-locating algorithms. In each case, the object-locating system is quite similar in design and perfor-

mance: a query from a client is usually routed to the appropriate server which has the the necessary information (in the form of either a file or a block) in $O(log(n))$ hops between other nodes in the system before reaching the result. Such object-locating systems are elegant and very efficient in identifying storage nodes.

Unfortunately, as with many peer-to-peer systems to date, these systems fail to adequately address a fundamental question of fairness, which we will discuss later.

2.5.1 CFS

CFS is a peer-to-peer read-only storage system that provides very good guarantees on efficiency and robustness in a decentralized network of storage nodes [20]. It is build on top of Chord, a peer-to-peer scalable lookup service which allows CFS to quickly locate where information is stored [54].

CFS creates a user-level virtual file system out of blocks, and distributes the blocks among multiple CFS nodes. Each node in the system has a unique hash associated with it based on the node's IP address. The key to the locations of the blocks is content-based hashing – the contents of the block are hashed into the same domain as the node hashes, and each node receives a portion of the hashed blocks. Each node maintains a list of (immediately) successive nodes and a list of shortcuts to nodes farther along the hash space. When a lookup happens (based on the hash of the block being searched for), a node can use its shortcut list to jump to what it believes to be the nearest node that has the hashed block. That node repeats the process, and so on, until the node holding the block is located. Without the shortcut list, a search would be linear to the number of nodes, but by using the shortcut list, the number of search hops is $O(log(n))$ of the number of nodes in the system.

A Unix file system is built on top of the blocks for a client, and the root directory block is encrypted and stored in the system. This root directory contains the hashes of subsequent blocks in the file system [19]. Quotas are enforced on a per-client-address

basis – in other words, each unique IP address can use only a certain amount of disk space on each server. This quota system is clearly very crude and does not take into account the amount of space each node contributes to the system.

2.5.2 Oceanstore

Oceanstore is intended as a commercial global-scale file system in which data resides on a large, distributed set of servers and can be accessed efficiently and reliably from anywhere in the world. Clients pay to store their data on these servers, and Oceanstore guarantees the accessibility of their data.

Oceanstore uses very widespread distribution to combat potential data loss. Like CFS, Oceanstore is built on top of an object-location system, similar to Chord in many ways, called Tapestry [60]. Tapestry performs hashing on disk blocks, but instead of hashing storage nodes on their IP address, it randomly assigns values to nodes and then routes queries based on a modified hamming-distance algorithm between the bits of the hash of the block in question and the current block's neighbors. This locator system also finds the proper location in $O(log(n))$ hops of the number of nodes in the system.

Oceanstore performs very sophisticated caching and pre-fetching techniques, since there is only a small semi-trusted set of servers on which the responsibility of storing the blocks falls – all other nodes in the system serve as caching nodes. These servers are financially responsible for keeping a user's information, thus solving the issue of fairness – a user "contributes" money, and receives a certain amount of storage space in return [36].

2.5.3 PAST

PAST is in practice very similar to both CFS and Oceanstore. Past is built on top of an $O(log(n))$ hop object location system called Pastry [47] which has minor differences

to Tapestry and Chord. One significant difference between PAST and the other two systems is that it is file-based, instead of block-based. This means that load-balancing works differently, since small files are balanced better while large files are distributed more poorly [48] [23].

Pastry assigns randomized hashes to both nodes and files in the system, and the routing algorithm maintains tables of "neighbors" to which it can intelligently forward a search request, again, in $O(log(n))$ hops to find the appropriate node containing the desired file. Since it is file-based and each file has a randomized hash, this architecture is closest to FreeNet [14] [15]. PAST does implement a rudimentary quota system, but it is again IP-address based with no accountability for how much a given node contributes and how much it consumes from the system.

2.6 Peer-to-Peer Trading Networks

The idea of trading resources among peers has been used in computer and other applications for quite some time. Usually, trades among people, organizations or systems are verified and supervised by a human operator thereby providing protection against many of the issues that might otherwise trouble a fully automatic system.

Recent work has been done by Cooper and Garcia-Molina at Stanford to explore the idea of trading within the context of peer-to-peer storage systems [16] [17], though there is very little emphasis on subversive behavior of peer agents in this work. Their design incorporates a system of bidding for space among peers which is effective on the assumption that the peers do not step outside the framework of the system. This type of system is a good approach when all peers are indeed interested in the best performance of the system as a whole. The system addresses poor bidding practices (always bidding or never bidding, for example) but does not address the problem of theft or dishonesty among the peers.

Experimental results show that trading groups with 10 peer trading members or fewer are the most efficient, and larger groups become progressively less reliable. This is due to the purely heuristic bidding approach taken and needs to be corrected by an on-line learning mechanism. In addition, the system as designed has no way of backing out of a trade, in the case of permanent loss of a peer or as would be necessary if a peer were to steal services from others.

There are a number of bidding policies that can be designed within this framework [18]. Deciding which will give better results depends on a number of dynamic factors including total system capacity and bidding order. In this design, the simplest type of bidding interaction is an algorithm called DEED_TRADING in which peers trade deeds, or promises of space with various reliability metrics affecting the relative sizes of the amount of space traded. All of these algorithms, however, can be thought of as fixed policy heuristics that may be fundamentally flawed with regard to unexpected situations.

2.7 Summary of Literature Review

The progression from server-based networked file systems to full peer-to-peer file systems is quite striking. Several scalable, efficient and robust peer-to-peer Internet-wide storage systems have been developed recently. Despite this, an agent-based approach which considers the problems of fairness, accountability, and reliability from a bottom-up point of view has not been previously considered. A peer-to-peer lookup-service would be a very good feature to add on top of an agent-based file storage system for the purposes of file sharing to other users, but we feel that there should be some form of leverage that a block's owner would have over blocks stored on other nodes. A zero-sum system is a logical solution as an alternative to a complicated quota-based system.

To make optimal decisions, the system needs to be able to adjust and dynamically alter its policy based on a changing environment. While a fixed trading policy might accommodate a certain set of conditions (for example, subversions or attacks), only a dynamic on-line policy has a chance of countering the whole class of problems which may arise from inappropriate agent behavior.

Chapter 3: System Description

3.1 System Overview

In this section, we give an overview of A-DFS, a peer-to-peer distributed file system using an agent-based architecture. An agent provides a self-centered utility-based view of file storage: the agents strive towards what is best for their owners. If the agents are properly constructed, they can behave in such a way as to improve their owner's data storage characteristics without being exploited by any of the other agents in the system.

The proposed system is in essence a peer-to-peer *personal* distributed file system. In other words, information will be distributed among multiple physical disks, but a user will retain sole ownership and a strong degree of control over their own files.

This personal nature of the A-DFS will be expressed by a strong encryption mechanism and a high degree of autonomy in each individual node or agent in the system. This autonomy between nodes means that while some assumptions of good faith may be taken, there is to be no trust between nodes in the system, since it is each node's goal to gain the very most for its owner in terms of disk space and reliability. An exchange of services should only be performed by agents if it is in the better interests of both to do so.

The desired goals of such a system are as follows:

- Scalability.

  The system should scale well and maintain performance and other characteristics as the number of users and computers involved increases.

- Personal Administration.

  There should be no higher authority for a user's files than that user himself. In other words, there is no "trusted" administrator who has access to all files.

- Security.

  The system should be secure, in that no unauthorized access or denial of service can take place.

- Reliability.

  The system should be failsafe and reliable in events of loss of connectivity or hardware failure.

- Reasonable Performance.

  The system should perform at acceptable speeds, and in particular, at speeds no less than a local file system.

- Mobility.

  The user should be able to access the same file space from multiple locations.

- Anonymity.

  If desired, the user should be able to retain a fair degree of anonymity with respect to his files. He should feel safe storing his file in the system.

With these goals in mind, the system has been designed as follows. We will give an overview of how the overall systems works, with following chapters focusing on the detailed components of the system, namely how information is stored, how authentication and authorization takes place, and lastly, how the agents will make their decisions for which actions to take.

Each user who chooses to participate in A-DFS allocates some of their local disk space for their use in the A-DFS. This allocated local disk space is broken up into chunks, or blocks, that are for efficiency and fairness reasons considered single indivisible units within A-DFS, similar to a disk block. Each of these blocks is "owned" by the user who created them and managed by the agent running on a workstation belonging to that user. It is quite possible that blocks initially belonging

to a user are physically located on multiple disks (and on multiple machines) that are all owned by the user; this could be the case if a user has several machines, at home and/or at the office, all participating in the A-DFS. In this case, all of the agents on these machines would have a shared pool of blocks under their collective control.

A virtual file system is built on top of all of a user's blocks, and files are written to and from the virtual file system by the client side of the agent as illustrated in Figure 7. These A-DFS blocks are the basic unit in which A-DFS agents interact with each other; they are traded one-for-one.

