A design for a distributed file system featuring peer-to-peer caching

Johannes Lehtinen
<johannes.lehtinen@iki.fi>

Master's Thesis
March 18, 2005

Supervisor:
prof. Juha Tuominen

HELSINKI UNIVERSITY OF TECHNOLOGY Department of Computer Science and Engineering	ABSTRACT OF MASTER'S THESIS

Author	Date
Johannes Lehtinen	March 18, 2005
	Pages: 81

Title of thesis	
A design for a distributed file system featuring peer-to-peer caching	

Professorship	Professorship Code
Software Systems	2131

Supervisor
prof. Juha Tuominen

Instructor
prof. Juha Tuominen

Distributed file systems are commonly used to share data within workgroups and organizations. They enable user mobility and high availability. Most of the problems associated with developing distributed file systems are directly related to the distributed nature of the system. Challenges include concurrency control, partial failures, scalability and security. File system consistency is an important issue. Distributed file systems focus on overcoming these issues.

Increasing network bandwidth has lead into research on large scale Internet-wide distributed file systems. Such systems require good scalability. Peer-to-peer architecture has been used to harness the aggregate processing power, network bandwidth and storage capacity of the participating systems.

This thesis explores the feasibility of a hybrid approach. The idea of client level peer-to-peer caching is taken from web serving and applied to distributed file systems. The goal is to utilize the processing power and network bandwidth of participating clients for better scalability without losing the advantages of well-managed trusted server environments.

A new distributed file system design is presented in this thesis. The design makes it possible for the participating clients to share cached data with each other while relying on trusted dedicated servers for long-term persistent storage. The servers help clients locate promising candidates for peer-to-peer caching. Cryptographic signatures ensure the integrity of data received through intermediate peers. An optimistic replication strategy is applied and version vectors are used extensively to maintain file system consistency.

The design was found to fulfill the requirements set for it. The overhead caused by data signing and version vector operations was measured to evaluate the practical feasibility of the design. Data signing overhead was found to be significant. A full implementation and a realistic test environment would be required to fully evaluate the performance effect of the suggested peer-to-peer caching strategy. However, the design shows that it is possible to apply the concept of peer-to-peer caching to distributed file systems and that the subject should be explored further.

Keywords
distributed systems, distributed file systems, file system consistency, peer-to-peer systems, peer-to-peer caching

TEKNILLINEN KORKEAKOULU Tietotekniikan osasto	DIPLOMITYÖN TIIVISTELMÄ

Tekijä	Päiväys
Johannes Lehtinen	18.3.2005
	Sivumäärä 81

Työn nimi
Suunnitelma vertaisverkkoa välimuistina hyödyntävästä hajautetusta tiedostojärjestelmästä

Professuuri	Koodi
Ohjelmistojärjestelmät	2131

Työn valvoja
prof. Juha Tuominen

Työn ohjaaja
prof. Juha Tuominen

Hajautettuja tiedostojärjestelmiä käytetään tavallisesti tiedon jakamiseen työryhmien ja organisaatioiden sisällä. Ne mahdollistavat käyttäjien liikkuvuuden ja korkean käytettävyyden. Haasteita ovat mm. yhtäaikaisuus, osittaiset viat, skaalautuvuus ja tietoturva. Myös tiedostojärjestelmän looginen eheys on tärkeä asia.

Verkkoyhteyksien kehittyminen on johtanut Internetin laajuisten hajautettujen tiedostojärjestelmien kehittämiseen. Tällaiset järjestelmät vaativat hyvää skaalautuvuutta. Vertaisverkkoarkkitehtuuria on käytetty järjestelmään osallistuvien koneiden yhteisen prosessoritehon, verkkoyhteyksien ja tallennuskapasiteetin hyödyntämisessä.

Tässä työssä selvitetään, voidaanko WWW-palveluista lainattua ajatusta vertaisverkon hyödyntämisestä jaettuna välimuistina soveltaa hajautettuihin tiedostojärjestelmiin. Tavoitteena on osallistuvien koneiden yhteisen prosessoritehon ja verkkoyhteyksien hyödyntäminen uhraamatta kuitenkaan hallitun palvelinympäristön etuja.

Työssä esitellään suunnitelma uudesta hajautetusta tiedostojärjestelmästä. Esitelty ratkaisu antaa asiakaskoneille mahdollisuuden jakaa välimuistissa olevia tietoja toistensa kanssa. Tietojen pysyvässä tallennuksessa turvaudutaan kuitenkin luotettuihin palvelimiin. Palvelimet auttavat asiakaskoneita löytämään lupaavia ehdokkaita yhteistyökumppaneiksi. Vertaiskoneilta vastaanotettujen tietojen eheys varmistetaan käyttäen digitaalisia allekirjoituksia.

Esitelty suunnitelma täyttää sille asetetut vaatimukset. Allekirjoitusten ja versiovektoreiden käsittelyn aiheuttama ylimääräinen kuorma mitattiin kokeellisesti suunnitelman käyttökelpoisuuden varmistamiseksi. Allekirjoitusten huomattiin aiheuttavan merkittävää lisäkuormaa. Suunnitellun välimuistikäytännön tehokkuuden luotettava arviointi vaatisi täydellisen toteutuksen ja realistisen testiympäristön. Suunnitelma osoittaa kuitenkin, että vertaisverkkovälimuisti on sovellettavissa hajautettuihin tiedostojärjestelmiin ja että asiaa kannattaa tutkia tarkemmin.

Avainsanat
hajautetut järjestelmät, hajautetut tiedostojärjestelmät, tiedostojärjestelmän eheys, vertaisverkkojärjestelmät, vertaisverkkovälimuisti

Preface

This thesis has been my intense hobby for the past year or so. I decided to pick a subject I would like to learn to know better and to work on it on my own time. I chose distributed file systems and I have to admit I have learned a lot. Most of all, I have learned to appreciate the amount of research done on the subject.

I would like to thank my supervisor, professor Juha Tuominen, for the feedback regarding this thesis and for the general advice on structure and style of scientific publications. However, he has had a much more profound effect on my studies. Almost 10 years ago I was welcomed to his research group as an undergraduate member. I have since left the group but assisting on research projects was a great introduction to the academic world. It made me curious about academic research and that curiosity has motivated me to finish my studies during all these years I have focused on other duties.

Antti Kantee did an excellent job commenting and proofreading this thesis. His critical comments dragged me down to earth and helped me to realize important areas I had not (and have not) covered. I am sure some errors remain but there would be many more were it not for him. On the other hand, the fact that he took this thesis seriously at all restored some of my confidence on whether this work made any sense technically.

I am most grateful to my parents for encouraging me to seek higher education and for supporting my computer and software hobby early on. When I did not yet have a computer, I was allowed to use an old typewriter to write down those untested VIC-20 program listings. My grandfather has also supported my technical hobbies in many ways. Later, the free and open source software community provided me the tools to continue and advance my hobby which became also my profession. I am looking forward to contribute something back to the community.

My beloved wife, Ansku, has provided me more support on this thesis than could be reasonably expected from anyone. She allowed me to work on this thesis for extended periods even while she must have felt compelled to advance her own thesis. Fortunately, we have now completed both theses. She also managed to calm me down when I had lost all hope of ever completing this work. Our children, Elias and Elsa, were always there helping me to realize how unimportant this thesis is compared to things that really matter.

Halikko, March 18, 2005

Johannes Lehtinen

Contents

Abbreviations

AFS Andrew file system

API Application Programming Interface

BSD Berkeley Software Distribution, a Berkeley Unix distribution

CFS Cooperative File System

CIFS Common Internet File System

CPU Central Processing Unit, processor

DNS Domain Name System

DSA Digital Signature Algorithm

DSS Digital Signature Standard

FTP File Transfer Protocol

GID Group Identifier

IDE Integrated Drive Electronics, an interface for storage devices

IO Input/Output

IP Internet Protocol

LBFS Low-Bandwidth File System

LRU Least Recently Used

MD5 Message Digest 5, a message digest algorithm

MTA Mail Transfer Agent

NFS Network File System

OPT Optimal Replacement Rule

RAID Redundant Array of Independent Disks, also known as Redundant Array of Inexpensive Disks

RFC Request For Comments, an Internet standards-related document series by The Internet Engineering Task Force

RPC Remote Procedure Call

SATA Serial Advanced Technology Attachment, a serial bus for connecting IDE storage devices

SCSI Small Computer System Interface, an interface for storage devices

SFS Self-certifying File System

SFSRO SFS read-only file system

SHA Secure Hash Algorithm

SPKI Simple Public Key Infrastructure

SSL Secure Sockets Layer

TCP Transmission Control Protocol

TLS Transport Layer Security

UCLA University of California at Los Angeles

UDP User Datagram Protocol

UID User Identifier

URI Uniform Resource Identifier

UUID Universally Unique Identifier

WAN Wide-Area Network

WLAN Wireless Local Area Network

WWW World Wide Web

Chapter 1

Introduction

Distributed file systems are commonly used to share data within workgroups and organizations. They also provide a conventional way to utilize the reliable storage provided by high performance storage servers and make it possible for the users to transparently access their personal files at different workstations.

The predominant architecture for the deployed systems is the traditional client-server architecture. This approach works well when clients and servers share the same reliable network infrastructure. However, increasing network bandwidth has made it plausible to use a file system over the Internet, enabling large scale wide-area file systems.

Large scale file systems require good scalability. Also the limited reliability and varying network conditions of the Internet must be taken into account. To some extent these issues can be addressed by placing multiple replicated servers in different parts of the network and using less strict consistency semantics. However, implementing and maintaining such a system requires more resources.

Increasing storage capacity and processing power of desktop computers has led into recent research exploring the possibilities of using the aggregate storage capacity and the aggregate network bandwidth of numerous cooperating computers. A fully decentralized peer-to-peer file system is inherently more scalable than the client-server scheme. The peer-to-peer file sharing applications, such as BitTorrent [13], have already demonstrated the potential of decentralized data transfers. However, peer-to-peer systems also involve new kind of security concerns and management issues. It is impossible to centrally control the system.

The goal of this thesis is to find out if it would be possible to develop a hybrid large scale file system design providing the reliability and manageability associated with the dedicated server environments while simultaneously harnessing the processing power and the network bandwidth of the participating desktop computers.

1.1 Research problem

Several designs and implementations of distributed file systems suitable for large scale usage exists. For example, Coda [48] supports replication and disconnected operation, Ficus [26] is a peer-to-peer file system suitable for distributed or mo-

bile workgroups and Ivy [38] uses aggregate storage capacity of the participating computers in a peer-to-peer fashion. Of these three file systems Coda provides centrally managed and controlled highly reliable storage and Ivy is a fully decentralized approach. Ficus uses peer-to-peer architecture to propagate updates but does not have shared distributed storage.

Large geographically distributed projects would benefit from a reliable and a controllable way to share large amounts of data. A distributed file system controlled by a trusted entity can provide such a shared data storage. The control aspect is important because users must be able to trust that the file system processes and stores their data reliably. Additionally, users must be able to trust the integrity of the data they receive from the file system.

Decentralized "anonymous" peer-to-peer storage, like that provided by Ivy, is problematic because no single entity can fully control it. It is possible to enforce confidentiality and integrity of data by technical means but availability is harder to guarantee [27]. A high enough replication factor can be picked to be fairly certain that no data is lost even if several peers fail at once. However, it is difficult to estimate the possible effect of exceptional events such as targeted attacks or quickly spreading computer viruses.

The problem is how could the unused resources of the participating desktop computers, or clients, be used to improve the overall performance of the system if trusted dedicated servers are providing the persistent storage. In fact, the clients are already using their resources to cache data locally. This helps their own performance and, as a side effect, also improves overall performance by reducing load on the servers.

In web serving, client level peer-to-peer caching has been suggested for handling flash crowds (sudden traffic surges due to high demand content) [53]. This idea could possibly be applied to distributed file systems as well. Typically, clients are already caching data for themselves. Peer-to-peer caching extends the cache usage by letting the clients also serve cached data to each other. The main objective is to explore the feasibility of this idea: client level peer-to-peer caching in distributed file systems.

1.2 Scope

This thesis explores the theoretical background of distributed file systems together with the most important practical issues and the techniques to address them. Some rationale is also sought for the existence and use of distributed file systems. Because most of the issues are directly related to the distributed nature of the system also the main characteristics and challenges of distributed systems in general are shortly described. One of the most important and challenging topics, file system consistency, is discussed in more detail.

After establishing the background, a new file system design is developed. The main purpose of the design is to see how a reliable and manageable dedicated server environment could be combined with client level peer-to-peer caching to improve the overall performance of the system. To ensure that the design is realistic a use case is presented and some more general requirements are set for the design. Finally, the design is evaluated based on the requirements. Implementing the design is beyond the scope of this thesis.

Security is an important aspect in highly distributed systems. Establishing

trust and privacy in peer-to-peer applications is a major research topic of its own. In this thesis the security issues are briefly introduced and some basic solutions are given to make sure that the design does not conflict with standard security requirements. However, more elegant solutions are left mostly for further research and development work.

1.3 Structure of this thesis

This thesis has four main parts with different objectives.

Introduction. The current chapter, Chapter 1, introduced the research problem and defined the scope of this thesis.

Background. Chapters 2 and 3 introduce the theoretical background related to the problem at hand as well as related work solving specific parts of the problem.

Design. Chapter 4 presents an envisioned use case and defines requirements for the file system design. With the guidance and tools provided by earlier chapters, Chapter 5 describes the file system design.

Results. In Chapter 6 the design is evaluated and the results are analyzed. Finally, a conclusion is derived and further related work is briefly discussed in Chapter 7.

Chapter 2

Distributed file systems

A file system is an essential service used by most applications. Applications generate and use information. Typically the input is read from a file and the output saved in a file for long-term storage and for further processing by other applications. In fact, the operating instructions for applications themselves are usually stored in files. [52]

The level of sophistication required from the file system depends on the operating environment. Design issues for a general-purpose file system include the *naming structure* for organizing the files, the *application programming interface* (API) for application developers, the *mapping* between the files and the physical media and maintaining *integrity* of stored data over undesired events such as hardware and software failures. In a multi-processing environment the file system must also implement *concurrency control* and in a multi-user environment *security* becomes an important issue. [47]

Distributed file systems face the same basic design issues as local file systems used in multi-user timesharing environments but now the processes and the users are dispersed in a network of autonomous computers [47]. The network imposes limits on the inter-node communication bandwidth, produces additional latency and limits the availability of nodes. In a distributed environment the main focus turns to overcoming these limitations. An additional problem is to determine the *file location* in a network [47].

Most distributed file system designs utilize an existing local file system for conventional storage and concentrate on providing the distributed functionality on top of it in a layered approach. For example, Network File System (NFS) version 4 is mainly a network protocol layer for accessing an underlying local file system without unduly favoring one file system or operating system over another [50].

Many of the characteristics of distributed file systems are due to their distributed nature and they share many common design challenges with other distributed systems. The next section characterizes distributed systems in general.

2.1 Distributed systems

In a distributed system, components located at networked computers communicate and coordinate their actions only by passing messages [15]. The Internet is an example of a large and complicated distributed system; it is actually composed of smaller networks which are distributed systems in their own right [15]. Spatially a distributed system may span multiple continents, like the Internet, or it may be enclosed in a device, like an in-car network.

The motivation for constructing distributed systems is to share different kind of networked resources ranging from hardware components to data [15]. Sharing makes it possible for a system to be easily extensible and scalable. Availability of redundant resources leads to better fault tolerance.