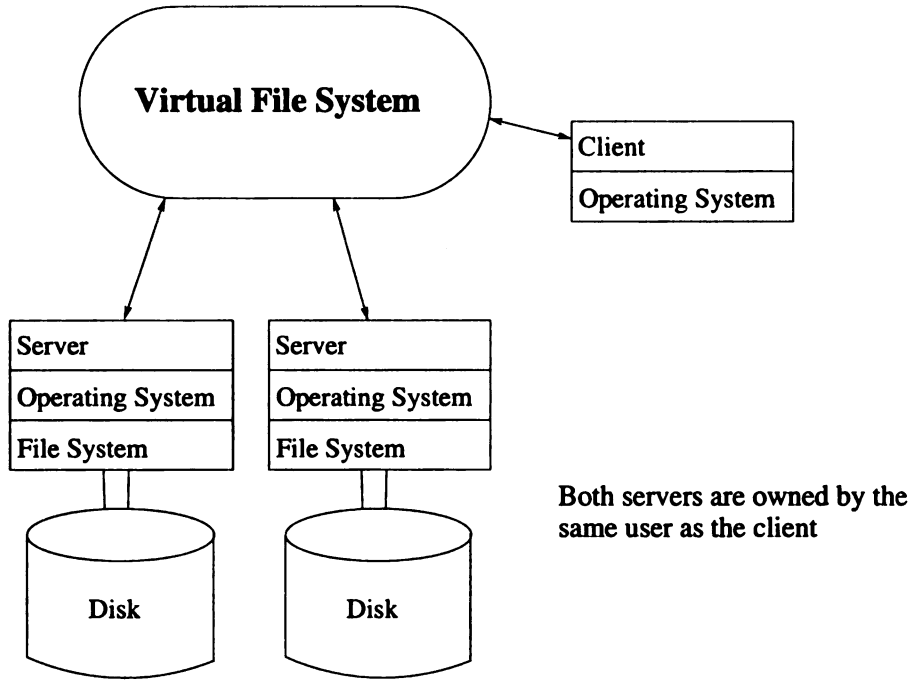


Figure 7: Overview of A-DFS

As illustrated in Figure 8, a user, or block owner, may have blocks which belong

to them on many different machines not under their immediate control, but the total number of blocks that they own is exactly the same as what it started at when all of the blocks were on their own machine. Of course, the block owner can do whatever they wish with the blocks, either use all of them to store files or split them up, perhaps mirroring some of their space.

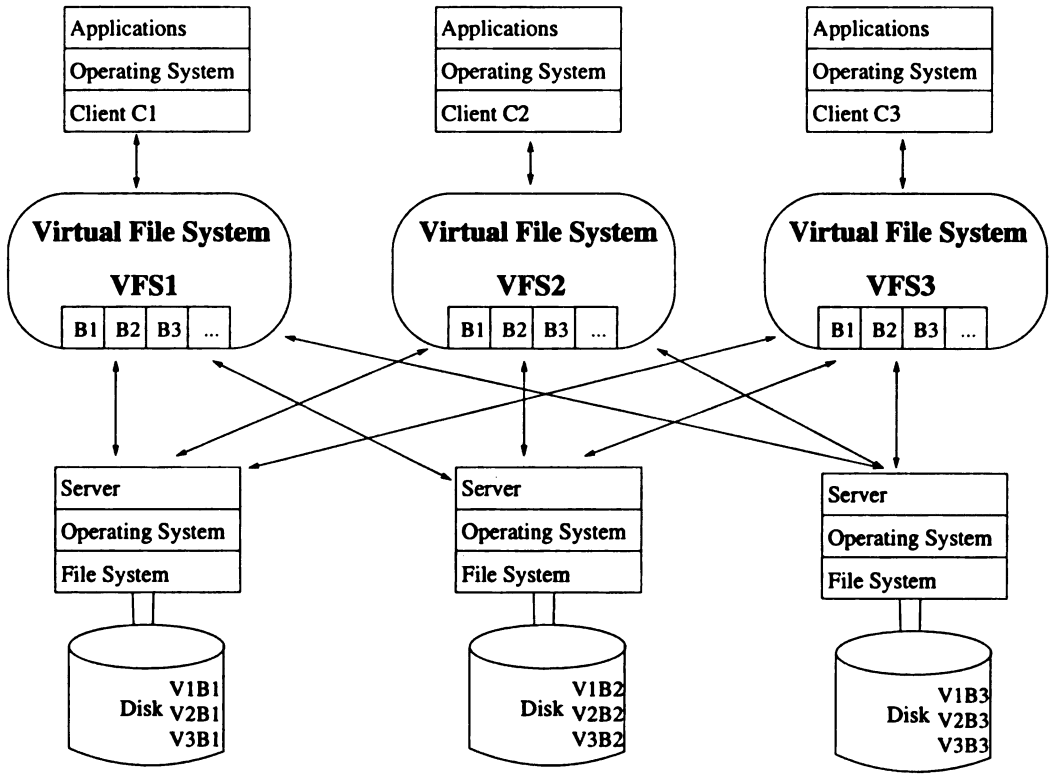


Figure 8: Sample A-DFS Interactions

Let's look at a specific example, as illustrated by Figure 9. Owner A decides to use 1GB of space, and at 1MB/block, this gives A's agent 1000 blocks to work with. Owner B decides to use 2GB of space, which gives B's agent 2000 blocks. Owner C decides to use 3GB of space, for 3000 blocks.

At their starting points, A's agent directly controls all 1000 of A's blocks, B's agent holds all 2000 of B's, and so forth. Owner A decides to use no backups, Owner B decides to back up his blocks 1:1, and Owner C decides to be really safe and back

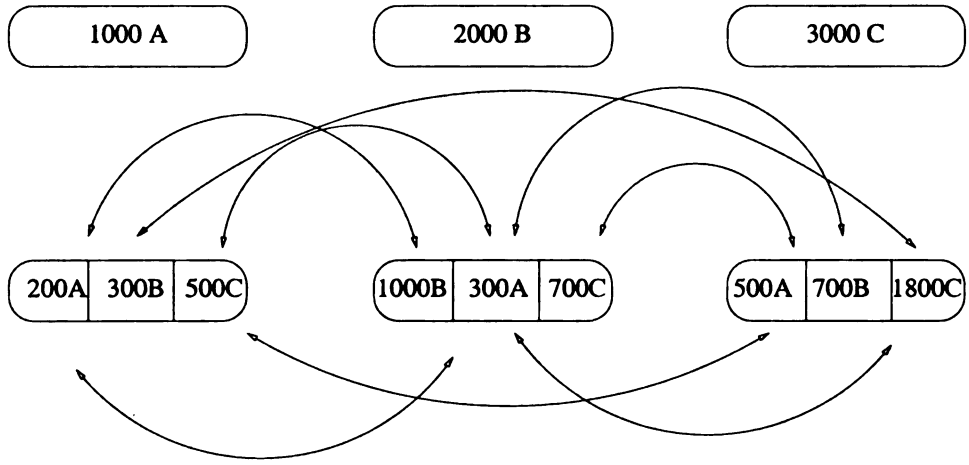


Figure 9: Disk space allocation immediately after joining A-DFS and after blocks have been traded

keep two backup copies of each block he owns. Each user then has 1000 'usable' blocks from their own point of view, but 1000, 2000, and 3000, respectively, from their system's point of view.

Based on the trading model, which will be discussed in a later section, Agent A trades some blocks with Agents B and C, B trades with A and C, and C trades with A and B. At some later time, each is holding blocks "belonging" to A, B, and C. When B needs to access some of his blocks, his agent locates the nearest appropriate block and requests that block from the remote agent charged with holding it [44].

All blocks are encrypted, and unless their owner decides to share the decryption keys, no one other than the block owner's agent may access the information in the block even though the encrypted block contents may certainly be read by anyone on the machine that the block is physically located on, as well as anyone snooping the

network during transfers.

In addition, each block has an owner identity associated with it in the form of a digital signature, so that only the legitimate owner can write to the block on a remote storage host [9]. This signature is a variation of a one-way content hash of the block in question.

The authentication and authorization scheme employed needs to be powerful enough to do the following: prevent unauthorized access to information stored within blocks, prevent unauthorized writes to blocks, and to detect "cheaters" or rogue agents who do not act in good faith in terms of preserving the blocks with which they have been entrusted. Examples of such cheating are discarding blocks or allowing them to become corrupted or overwritten in an unauthorized fashion, in other words, double-booking their disk space.

However, a number of interesting features are available for sharing access to blocks. If a user were to distribute the public key associated with a set of blocks then other users would be able to decrypt and thus read the files stored in these blocks [39]. This will allow for sharing of data without replication. Also, empty blocks can be effectively compressed by giving them a special designation; this will allow the agent entrusted with storing a block to "borrow" space from these blocks to use as a cache for other frequently used blocks that would otherwise have to be retrieved from possibly remote locations. This would establish a proximity cache as space allows. When the owner of the borrowed blocks requests allocation of that space, the cache would simply be flushed.

3.2 Self-Moderation

Central to A-DFS is the ability of each agent to make its own decisions. If properly implemented, this ensures that no agent takes advantage of another. To this end, it

is important that each agent have a view of their "world" state and some idea of the state of other agents in order to be able to make intelligent decisions about which actions to take. An agent's state consists of both internal and external information. An agent's internal information is straight-forward: its own network parameters, its bitmap of blocks and its usage patterns. External information is more difficult to gather. An agent should have an idea of how it might expect other agents to act. In other words, there should be some sort of information sharing between agents, but not in a trusted manner.

Information shared between agents should be consistent – in other words, agent A's idea of agent X's behavior should closely correspond to agent B's view of the same agent X. This problem is very related to the classic Byzantine Generals Problem [38] in which a number of generals who may have traitors in their midst must exchange information and come to identical conclusions. This can only be the case if the non-traitorous generals each have the same information.

A reasonable solution to this scenario is a single authenticated broadcast channel shared by all the agents. Such a multicast channel is lightweight relative to other network costs involved and ensures that each agent has a voice which other agents can hear – but it will be their decision to ignore or pay attention to the information other agents would propagate. Since each agent can cryptographically sign their messages, only dishonest agents can forge each others message and no honest agent's messages can be forged.

In a system of $3m + 1$ agents in which there are at most m traitorous agents, non-broadcast solutions to this problem exist, but they are recursive in the number of possibly traitorous agents in the system. Essentially, each agent takes a vote among its peers to ensure that it is indeed receiving each bit of information correctly from a specific peer, and this process must be repeated m recursive steps to ensure the top level of information is transmitted without interference.

3.3 Trading

Block trading is a zero-sum system; once in the system, blocks are neither created nor destroyed, but simply moved around between different agents.

At its most basic level, block movement can be thought of in terms of a simple trade from each agent's point of view: an agent must decide which block it should offer up for trade to its neighbor, and whether it should accept a given block in return.

There are a number of factors that affect block trading policy both adversely and positively. First and foremost, an agent must weigh the consequence of a possible trade against the resulting reliability of the blocks it is entrusted with. Such reliability factors include having multiple identical (mirrored) blocks owned by the same user. The intention of distributed identical blocks is to avoid catastrophic failure and loss of data; this should be accomplished by widely dispersing identical blocks and thus a receiving agent should not accept duplicate blocks. In addition, a storage agent should not accept blocks from a known cheater (see Security Model section below). Trading blocks with a cheater will likely result in data loss.

Overall performance of the A-DFS is another concern that should govern trading policy, and so a way to encourage agents to help their neighbors must be devised. Blocks that are often used in a given local network neighborhood should be attracted into that neighborhood. Caching achieves similar results, but congregation of frequently use blocks to their users will tune A-DFS for better performance by lowering network overhead [35].

The trading schedule (how frequently a block is traded) should be proportional to the aggregate of how much an agent wants to trade all of its blocks. For example, at the initial state of a new A-DFS member, when all mirrored blocks reside on the local disk, the aggregate "desire to trade" is very high – the agent should want to offload as many of its mirrored blocks as possible. A probabilistic model can be used in this case; the mean period between trades should increase as the agent stabilizes

its reliability and performance characteristics.

The next three chapters go into detail describing the following subsystems: storage, communication (authentication, authorization, and feedback), and internal agent design.

Chapter 4: Storage Subsystem

In this section, we will discuss how the actual storage of information takes place – how to implement the user-level and OS-interactions. While straightforward, these specifics would be necessary to implement in any such working system and should therefore be provided to aid in development of such a system.

There are two possible user interfaces for such a system. First, there should be a command-line interface that portrays remotely stored files as normal local files and provides with the same file operations as with normal files. On top of such a command-line interface, a graphical user interface might be constructed that provides an icon-based view of the same files. Essentially, what is necessary is a set of file operations that mimic the standard set of file calls, such as fopen(), fclose(), fseek(), and so forth. These function calls could either be implemented as a shell process, much like the Samba file system shell or as a separate file system which would be implemented as a kernel module. For a first implementation, taking the husk of the smbfs (Samba File System) shell and modifying it to this purpose would be considerably easier. To implement each of these calls, several operations must happen.

First, since the agents in the system deal with each other at the block level, sets of files must be mapped to set of blocks. To do this, an abstract or virtual file system must be built on top of all of an agent's allocated blocks. This can be done in a number of ways, but it would be easiest to re-use an already developed and mature filesystem such as FFS, and only point it at a set of blocks (within an otherwise larger filesystem) instead of a physical disk. FreeBSD 4-STABLE is an operating system which runs on the Intel CPU-based architecture [1] and FreeBSD has exactly this capability. This is known as a vnode file structure and allows a file to be treated as a raw device. This allows an easy (although perhaps inefficient) way of creating a prototype A-DFS system between FreeBSD nodes. A normal FFS file system is

created via a vnode device within a single large file, and this single large file is then shared by a user-level A-DFS agent.

This single large file is then broken up into the virtual A-DFS blocks and this then leads to the second issue of transferring these blocks between nodes in A-DFS. This is implemented by having a single master-agent process have this file open with write access perpetually. Then it would use a utility similar to netcat or, perhaps, its own sockets-based transfer mechanism to transfer blocks between hosts.

To increase performance by decreasing network overhead, numerous mature caching and hoarding techniques [29] [37] [11] could be implemented. Both the Oceanstore and PAST systems use locality-based caching and data migration techniques which could easily be adapted to such a system. At the moment though, our interests lie in exploring an agent-based paradigm and not seeking out optimizations on existing techniques, so our efforts in this area will be minimal in order to obtain a reasonable level of performance.

Rudimentary benchmarks for transferring 1MB-blocks between point-to-point or switched networked hosts via the netcat utility and reading and writing from them via a vnode device structure yield comparable results to transferring the same amount of information via NFS. This leads to an expected performance framework roughly in the same ballpark as typical server-based filesystem.

Chapter 5: Authentication and Authorization Subsystem

In order for A-DFS to function as if it were a local file system from the user's point of view, the user has to be certain that he is the only person who has administrative access to his blocks. In addition, a scheme must be devised to ensure that cheaters (rogue agents that do not adhere to guidelines) are caught and are not permitted to abuse other people's resources.

This first part, block-level security, is done through an application of public-key encryption, in which a block owner uses the private key to encrypt the block upon writing, then the public key is used to decrypt it [32]. In this way, the owner can share the public key with others, thereby giving them read access. If he wants to share write access, he can also share the private key for that block.

We will briefly outline RSA, a public-key cryptosystem defined by Rivest, Shamir, and Adleman [46]. In RSA, keys are quadruples (p, q, e, d), with p a large prime number, q a large prime number, and d and e large numbers with $(de - 1)$ divisible by $(p - 1)(q - 1)$. The encrypting function is $E_K(P) = P^e \bmod pq$, and decrypting function $D_K(C) = C^d \bmod pq$ where P is the plain-text and C is the encrypted message. All quantities are readily computed from classic and modern number theoretic algorithms (Euclid's algorithm for computing the greatest common divisor yields an algorithm for the encrypting and decrypting functions, and newly explored computational approaches to finding large 'probable' primes, such as the Fermat test, provide the the key quadruple generation algorithms). E_K is easily computed from the pair (pq, e) — but, as far as anyone knows, there is no easy way to compute D_K from the pair (pq, e). So, whoever generates K can publish (pq, e). Anyone can send a secret message to him; he is the only one who can read the messages.

Relating private and public keys to blocks is a straight-forward process: a group

of blocks with the same read/write permissions will have a unique public/private key pair associated with them. Read permission is granted all who have the public key, and write permission is granted to those who have the private key.

If the owner wants to make a block world readable, he can openly distribute the public key along with the block or on a common white-board-type server that facilitates file sharing on a large scale. In this way, the owner of the blocks remains anonymous while retaining write access to the blocks, since he is the only one able to produce the proper private key to perform a write operation.

Although ensuring access control is a fairly straight-forward application of public key encryption as outlined above, the detection of cheaters is a bit trickier. The basic operation is similar to determining whether another server (and its corresponding stored blocks that belong to you) are off line. The added complication is that we must assume that the agent entrusted with storing the block is attempting to cheat. A reasonable system that will (eventually, after a period of time) detect cheaters involves spot checks, or periodic challenge-response queries that can only be answered by an agent that indeed still has the block being verified [9].

We will start by analyzing a simple base case from which more complicated cases can be further understood. A and B are nodes in the system which have traded blocks with each other; node A has given one of its own blocks to node B and node B has given one of its blocks to node A. Suppose that B is a cheater that does not retain blocks that have been given to it, and that B wishes to fool A into thinking that it still has A's block. Node B might extract some information from A's block before discarding it, hoping to use this information if node A ever requests a verification.

If A retains a copy of the block it gave to B (for example, if A is using B as a backup mirror), then this checking is fairly straight forward: A can request that B perform a randomly seeded checksum on A's block which A can then verify by performing the same operation on its own copy.

However, if A does not have a local copy of the block it wishes to check, then the verification process becomes more difficult. One method of cheat detection in this situation is to store pre-computed checksums. Prior to transmitting the block to B, A would compute and store a checksum value for that block. A can then periodically request the blocks from B and verify that the checksum of the received block matches A's stored checksum. Although this solution guarantees that A will detect any dishonest behavior from B, the overhead involved in transmitting entire blocks just to ensure fair play is undesirable.

At first glance, a potential optimization might be to request for B to calculate a checksum on the block in question and transmit just the checksum result for comparison. However, this (and variations on this) does not work because upon receiving the original block, B can calculate the checksum and discard the entire block, thus fooling A into believing that the block is safely stored in subsequent check-ups.

If A is not going to store a local copy of the block, then an adequate solution is for A to pre-compute a series of challenge-response pairs and store them for future checkups on B. These challenge-response pairs consist of a random phrase symmetrically encrypted by an arbitrary portion of the block. A will subsequently send a challenge to B in the form of a block id, an encrypted phrase and an offset within that block, and B will have to respond with the proper decrypted phrase. B decrypts the challenge by using a portion of A's block as a key to the challenge phrase. If B's response does not match the original phrase, A can conclude that B does not have a legitimate copy of the block.

If A is storing any of B's blocks and A has detected that B is a cheater, then a clear solution is for A to discard B's block in retaliation (reclaiming its local disk space by a forced trade) and cease further trading activity with B. The information in the lost block can be recovered either from a local copy or requested from another host storing it.