2.1.1 Characteristics

All of the vastly different distributed systems have some common characteristics that the system design must account for. The following consequences of the definition are listed by Coulouris et al. [15]

Concurrency. The components of a distributed system execute their programs concurrently [15]. The components may also perform operations slower or faster depending on their hardware. This makes the system extensible and flexible. Available resources can be increased by adding more components or by replacing existing components with better performing ones. However, some resources can not be used concurrently by the components. For example, a simple network printer can only receive and print one document at a time. Such resources need to be identified and components that share such resources must coordinate access.

Lack of a global clock. The components of a distributed system do not have a single global notion of the correct time [15]. Each component has its own local clock. While the clocks can be synchronized by passing messages the accuracy is limited by variations in the message transmission time [15]. The lack of a global clock makes it more difficult for the components to coordinate their actions. The components must rely on messaging to know the state of other components and the global state of the system.

Independent failures. Distributed systems are made of networked computers. Each component can fail like any computer system. However, the system as whole has new kind of failure scenarios. Network faults may split the system into isolated segments and computers may fail bringing part of the system down. Furthermore, these kinds of errors may be difficult to detect. For example, a program may have difficulty in detecting whether the network has failed or has become unusually slow [15]. It is also difficult to know whether just the network connection to a computer failed (leaving the computer running its software in isolation) or the computer itself failed (a system crash). Each component of the system can fail independently, leaving the others still running [15].

2.1.2 Challenges

Some of the challenges in constructing distributed systems were already mentioned. The inherent *concurrency* means that all shared resources must either ensure proper operation even when simultaneous requests are being processed or access to them must be serialized. The latter approach limits throughput and therefore it is generally better to allow concurrency when possible [15].

Another topic already mentioned is *failure handling*. The fact that makes it challenging is the partial nature of failures: some components fail while others continue to function [15]. The first step to handling failures is to detect them. For example, acknowledgements should be used to verify that an operation completed successfully and integrity of data transfers should be verified. Even when a particular failure mode can not be readily detected by other components they must be designed to allow for it. The components should also be designed so that they tolerate failures of other components and recover from them. Some failures can also be masked, for example by retransmitting a lost message [15].

The flexibility and extensibility of distributed systems often leads to *heterogeneity*. For example, the Internet is a heterogeneous collection of computers and networks [15]. In a distributed system of that scale it would be very awkward to require everyone to use exactly same kind of hardware and software. In fact, such a strict monoculture would be a security threat because a single fault in the implementation could bring the whole system down. The heterogeneity requires that common standards — such as Internet protocols — need to be adopted for communication between the components [15].

It takes time and effort to create and establish an official standard. It does not make sense to standardize all interfaces of every distributed system because some interfaces are very particular to the specific design. However, for a distributed system to be extensible its key interfaces must be documented. The *openness* of a system determines how easily it can be extended and reimplemented [15]. Open systems have published key interfaces and their design is documented to the extent making it possible for an independent developer to implement a component that interacts with the existing system.

Distributed systems are typically motivated by resource sharing. Many of the resources, be they hardware or data, are very valuable to their users and therefore their *security* is an important issue [15]. The system must be designed so that the ease of extensibility and openness can not be used against the system. Unauthorized access to resources must be prevented and confidentiality, integrity and availability of data must be ensured. Especially a system deployed in a very open network like the Internet must be designed to be robust against intentional abuse.

Scalability is important for distributed systems designed to be extended. A system is described as scalable if it will remain effective when there is a significant increase in the number of resources and the number of users [15]. A scalable system should be able to support n users using at most $O(n)$ physical resources [15] and there should not be predetermined constants or fixed size data structures that would prevent the system from scaling up. Possible performance bottlenecks should be avoided by using decentralized algorithms [15].

Distributed systems have some extra complexity caused by the separation of components: the partial nature of failures, concurrency and resource sharing, communication between the components and the redundant resources possibly

available in the system. Ideally these issues should be hidden from the user. *Transparency* means the concealment of these issues from the user and the application programmer [15]. Some issues can not be transparently handled such as repeated communication failures preventing the operation requested by the user.

2.2 Rationale for distributing a file system

Distributed systems are more complicated than centralized ones because they need inter-node communication and coordination. Limited bandwidth, latency and ever changing network conditions of the Internet makes it challenging to develop efficient large scale systems. On top of that one must not forget security considerations in a network full of rogue nodes. Why go through all this trouble for creating a wide-area distributed file system when a local hard disk can store the information just fine? This section explains the motivation.

2.2.1 Mobility

In the office environment distributed file systems have been used to make it easy for the user to move from one computer to another and yet have the same data set, such as the home directory, available in both. The information is physically stored in a file server and made accessible from any workstation. This setup is especially useful in environments where users do not have dedicated workstations such as in student computing facilities. Distributed file system also allows for flexible customization of computing environment as all the data and preferences move with the user.

The advances in wireless data communications are making Internet access an ubiquitous resource. The idea of *nomadic computing* with access to remote data [2] is reality for anyone carrying a mobile phone. Users may leave the traditional office environment behind and perform computing tasks in whatever location provides the best resources or the environment for the task at hand.

Wireless local area networks (WLANs) make it feasible to access distributed file systems using a portable device. The user can access the data stored at the file server while being free to move from one location to other. The distributed file system can provide a generic interface to large amounts of data even though the portable device itself may have very limited storage capacity. Research is also being done to enable data sharing in ad hoc networks [8].

2.2.2 Data sharing

Distributing a file system is one of the most general and straightforward ways to share large amounts of data among users scattered in a network of computers. When transferring files as e-mail attachments or by FTP it soon becomes difficult to track what copy is the latest or authoritative version of a file. Distributed access to a single file system provides users an up-to-date view to shared data. From the application point of view a distributed file system hooked to an operating system I/O API is transparent so applications do not need to be specially designed or rewritten to make use of it.

However, a distributed file system alone is not always sufficient when several users are simultaneously using and updating the same files. Conflicting updates lead to data corruption and lost changes. Therefore a special version control system is often needed when a group of people actively works on the same set of files.

2.2.3 High availability

Data stored locally in one computer can be easily lost or corrupted due to a hardware or software failure. Hard disk drives themselves are mechanical devices and have a typical manufacturer defined service life of 3–5 years [28]. Periodical backups help restore most of the data should there be a failure, but they won't prevent the latest modifications from being lost. The data will also be unavailable before it has been restored from the backup.

Redundant hardware such as a *redundant array of independent disks* (RAID) can be used to raise the availability of a computer system. A redundant system is more prone to component failures than a non-redundant one because it has more components that can fail. However, since the failure of a single redundant component does not render the system inoperable the redundant system has higher availability. [28]

But computer systems designed for high availability cost more than commodity systems. It makes sense to have a highly reliable file server serving files to several less reliable client systems. A failed client system based on commodity hardware and software can be quickly replaced by a new one while all the data remains continuously available at the server. This can be achieved by *distributed access*.

A single computer is always subject to undesired changes in its operating environment, such as a fire or a power outage. It is also possible to make the file service redundant on server level. The file server can be replicated by introducing another identical server that can take over should the primary server fail. This requires replication of the file system between the two servers, i.e. the use of *distributed storage*.

Finally, a threat not obvious but important from a historical perspective is authorities controlling information to their liking. Development of moveable type printing by *Gutenberg* once helped to break the religious monopoly by enabling independent dissemination of written information. Today electronic publishing is putting the Gutenberg inheritance at risk by placing a document on few servers worldwide. The owners of these servers can be coerced into removing or modifying the information. To prevent this scenario *Anderson* suggests *The Eternity Service* — a kind of distributed file system — which would distribute published information over numerous independent servers round the world. [4]

2.2.4 Scalability

Scalability issues of a fast local network can also be addressed by replicating the file server as described above. The replicated servers can share the load thus making the file service more scalable.

In the Internet the network is typically the bottleneck. Even if the endpoints have a fast local interface to the network a slow or congested link somewhere in-between usually limits the effective transfer rate.

Peer-to-peer file sharing applications try to overcome network limitations by locating a copy or copies of the data closer (network wise) to the receiving node and downloading them instead. This does not only make the transfer faster for the downloaders but also offloads traffic from the nodes serving the original data. [13]

A generalized approach would be to implement a wide-area distributed file system that could maximize the effective transfer rate by distributing the storage over the network and transferring data from nodes closest to the receiving node.

2.3 Evolution

2.3.1 The beginnings

In the beginning remote files were accessed by user-initiated file transfers. The first version of the later widely used *File Transfer Protocol* (FTP) was completed in early 1970s. While this approach made it possible to share data it was impractical especially regarding the use of frequently changing remote files. A major step in the development of distributed file systems was the concept of *network transparency*: being able to access remote files similarly to local files. Transparent access made it straightforward for applications to access remote storage. [47]

Concurrency control, flexibility of naming structure, access control and caching were issues addressed by numerous experimental file systems developed during the decade from 1975 to 1985. *Locus* emphasized *location transparency* and explored the use of *optimistic replication* to support partitioned operation. The possible inconsistencies were detected using *version vectors*. An alternative approach to preserving consistency, *weighted voting*, was proposed by Violet. A new rationale for distributed file systems was given by the introduction of diskless workstations. [47, 26]

2.3.2 Maturing

Arguably the two most long-lived successful distributed file systems are the *Network File System* [12, 50] (NFS) and the *Andrew file system*[1] [29, 49] (AFS). Both have evolved during their existence from mid-1980s to present day but are still mostly faithful to their original characteristics and have a large user base.

NFS, introduced in 1985, was developed by Sun Microsystems. It is a straightforward stateless[2] file access protocol that approximates Unix consistency semantics and supports client caching [47]. The strength of NFS is its *simplicity* as well as independence of the transport protocols, operating systems and underlying file systems [47]. It has become the de facto file access protocol of the Unix world and several implementations are available from different vendors. An important reason for the popularity of NFS is the fact that it is an *open standard*. The latest version of the protocol is NFS version 4 defined by RFC3010 [50].

[1] Andrew file system was named after the Andrew computing network of Carnegie-Mellon University which in turn was named after Andrew Carnegie and Andrew Mellon, the benefactors of the university.

[2] NFS version 4 is a state-full protocol due to the integration of file locking. [50]

AFS was pioneered at Carnegie Mellon University starting from 1983 and was commercialized by Transarc Corporation which was later acquired by IBM. Major AFS design goal was *scalability* which was sought by strictly separating the trusted server environment from untrusted clients, by server replication and by client caching [49]. At present well-known implementations include IBM AFS, OpenAFS[3] and a client implementation Arla[4] implemented at Royal Institute of Technology, Stockholm. AFS scales well in campus-wide networks [49] and it is being used in many networks with massive amounts of users and clients.

Later distributed file systems seeking for new features have, for now, been of more experimental quality and have been adopted mostly by small communities closely associated with the original developers.

2.3.3 Later development

AFS was fairly scalable regarding the number of clients but as installations got larger and mobile clients were introduced the availability suffered. *Coda* [48], also developed in Carnegie-Mellon University since 1987, was chartered to address the *availability* limitations of AFS. It provides both server replication (also known as first-class replication) and *disconnected operation* (known as second-class replication). Coda follows the client/server model of AFS but is usable in the presence of network partitions. [31]

Ficus [26], a file system contemporary to Coda, is a fully *decentralized* file system with optimistic replication. It provides *one-copy availability*; a file can be read and updated as long as any replica is available. Updates are propagated among servers using a peer-to-peer oriented scheme. Ficus uses NFS for the client interface and therefore does not support second-class replication. The design emphasizes the *stackable layers* architecture and focuses on providing a large scale file system for geographically scattered installations that are frequently partitioned.

The very beginning of 1990s saw the birth of the *World Wide Web* (WWW) or just "the web". Publishing information Internet-wide and linking to external information became easier than ever. The web focuses on the publishing aspect and does not solve the issues regarding large scale read/write sharing of data. However, the read-only distributed file systems and distributed web caching are in many ways closely related.

Another decentralized file system, *xFS* [5] from mid-1990s, explored the possibilities of a *serverless* file system that distributes storage, cache and control over cooperating trusted workstations in a fast local network. *Frangipani* [55] was intended for same kind of trusted environments but it took a different approach by using a distributed virtual disk, Petal [33], as basis and built a file system on top of that.

Later, in the beginning of 21st century, *Farsite* [1] took the concept behind xFS further by recognizing the fact that it is not safe to trust workstations even if they were centrally administered. Farsite utilizes *byzantine fault tolerance* to implement a file system provided by cooperating workstations in an incompletely trusted environment.

[3] OpenAFS is an open sourced branch of the IBM code base. Web site is available at `http://www.openafs.org/`.

[4] `http://www.stacken.kth.se/projekt/arla/`

Also the research for large scale Internet-wide file systems continued. *JetFile* [25] from late 1990s explored the use of multicast messaging for better wide-area scalability while *OceanStore* [43] introduced an architecture for a massively scalable maintenance-free file system. While components of OceanStore — such as the Tapestry location and routing infrastructure — have already been implemented a complete prototype is still under development.

An interesting approach to archiving data was suggested in the form of Archival Intermemory [24]. It is strongly related to the concept of Eternity Service, an attempt to study the technical and architectural issues of such a system. By participating in a system for only a finite time a node can ensure archival preservation of its data. The design is based on the assumption of continuous growth of the memory capacity.

In 1999 another killer application emerged. Napster led the wave of *file sharing applications* by going from nowhere to presenting as much as 30% of Internet traffic in just three months [40]. File sharing strives to provide better availability and scalability (regarding the number of users accessing the same file) than the web by applying peer-to-peer semantics to online publishing. The file sharing applications are typically user-initiated and concentrate on read-only data like the web. Regardless, many of the techniques used to achieve scalability are applicable to large scale distributed file systems as well.

The *Self-certifying File System* [36, 35] (SFS) introduced *self-certifying pathnames* as a means to provide strong security in an Internet-wide file system without demanding a specific global scale key management scheme. With the focus on *security* SFS provides a highly secure distributed but non-replicated file system. The *SFS read-only file system* [23, 35] (SFSRO) builds on the SFS framework to provide a fast and secure replicated read-only file system.

The *Cooperative File System* [17] (CFS) was inspired by file sharing. It layers an SFSRO-based read-only file system on top of a *distributed hash*, DHash, with a provably efficient lookup algorithm, Chord [54]. The file sharing also raised the issue of anonymity. An anonymous storage service, *The Free Haven Project* [20], was inspired by anonymous publication services, like Freenet, to provide distributed anonymous storage.

The development of wireless communication has made Internet access ubiquitous. However, distributed file systems are rarely used over low-bandwidth wide-area networks as the performance would be unacceptable. The *Low-Bandwidth File System* [37] (LBFS) is a file system which exploits inter-file similarities to reduce the amount of data to be transferred. It is based on NFS version 3 and provides traditional file system semantics.

Finally, *Ivy* [38] is a multi-user read/write peer-to-peer file system. It also uses DHash for distributed block storage but is totally log-structured. Each participant maintains a log for file operations and others decide which logs to trust. Data is found by consulting all relevant logs. Ivy supports operation in a partitioned network but works best in a fully connected network.

2.4 File naming and location

Files are typically organized in a hierarchic file directory to make it easy for a user to find and manage information. In multi-user file systems it is also important to be able to refer to a file with a unique name when communicating

with other users of the system. Distributed file systems are not an exception. They should integrate transparently with the conventional file name space used by the user.