In addition any retaliatory block dropping, A should notify other A-DFS members of B's rogue tendencies. While there are point-to-point relaying methods for doing this, the communication overhead is too high. A viable alternative is to implement a single multicast channel which all agents will monitor. Each message sent along this broadcast line needs to be attributable to the sender, otherwise a rogue agent might fabricate accusations in order to confuse the situation for others. Like all other communication between agents, each message will be cryptographically signed by the sender with their private key and can be authenticated with their readily available public key. To minimize the computational resources necessary to encrypt and decrypt these messages, key size can be kept relatively low by embedding source address information and cycling keysets on an as-needed basis.

The effect of this broadcast information on agents' behavior is a separate issue. For one, since there is no way of identifying the validity of each given accusation, it is safest to consider an accusation from A to B as detrimental to both A and B and to expect that probabilistically whichever one was indeed misbehaving would be identified by a number of other accusations as well, while the one who was truthful would only have a slightly downgraded public opinion, along with all of his fellow co-accusers or co-accusees, as the case might be. This public opinion should have a time element to it as well, since agents should be allowed to change their behavior or recover from crashes. We will refer to this public opinion as an agent's reputation rating and its calculation will be discussed in the next chapter.

Chapter 6: Agent Behavior

In this section we will discuss how to organize agent perceptions and actions in order to allow them to act in the most beneficial manner with respect to the their individual storage characteristics. In a large, complex system such as this, there are quite a number of factors to consider. Since the number of states each agent could possibly be in is very large, agents need to be able to adapt to a changing environment. This suggests that an on-line learning algorithm be a part of the agent decision process. As with any learning system, in order to minimize the complexity of the learning, we will include certain heuristic simplifications in the form of quantization of input data as well as attempt a partial mapping between certain states and utilities. This partial mapping will allow us to decide whether agents are acting reasonably by looking at their states and actions while the learning system should pin-point optimal behavior.

6.1 Agent Perceptions

Each agent is able to gather information about itself and its environment, including other agents, and base its actions on the following:

- Block map

  Each agent records the location of its blocks, whether they are stored on the local disk or have been traded away to other storage agents. Along with each block is a set of information including frequency of read and write access as well as latency and bandwidth estimates from the storage host to the owner.

- Characteristics of blocks it is hosting

  Each agent should keep track of the owners of its blocks and their network properties as well access frequencies of the blocks.

- Network properties of peers

  Each agent should be aware of its network connection properties, such bandwidth and latency to a set of its near neighbors.

- Reputation

  By monitoring the multicast channel each agent may keep track of its own reputation as well as the reputations of other agents in the system. Specifically, we will impose a heuristic scale for reputation. A zero reputation is unblemished, in that there have been no complaints lodged by or against a given agent. As a "mark" against an agent is perceived, its negative reputation score is increased by $1-z$ where z is the z-score of the agent's reputation on a normal distribution relative to the normalized distribution of all the reputation scores of all the agents in the population. This is capped, of course, at a distance of $1-\epsilon$ standard deviations for a minimum of slightly higher than zero and maximum of 2. Thus, an agent with already exactly average reputation would have their negative reputation increase by a value of 1, while an agent that is one half of a standard deviation better than the mean population reputation would only drop by a value of 0.5. This scaling accounts for the chances that an agent with a good reputation is more likely to be honest relative to an agent who already has a bad reputation.

We feel that this set of perceptions gives the agents to the necessary information to reach near-optimal behaviors, there are many other possible factors that agents might take into account as the sophistication of the system grows.

6.2 Agent Actions

The primary action of each agent is to perform mutually-beneficial trades, while secondary actions are also necessary. Agents may initiate the following actions within

our system:

- Trade with another agent

  This would involve selecting another storage host and offering a trade. The host can accept or decline based on its own perceptions.

- Cheat another agent

  Once a trade is performed, an agent might choose to cheat by throwing away a block that it has been entrusted with and using the space for itself. There are other possible ways to cheat, but this is the simplest and will be the focus of our discussion. We believe that with a properly implemented authentication and authorization scheme all other forms of cheating may be reduced to this instance.

- Verify a block

  As discussed in the authentication and authorization chapter, an agent may issue a challenge involving a randomized checksum to verify that its block is indeed stored properly by another storage host. A host may fail such a challenge by responding improperly or not responding at all, as could be in the case of power-off or network disconnection. Such a failure would lead to a broadcast "accusation" and ensuing negotiation with other agents for blocks owned by this host.

- Accuse another agent of cheating (or being unavailable)

  Whether or not a block verifies correctly, an agent may use the multicast channel to accuse another agent of cheating. Since there is no objective way of determining who is at fault, both the accuser and accused suffer a drop in reputation as a result. Implicit in the accusation is a request to other network participants for trades of blocks owned by the alleged cheater. Once these trade have been performed, the accuser can drop the alleged cheater's blocks and replace

them with their own (since the accused cheater has already failed to store the accuser's blocks). Once another agent responds with an offer to trade blocks, another decision based on their reputation must occur, otherwise a group of conspiring agents may take advantage this guaranteed trade to shuffle fictitious blocks to a target agent indefinitely.

6.3 Agent Utility

Here we decide what makes an agent "happy" or "unhappy". Clearly, this should be related to how well an agent is performing its duties to its human owner:

- Not storing mirrors of its own blocks

  One of the important goals of using a distributed file system is to avoid data loss – an agent should not leave itself vulnerable by storing multiple copies (mirrors) of the same block on its own disk.

- Correctly verifying or using a block on another storage server

  In this case, the agent is functioning properly and has verified that its data is safe.

- Finding out that it has suffered data loss

  An agent's state of well-being is affected negatively when it learns that its blocks are stored on a cheating or unreliable storage server.

- Cheating

  An agent who cheats may gain storage space, which is beneficial to its user. Another way to compute this value would be to simply calculate the total used storage space by that agent. Since all agents have a fixed amount of storage that is their own, this simply introduces a universal shift in the utility value associated with this factor.

- Network properties

  Excessive network usage, timely (or laggy) response times also factor into agent utility. Unfortunately, it is often difficult to isolate the cause of these effects. This will be the focus of future study.

6.4 Learning Algorithm

Previous attempts at multi-agent trading networks have implemented static control policies designed to optimize performance and reliability characteristics within the trading network. These systems rely on a relatively stable and permanent trading network – for example, once a trade has been decided upon, it becomes a permanent relationship. In addition, each trade is limited so that an agent is only allowed to trade blocks that they themselves own, avoiding situations in which blocks are traded multiple times. In this way, "greedy" approaches lead to near-optimal strategies.

These assumptions prove incorrect within our system since being able to back out of a bad trade is central to our model. In addition, we must make decisions in a continuously changing environment in which other agents may not only come and go, but also dynamically change their trading strategies. In this way, we know neither the exact state transition function, nor the exact reward function that any of the actions our agents take will lead to.

It would be very difficult to properly control the actions of our agents through heuristics due to these many inter-related and changing variables in the system. A solution to this problem is to employ machine learning techniques to have the agents learn for themselves in an on-line fashion. The general category of numerical on-line learning algorithms that are suited to this type of dynamic and only partially observable environment is known as reinforcement learning. The basic idea behind reinforcement learning is that an agent stumbles around in an environment observing

what inputs and rewards that it can and continually uses this information to build a progressively more accurate view of a what could be a good behavior policy. We will use a specific type of reinforcement learning called Q-learning which avoids time-delays in the reward function.

Q-learning [56][33] is an efficient learning framework in which an agent learns the expected utilities, called Q-values, for state-action pairs. The Q-space is initially set so that all actions for a given state are equally probable. Actions are chosen probabilistically, usually exponentially, and weighted towards the highest expected payoff actions for any given state. The balance between exploring the environment through actions that are perceived as less profitable versus taking the best known profitable actions is a tunable parameter known as the learning rate. As the agent performs actions, positive and negative payoffs incur and the agent learns which actions generally yield higher payoffs.

6.4.1 Traditional Q-learning

Within the framework discussed above, learning the optimal policy directly is difficult because we do not have access to training data in the form of state-action pairs, we only have a series of immediate rewards to consider.

Given this type of input, in addition to any observation that the agent makes, we can progressively approximate closer to the optimal policy using an iterative refinement process. Q-learning differs most notably from traditional function approximation techniques in that the states are only partially observable and that the agent must explore the state space rather than look ahead in its own imaginary model as it might in a Markov Decision Process.

As illustrated in Figure 10, the Q-value of state s and action a is the maximum cumulative reward that can be achieved by starting at state s, taking action a as the first action and then following the maximal known reward policy thereafter. This

Q-value for a state-action pair is defined as

$$Q(s,a) = r(s,a) + c \cdot \max_{a'} Q(s',a'),$$

where $r(s,a)$ is the immediate reward of the resulting action a taken at state s and the second term is a discounted (by factor c) expected maximum cumulative reward (including subsequent actions) over possible actions a' of the newly resulting state s' reached by applying action a at state s. In other words, the Q-value of the state-action pair (s,a) is determined by the immediate reward plus the best possible Q-value of the next state reached by taking action a as determined by fixing state s' and maximizing Q over all a''s.