2.4.1 Location transparency

Early implementations simply prefixed the remote file name with the name of the site storing the file [47]. For example, to copy a local file to the remote site "`unix2`" using *The Newcastle Connection* [11] the user would use the following command and file path.

```
cp a /../unix2/user/brian/a
```

The first "`/`" denotes the conventional Unix root directory, "`..`" extends the directory hierarchy by introducing a new level to specify a remote site, "`unix2`" specifies the remote site and "`/user/brian/a`" is the target location at the remote site. User could also switch the current working directory to a remote directory and access the files in the directory as usual. This was implemented by extending the system calls that receive pathnames or file descriptors as parameters. [11]

The benefits of the approach is that the user can access any remote site by using the standard Unix file system path mechanism (subject to access control restrictions). On the downside the method places the burden of remembering site names on the user and prevents storage from being transparently moved from one location to another. Nor does it allow for a remote directory to appear as a descendant of the conventional Unix root directory.

Location transparency is a better approach. It means that the name of a file is devoid of location information. User does not need to know if a file is remote or not and where it is stored. However, the distributed file system implementation needs to be able to determine the servers or peers that can provide the file and receive updates on it. The implementation needs a file location mechanism. [47]

2.4.2 Volumes

Ideally, users should be able to organize the files however they wish regardless of their physical location. In practice storage management is often made easier by physically grouping files with logically related names. Such a singly-rooted, self-contained, connected set of files and directories is sometimes called a *volume*. Name space can then be structured as a hierarchy of volumes. [26]

Volumes can be used to separate file hierarchies with different purposes or with different administration. This gives administration autonomy for the volumes they manage and provides a practical unit for managing storage. Storage can be allocated per volume, access and replication can be controlled on volume level and more frequent backups can be made of volumes containing more important data.

Unix has a concept of *mounting* file systems (volumes). First a *root* file system is mounted at "`/`" then other file systems are successively mounted at specified nodes or mount points as subtrees. This makes it possible for separate file systems to appear as one tree-structured file directory like illustrated by figure 2.1. Many distributed file systems utilize the mounting mechanism. For

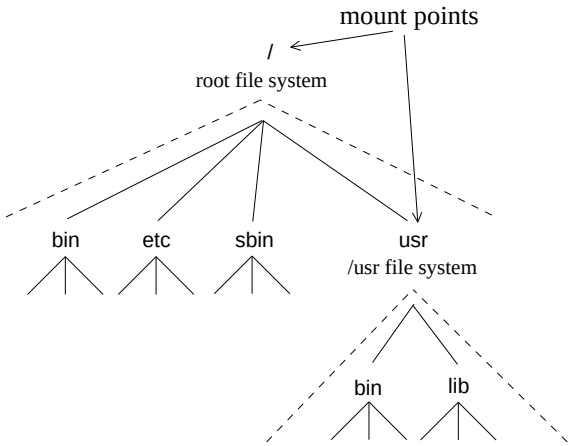

Figure 2.1: A hierarchy of mounted file systems

example, an NFS server exports specified subtrees of its own file hierarchy. Clients then mount these remote subtrees according to a locally specified mount table. [26]

In distributed systems the use of locally specified mount points is problematic. There are no guarantees that a specific volume would be mounted at the same mount point on two different computers. Should the volume storing a file be mounted at different mount point the file will also have a different file path. This makes it harder for users to refer to a specific file while communicating with each other or when moving from one computer to another within the system. It also makes it more difficult to configure file paths into applications.

Another problem is that the user can only access files located in mounted volumes. If there is a volume that was not specified in a mount table, the user is not able to access the files on it because mount table modifications typically require a privileged account.

In small scale these problems can be solved by negotiating a common mount table among all the nodes of the system (like in Locus) or by maintaining a shared volume database (like in Cellular AFS). However, neither of the approaches work in large scale environments spanning numerous administrative domains. [26]

2.4.3 Graft points

Graft points and *grafting* as used by *Ficus* are similar to mount points and mounting, respectively. However, graft points are not specified in a separate table but are special files stored in an ordinary volume. They designate points where other volumes should be mounted and contain the necessary volume identification information and a list of available volume replicas and the hosts providing them. Graft points themselves can be copied, renamed and replicated just like any other file. Figure 2.2 depicts a file hierarchy constructed by grafting. [26]

Grafting makes the name space more unified. Two computers mounting the

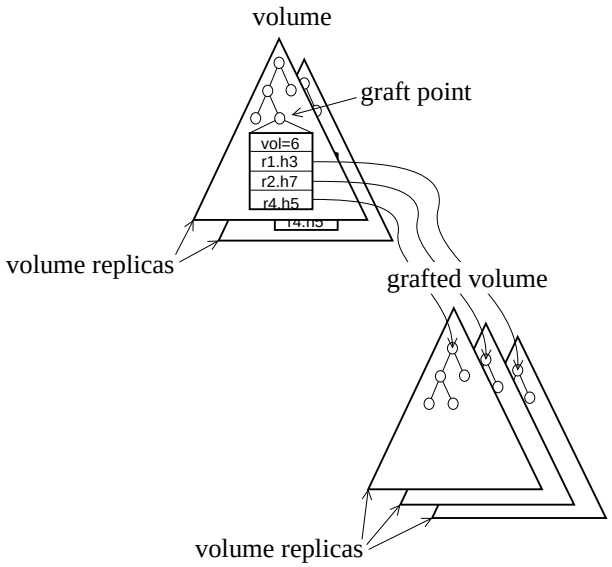

Figure 2.2: A graft point with both parent and child volumes replicated [26]

same volume have identical file directory structure beneath the volume root including the other volumes grafted into it. From the administrative point of view graft points make things easier because they naturally fall within the administrative domain of the volume containing them, yet they are accessible to anyone having access to the volume. On the other hand, the administration of the grafted volume retains full control over it because the replication factor and access control characteristics of the grafted volume do not depend on the grafting point which acts merely as a pointer.

Because a global scale distributed system would have vast amount of volumes and the complete file directory would therefore be massive (or even infinite if cyclic grafting is allowed) it does not make sense to construct the complete file directory at start up. Ficus constructs the file directory on demand as a file path is expanded and new graft points are hit [26]. This is called *autografting*.

Conceptually grafting is similar to hyperlinking to external resources from within a document. An important difference is how URIs as hyperlinks are mostly human-readable and easy to communicate to other people as references to information. Graft points, on the other hand, include more numeric information and do not have an URI-like standard human-friendly encoding because files are usually referred to by their pathname. However, grafting alone does not guarantee that a full pathname would uniquely identify the same file on two computers of the system. They also need to mount the same highest level volume at the same point in the conventional file directory.

In practice the use of pathnames as unique references in an autografted system would still require that there exists a system-wide agreement on a highest level volume that is mounted to a specific mount point and that has all other volumes as its descendants. This would effectively provide the complete system-wide namespace needed for unique names and name transparency. However, it

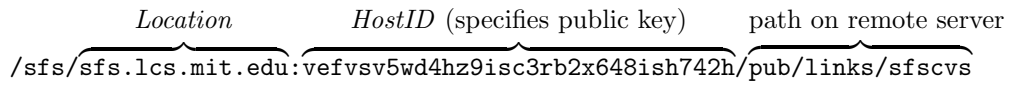

Figure 2.3: A self-certifying pathname [35]

would be a strictly hierarchical system where those administering higher level volumes would have the power to render entire branches of name space invisible by removing graft points pointing to them. In a global scale fair management of such a name space would be difficult as has been demonstrated by name space grabbing and the domain fights associated with the hierarchical domain name system.

2.4.4 Self-certifying pathnames

Security is an essential requirement for a large scale distributed system spanning several administrative domains. Being able to locate and access a file by using an obtained reference is not in and of itself a sufficient guarantee for file integrity. Someone may have changed the file without authorization, a file replica may have been compromised at the storage site, the file may have been corrupted due to hardware error or someone may have been wiretapping and altering the data while it was transferred.

Public key cryptography is a customary solution for securing the communication path. A client uses the public key of the server to authenticate a secure channel to the server and the server grants access based on the public key of the client or other credentials exchanged over the secure channel. However, for public key cryptography to be applicable the public keys must be first exchanged in a way that the recipient can trust the received public key. This is an issue of key management.

Key management at the scale of the Internet is not trivial. There are several alternatives which are appropriate for different situations. It is unlikely that any one key management scheme would be always applicable in a global scale [35]. *SFS* [36, 35] bypasses the obstacle by not requiring key management at the file system implementation level. It integrates public keys into file names to form *self-certifying pathnames*. Key management is then an issue of whatever method the user uses for selecting file names, therefore allowing multiple key management schemes to coexist. In fact, SFS itself provides a key management infrastructure to embed various key management mechanisms such as the SSL public key infrastructure or SPKI.

A self-certifying pathname includes a server name and a one-way hash of the public key associated with the server. Figure 2.3 shows a pathname split into components. When contacting the server, a client can verify that the public key presented by the server matches the hash embedded into the pathname. If the pathname has been obtained from a trusted source it can be used as a certificate which certifies the public key of the server. Symbolic links are used to provide more convenient file paths. A trusted entity can function as a certification authority by providing symbolic links to servers. [35]

2.4.5 Dynamic peer lookup

A problem common to mounting, grafting and self-certifying pathnames is that the location information must be maintained manually. In case of mounting the location information needs to be maintained locally in mount tables. Grafting makes the location more transparent to most users by making it possible to share the graft point information. However, someone still needs to maintain the graft points and the others need to trust the maintainer. In this regard SFS is similar to grafting as location transparency is sought through shorthand symbolic links maintained by a trusted party.

The domain name system (DNS) already provides a level of indirection and location transparency. It is true that a specific DNS name (and to a lesser extent, even a specific IP address) can be associated to a highly available redundant service. *SFSRO*, the read-only version of SFS with support for replication, suggests the use of round-robin DNS or more advanced methods to overcome the limitation [23, 35]. However, these methods require active centralized administration to maintain the DNS information or network configuration.

A more decentralized and a scalable approach is for the nodes to cooperate as peers to dynamically lookup the node providing the desired information. *CFS* [17] adopts the SFSRO naming, authentication and file system structure but provides the ability to dynamically find nodes serving specific data. The storage is distributed among the peers so CFS needs to be completely location transparent. It uses the *Chord* [54] lookup protocol to find a node that stores a specific data block.

Chord places nodes of the system in a circular identifier space. Node identifiers are calculated by a hash function and the data block identifiers are taken from the same identifier space by hashing the block contents. Each block is stored at the successor of its ID and the block replicas at immediate successors of the node, spreading the data quite evenly among the nodes. Each node maintains a *finger table* of succeeding nodes at power-of-two intervals on the ID circle. Lookup is performed by choosing a node closest to the target ID from the finger table and querying it for possible next hops. In a stable ring each hop eliminates half the remaining distance to the target ID. Figure 2.4 shows a typical lookup path. [17]

With N total nodes participating in a Chord ring each node maintains information only about $O(\log N)$ other nodes and looking up a node requires $O(\log N)$ messages being sent to other nodes. A node joining or leaving the ring requires $O(\log^2 N)$ messages. All in all Chord is a provably efficient lookup protocol. [17]

2.5 Transferring data

Inter-node communication is by definition an essential issue in any distributed system. For distributed file systems the whole point of existence is to share and transfer data among a group of computers. The file system data transfers must be efficient while retaining data integrity and providing the desired level of security.

Modern local area networks have low latency and bandwidth comparable to

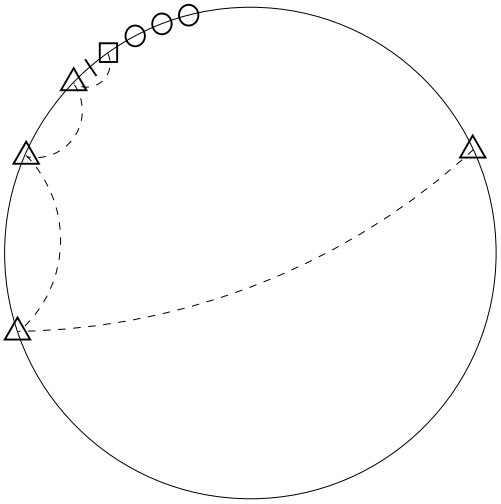

Figure 2.4: A typical Chord lookup path. Lookup advances hop by hop (the dotted arcs) through the servers along the lookup path (the triangles). The block is stored at the server (the square) being the successor of block's ID (the tick mark). Replicas of the block are stored at immediate successors of the server (the circles). [17]

that of local storage device interfaces[5]. In fact, it is feasible to access storage devices over high performance IP networks using iSCSI protocol, for example. However, large scale distributed systems typically use the Internet for global connectivity. In global scale Internet connections often have latency of hundreds of milliseconds or even more, packets are dropped and the effective transfer rate is severely limited by bandwidth and congestion [7, 40]. Consequently many design decisions depend on the targeted scale of the system.

Most of the low level work is performed by the standard Internet protocols: the *Internet Protocol* (IP) and a suitable transport layer protocol. In small scale environments the *User Datagram Protocol* (UDP) has been used, for example by NFS[6], as a low overhead stateless transport for transferring blocks of data on demand. For Internet-wide transfers higher latency makes it worthwhile to preemptively fetch larger blocks of data. Longer continuous transfers and the high probability of congestion suggests the use of the *Transmission Control Protocol* (TCP) as the large scale transport layer protocol. In a related note, file sharing is already changing the Internet traffic characteristics by causing a substantial increase in number of long flows, called elephants [40].

On top of the transport level protocol some kind of *remote procedure call*[7] mechanism is typically used for communicating the requests between the nodes.

[5]Gigabit Ethernet has a nominal transfer rate of 1 Gbps while the modern IDE SATA-150 interface is rated at 1.2 Gbps and the high performance SCSI Ultra320 interface at 2.6 Gbps.

[6]NFS version 4 requires the use of TCP or a transport with comparable congestion control [50].

[7]In this thesis "remote procedure call" refers to the general concept while "Remote Procedure Call" or "RPC" denotes the widely adopted XDR/RPC protocol defined by RFC1831 and RFC1832.

Distributed file systems usually emphasize the performance and they rarely use higher level concepts such as distributed object models. In its most simple form a distributed file system is merely a layer redirecting the file system interface calls to a remote server.

2.5.1 File system usage characteristics

Optimizing file system data transfers requires understanding the dynamic file system usage characteristics. Trace-driven file system measurements [39, 6, 56] have revealed several general characteristics that have prevailed despite growth of main memory size, disk storage size and CPU power.

Files are often read sequentially. Majority of read-only accesses are sequential transfers of whole files and majority of read-only data is transferred sequentially. Most of the all sequential runs (portion of a file read or written sequentially) are short while most bytes are transferred by longer runs. A similar pattern can be observed about the size of files in relation to the number of sequential IO operations; most operations are to short files while most bytes are transferred to large files. [6, 56]

The typical period of time a file remains open is short and has been shortening further possibly signifying the growing CPU performance. According to the Windows NT measurements 75% of files were open less than 10 milliseconds [56]. Earlier studies of BSD and Sprite environments placed the 75% mark at a half-second and a quarter-second, respectively [39, 6].

Most files do not live long either. In the Windows NT study 80% of newly created files were deleted within 4 seconds of their creation [56]. Earlier studies show between 65% to 80% of new files being deleted within 30 seconds [39, 6] so the typical life time has been shortening, too. It appears that the likelihood of file being overwritten decreases sharply after it has been opened for reading several times [10].

It has been demonstrated that the file system activity is extremely bursty and remains so even when averaged over longer time periods [56]. The extreme variance at all time scales means that resource usage predictions can not be made based on the observed mean and variance nor can the arrival rate of IO requests be modeled as a Poisson process. Instead the distribution of meantime between two requests seems to better match a power-law distribution[8] [56].

Even if the file system activity level was impossible to predict, luckily it is possible to predict, to some extent, the files to be accessed next. File accesses have strong temporal locality. Trace results have shown that after two hours of tracing over 66% of files opened for reading were opened previously during the trace [10].

2.5.2 Caching

Avoiding unnecessary communications and data transfers altogether is a straightforward way to minimize network related performance problems. *Caching* is one of the most important concepts used to achieve high performance in distributed file systems. The remotely obtained data that is likely to be needed again in the near future is stored in a local fast *cache* storage. When cached data is needed

[8] In one sample the tail of the distribution of meantime between two file open requests was found to match a Pareto distribution $P[X > x] \sim x^{-\alpha}$ with α having a value of 1.2 [56].

again it is available in the local cache and need not be requested or transferred from the remote storage.

In practice caches have a limited storage size and it has to be decided what data to keep in the cache. This is determined by the *placement policy*, telling when to put something in the cache, and the *replacement policy*, telling which entries to replace when cache is full [10].

Typically data is placed in the file system cache when it is read. This policy is known as *demand caching*. In demand caching the *optimal replacement rule* (OPT) is one that always replaces the item that will be used the furthest in the future [10]. OPT is not practical because it requires that the sequence of future file operations is known in advance but it can be used to evaluate the efficiency of more practical rules. The most common practical replacement policy is to simply replace the *least recently used* (LRU) entry [10]. Fortunately, trace-driven simulations have shown that miss rates of the LRU and OPT rules are close to converging at reasonable cache sizes [10], making the LRU policy both simple and efficient.

Many distributed file systems handle data in fixed size blocks (or pages) [50, 17] similarly to how physical block I/O devices store data. Whole pages are transferred as they (or parts of them) are needed. The approach makes it feasible for an application to read one byte at a time even if it would not make sense to transfer one byte per request due to overhead associated with the request. It also makes cache storage bookkeeping more efficient (it can be done on a block level rather than on a byte level).

Locally cached data may be stored in the main memory or in a local disk. The main memory is fast but limited in capacity while a disk is slower but provides higher capacity and persistence. Under low-bandwidth conditions it is beneficial to use the local disk as a large persistent cache storage [37].

File system measurements show that caching efficiently reduces the amount of information that needs to be transferred [6]. At the same time caching also introduces consistency problems because file system clients have a local cached file system state which does not necessarily correspond with the authoritative state of the server. Methods for maintaining and restoring consistency are described in Chapter 3.

2.5.3 Prefetching

Reading ahead is used in operating systems as part of I/O buffering [52]. It tries to account for the fact that data is often read sequentially by reading and buffering more data than was actually requested by the application. This enables uninterrupted data transfers and lowers the latency associated with subsequent reads.

Similarly a distributed file system client can *prefetch* and cache more data than was actually requested by the application [10]. Prefetching pays off as lower latency and potentially higher average transfer rate if the data is being read sequentially. The average transfer rate can be higher because the pending prefetch requests cause data to be transferred while the application is still processing the data obtained earlier. On the other hand, if the extra data is not needed or it gets replaced in the cache before it is actually needed then the bandwidth used to transfer the data was wasted.

Some file systems prefetch and cache whole files at a time [31]. The whole file caching wastes bandwidth and cache space by retrieving and caching lot of unneeded data. However, during disconnected operation it ensures that either a file can not be accessed at all or it can be successfully read in its entirety. This makes the behavior more consistent from user point of view and also applications tend to fail more gracefully on a failed open than on a failed read following a successful open [31]. Whole file caching has been feasible because most accessed files are small (more than 60% of accessed files being less than 10 KiB [6, 56]). However, the increasing trend of maximum and average file sizes makes it more wasteful or even impossible to cache the biggest files in their entity. One solution is to use whole file caching only for relatively small files and handle larger files in a block oriented fashion.

2.5.4 Write-behind

While caching is often associated with read requests, *write-behind* cache policy is about buffering the write requests. When an application writes to a file the written data is buffered and control is returned immediately to the application. Other file system modifications can be buffered as well. The changes are delivered to the server only after the fact but locally the new file system state is available immediately. This improves application performance by eliminating the latency associated with the write request being delivered to the server.

The data written using write-behind needs to be cached so that subsequent reads of the data return the new version not yet delivered to the server. The situation becomes more complicated if other users of the file system are considered. When they query the server for the data they would receive data that is not current.

An extreme example of using write-behind caching is disconnected operation [31]. File systems supporting disconnected operation can handle local file updates even when they can not currently contact any file servers. The updates are propagated to the server later when the connection is restored. During disconnected operation update conflicts are unavoidable.

2.5.5 Compression

Most files transferred by FTP or similar user-initiated methods are compressed to save server disk space and to make the transfer more efficient. *Compression* can also be used by distributed file systems to make more efficient use of the available bandwidth. For example, LBFS compresses all RPC traffic on-line using conventional gzip compression to save bandwidth [37]. Compression is especially well-suited for low-bandwidth communications if there is some extra CPU time to trade for higher efficient transfer rate.

While conventional compression algorithms eliminate redundant information in the binary stream being compressed a file system can also exploit similarities between files or versions of the same file. LBFS demonstrates this by dividing all files to chunks. The chunks of all stored files are then indexed in a database based on the SHA-1 hash of each chunk. File system reads and writes transmit the associated data initially only as a sequence of SHA-1 content hashes corresponding to the chunks being transferred. If a received hash is found in the index database then the receiver can replace the hash by the corresponding data

found in the local database and the actual data need not be transferred at all. Otherwise the actual data is requested separately from the remote computer. The server maintains a database of all stored files divided to chunks while the client maintains a database of cached data. [37]

If data was divided to fixed size chunks then insertion of intervening data — such as addition of this comment — would totally change the contents of the succeeding chunks. This would render the index inefficient by requiring that the similar data being eliminated appears at file positions divisible by the fixed chunk size. Instead the LBFS divides data to chunks based on a Rabin fingerprint calculated by sliding a 48-byte window over the data. A boundary is determined by low-order 13 bits equalling to a chosen value giving an expected chunk size of 8 KiB. Thus chunk boundaries move accordingly when data is inserted or removed and similar sequences of data are chunked similarly. [37]

In application performance tests the LBFS was up to an order of magnitude faster than a conventional distributed file system (compared to CIFS and NFS) when tested over a simulated cable modem link [37]. This demonstrates the potential benefits of not transferring redundant information, especially under low-bandwidth conditions.

2.5.6 Compound operations

Traditionally remote procedure calls are performed one after another to catch possible errors and to make sure that the remote state is suitable for the next intended operation. For example, to read data from a file in NFS version 3 and older versions the client first performs the LOOKUP procedure to obtain a filehandle and then passes the handle to the READ procedure to read data from the file [12].

In wide-area networks the latency involved with invoking a remote procedure is often significant compared to the time it takes to actually perform the procedure. Having to perform two procedure calls to first lookup the file and then read the data doubles the total latency associated with the operation. Furthermore, each element of the pathname needs to be looked up separately. In NFSv3 this particular case of LOOKUP inefficiency is solved by suggesting the client implementation to cache filehandles [12]. However, this does not remove the bottleneck for other similar operation sequences.

NFS version 4 provides *compound operations* to eliminate this bottleneck. A sequence of operations is sent to a server in a single RPC. Filehandles are passed from one operation to another using a concept of a current filehandle and saved filehandle. The client can then, for example, request the server to lookup a file, read data from file and replace the data in a single request. The server performs the specified operations one after another and stops at the first unsuccessful operation. The server then sends the compound results to the client. [50]

While compound operations reduce latency and network traffic they make it more difficult for the client to recover in case of communication or server problems. If the results of a compound operation are lost then the client must try to recover without knowing how far the server got performing the operations. Therefore it is recommended that overly complicated compound operations would be avoided [50].

2.5.7 Distributed data transfers

Globally publishing high demand data objects such as new versions of popular software or important current news is done mostly in the web nowadays and not through file system interfaces. However, the ability to cope with sudden surges of demand (known as *flash crowds* and *hot spots* in the context of web serving) is a useful feature for a global scale file system as well. The core problem is that the uplink bandwidth available to a single publishing site is often not enough for serving a global audience with far greater aggregate downlink bandwidth. If the audience is large enough their download requests may overload the server or its network connection to the point that no one can successfully download the data.

This problem is reduced somewhat by web proxies that cache content but for a globally interesting data there are still vast amounts of proxies and directly connected clients making requests. For popular web sites companies such as Akamai[9] provide content distribution services. The content is mirrored to several servers around the world and each download request is dynamically redirected to a server that can most efficiently serve the specific client. This kind of commercial content delivery networks are typically targeted for business use. Backslash[10] [51] and Coral [22] are examples of collaborative peer-to-peer content distribution networks balancing load between the servers.

An alternative solution is to harness the uplink bandwidth of the downloaders themselves. Network links are typically full-duplex links making it possible to upload data without adversely affecting the download rate. PROOFS [53] is a distributed system enabling the participating web clients to form a peer-to-peer network for serving cached copies of data objects to each other. Also the BitTorrent [13] peer-to-peer file sharing application uses the uplink bandwidth of downloaders. The downloading clients do not only download data put also upload data they have already obtained to each other.

The BitTorrent has been designed so that the peers have incentive to upload data to each other; they reward each other by preferring to upload chunks of data to those peers they have received new data from [13]. Very much like swapping collectibles. The rarest data chunks (chunks that not many active peers have yet) are worth downloading first because they are most valuable in this respect. This spreads the data efficiently over the peer network and eases the burden of the original publishing site which becomes just one of the peers, the one that had all the data chunks to begin with.

These very same ideas can be used in large scale distributed file systems. They are a form of caching: *peer-to-peer caching*. Another approach is to distribute the whole data storage over the participating peers. A large popular file is then split over a large number of peers and each of them only needs to provide and transfer the part it is storing. This kind of distributed storage is discussed more thoroughly in Section 2.6.4.

[9] `http://www.akamai.com/`

[10] One incarnation of flash crowds is known as "slashdotting" after a popular "news for nerds" site, *Slashdot* (`http://slashdot.org/`), which is known for indirectly overloading web sites as large number of users follow the hyperlinks embedded in the front page articles.

2.6 Replicated storage

High availability and scalability can be achieved by distributing and replicating storage to several computers. Distributed storage without replication requires merely the ability to locate and access the desired data. This was addressed by the earlier discussion. Replication, however, introduces a new set of problems.

Replicated storage is more available and scalable because there exists several physical copies of a logical data object (such as a file). The copies are called *replicas*. They can be placed on different disks or servers to balance load and to provide fallback alternatives in case of system failures. If each of the n replicas has an independent probability p of becoming unavailable over a given time period, then the availability of the data object is $1 - p^n$ [15]. Increasing the number of replicas increases the availability[11], but also requires more physical storage and causes more overhead.

Replicas are managed by *replica managers*: software that is responsible for managing the local replicas at each server. Replicas representing the same file must remain somewhat consistent for replication to be useful. For example, it is undesirable for two replicas representing the same file to have different contents. A user would otherwise see one version or the other depending on the availability of servers and the replica selection strategy used. The user might even see a mix of contents taken from multiple replicas.

The consistency issues with replication are similar to consistency issues with caching. This is because caching is also a form of data replication [15] even if the replicas may be short-lived. In fact, local caching can be used to achieve disconnected operation so that it is possible to read and write files even when being disconnected from all the servers [31]. It can then be considered *second-class replication* apart from server replication, or *first-class replication* [31].

Rest of this section introduces different kind of approaches to storage replication. Consistency issues are discussed more thoroughly in Chapter 3.

2.6.1 Passive replication

File service can be replicated in a local server cluster. Typically the cluster has two or more servers in the same logical network (not necessarily in the same physical location, however). In the simplest mode the primary server provides the service and the secondary server (also known as backup server or slave) in reserve maintains a replica of the file system or a replica of the underlying block device storage. The servers monitor each other, typically using a heart-beat protocol of some sort. Should the backup server notice the primary server to have failed it takes over the responsibility for providing the service.

Because the client only communicates with the server currently in charge the setup can be transparent from the client point of view. This can be achieved by assigning an IP address to the service. During normal operation the service IP is associated with the primary server but on failure the backup server takes over the IP and continues providing the service. Figure 2.5 illustrates this process. Once the primary server recovers it synchronizes the state with the backup server and again assumes the responsibility for the service.

[11] $1 - p^{n_1} < 1 - p^{n_2}$, $0 < p < 1$, $1 \leq n_1 < n_2$

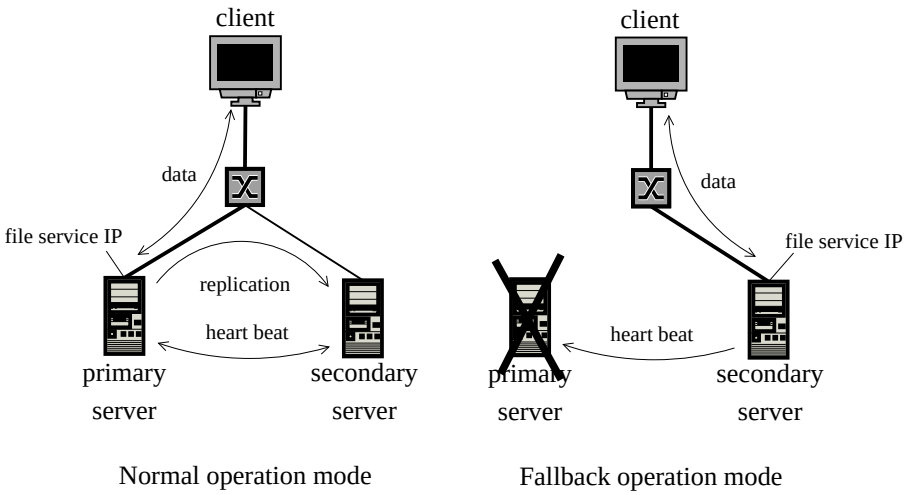

Figure 2.5: A server cluster with a primary server and one backup server for fallback. In normal operation mode all requests are delivered to the primary server which replicates the updates to the backup server. If the heart beat from the primary server is lost the backup server takes over the file service IP address and starts providing the service. Client is not aware of two separate servers.

This kind of setup where the clients only communicate with the server currently responsible for the service is known as *passive* or *primary-backup* model of replication [15]. The server side is simplified by the fact that the changes are being replicated to one direction only at a time (normally from the primary server to the backup server, or vice versa when the primary server is recovering from a failure). Optionally there can be several backup servers. The configuration also allows for the administration to take one of the servers offline for maintenance.

Possible inconsistencies between the servers are visible to the client only when the service is being switched over from one server to another. During normal operation mode the inconsistencies can be avoided by making sure that all updates are replicated to the backup server before the primary server commits to the update. It is then known that during an emergency fallback to the secondary server the file system contents remain the same. View-synchronous group communication can be used to ensure this [15]. After the primary server recovers the servers can verify that they are actually in sync before the service IP is restored to the primary server.

Load balancing cluster setups where multiple servers serve the clients simultaneously require more complex coordination to maintain the consistency. In that respect they are similar to systems with several distinguishable servers.