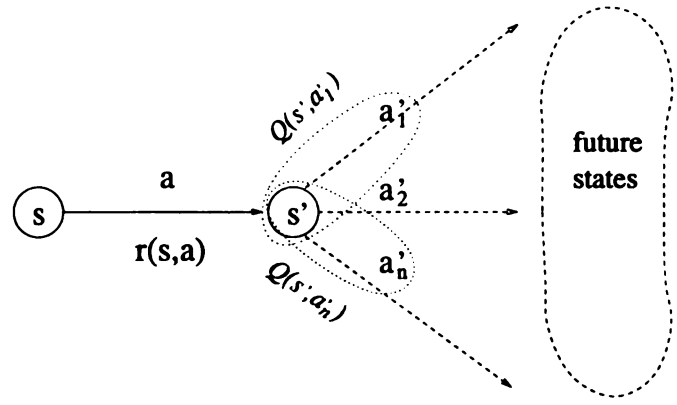


Figure 10: Q-learning update rule state diagram

Algorithmically, this leads to an iterative refinement process as follows:

- For each (s,a) initialize $Q(s,a)$ to zero
- Observe current state s
- Do forever:
  - Select an action a and execute it
  - Observe immediate reward r and new state s'

– Update the entry for $Q(s, a)$ based on the update rule:

$$Q(s, a) \leftarrow r + c \cdot \max_{a'} Q(s', a')$$

– Move to the next state: $s \leftarrow s'$

This algorithm is guaranteed to eventually converge to the optimal policy under conditions of modeling an underlying deterministic Markov Process in which the reward values are bounded and that the agent performing the simulation visits each possible state-action pair infinitely often. All of these conditions hold true in the A-DFS learning system.

6.4.2 Modified Q-learning Algorithm

The actions of trading, cheating, and accusing are not mutually exclusive, so each decision must be made on its own without regard for the others. This leads to learning a separate Q-space for each of these actions.

Within each Q-space, the states are identical; each state consists of the agent's reputation, the number of blocks in "home" position (blocks that are local to the originating host), the mean reputation of each of its blocks' storage hosts, and a measure of diversity – how many different storage hosts it is using for remote storage. Each of these values is broken up into ranges to make the learning space manageable.

At this point, we must examine the state-space considerations of the reputation of each agent. Most importantly, learning a state space in which each state encompasses the reputation of both agents would be very large, even if quantized into ranges. With 10 ranges, we would see a 100-fold increase in the state space. Because of this, the traditional Q-learning algorithm must undergo a reorganization if it is to be effectively applied to the learning problem at hand. Instead of learning state-actions, agents learn state-rule pairs in which each rule encompasses a range of possible input values

based on the reputations of the nodes involved in the trade. The rule would then be applied to make a decision based on the ratio of the reputation of the deciding agent versus the reputation of the partner agent – this single rule could then be used at a single state that would otherwise have to be split into 100 different states.

Since each action, whether it is trading, cheating, or accusing, is to be performed on an un-disclosed target (the trading or storage partner to the agent in question making the decision), the action component of each learned Q-space is actually a *rule* for deciding whether to perform the given action or not. This rule is a simple cut-off rule: "is $F(rep_1, rep_2)$ below a certain threshold?" in which F a function of the reputations of the two agents in question. If yes, perform the action, and if no, do not. These thresholds should, as above, be broken up into ranges. This leads a space of nm rule-state Q-values for each agent, where n is the number of rule groups and m is the number of states each agent may be in, again subject to the discretization used. In more traditional Q-learning terminology, each of the m rules can be thought of as a mutually exclusive action to apply in order to reach a final yes/no decision.

The reward is calculated based on whether the verification step succeeded or failed (and if failed, a deduction is made for each block stored on the bad host), whether an agent has any of its own blocks or not, and whether the agent has any extra storage space that was gained through cheating.

The Q-learning update rule for each Q-space is:

$$Q_{\mathbf{new}}(s,r) \leftarrow (1-\alpha)Q_{\mathbf{prev}}(s,r) + \alpha(P(s,r) + c \cdot \max_{r'} Q(s',r')),$$

where r is the chosen cutoff rule, s and s' are the current and new states, respectively, and $P(s,r)$ is the payoff for that iteration. Each action-rule is chosen probabilistically over a normal distribution of the expected payoff for each action rule. The decay $0 < \alpha < 1$ and future income discount $0 < c < 1$ are fixed and should be empirically

set to yield good learning behavior.

In the following chapter, we will describe an implementation of this learning algorithm in a simulation and emulation of A-DFS.

Chapter 7: System Evaluation Through Simulation and Emulation

To implement a multi-agent reinforcement learning simulation, we need to make some further assumptions about the system and the agents that inhabit it. The goals of our simulation include verification of the general concept and attainment of a better understanding of how large-scale agent interaction might take place. In order to focus on agent interaction, we make some simplifying assumptions with respect to network reliability and bandwidth. To this end, we impose a time-scale on trades: one trade per agent per iteration of the system, and thus, for the moment, do not need to figure network costs any further. We will now focus our attention on having each agent learn the payoffs associated with each action it can take in relation to its environment.

At each time-interval, or iteration, each agent randomly picks (without replacement) a candidate agent to consider a trade with. If both agents agree, then a trade occurs. Once a trade occurs, each of the agents must decide if they will cheat at this trade or not. Additionally, each agent will randomly verify a block that it has stored somewhere else. If the verification fails, the agent automatically performs an accusation action, and if the verification succeeds the agent may decide to perform an accusation anyway. These accusations hurt both the accused and accusing agent's reputations, and requests a trade for the next round between the accuser and any other agent who is holding blocks belonging to the accused. Reputation is a scaled measure of the number of complaints that have been broadcast by or towards an agent. We will have this value decay over time to account for a cheater learning not to cheat as well as for temporarily down hosts.

To make the learning problem tractable, we will discretize the ruleset into to 10 action-rules for each possible state, and break states up into 10 ranges for each perceived feature group, for a total of 1000 state-rule pairs for each agent. These

simulations were implemented in a standard FreeBSD 4-STABLE development environment.

7.1 Stability

The first simulation that was performed intended to verify some basic properties of the system. These included the existence of non-trivial equilibria and the durability of the system to rogue agents. We assumed a stable set of files and blocks, with no change in network bandwidth due to file updates.

A system was populated with 1,000 agents, with evenly randomized block contributions between 100 and 1000 blocks each, mirroring factor of 2, and a flat network topology. A certain percentage of agents, C, was programmed to always cheat, while the rest employed a simple greedy decision based on their expected utility measure. No learning was implemented. Each agent was given a chance to trade once each epoch. Since cheat-detection is not immediate, cheaters were found out at a random rate at somewhere between 1 and the number of blocks that the owner of the blocks had, assuming that the owner verified one block per epoch. This cheat detection information was not shared between agents. Each percentage C of cheaters was simulated 100 times.

From Figure 11, we can see that even for a reasonably high fraction of cheaters (10%), the system still manages to converge. It is worth noting that if an agent with N blocks and a mirroring factor of M is added to an already stabilized system, it indeed takes approximately $N * (1 - \frac{1}{M})$ epochs to reach a stable state along with the rest of the system according to the simulation, which agrees with common sense.

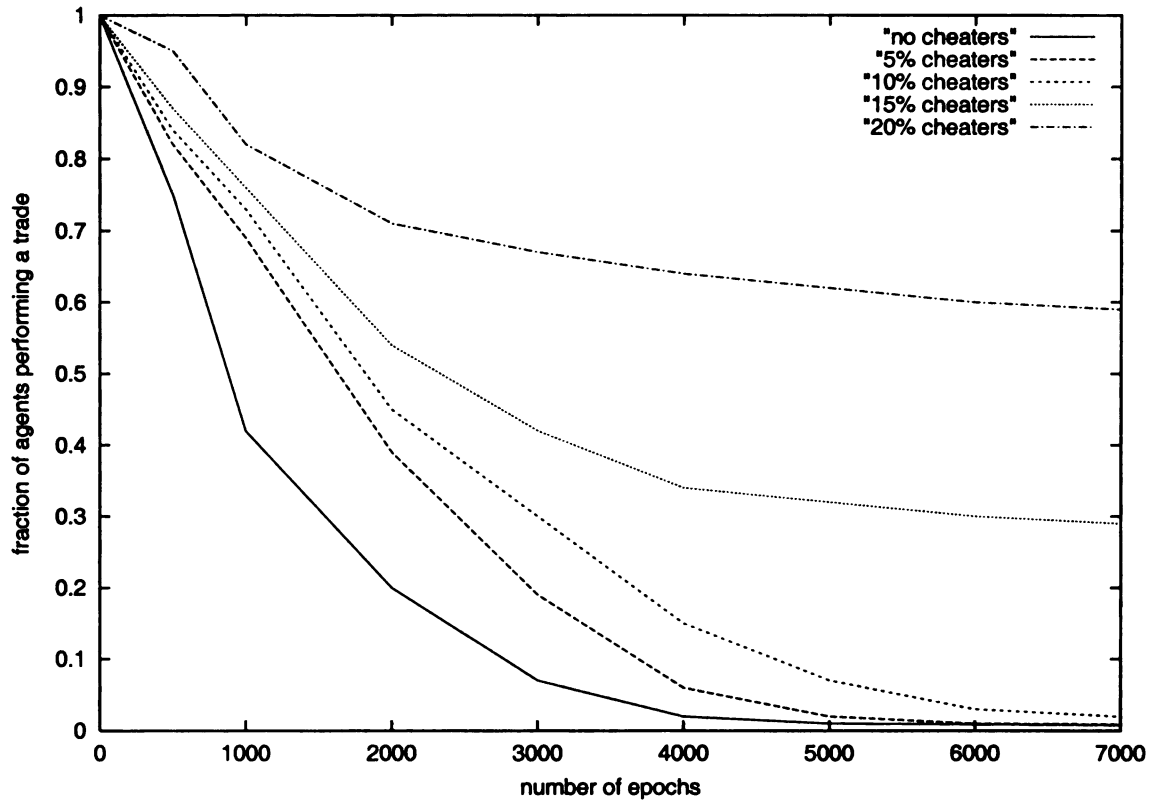


Figure 11: Convergence Results

7.2 Learning simulation

In our second, more detailed, simulation, the system was run with 100 learning agents, each contributing 100 blocks of storage. After every 1000 iterations, the system was reset to initial starting conditions, but each agent kept their memory of their learned Q-spaces. Each group of 1000 iterations will be referred to as an epoch. None of the agents had any heuristic behaviors, so all actions were learned.

Figure 12 shows that as the number of training epochs grows, the agents learn that cheating and false accusations generally do not pay off in the long run, since their reputation drops and other agents will not trade with them, thus penalizing them by keeping many of their blocks in "home" position. Since there is no immediate reward for false accusations, agents learn quickly that this behavior is undesirable. Cheating,

on the other hand, carries a short-term benefit and thus this behavior takes longer to learn not to perform. Many agents continue to perform small amounts of cheating even after they have learned their other two behaviors well. This is due to the contrast between the top two brackets of reputation values. Agents learn how to cheat just enough to stay in the top bracket. A way to combat this might be to lower the decay rate for reputation, at the cost of slower recovery from legitimate down-time.

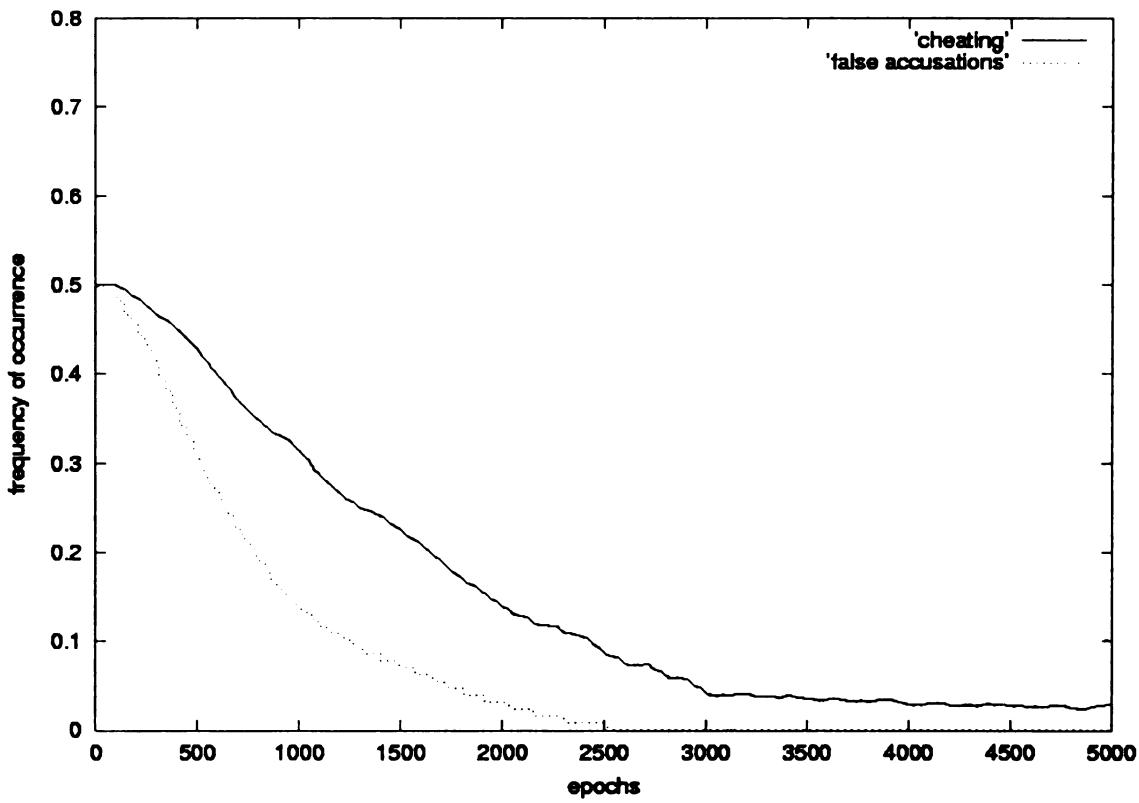


Figure 12: Frequency of trade and cheat actions over learning epochs.

The next view of our simulation data comes from the same set runs, shown in Figure 13. We have extracted the actual trading pattern of an epoch towards the end of the run, namely number 4000. It is clear from the plot that the agents are very motivated to trade at the beginning, since they are penalized for having their own blocks. After several hundred iterations, the trading rate quickly relaxes and stabilizes.

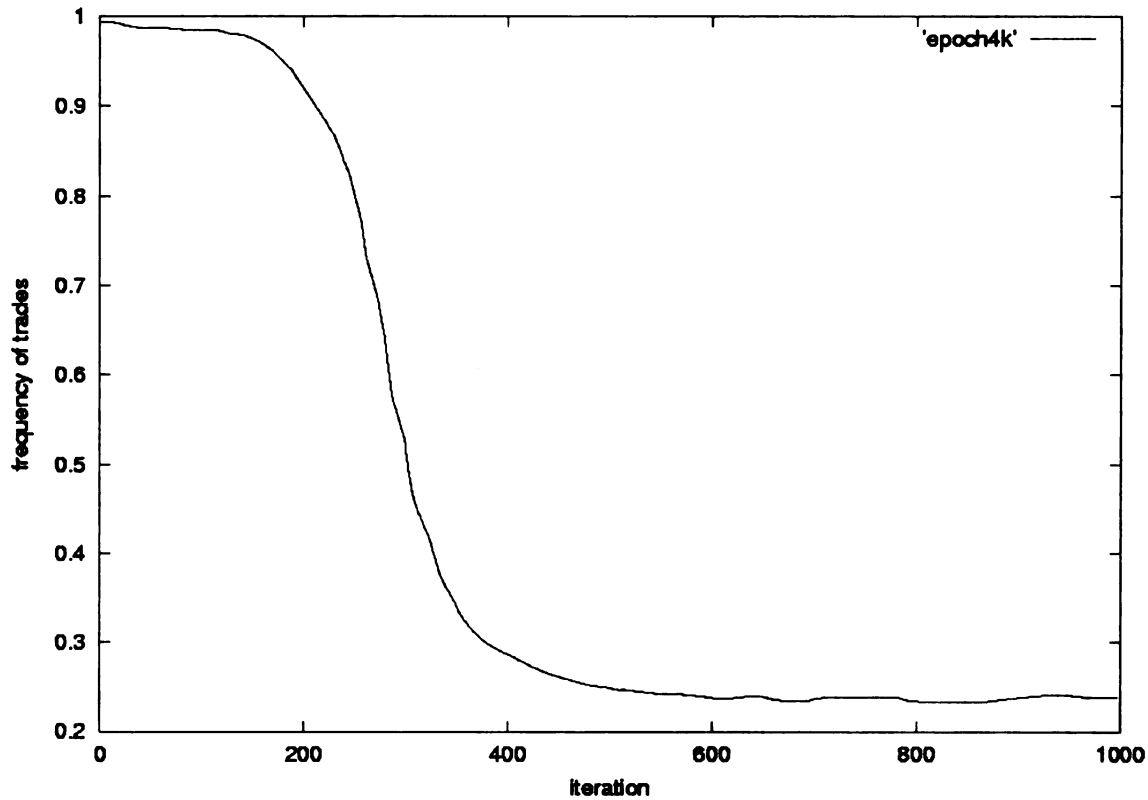


Figure 13: Number of trades at each iteration of epoch 4000

Our second set of runs was very similar to the first with one major difference. 5 of the 100 agents were given heuristics to be cheaters from the very start. This gives them a strong advantage over the other agents in the early epochs. Figure 14 shows the ratio of stolen blocks at the end of each epoch. The learning agents eventually recover and learn to easily identify the cheaters by their reputation and simply not trade with them.

7.3 Long-Term Cheating

As we noticed in Figures 12 and 14, in cases where there is a strong boundary between the top two reputation brackets we notice a bothersome behavior: agents will try to cheat "minimally". In other words, they will learn the reputation decay function in