2.6.2 Active replication

In the *active* model of replication the client requests are delivered to a group of servers. The system uses group communication to deliver the requests to

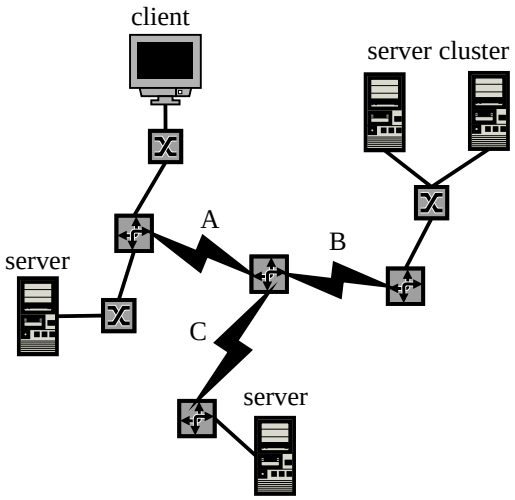

Figure 2.6: A distributed system with servers located in various parts of the network. Flash arrows are wide-area network (WAN) links. If link A fails the client can still contact the server connected to the same router as the client. If link B or C fails the client can contact at least two servers. On the other hand, connection to the server cluster in the upper right corner would go down if either of the links A or B failed.

every correct server in the same (total) order. If only crash failures are to be tolerated it is enough for the client (or its front end) to wait for the first server response. It is also possible to mask up to f byzantine failures as long as the system incorporates at least $2f + 1$ servers and $f + 1$ identical responses are being waited for before proceeding. A byzantine failure is an arbitrary failure of a node possibly caused by malfunctioning hardware, erroneous software or even a malicious person intentionally interfering the system. [15]

Due to group communication system all servers process the requests in the same order and the system achieves *sequential consistency*. A reliable multicast messaging is needed to ensure that all servers receive the same set of requests. The active replication system does not provide *linearizability*; the processing order of the requests does not necessarily follow the real-time order the requests were made in. [15]

2.6.3 Wide-area storage replication

In wide-area systems the downside of local server clusters is the limited reliability of the network connection between the client and the cluster. If the connection is down the client can not access the data. However, the client might still have connectivity to large part of the Internet. If servers are distributed geographically and placed in different local networks it is more likely that the client can establish connection with at least one server. Figure 2.6 illustrates the effect of distributing the servers geographically.

Storage replicated to several servers in different parts of the network gives better availability but needs to overcome the varying networking conditions

and has the potential to induce additional inconsistency among the servers. In global scale systems it is not feasible for a client to reliably multicast its requests to all the correct servers in the system. Instead the client may now contact any of multiple servers to access a file. *Replica control* refers to mapping read/write requests on logical data items onto physical copies of objects [31]. The updates are then propagated among the servers. The *gossip architecture* is about replicating data close to the points where groups of clients need it [15].

Applying strong consistency in a large scale system would require a lot of inter-node communication and hinder the performance of the system. Relaxed consistency semantics may be applied instead.

2.6.4 Distributed hash as storage

Replicating all file system data to all servers is practical only for file systems with a small number of highly available servers. For large peer-to-peer systems with vast amounts of data it would not be feasible to replicate all the data to every peer. Replicating each individual file over a limited random set of peers also leads to uneven distribution of load because some files are in high demand while most are not and some files are huge while most are small. Instead, the file system data can be split to fixed size blocks of data (just like how it is stored on physical storage devices) and each block can be replicated to a random set of peers. In this way the storage requirements and traffic load are more evenly distributed over the peers.

CFS [17] and Ivy [38] are examples of distributed file systems that use *distributed hash*, in this case *DHash*, as data storage. Distributed hash is a hash table implemented by cooperating nodes. Each data block is associated with a hash key which can be mapped to a node or set of nodes (for replication). These nodes are responsible for storing the data and others can locate data blocks by using a lookup algorithm to find the hosts that store data associated with a particular key.

DHash can store arbitrary values. Data blocks are replicated and the integrity of each data block stored is ensured by using the SHA-1 content hash of the block as the key or by signing the block with a public key used as the hash key. A node accessing the data can then check the integrity by checking the content hash or the signature. This prevents malicious or buggy nodes from forging data although they can deny existence of a block or produce a stale copy of a signed data block. Chord [54] introduced in Section 2.4.5 is used as the lookup algorithm. [38]

CFS is a read-only file system in a sense that clients can not update data. Only the publisher of each file system (a single CFS distributed storage may simultaneously store several unrelated file systems) can update the data. The file system data is stored as blocks much like a regular file system is stored on a disk. The root block of the file system is signed by the publisher while all other blocks use content hashes as their keys. The blocks refer to other blocks using their keys, or content hashes, so to update a file system the signed root block needs to be updated by the publisher. Old data is garbage collected by setting a predefined finite storage period for data and extending it if necessary. [17]

Ivy is a distributed log-structured file system. Each storage node maintains a log of the updates it has made to the file system and stores it to DHash. To read the file system a node consults all the logs of the other nodes, putting the

updates in total order. To avoid having to read the whole log all the time each node maintains a private recent snapshot summarizing the file system state. Log records are kept indefinitely. Nodes can be selective about which logs to trust and updates can be controlled by ignoring updates of a data object in the logs of those nodes that are not authorized to update the data object. Cooperating users share a view which is a set of log heads that comprise a file system. Both snapshots and views are also stored in DHash. [38]

Chapter 3

File system consistency

File system *consistency* is an important issue for distributed file systems that replicate data. Due to replication there are multiple instances, replicas, of a logical data object. *Replication transparency* means that users and applications need not be aware of the replicas but they only see the single logical data object [15]. This is an essential property because it simplifies the applications and provides convenience to the users. Otherwise users or applications would need to decide which replica to use and ensure that updates are applied to every replica. Instead, it is the file system implementation that takes care of these issues.

Caching is one form of replication because the copies of data in the cache storage can be considered replicas [15]. Thus also file systems like NFS are exposed to replica consistency issues even if they do not have persistent replica copies. On the other hand also local file systems of multi-processing systems must have concurrency control to handle multiple processes concurrently accessing the same file.

The availability of several replicas has the potential to make the file system behave inconsistently from the user point of view. For example, consecutive read requests might return information from different replicas (the first replica used might become unavailable or this could be due to load balancing). If the replicas are out of sync and represent different versions of the file, the user might first get the current version of data from one replica and later some earlier version. He would see the latest actual updates reversed. Or he might see his own updates undone. This kind of view would definitely be unintuitive and inconsistent with respect to the actual file system events.

It would be useful for the user and applications to always have a consistent and up-to-date view of the file system. However, maintaining these properties may affect performance and availability. For example, if all replicas were always updated concurrently in an atomic fashion, it would be impossible to update any data if one of the storage servers was unreachable due to a network problem.

3.1 Consistency semantics

Different applications need different kind of *consistency semantics* or *correctness criteria*. Linearizability and sequential consistency are generic formal cor-

rectness criteria for replicated object services. One-copy update semantics and serializability capture the same basic ideas in the context of file systems. These concepts will be examined next. Finally, it is observed that a consistently behaving file system is not always adequate. A separate notion of application level consistency and concurrency control is also required.

3.1.1 Linearizability

The most strict correctness criteria for replicated objects is *linearizability* (or *atomicity*) [15]. The criteria is defined using a virtual interleaving of clients' read and update operations on the objects. Such interleaving does not necessarily physically occur anywhere but the concept is used to establish the correctness of the execution.

The following formal definition has been presented by Coulouris et al [15].

> A replicated shared object service is said to be linearizable if for any execution there is some interleaving of the series of operations issued by all the clients that satisfies the following criteria:
>
> - The interleaved sequence of operations meets the specification of a (single) correct copy of the objects.
> - The order of operations in the interleaving is consistent with the real times at which the operations occurred in the actual execution.

This means that every read operation will return data that correctly reflects the update operations by all clients that occurred before the read request (in real time). Thus the data returned is always up-to-date.

The problem is that this criteria is not necessarily very practical to implement. As discussed earlier, distributed systems do not have a single global notion of real time. The criteria also requires that updates be simultaneously performed on all replicas (hence the term atomicity). This limits the update availability.

3.1.2 Sequential consistency

Sequential consistency is a weaker correctness criteria. It does not rely on real time for interleaving the operations. Nor does it require any other total order on all operations. The possible interleavings are limited only by the order of events at each separate client: the program order. [15]

The following formal definition has been presented by Coulouris et al [15].

> A replicated shared object service is said to be sequentially consistent if for any execution there is some interleaving of the series of operations issued by all the clients that satisfies the following criteria:
>
> - The interleaved sequence of operations meets the specification of a (single) correct copy of the objects.
> - The order of operations in the interleaving is consistent with the program order in which each individual client executed them.

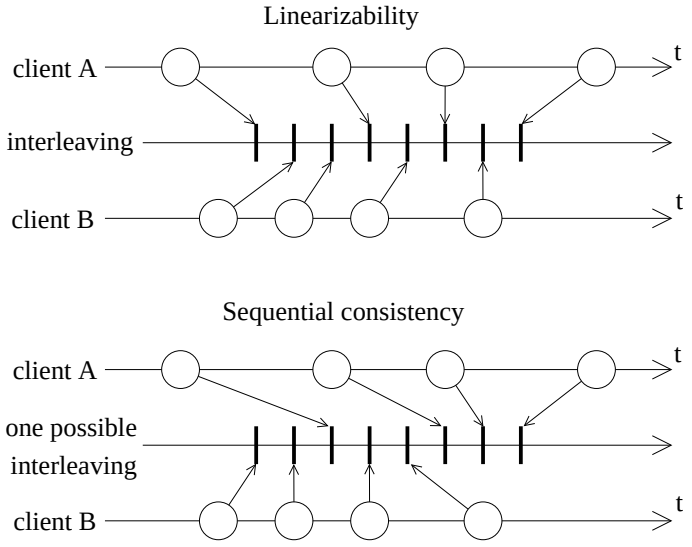

Figure 3.1: An example of possible interleaving of operations for a linearizable service and for a sequentially consistent service. In this example there are two clients performing operations. The circles represent the operations and when they occurred in real time. For linearizability the presented interleaving is the only possible interleaving but for sequential consistency it is only one of the possible interleavings.

A client of a sequentially consistent service will get back data that correctly reflects the earlier (in real time) update operations by the client itself as a result of a read operation. The returned data does not need to reflect every update operation by other clients that occurred before the read operation. However, if the data reflects an update operation performed by any client it is also guaranteed to reflect every earlier update operation performed by the same client. Figure 3.1 illustrates the differences between linearizable and sequentially consistent service.

Sequential consistency highlights the fact that clients are executing their programs independently. If operations performed by a client are placed later in the interleaved sequence than what real time would suggest, it will appear like the client would be behind its actual execution point compared to other clients. If the clients only communicate through the replicated object service the difference is indistinguishable.

3.1.3 One-copy update semantics

According to *one-copy update semantics* of a file system the processes (possibly concurrently) accessing a given file see the file contents like only a single copy of the file contents existed [15]. These semantics are also known as single-machine *Unix semantics* [10] and the criteria corresponds with linearizability: the updates are immediately available to all readers. In any cached distributed file system such strict consistency criteria would mean that clients must verify

the validity of all cached data for each I/O operation [10]. Implementing such consistency semantics would be very expensive [10].

In practice most distributed file systems only approximate the one-copy update semantics in favor of performance [10]. Caching is essential to performance of distributed file systems and because it is based on the idea of having multiple (temporary) copies of the data it does not go easily together with one-copy semantics. Some implementations set measurable upper limits to how much the client view may deviate from the current file system state. The level of consistency may also vary depending on networking conditions; a close approximation may be provided under normal conditions but a more relaxed approximation allows the system to continue operating during network partition.

3.1.4 Time-bounded update semantics

The *time-bounded consistency semantics* sets temporal constrains on inconsistencies. The clients make sure that the cached data they use is never older than preset upper bound. If necessary, they revalidate the cached data with the server. If a client can not contact a server to revalidate the data then the operation fails. [10]

3.1.5 Serializability

Many file system replication mechanisms adhere to *serializability* [26]. It is similar to the database concept of one-copy serializability (1-SR) [10] and corresponds with sequential consistency. Informally serializability means that all client actions be logically orderable in serial fashion though they may be physically concurrent [26].

The logical ordering can be relaxed by recognizing that some operations *commute*. The order of commuting operations is interchangeable while producing the same result in the end. For example, a contiguous sequence of read-only operations can be permutated to any order because they do not change the file system state. Any two operations that operate on distinct files also commute. [15]

Because read operations and operations on separate files commute the serializability enforcement techniques focus on updates and how to intertwine them into the sequence of other updates and consequent runs of read operations.

3.1.6 Eventual consistency

Eventual consistency is a fairly weak consistency criteria. It means that the replicas will eventually reach the same final value if new updates are not performed [46]. In the meanwhile a replica may have a stale or even inconsistent state which is visible to the user. Bounded consistency semantics can be build on top of eventual consistency by selectively preventing the use of replicas that are too out of date by some measure.

In practice, updates are occurring all the time in file systems. The dynamic nature of eventual consistency is better captured by the following definition by Saito and Shapiro [46] (adapted).

> Let replica *schedule* be a sequence of operations that has taken the replica from the initial state to the current state. Two schedules

> are equivalent when given the same initial state, they produce the same final state. This allows for commuting operations and other application-specific concepts of equivalence. A schedule may also include *voided* operations, operations that are not executed. A replicated object is eventually consistent when it meets the following conditions, assuming that all replicas start from the same initial state.
>
> - At any moment, for each replica, there is a prefix of the schedule that is equivalent to a prefix of the schedule of every other replica. This is the *committed prefix*.
> - The committed prefix of each replica grows monotonically over time.
> - All non-voided operations in the committed prefix satisfy their preconditions.
> - Every submitted operation is eventually included in the committed prefix, possibly in a voided form.

3.1.7 Application level consistency

Many applications with several programs accessing the same files need additional *application level consistency* semantics and *concurrency control* even assuming the underlying file system supports one-copy update semantics or linearizability.

For example, a mail transfer agent (MTA) may deliver e-mail to the inbox file while a mail reader is accessing or updating it. What if the user has just removed mails from the inbox and the mail reader is updating the inbox file while the MTA is appending new mail to the same file? To update the inbox the mail reader first needs to read the file contents into a buffer and then rewrite the file leaving out the deleted mails. If the MTA happens to append the new incoming e-mails to the file just after the mail reader has read the file and before it has replaced it with an updated version then the mail reader may overwrite the new e-mails. Either the new mails are lost or the inbox file ends up corrupted including mixed data from both update operations.

The above scenario is possible given the most strict consistency criteria for the underlying file system if there is no atomic operation that could be used to update a file depending on its current contents. Many database implementations support isolated transactions composed of multiple operations to handle situations like this. The widely adopted file systems and file system interfaces do not directly support such a sophisticated transactional approach. Instead, they provide advisory or mandatory file locking mechanisms that applications can apply to ensure their internal consistency. This often requires that all programs accessing the shared files comply with the locking protocol to control concurrent access.

3.2 Cache invalidation

In a single-server model there is one persistent copy of the file system data. If every client operation would be synchronously executed at the server the

server could easily serialize the requests and would even provide one-copy update semantics. However, clients typically cache data for performance reasons. Maintaining serializability then becomes an issue of what data to cache and when the cached data should be invalidated (removed from the cache).

Data cached at a client can be invalidated by the client itself (*client-driven invalidation*) or by the server sending invalidation messages to the client (*server-driven invalidation*) or the algorithm may fall somewhere in between. Client-driven invalidation allows for "stateless" servers because the servers do not need to keep track of active clients but just answer client requests. Server-driven invalidation, on the other hand, requires less server traffic when the widely shared files change infrequently. [10]

3.2.1 Client-driven invalidation

Client-driven invalidation can be implemented, for example, by having the server include a "last updated" timestamp with any data it sends to the client [10]. The client stores the timestamp along with cached data. To check the validity of cached data client could request the current timestamp for the data and compare it with the cached copy. Or when requiring up-to-date data client could include the timestamp of the cached copy in the request and the server would either confirm that the data is still current or send the current version of the data. This approach could be used to implement one-copy update semantics combined with data caching.

While redundant data transfers would be eliminated by including cached timestamps in read requests each operation would still have some communication overhead. By relaxing consistency semantics the client-driven invalidation can be made more efficient [10]. The client could enforce an upper limit for the age of cached data or it could check the validity of data only at file open time [10]. When updating data that is also cached at the client, the client must update (or invalidate) the cached copy as well.

NFS is an example of a distributed file system with client-driven invalidation. It uses a relaxed consistency model where cached data is revalidated on file open [50]. For applications that require file sharing NFS also provides locking facilities [50].

One way to enforce serializability with client driven invalidation is to make sure the data cached at a client is up-to-date immediately after each (synchronous) update performed by the client and each time new data has been fetched from the server. Each cached read operation can then be logically ordered right after the previous update or fetch. The operations by different clients can be interleaved by serializing updates at the server and placing read operations logically after the update that corresponds with the file system state just after the previous update or fetch by the client. The cached data need not be revalidated immediately on updates and fetches but it could be flagged as needing validation ("lazy" revalidation). Figure 3.2 shows an example of how operations could be interleaved.

3.2.2 Server-driven invalidation

Server-driven invalidation requires that the server keeps track of each active client and their cache contents. When data is updated the server sends invali-

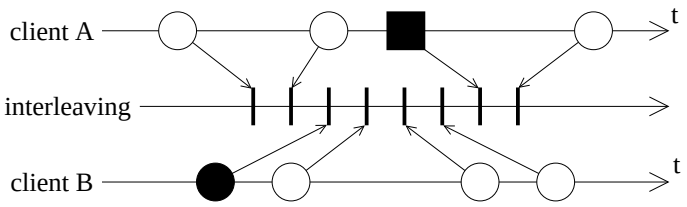

Figure 3.2: A possible interleaving of operations for a serializable file system with read caching. Reads are satisfied using the data in the local cache (unfilled circles) or by fetching data from the server (the filled circle). Notice how two cached read operations by client B actually occurring after the update (the square) and possibly returning outdated data can be considered to have logically occurred before the update [10].

dation messages to the clients that have cached the data. Server-driven invalidation raises new concerns because the invalidation messages could be delayed or lost possibly compromising the consistency [10]. Additionally, write response times can be very poor when the server waits for the invalidation messages to time out due to network or client failure [10].

Implementations of server-driven invalidation must allow for invalidation failures. For example, if an invalidation message can not be delivered to a client due to a network problem or software crash it would be unreasonable to prevent updates of any data the crashed client had in its cache. The crashed client might never recover so that it could process the pending invalidation messages. Additionally the invalidation messages should be delivered to each client in the order the updates occurred to keep the per-client cache contents consistent with some persistent file system state.

AFS is an example of a distributed file system with server-driven invalidation. Servers maintain records of cached copies and issue a *callback* to the client when the file is updated. However, AFS is not strictly server-driven. The consistency of a cached copy is only guaranteed for a fixed time, after which the client must revalidate the data before use. This helps avoiding the pitfalls associated with server-driven invalidation. [10]

Obtaining serializability with server-driven invalidation is straightforward when there is only one server. Let all updates be applied synchronously to the persistent copy at the server while read operations may return data from the local cache of the client. The server serializes updates and provides invalidation messages to the client when cached data changes. All reads that return outdated data from the cache can be considered to have logically occurred before the data was updated at the server [10] (see figure 3.2). If the client is writing data based on out of date reads, any pending invalidation messages can be logically considered to have occurred just after the write [10].

With multiple servers or hierarchical caching enforcing serializability is more complicated. It is sufficient to ensure that once the updated data has been read by some client, future reads by the updater must be current as of just after the write [10].

3.3 Replica control

Several replica control strategies have been developed to enforce consistency in the presence of multiple replicas. One of the most important points of replication is to provide better availability if some of the servers fail or network is partitioned. Therefore the replica control algorithm should limit the availability as little as possible while providing the desired level of consistency.

3.3.1 Serializing replica updates

To keep multiple replicas consistent updates must either be applied to all the replicas at once or the updates must be propagated among the servers hosting the replicas.

Applying updates to all the replicas simultaneously could be done using a reliable totally ordered multicast protocol. However, an update of a file would then require that all the replicas were available at the time of update. Failure of a single server would prevent updates of any data replicated to the server. In a network partitioned to two isolated segments, files with replicas present in both segments would be available for reading to all clients but the files could not be updated by any client. Since file systems are most often used for hosting dynamic data the availability limitations would be rather severe in any large scale system.

Several alternative strategies have been developed to prevent concurrent updates while still allowing some updates to occur even in a partitioned network. All the following approaches confine possible updates to a single segment of the partitioned network, thereby preventing concurrent updates.

Primary copy strategy allows updates to a file only through a designated replica, the primary copy [26]. Other replicas can only fulfill read requests. Updates to a file can then be serialized by the server hosting the primary copy. The updates are propagated to other replicas as they are applied to the primary copy. During a network partition some replicas may be out of date but they always represent some version of the primary copy. Using the primary copy strategy a file can be updated as long as the primary copy is available. For example, in a network partitioned to two isolated segments, the clients in the segment with primary copy could still perform updates. On the other hand, if the server hosting the primary copy fails then no updates can be performed at all.

Majority voting technique allows for single-server failures. It ensures that each update is applied simultaneously to (at least) majority of the replicas [26]. When making updates based on reads the read access must also involve at least majority of replicas to guarantee that the latest version of the file was read [26]. Majority voting does not limit read or update availability when a single server or few of many servers fail. If network is split to two segments the majority voting allows updates only in the segment that has the majority of the replicas therefore preventing conflicting updates across the split.

Quorum consensus is a generalization of voting techniques [26]. Replicas are a priori allocated fixed quantities of votes and reads and updates can require different quorum thresholds [26]. The sum of the thresholds must be one greater than the number of replicas and the update threshold must be greater than one half of the votes [26]. This makes it possible to optimize the availability

according to relative frequency of reads and updates. However, the thresholds can not be selected so that both read and update availability would be high.

3.3.2 Optimistic replication

Enforcing serializability limits the update availability especially in large scale systems where partial operation is common. It is conceivable for a global scale distributed file system never to be fully connected.

The traces of distributed file system usage at the Princeton University Computer Science Department and the Digital Equipment Corporation's Systems Research Center show that write sharing of files is not as common as read sharing [10]. Writes of shared files drop off exponentially with the number of previous readers and less than 5% of the writes occur on files that have been previously opened by two or more clients [10].

The results of empirical studies suggest that update conflicts (two clients concurrently updating the same file) are not very frequent. This observation has enabled the development of *optimistic* replica consistency strategies. Optimistic view to replication is that rare conflicts do not justify the expense of locking and the reduced update availability associated with the pessimistic approach [41]. Instead, optimistic policies allow conflicts to occur and deal with them after the fact [41].

Ficus [26] is an example of an optimistically replicated distributed file system. It provides *one-copy availability*: being able to read and update data as long as any of the replicas are available [26]. This has the benefit that updates are not denied (and lost) just because they might cause a conflict.

Optimistically replicated systems can provide better performance because of the reduced locking overhead. They can also provide highly available low-latency updates because the update operation can be applied to any available replica. On the other hand there is a delay before a client using another replica can see the new version. Most optimistically replicated systems only offer *eventual consistency*: the state of the replicas will eventually converge [46].

For global scale distributed file systems and large peer-to-peer systems optimistic replication is a necessity. Ficus is targeted for geographically distributed workgroups. Its peer-to-peer nature combined with optimistic replication makes it possible to perform and propagate updates even under very challenging networking conditions. Full connectivity is not required at all and no more than two parties need to be able to communicate at the same time [41]. To propagate updates to all nodes it is only required that information flows indirectly among any pair of nodes in finite time [41].

Even if update conflicts might be rare they do occur. An optimistically replicated system must be prepared for detecting conflicts and to resolve them. It needs an update *reconciliation* service to prevent updates from being inadvertently lost [26].

3.3.3 Version vectors

The pessimistic approach avoids update conflicts altogether. In optimistically replicated file systems update conflicts must be detected after the fact. Figure 3.3 illustrates how an update conflict might occur during optimistic replication.

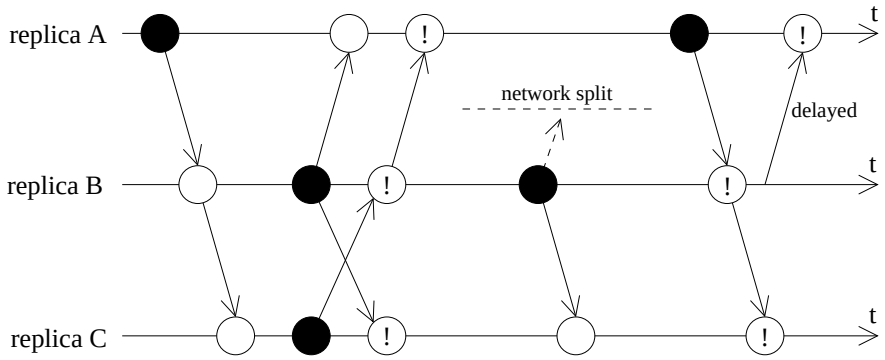

Figure 3.3: Update conflicts occurring between three replicas in an optimistically replicated system. The initial updates (filled circles) applied to a single replica are propagated (arrows and unfilled circles) to other replicas. Update conflicts (exclamation marks) may occur when an object is simultaneously updated through different replicas or when networking or node failures delay the propagation of updates.

The logical order of updates and file versions must also be determined to be able to propagate updates correctly. If two interacting servers concur that they have different non-conflicting versions of a file they also need to determine which version is newer and should be adopted by both.

Version vectors (also known as *vector clocks* [46]) are one way to detect the happen-before relation between events in a distributed system. Version vector summarizes the set of all updates applied to a copy of an object by including per-node *timestamps*. A timestamp can be any monotonically increasing value, typically a counter which is incremented on each update. By comparing version vectors component by component it is possible to detect if one vector dominates the other (represents a newer non-conflicting version) or if they are equal or if they are conflicting. [45]

Version vectors are suitable for distributed systems because they do not need any notion of total order. Timestamp values are only used to establish the order of events at each node individually. On the other hand, in the basic algorithm every node that has ever updated an object must be kept part of the version vector of the object. In large scale systems this might eventually bloat the vectors of long-lived actively updated objects thereby increasing storage requirements and hindering performance.

To limit the growth of version vectors in such systems they need to be *pruned*. A pruning algorithm removes the information that is not needed anymore to correctly determine the relation of two version vectors. This can be done by using distributed consensus protocols to agree on entries to be pruned [45]. However, dynamically changing distributed systems need an algorithm that is localized and capable of removing entries even if some nodes remain unavailable [45].

If the nodes of a system have loosely synchronized clocks and an upper limit exists for the time it takes for an update to propagate to all nodes then the *unilateral version vector pruning* algorithm as described by Saito [45] can be

used. The algorithm associates a physical-clock update timestamp with each entry [45]. Entries are inactivated and then deleted after time periods (D_{retire} and D_{delete}, respectively) which depend — as specified by equations 3.1 and 3.2 — on the upper limits for propagation time (D_{prop}), clock skew (D_{skew}) and network delay (D_{net}) [45].

$$D_{retire} > D_{prop} + D_{net} + D_{skew} \quad (3.1)$$

$$D_{delete} > D_{retire} + D_{net} + D_{skew} \quad (3.2)$$

3.3.4 Conflict resolution

When an update conflict is detected it must be resolved. Resolving an update conflict is about deciding which version of the file a replica manager should use for future reads and updates. The passive solution is to not apply conflicting updates at all. However, this leads to increasingly different versions representing the same logical file. Consider what happens if an update conflict occurs between two replica managers and they both stick to their own version of the file. Future updates applied to either of the replicas will also conflict and the two replicas will drift further apart content wise. The system would not be eventually consistent.

When using *manual resolution* the user is responsible for picking one version or the other as the version which should be adopted by the conflicting replica managers. It might be also necessary to manually merge the two versions to create a new coherent version. For example, manual resolution and decision making is required when there are two conflicting versions of program source code. It is typically the fallback solution when conflicts can not be automatically resolved.

Fortunately, many file system conflicts can be reasonably resolved using automatic *resolvers*. One important class of conflicts is *directory conflicts* (concurrent updates of file directories). Directory semantics are less application specific than application data file semantics, making them a good candidate for automatic resolution. Many conflicts are solved simply by producing union of directory changes [42]. For example, if a new file is created and another file concurrently deleted in the same directory then it is logical for the resolved version of the directory to include the new file but not to include the deleted file.

Two special kind of directory conflicts exist that are more problematic and often require some sort of manual resolution. A *name conflict* occurs when two files are independently created with the same name [42]. The conflict can be resolved by having a user rename one of the files. A *remove/update* conflict occurs when a file is concurrently updated and removed [42]. Simply removing the file is not always the desired solution because the removal of the file might depend on its contents. For example, a shared to-do list with all the tasks marked as completed could be removed by one of the team members while another member concurrently added a new task. Removal of the file would mean the new task was lost. While this scenario is possible even in local file systems as long as file locking is not used[1], the propagation delays make it more likely in optimistically replicated systems.

[1] One user reads the contents of the file and decides to remove it while another user updates it just before it is removed by the first user.

Ficus can automatically resolve all directory conflicts, except for name conflicts [42]. Remove/update conflicts are resolved by moving the updated file to a special *orphanage* directory so the file owner can inspect it to decide whether to keep it or to remove it [42]. Usage experiences show that this is enough to cover most of the directory conflicts. During a nine month period of use 708 780 directory names were created resulting to 128 name conflicts [42].

Update conflicts of regular files can not be resolved without application specific logic. For example, two concurrent changes on a plain text file could be merged automatically if they are not overlapping but the same logic would probably break most binary file formats. On the other hand, some data files appear to be plain text files but require proper ordering or enumeration of entries. Application specific resolvers can resolve conflicts based on application semantics. For example, a calendar application could check that two concurrently added bookings are not overlapping before merging them both into the calendar database.

Ficus allows for the user to specify the resolvers that should be tried. The resolvers can be implemented as scripts, binary executables or any form of executable programs. The system provides some default resolvers but the user can also implement user specific resolvers. [42]

It turns out that between $\frac{1}{3}$ and $\frac{2}{3}$ of update conflicts can be expected to be resolved automatically. However, update conflicts are indeed so rare that this is not necessarily a problem. During the previously mentioned nine month period of use the UCLA Ficus environment processed 14 142 241 non-directory updates which caused only 489 update/update conflicts [42]. 162 of the conflicts were automatically resolved, 176 conflicts could have been resolved if suitable resolvers had been implemented and the remaining 151 conflicts would have not been easy to resolve automatically [42]. It is worth noting that in the UCLA environment replicas were reconciled every hour [42] which leaves plenty of time for potential update conflicts.

Chapter 4

Requirements

This chapter defines the core requirements for the forthcoming file system design. First, an envisioned use case is given and then the evaluation criteria is specified. Implementation details are not considered because implementation of the file system is beyond the scope of this thesis.

The technical focus is on applying the peer-to-peer communication model for scalability and higher availability while trusting on highly reliable dedicated servers for long-term persistent storage. However, broader requirements are specified to ensure the validity of the design for some practical use.