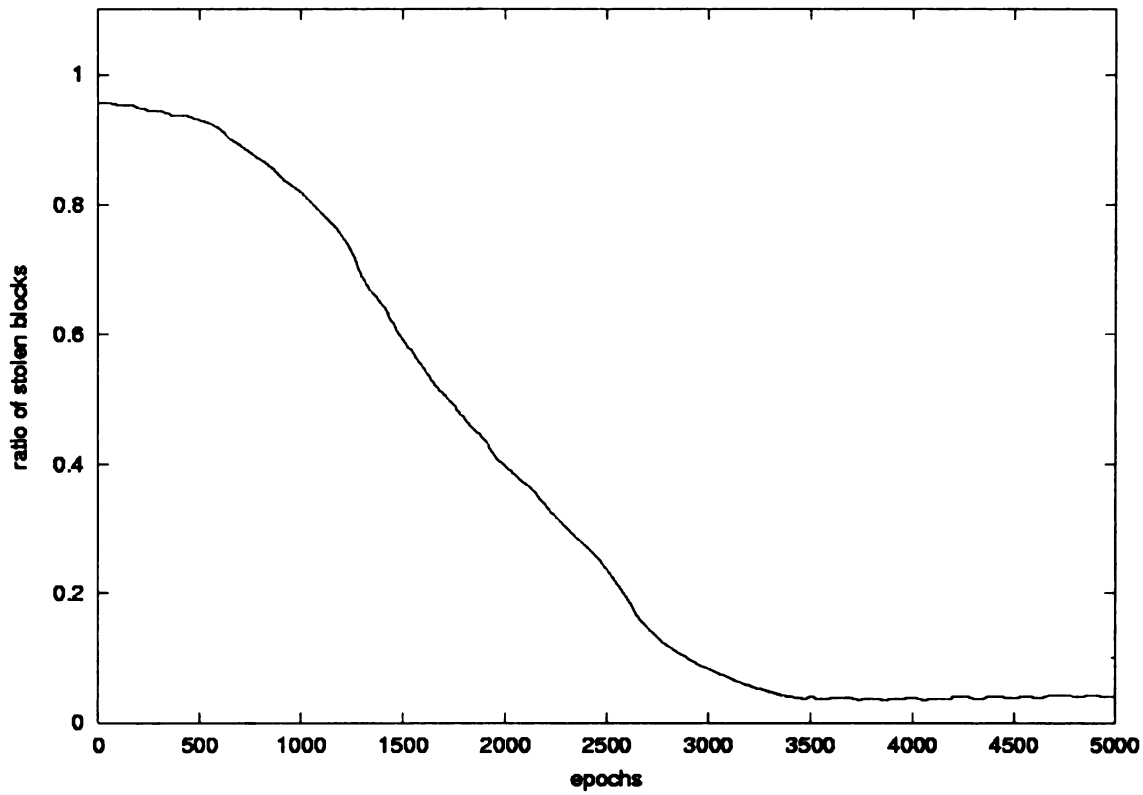


Figure 14: Ratio of stolen blocks over learning epochs with 5% non-learning cheaters.

order to stay in the best reputation bracket. This long-term cheating behavior does not go away over time, since it is the highest-paying strategy for the agents.

This problem resolves itself if we introduce a fraction of agents which are incapable of cheating, as illustrated in Figure 15. Since some agents are incapable of cheating, they create a distinct "best" bracket, and thus provide incentive for other cheating-capable agents to also attempt to be in this bracket. In a real-life setting, this is still a fairly pessimistic value for the number of fully honest agents; we would expect that a considerably larger number of agent owners will instruct their agents to not cheat, even if they could gain by it.

Unfortunately, since down-time also affects reputation, we will likely still see some minimal cheating in a non-perfect system, as may be implemented and deployed for use.

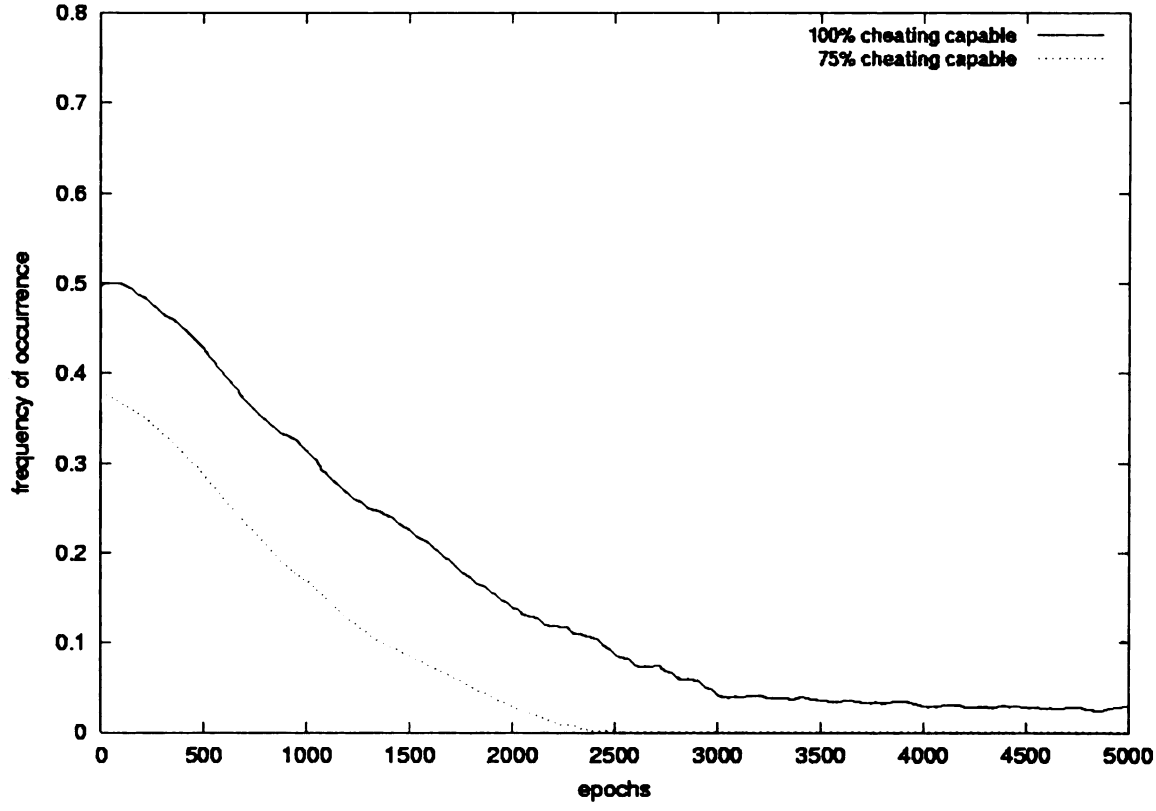


Figure 15: Frequency of cheat actions over learning epochs with 25% of the agents incapable of cheating compared to all agents being capable of cheating.

7.4 Emulated Peer-to-Peer System

As a step towards a fully functioning prototype agent-based peer-to-peer filesystem, we have implemented a multi-process emulation of our agents with the following design, also within FreeBSD 4-STABLE.

Each agent is composed of a concurrent process where each agent's actions are implemented exactly as in the simulation, along with a 'dump' function to collect agent statistics and internal variables. The required multicasts for reputation updates were implemented using a shared filesystem with unique filenames for each agent, and each agent ensuring its own file locking. Actual block storage, as in the simulation, was purely fictitious, although this functionality can be easily added via FreeBSD's vnode capability.

Statistically, the emulated and simulated systems behave identically, and as of yet we have found no emergent behavior due to the implementation.

Chapter 8: Conclusions and Discussion

We have developed a model applicable to a personal-computer environment where storage hosts are administrated by independent individuals and there is no central authority. This is considerably different and more general than either a set of peer hosts that simply copy files amongst each other or a set of trading peers that assume a well-meaning intent towards other peers in the system, such as a group of libraries sharing resources. In addition to the storage and authentication mechanics, we have successfully designed a behavioral policy that allows agents to successfully learn that neither cheating nor false accusations are beneficial in the long run.

There is strong merit to such an on-line learning agent-based approach toward self-moderating peer-to-peer systems. We have successfully simulated a Q-learning algorithm to allow the agents to adapt and perform their storage duties well. One bothersome aspect to the agent behavior is that the agents learned how to cheat, albeit slightly, without being penalized by learning the reputation decay function and mimicking it in their cutoff rules. The problem lies in the calculation of the reputation function. While the injection of fully-honest (incapable of cheating) agents creates a definitive top bracket, this does not solve the fundamental problem of the relative nature of our reputation metric. The reputation metric needs to be studied in further detail if we hope to find a solution, perhaps by adding a "repeat-offender" term to the reputation calculations. In addition, the reputation metric used was purely negative: an agent who has had many successful transactions has the same reputation score as an agent who has just recently joined the network and has had no transactions at all. This is somewhat justified in order to avoid the possibility of a small group of cheating agents conspiring to increase each other's reputations. This needs to be addressed further to give a reputation advantage to positive transactions over no transactions. A possible but costly way to avoid this type of abuse is to keep track of the number

of unique feedback sources for a given agent rather than the straight count.

As far as developing a full peer-to-peer file sharing and storage system, this self-moderation through an individual agent-based approach can be mated with current peer-to-peer lookup services such as Chord in order to provide the best of both worlds with regard to personal storage as well as file sharing and distribution.

Relevant to any network of peer-level service providers and consumers, results are more generally applicable than any previous approaches. In addition to peer-to-peer file systems, these results can be generalized to any peer-to-peer system with the following properties:

- The system is a network of peer-level service providers and consumers in which each host is both a provider and a consumer to varying degrees.
- Loss of service can be recovered from, but at considerable cost. This is certainly the case with most networked systems. There should be a backup alternative or way of re-routing services but it would be desirable to minimize these events.
- Service quality and availability can be verified, but at some cost. If this cost were sufficiently low, then there would be no estimation necessary as to the reputation and reliability of other nodes in the system since each node would know exactly the state of the system and optimal actions could be computed directly.
- Service between hosts can be broken down into host-to-host pieces. In the case of file systems, this means blocks. In the case of network traffic, it may be allowing access to or from an ISP via another party.
- Services can be exchanged equitably. In other words, each agent should be able to reciprocate service or denial of service to another agent.

Examples of such network systems might be wireless network or telephone access

across varying service providers, risk assessment of online transactions such as auction purchases, load balancing among web servers operated by different organizations, and even passenger balancing among different airline flight providers. In addition, we hope to identify other weaknesses of this model and ways that agents can be made to misbehave.

Chapter 9: Future Work

The current work was focused on demonstrating that the issues of resource abuse, theft, denial of service and fairness within peer-to-peer systems can be successfully dealt with in a relatively general manner. Further work can be done in the following three directions.