4.1 Use case

Consider a team of people working on a large project with lot of shared data. Parts of the team may be located at different locations. They can be from different departments, there can be several subcontractors involved or perhaps it is a community project with large number of individuals and organizations working together.

This kind of a project most likely uses some fairly advanced distributed version control system to share data like source code and documentation (although version control can be also implemented as part of a file system like done in Wayback [14]). However, projects often have large dynamically changing shared data sets that do not need rigorous version control. There could be large amounts of generated input test data or collections of external documents. The project could also share a testing environment. A distributed file system is a conventional way to share such data.

The team as a whole does not share a reliable network infrastructure. In fact, some members of the team might be gathering requirements and feedback at a customer site, requiring mobile access to the project data. Community projects typically have some centralized resources but the members are geographically distributed. Some members of the community occasionally gather together to more efficiently work on specific issues[1]. Commercial and academic projects with several participating organizations also have workshops.

[1]For example, the Debian project (`http://www.debian.org/`) has about one thousand registered developers worldwide. Occasionally there are "bug squashing parties" where local developers gather together to spend a weekend fixing bugs and working on other issues.

Setting up a conventional shared file system using NFS or an equivalent protocol to share project data does not serve such projects very well. It exposes remote members of the team to limited availability and does not scale well. Placing replicas to each major location and synchronizing the replicas using optimistic replication would provide better availability and could be done using a file system like Coda [48]. However, community projects, workshops and gatherings could use a more flexible system with less dependence on few highly trusted servers.

Peer-to-peer file systems using distributed hash as storage, such as Ivy [38], do not require dedicated servers but they require that peers have fairly good global connectivity. Ficus [26], an optimistically replicated peer-to-peer file system, allows participants of a gathering to share and synchronize data among themselves even if they could not connect to the centralized resources. The downside is that the storage capacity does not scale easily beyond that of an individual node. The lack of centralized dedicated servers also complicates maintenance tasks such as making periodic backups of data.

Combining the dedicated well-managed servers with peer-to-peer caching provides stable and trustworthy long-term persistent storage while simultaneously utilizing the resources of the participating desktop computers. Peer-to-peer interaction also enables temporary serverless operation. For example, during an ad hoc gathering in a coffee shop the participants could share data cached by their laptops. A single peer could also serve as a gateway to a server.

4.2 Evaluation criteria

This section lists the evaluation criteria for the file system design. The criteria has been partly derived from the use case described in Section 4.1. Each requirement is described in detail and some rationale is given clarifying the intent.

4.2.1 Scalability

The file system should be scalable regarding both the storage capacity and the data throughput rate. In other words, it should be possible to increase the storage capacity and data throughput limits by adding new resources to the system. The system should be able to serve at least thousands of users.

The biggest existing community software projects have in the order of hundreds or thousands of participants. The Debian project[2] has about one thousand registered developers and the current Linux kernels have almost 500 credited contributors. The design should be able to support such projects.

Software projects do not necessarily need huge data sets but scientific data sets can be large. For example, SETI@home[3] is processing tens of terabytes of data. One could also envision a community oriented movie project requiring terabytes of storage space. Data set sizes are constantly growing as capacity of storage media grows and more processing power becomes available.

[2] `http://www.debian.org/`
[3] `http://setiathome.ssl.berkeley.edu/`

4.2.2 Peer-to-peer caching

Nodes other than the dedicated storage servers, i.e. clients, should not be obligated to store any data. However, the design should make it possible for the clients to voluntarily share cached data with each other to level server load and to overcome network bottlenecks.

This is the core requirement differentiating the design from many client-server and peer-to-peer file systems. Such a peer-to-peer caching design should be efficient enough to improve performance in some realistic usage scenarios.

An important aspect is that the user should be able to decide how her resources are used. Some peer-to-peer file sharing applications do not let the user to choose which files or data blocks their node can store or relay to others. This is a problem because in an loosely controlled system some data might be illegal to store or distribute. Or the user might find distributing the data immoral or simply prefer to use her resources for some better purpose.

4.2.3 Ad hoc networking without dedicated servers

The file system should support ad hoc networking by making it possible for the clients to persistently store data and to share the data with local peers in a peer-to-peer fashion when there is no global connectivity. The peers should be able to update data and to propagate the updates to each other even when the local ad hoc network is the only connectivity available. The updates should be propagated to other nodes of the system as soon as connectivity permits.

The idea of persistently caching selected data is related to the idea of *pre-warmed cache*: The client preemptively fetches the data that is likely to be needed during disconnected operation. Ideally the design should make it possible to select the data that should be available for ad hoc gatherings despite of the possible lack of connectivity.

This enables the flexible use of the distributed file system data in ad hoc gatherings despite of the possibly limited connectivity. It also ensures that the updated data is eventually available to all the users of the system. Disconnected operation can be considered a special case of this requirement. When a node is disconnected it is the only node of the system in the local network segment. The user of a disconnected node can access the locally stored data and update it.

4.2.4 Hierarchical control and trust

The file system design should support hierarchical control of the file system. Control involves controlling the access to the system and the content of the system. Also the trust relationships between the users and the components of the system should follow a hierarchical structure; a user is only expected to trust components controlled by himself or components that have been certified as trusted by people who are higher in the hierarchy.

Large projects typically have some level of hierarchical organization structure. In commercial environment there is typically a project manager and possibly a steering group supervising the workgroups working on subprojects. Even community projects often have the core members or the lead developer who have the highest level of control. The Linux kernel development community can be

used as an example; Linus Torvalds ultimately decides what goes to his branch of Linux but he has several trustees and the responsibility for the development of specific subsystems has been delegated to yet other individuals.

Chapter 5

Design

This chapter describes the design of the *Cote*[1] file system. Cote combines client-server architecture with client level *peer-to-peer caching*. The idea is closely related to the concept of *hierarchical caching* which has been studied by Blaze [10]. Both client and server level peer-to-peer caching has been explored in the scope of web serving [51, 53, 22].

The major design objectives have been the high reliability and manageability of dedicated server environments — AFS inheritance one might say — and the potential of a peer-to-peer oriented approach as illustrated by many earlier file system designs [1, 17, 26, 38] and the BitTorrent file sharing application [13]. Also the support for disconnected operation, highlighted by Coda [48], has been an important motive.

5.1 Overview

This section provides an overview of the Cote design. Cote is like a traditional client-server system where the single central server has been replaced by several cooperating servers. However, the clients may also share cached data among themselves. Figure 5.1 illustrates the structure of the system and the relationships between the nodes.

5.1.1 Roles of participating nodes

Distributed persistent storage can be implemented using few dedicated servers, like AFS and Coda does [48], or the aggregate storage capacity of large number of participating nodes can be leveraged, like done by Ivy [38] and Farsite [1]. However, while the latter approach is more scalable, locating and transferring pieces of data scattered to numerous distinct locations incurs more overhead than accessing data placed at few high performance file servers. Cote inherits features from both approaches.

[1] People like to give names to things and to refer to things by their name. Originally I started describing this file system design by referring to it as "the design" or some such but it felt awkward and somehow very unnatural. Therefore I decided to name the file system after its intended use for serving a cooperating team. To cover my overly ambitious plans, I named it Cote.

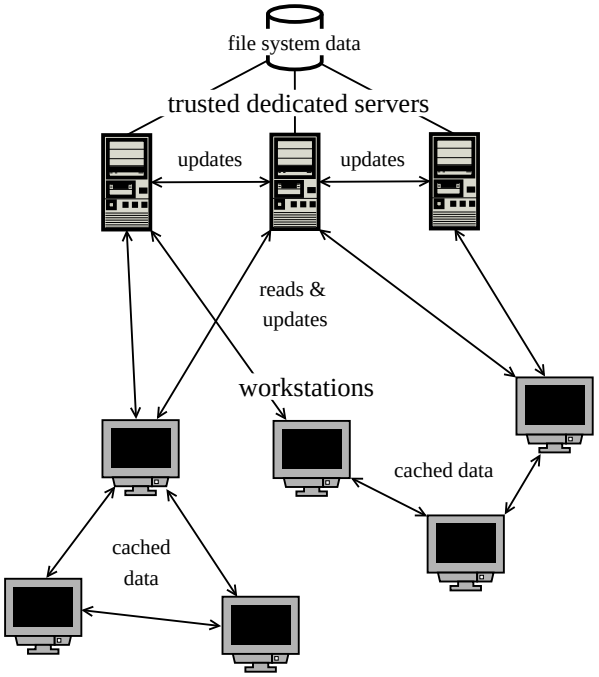

Figure 5.1: Structure of the Cote distributed file system. The persistent storage is provided by dedicated servers. The workstation clients access the data through the servers but they may also serve cached data to each other in a peer-to-peer fashion.

Each node participating in the Cote distributed system may persistently store file system data. However, some nodes are more reliable than others. The highly reliable trusted storage servers are equivalent to AFS servers. Their role is to provide a long-term persistent data storage. Other nodes can persistently cache and store data but they are not trusted to provide long-term persistence. They can, nonetheless, serve the locally cached data to each other reducing load on dedicated servers. Nodes may also buffer updates locally and lazily propagate them to servers responsible for persistent storage. A special case of this is a disconnected node which can use the local cache to satisfy reads and to buffer updates, like a Coda client does [48].

Cote nodes not acting as dedicated servers are not pure clients because they can also serve data to each other. However, in this thesis these nodes are called clients to differentiate them from the dedicated servers.

Cote nodes are identified by a public key. A node can simultaneously access several distinct file systems, volumes, which may not fall under the same administrative domain. The public key provides a globally unique provable *node identifier*. Notice that this key is not directly associated with a user. Multiple users could use the same Cote node and a single user could use multiple nodes.

5.1.2 File system structure

The file system structure has been strongly inspired by Ficus [26]. To enforce a coherent file system structure while still making the storage management flexible this design adopts the use of volumes and graft points similar to those used by Ficus. However, there are also important differences, particularly regarding the organization of persistent storage.

The file system is divided to singly-rooted, self-contained *volumes* for more flexible management. Each volume is managed and controlled ultimately by a *volume owner*: a person or an entity identified by a public key. A *volume number* is assigned to a volume by its owner when it is first created. The volume number is unique among the volumes belonging to the same owner. Each volume can then be identified using a globally unique *volume identifier* which is based on the owner public key and the volume number.

A volume contains a hidden *volume header* file and a conventional tree-structured file directory. The volume header includes volume specific parameters specified by the volume owner. These parameters are related to the organization of the storage and they are discussed in detail later. A volume can have other volumes mounted at specified points called *graft points*. The general idea of grafting was described in Section 2.4.3. Graft points are stored in special files in the volume. They include the identifier of the grafted volume and a list of storage servers possibly carrying the volume data.

Volumes are *autografted*; the clients consult the respective storage servers as the directory tree is traversed making the graft points fully transparent for a user. It is possible to create a cyclic structure by having two volumes graft each other, possibly via several intermediate volumes. Clients must be prepared to refuse grafting of cyclic references using the knowledge of volume ancestors or otherwise gracefully handle the situation.

Volumes can be replicated to several servers. Volume replication is independent from graft points. Any node can start replicating a volume but only the servers certified by the volume owner are automatically trusted by other partic-

ipating nodes. The server list included in a graft point is only a hint making it easier for the client to locate a suitable server carrying the data of the grafted volume. The graft points are in the administrative domain of the containing volume and their server lists are periodically updated to be consistent with the set of trusted servers carrying the grafted volume.

Logical files (including special files such as volume headers, directories and graft points) can be replicated as part of a volume but they can also be persistently cached by other nodes. Each logical file is identified by a globally unique permanent *file identifier*. File identifiers are produced by a one-way hash function in a way that prevents collisions while still allowing a node to generate new identifiers without interacting with any other node. A detailed description of the used method can be found in Section 5.2.4. The volume header and the root directory always have identifiers based on predetermined fixed parameters so they can be accessed directly without doing any directory lookups.

File replicas stored at persistent storage servers and possibly cached at other nodes are versioned using the basic version vector algorithm [45]. Version vectors were described in Section 3.3.3.

5.1.3 Persistent storage

Cote depends on trusted dedicated servers for long-term persistent storage. The servers are certified as trusted by the volume owner and they are assumed to serve correct volume data and to correctly receive updates on volume data.

Volume data can be replicated over several servers. Highest availability is provided by a configuration where all servers store all files of the volume. Clients can then obtain service from any available server. However, such configuration does not scale well in storage capacity because every server must be capable of storing the entire volume data. Adding volumes to overcome space limitations would cause additional maintenance overhead and would lead to artificial splitting of volumes.

The aggregate storage capacity of the servers can be more efficiently utilized by making each server only store a specific part of the volume data. CFS does this by distributing individual file blocks over the servers according to their content hashes [17]. This leads to even distribution of storage and network load. However, it also means that each large file is spread over numerous servers making it difficult to track file versions and requiring large number of network connections to read the whole file. Version tracking is not a problem for CFS which is a read-only file system. Similarly distributed Ivy uses a log-based file system to work around the problem [38].

Cote assigns whole files to servers. The files are assigned to servers according to the highest bits of file identifiers which were calculated by a one-way hash function. As long as the largest files are relatively small compared to the total size of the volume the data is distributed rather uniformly among the servers. Network traffic of servers is distributed in a sense that clients are free to use heuristics to determine which servers to use. Large individual file transfers can also be distributed by requesting different parts of the file from different servers carrying the same file. All in all, Cote leaves it up to the client implementation to decide how network traffic should be distributed. The rationale is that each client trying to maximize efficiency for itself ultimately balances the overall load

server	*identifier pattern*	*storage requirement*	*replication factor*	*share of random load*
A	any	full volume	2–4	$19/48 \approx 40\%$
B	00...	1/4 volume	2	$1/8 \approx 13\%$
C	01...	1/4 volume	2	$1/8 \approx 13\%$
D	10...	1/4 volume	3	$1/12 \approx 8\%$
E	10...	1/4 volume	3	$1/12 \approx 8\%$
F	11...	1/4 volume	4	$1/16 \approx 6\%$
G	11...	1/4 volume	4	$1/16 \approx 6\%$
H	110...	1/8 volume	4	$1/32 \approx 3\%$
I	111...	1/8 volume	4	$1/32 \approx 3\%$

Table 5.1: An example of a storage replication configuration. Each server stores the files matching the bit pattern assigned to it. This allows for varying server storage requirements depending on relative storage capacities of the servers. By assigning an overlapping bit pattern to specific number of servers the replication factor of the data carried by these servers can also be adjusted to account for the reliability of the servers. The table also lists the share of load each server would receive assuming the clients would randomly select a server among those carrying the file to be accessed.

in an effective way (each server receiving as much load as it can handle without its relative performance deteriorating too much).

The files are assigned to servers seemingly arbitrarily (using a hash function). However, the volume owner can control the relative storage capacity requirements of each server by assigning differently sized subsets of the file identifier space to servers. This is done by specifying a bit pattern for each server. Files with an identifier matching the bit pattern are replicated by the server. Table 5.1 shows an example of a *volume replication table* mapping each server to a file identifier pattern.

The replication table can be used to balance storage requirements between servers having different storage capacities. Also the replication factor of data is controlled through it. The example configuration has one server which replicates all files. This would enable easy snapshotting and archiving of data while distributing the serving load among several smaller servers. The downside of such high level of control is the need for centralized management and the need for cooperation between the volume owner and server administrators. On the other hand, ensuring the trustworthiness of the servers already require close cooperation.