First, in the area of a specific peer-to-peer filesystem, work should be done to expand the model to take realistic network characteristics, workstation down-time, and actual use of blocks into consideration as well as merging this system with a modern information lookup service such as Chord. This will lead to a workable and useful application that might benefit a large number of users world-wide.

Secondly, work can be done to formalize our model further to analyze the types of systems that would benefit from an on-line learning algorithm based on a peer reputation-feedback metric. This metric should be analyzed and generalized from a straight count to something more expressive of an actual reputation, including both positive and negative interactions with others while avoiding disingenuous feedback meant to artificially boost or lower a reputation.

Finally, one could apply these techniques to other domains that satisfy the properties outlined in the previous chapter directly. A number of applications come to mind including packet- or connection-level load balancing on servers. Here, we will outline a specific case of this type of load balancing with regard to email services. It is well known that email systems are heavily loaded by mass-advertisements, viruses and other automated messages from unwelcome hosts, commonly known as "Spam".

Currently, there are two ways of blocking unwanted email: at the server (per-host) level and at the user (per-message) level. These two drastically different approaches have differing performance characteristics and lend themselves differently to automation. Per-message filtering can be automated via machine learning and

hand-crafted heuristics to filter messages by their headers or content, while per-host blocking is done by a human operator entering an IP or range of IPs into a local block list or possibly adding these bad sites to a central database – in either case, the entry must be done by hand.

If we could formulate this problem in terms of a peer-to-peer reputation-feedback problem, we might be able to combine both approaches to augment each other.

Consider an email send-receive transaction as an exchange that the receiver can accept or refuse: positive feedback might be collected if a message is read or filed and negative feedback might result from a message being deleted while still unread or marked as spam. This would happen at the per-message level, with a site agent collecting site-wide statistics on the originating hosts of both "good" and "bad" messages. This feedback information should be quantized and broadcast or placed on a whiteboard space in the same way as our peer-to-peer reliability information.

Each site agent, now armed with this feedback information can employ either heuristics or machine learning techniques to block at the per-host level in an automated fashion, thereby increasing email efficiency considerably since spam would get blocked sooner in the delivery chain.

Bibliography

[1] FreeBSD. `http://www.freebsd.org`.

[2] gnutella. `http://www.gnutella.org`.

[3] Morpheus. `http://www.morpheus.com`.

[4] Ohaha. `http://www.ohaha.com`.

[5] OpenBSD. `http://www.openbsd.org`.

[6] T. Anderson, D. Culler, and D. Patterson. A Case for NOW (Network of Workstations), February 1995.

[7] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating System Principles. ACM*, pages 109–126, Copper Mountain Resort, Colorado, December 1995.

[8] M.G. Baker, J.H. Hartman, M.D. Kupfer, K.W. Shirriff, and J.K. Ousterhout. Measurements of a distributed file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 198–212. Association for Computing Machinery SIGOPS, 1991.

[9] M. Bellare and P. Rogaway. The exact security of digital signatures – how to sign with RSA and Rabin. *Advances in Cryptology – Eurocrypt 1996 in Lecture Notes in Computer Science*, 1070, 1996.

[10] A.D. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart. *The Echo Distributed File System.* Digtal Systems Research Center, Palo Alto, CA, September 1993.

[11] M. Blaze. *Caching in Large-Scale Distributed File Systems.* Princeton University, January 1993.

[12] W. Bolosky, J. Douceur, D. Ely, and M. Theimer. Feasibily of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proceedings of SIG METRICS*, June 2000.

[13] F. Cheong. *Internet Agents: Spiders, Wanderers, Brokers, and Bots.* New Riders, 1996.

[14] I. Clarke. A distributed decentralised information storage and retrieval system. *Master's Thesis, University of Edinburgh*, 1999.

[15] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, July 2000.

[16] B. Cooper and H. Garcia-Molina. Creating trading networks of digital archives. *ACM/IEEE Joint Conference on Digital Libraries*, pages 353–362, 2001.

[17] B. Cooper and H. Garcia-Molina. Bidding for storage space in a peer-to-peer data preservation system. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*. IEEE, 2002.

[18] B. Cooper and H. Garcia-Molina. Peer to peer data trading to preserve information. *ACM Transactions on Information Systems*, 20(2):133–170, 2002.

[19] F. Dabek, E. Brunskill, M.F. Kaashoek, D. Karger, R. Morris, I. Stoica, and H. Balakrishnan. Building peer-to-peer systems with Chord, a distributed location service. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 71–76, May 2001.

[20] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, October 2001.

[21] J.R. Douceur and W.J. Bolosky. A large-scale study of file-system contents. In *Proceedings of SIG-METRICS '99*, May 1999.

[22] J.R. Douceur and R.P. Wattenhofer. Large-scale simulation of replica placement algorithms for a serverless distributed file system. In *Proceedings of 9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2001.

[23] P. Druschel and A. Rowstron. Past: Persistent and anonymous storage in a peer-to-peer networking environment. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 65–70, May 2001.

[24] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol - HTTP/1.1. *RFC 1094*, June 1999.

[25] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, 1997.

[26] B. Gronvall, A. Westerlund, and S. Pink. The design of a multicast-based distributed file system. In *Operating Systems Design and Implementation*, number Special Issue, pages 251–264. ACM Press, 1999.

[27] J. Hartman and J. Ousterhout. The Zebra Striped Network File System. *ACM Transactions on Computer Systems*, August 1995.

[28] A.L.G. Hayzelden and J. Bigham. Agent Technology in Communications Systems: An Overview. *Knowledge Engineering Review*, 1999.

[29] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West. Scale and performance in a distributed file system. *ACM Transaction on Computer Systems*, 6(1), February 1988.

[30] L.B. Huston and Peter Honeyman. Disconnected operation for AFS. In *Proceedings of the USENIX Symposium on Mobile and Location-Independent Computing.* USENIX, 1993.

[31] C. Iglesias, M. Garrijo, and J. Gonzalez. A survey of agent-oriented methodologies. In Jörg Müller, Munindar P. Singh, and Anand S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555, pages 317–330. Springer-Verlag: Heidelberg, Germany, 1999.

[32] Network Associates Inc. An Introduction to Cryptography. `http://www.pgpi.org`, 1999.

[33] L.P. Kaelbling, M.L. Littman, and A.P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[34] J.J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1), 1992.

[35] T.M. Kroeger and D.D.E. Long. Predicting file system actions from prior events. *USENIX Conference Proceedings*, January 1996.

[36] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (APLOS 2000)*, pages 190–201, November 2000.

[37] G.H. Kuenning and G.J. Popek. Automated Hoarding for Mobile Computers. *Operating Systems Review*, 31(5), December 1997.

[38] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3), July 1982.

[39] D. Mazières, M. Kaminsky, M.F. Kaashoek, and E. Witchel. Separating key management from file system security. *Operating Systems Review*, 33(5), December 1999.

[40] M.K. McKusick, W.N. Joy, S.J. Leffler, and R.S. Fabry. A fast file system for Unix. *ACM Transactions on Computer Systems*, 2(3), August 1984.

[41] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *IEEE/ACM Transactions on Networking*, 6(1):134–154, February 1988.

[42] J.K. Ousterhout, A.R. Cherenson, F. Douglis, M.N. Nelson, and B.B. Welch. The Sprite Network Operating System. *Computer Magazine of the Computer Group News of the IEEE Computer Group Society, ; ACM CR 8905-0314*, 21(2), 1988.

[43] G. Papadopoulos. Models and technologies for the coordination of internet agents: A survey, 2000.

[44] C. Plaxton, R. Rajaraman, and A. Richa. Accessing nearby copies of replicated objects in a distributred environment. In *Proceedings of the ACM SPAA*, pages 311–320, June 1997.

[45] J. Postel and J. Reynolds. File Transfer Protocol (FTP). *RFC 959*, October 1985.

[46] R.L. Rivest, A. Shamir, and L.M. Adelman. *A Method for obtaining digital signatures and public-key cryptosystems*, volume 21. 1978.

[47] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, November 2001.

[48] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2001.

[49] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 Usenix*, 1985.

[50] M. Satyanarayanan. Scalable, secure, and highly available distributed file access. *IEEE Computer*, May 1990.

[51] M. Satyanarayanan, J.J. Kistler, L.B. Mummert, M.R. Ebling, P. Kumar, and Q. Lu. Experience with disconnected operation in a mobile computing environment. In *Proceedings of the USENIX Symposium on Mobile and Location-Independent Computing.* USENIX, August 1993.

[52] M. Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering. In *Proceedings of the 1995 Winter USENIX Conference*, January 1995.

[53] T. Spalink, J. H. Hartman, and G. Gibson. The Effects of a Mobile Agent on File Service. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.

[54] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. *Chord: A scalable peer-to-peer lookup service for internet applications.* MIT, Cambridge, MA, March 2001.

[55] Sun Microsystems, Inc. NFS: Network file system protocol specification. *RFC 1094*, 1989.

[56] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, 1998.

[57] A.S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, 1992.

[58] C.A. Thekkath, T. Mann, and E.K. Lee. Frangipani: A Scalable Distributed File System. *Operating Systems Review*, 31(5), December 1997.

[59] R. van Renesse, H. van Staveren, and A. S. Tanenbaum. The performance of the Amoeba distributed operating system. *Software Practice and Experience*, 19(3), March 1989.

[60] B. Zhao, J. Kubiatowicz, and A. Joseph. *Tapestry: An infrastructure for fault-tolerant wide-area location and routing.* Computer Science Division, U.C. Berkeley, April 2001.