5.1.4 Peer-to-peer caching

Most practical distributed file system implementations cache data at a client. Caching efficiently increases performance because file accesses have strong temporal locality. Cote takes caching further by letting clients share the data they have cached with other clients.

Let there be two clients in the same local network accessing a file system which is hosted on a remote server. If the connection to the server is slow, perhaps going over the Internet, and the clients are accessing the same files

then it would make sense for them to have a collective cache. This would be similar to a web proxy server which provides a common cache to the clients using it. Cote does not use proxy servers but enables clients to directly serve cached data to each other.

Potential local peers can be discovered by broadcasting. Being able to easily cooperate with local peers is important for two reasons. Firstly, if there are no servers available (ad hoc networking) it enables the nodes to access the data persistently cached by other local nodes. They can also propagate updates to each other. Secondly, the peers in the same local network can reliably and efficiently communicate with each other and they potentially share file usage patterns. It is supposed that two peers in the same local network (say, in the same office or department) are more likely to access the same data than two random nodes.

Cote servers help clients to locate peers in remote networks. By querying a server a client can find remote peers which have been accessing the same data it is currently accessing. The primary reason for two clients in different parts of the Internet to cooperate would be to compensate for an overloaded and slow server. If reading data from the server is slower than transferring data between the clients then it makes sense for the clients to serve cached data to each other. This will also ease the burden on the server.

Caching data, especially persistently caching large amounts of data, will consume some client resources. Users must be willing to dedicate extra resources to be able to efficiently share cached data. To motivate the users, Cote design provides incentives similarly to BitTorrent [13]. When serving peer requests, the clients will give priority to those peers that have been most effectively serving them. This strategy also has the benefit of implicitly favoring peer-to-peer relationships where peers are close to each other thus localizing the network traffic.

Clients typically have multiple open connections to different peers. One reason is that this gives better coverage of cached data. A client can transfer blocks of one file from one peer and blocks of other file from some other peer. The second reason is that this makes it possible for the client to constantly monitor the relative performance of several peers. Even if the performance of the local network was varying the client would know which of the peers are best performing. When probing for potentially better peers the client can then drop the connections to redundant poorest performers to keep the number of open connections within the predetermined limits.

Consistency is easily sacrificed by naive caching, especially in a system like Cote where a client can use several redundant data sources simultaneously. In Cote all data updates are propagated optimistically including server-to-server, client-to-server and client-to-client updates. This means that update conflicts can occur between any two node. The possible conflicts are solved manually. Version vectors are used extensively and clients maintain a local version cache of recently visited files ensuring that only versions no earlier than the last visited version are seen by the user and applications.

5.2 Security mechanisms

Security is an important issue in any large scale distributed system. When there are numerous participants involved one can not just trust every one of them. On the other hand, security features must not hinder the use of the system. Ideally, they should be transparent to the user.

Cote uses *public key cryptography* to identify entities, to sign file system data and for the trusted entities to certify other entities. At the highest level of the per-volume hierarchy of trust is the volume owner who is trusted to manage the volume storage and to control access to the volume data. The public key of the volume owner is the only key a user must receive and verify when starting to use a volume.

5.2.1 Data integrity

The volume header includes all the information about how the long-term persistent storage is organized, including the list of trusted servers and their public keys. This file is signed by the volume owner effectively certifying the listed servers as being trusted to store and to process data properly. Data is stored as is on these servers and clients assume them to return correct data and to correctly store the updates delivered to them. However, external security mechanisms such as file signing and encryption can be applied, if necessary. A periodic update of a volume header by all nodes can be forced by including in the header a date after which the header is not valid.

All data returned by a server is signed by the server together with the current timestamp. When caching data a client caches the signature along with the data. When cached data is delivered to other clients participating in peer-to-peer caching the associated server signature is transferred as well. The receiving client can then verify the integrity of the data without having to trust intermediate peers. It does this by checking the server signature and checking that the signing server is actually listed in the volume header. The timestamp can be used to enforce an upper limit to the age of cached data accepted from peers.

5.2.2 Access control

In a Cote system, servers are not the only nodes responsible for access control. Nodes participating in peer-to-peer caching must independently enforce access restrictions when they process read requests and receive updates from their peers. This requires that access control checks can be done using the file meta data and the information provided by the requester.

Unix style user/group level access control based on the notion of file ownership can be implemented by having user and group ownership attributes on files together with an access mask attribute. Users and groups are identified by public keys. A client proves the public key of the current user to the other party by using a standard challenge/response protocol. Group membership is certified by the group owner signing a membership certificate for a user. Each user stores her own group membership certificates. They are provided on demand when group membership is required for access.

To integrate the distributed file system into the local view of the file system, selected user and group keys can be locally mapped to local user and group

identifiers (UIDs and GIDs). Unknown users and groups are mapped to default values. This local mapping is only used for viewing file ownership and access mask information, not for authorization.

The previously described access control mechanism provides compatibility with Unix access control semantics but it is not very flexible. For large scale use, a better alternative mechanism is to use *authorization certificates* provided by the Simple Public Key Infrastructure (SPKI) [21].

Authorization certificates are issued by *issuers* to authorize *subjects* to perform some action. If allowed by the issuer, the subject may further *delegate* the authorization by issuing another certificate to another subject. Delegation creates a *certificate chain*. To be able to perform the action, a subject must present a valid certificate chain proving that he has been properly authorized. A certificate may only be valid after or until a specific date to allow for temporary permissions and revocation of permissions. [21]

In Cote, authorization certificates are used to authorize users to perform read or update operations to specific parts of a volume. The initial issuer is always the volume owner. In a large system the owner may issue authorization first to bigger entities, such as organizations or departments, which then delegate the authorization further to groups of users and finally to individual users. On the other hand, users may delegate authorization to access selected personal files to other users. Read and write permission can be limited to a specific file or a directory (optionally recursively including all files and subdirectories) and they never cross volume borders.

Each user is responsible for storing the certificate chains needed to prove one's own authorizations. Servers and nodes participating in peer-to-peer caching request proof of authorization when necessary and cache most recently used certificate data for better efficiency. They know the public key of the volume owner and can thus verify that the presented certificate chain is valid. Managing and issuing certificates requires additional tools and utilities not considered in this thesis.

5.2.3 Update propagation

Propagating updates among the clients requires two access control checks. The peer sending the update must check that the user associated with the receiving peer is authorized to read the data while the receiving peer must check that the user who initiated the update was authorized to update the data. An update can be propagated via intermediate peers so the receiver must be able to verify the initiating user and to check his authorization.

Every update is signed by the initiating user *and* the initiating node. The user signature is used to check the update authorization. However, this is not enough. A malicious user being authorized to update a file could construct a new version vector (included in the update) which is equal to a version vector created by some other node, possibly overriding other updates. The node signature is used to check that the base version vector (also included in the update) and the new version vector only differ for the component of the signing node.

Also the server-to-server replication could use the same security mechanisms for update propagation. Each server would check the update authorization independently using the original update message as signed by the originating user and node. This would make it impossible for a malicious server to inject

invalid data to other servers. However, each server would have to store the received update messages until it has successfully propagated them to every other replica. This could be a problem if a replica becomes unreachable for an extended period of time. Therefore Cote allows for the servers to synchronize periodically with each other and to delete the update messages after they have been applied to the local replicas.

5.2.4 Collision resistant identifiers

Volume and file identifiers are constructed so that a node can generate new identifiers without interacting with any other node. This enables disconnected operation and avoids possible bottlenecks caused by centralized components responsible for assigning identifiers. Because a centralized counter can not be used to generate identifiers, other methods must to be used to avoid identifier collisions.

One possible solution would be to use universally unique identifiers (UUIDs) based on timestamps and pseudorandom data. However, it would be difficult to prevent a malicious person from intentionally causing identifier collisions which might have security implications.

Instead, Cote uses one-way hashes calculated over a tuple composed of a public key and a number which is obtained from a monotonically increasing counter associated with the public key. Volume identifiers are based on the public key of the volume owner and file identifiers are based on the public key of the creating node. The public key uniquely identifies the creating entity while the monotonically increasing number provides uniqueness among the identifiers generated by the entity. One-way hashes provide more compact identifiers while preserving uniqueness with high probability[2].

A file creation request includes the public key and the signature of the initiating node as well as the file number. This data can be used by the node processing the request to determine the hash identifier which does not need to be included in the request at all. It is then computationally very expensive (practically impossible) for a malicious person to cause targeted identifier collisions without somehow obtaining the secret key used to generate the targeted identifier. He would have to try huge number of different file numbers or key pairs to find a collision with an existing identifier not originally created by him.

5.2.5 Self-certifying volume identifiers

Cote volumes can be uniquely referred to using the volume identifier. The volume identifier can be specified in a graft point or as part of an URI (a possible Cote URI scheme is demonstrated later in Figure 5.2). A client following such a referral should be able to verify the integrity of the data.

To verify the integrity of downloaded volume data, a client needs a valid verified volume header which provides the public keys of the trusted servers. A volume header can be easily downloaded from any reachable server but its integrity must be verified before it is used. The volume header is signed by the

[2]One-way hash functions are designed to be collision resistant. Assuming the hash function behaves randomly, the probability of two different inputs producing the same output is $1/2^n$ where n is the length of a hash value in bits. The probability of a collision occurring for a set of k different inputs is $1 - \frac{2^n}{2^n} \cdot \frac{2^n-1}{2^n} \cdot \ldots \cdot \frac{2^n-k+1}{2^n} = 1 - \prod_{i=1}^{k-1} \left(1 - \frac{i}{2^n}\right)$.

volume owner and it also includes the signing key so it is easy for the client to check the signature. However, the client does not necessarily know if the included public key really belongs to the correct volume owner. A malicious person could create a fake but correct looking volume header by generating a new public key, including the key into the header and finally signing the header with the generated key.

The volume identifier itself can be used to verify the public key of the volume owner. Remember that the volume identifier is actually a content hash calculated over the volume owner's public key and the volume number. The client can repeat this calculation to verify that the key and the number in a downloaded header really correspond with the volume identifier. Having verified the public key it can then verify the integrity of the volume header. Cote volume identifiers are therefore *self-certifying*, an idea derived from the self-certifying pathnames as used by SFS [35].

5.3 Performing file operations over a network

Once the connection has been established, client-server and peer-to-peer operations (reads and updates) are very similar in Cote. Also the server-to-server update propagation uses the same protocol. The biggest differences are due to security issues, more specifically due to different kind of trust relationships. These security related aspects were discussed in Section 5.2.

Update operations are used by clients to store the updates persistently to servers and by servers propagating the updates to other servers. Clients may also share updated data with each other without first storing the updates on a server but they do this only through read operations (upon request). By allowing clients only to fetch data from other clients, rather than to push data to them, makes it easier for the clients to control the use of local resources.

5.3.1 Initial volume access

Initially when a client is about to access a volume for the first time, it only knows the volume identifier. To get a list of volume servers and other volume parameters the client must locate and download the volume header. If the volume is being accessed through a graft point then the server list included in the graft point can be used to locate a server. The client can also use the mechanisms described later in Section 5.4.1 to discover a local peer which can provide the volume header. Otherwise the user must point the client to a suitable source.

It would be useful to be able to refer to a specific file also outside the file system, perhaps in an e-mail or on a web page. A full pathname is not necessarily unique because the volume can be mounted at different points or it can be dynamically grafted through a graft point. The referral should also include enough information for the recipient to locate a server that provides access to volume data.

A Uniform Resource Identifier (URI) [9] is a standard way to name and to locate resources. Cote could use a URI scheme like the one suggested in Figure 5.2. Such an URI could be used to uniquely identify a volume or a file in a volume and to also specify a server which can be used for initial volume access.

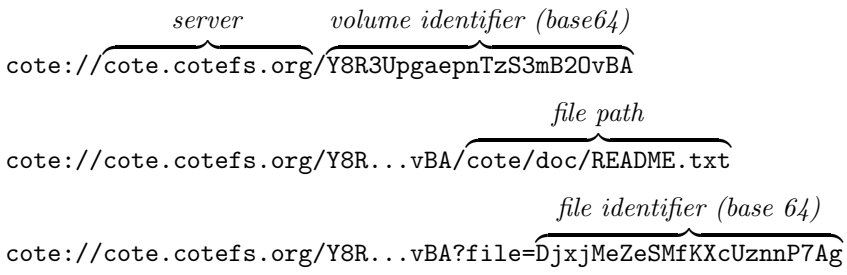

Figure 5.2: Examples of possible Cote URIs. The first URI identifies a Cote volume and specifies a server that can be used to access the volume. The volume identifier is base64-encoded. The second URI extends the first by specifying a file on the volume using a file path. The third URI also specifies a file but this time using a file identifier making the URI immune to pathname changes.

While the illustrated URI scheme is not necessarily very human-readable due to length of the identifiers used, it is compact enough to be embedded on a web page or to be sent by e-mail.

5.3.2 Communication channel

TCP sockets are used for all data transfers to have proper congestion control for large scale usage. The same connection may be used for transferring requests to both directions. For client level peer-to-peer communication the connection is initiated by a client seeking for new peers to share data with.

Communication is asynchronous. There can be several pending operations, or remote procedure calls, in both directions. However, these are not compound operations or transactions. A queued operation must not depend on the success of earlier pending operations. In practice this means that multiple read operations and unrelated updates can be queued. Queuing multiple operations reduces the undesired effect latency would otherwise have on the data throughput. One way to take advantage of operation queueing would be to prefetch data that is likely to be needed next. Latency is less of an issue for updates because they are propagated lazily in the background.

The protocol used is a straightforward binary protocol for data transfers. Each request includes a *request number* which is a connection and direction specific monotonically increasing number for mapping each response to the corresponding request.

For secure communications and to apply the security mechanisms described earlier nodes can use the Secure Sockets Layer (SSL) or Transport Level Security (TLS) to exchange keys and to secure the communication channel.

5.3.3 Data transfers

Cote transfers data in fixed size pages. Both reads and writes are always done on full pages and cached data is managed on page level. Using the same page size in all parts of the distributed system simplifies storage management, communication and caching. However, because the optimal page size is an application specific parameter and tends to be growing over time, Cote makes it a volume

specific parameter. A volume containing only relatively small files can use a small page size while a volume containing large files, such as media files, should probably use a larger page size. A volume can be grafted on to other volume with different page size.

There is no separate open/close procedure associated with file access. File pages are just read and updated directly. This makes it straightforward to lazily propagate updates among the nodes. According to Unix semantics an open file remains available to those having the file open even after the file has been deleted. In Cote, deletion of a file immediately removes the file from a directory (hiding it from future lookups) and creates a new file version, the tombstone. However, a server may temporarily keep the previous version of the file around to continue serving clients still having the file open, thus approximating Unix semantics. The tombstones are purged after a suitable grace period.

Read and update requests include the identifier of the file and the starting offset for the data to be read or updated. Update requests and read responses naturally include the file data associated with the operation. Also the total length of the file is included along with file data. File can then be extended or cut to specified length as part of any update operation. Including the file length in read responses enables fast and reliable seeking.

Read and update operations involving the last incomplete page of the file transfer only the bytes included in the file. File attributes (version, type, length, owner, last modifier, create and last update timestamps, access control information, etc.) can be read and some of them updated using file attribute read and update operations.

Volume headers, graft points, file directories and symbolic links are special files that are not transferred as pages. File directories have special semantics and usage patterns so they are managed using per-entry granularity. Cote includes operations for creating a file, removing a file, moving/renaming a file, looking up a file and listing files. The contents of other special files are transferred as raw data but special atomic whole file read and update operations are used to maintain the integrity of the files.

5.3.4 File version information

All logical files are versioned enabling lazy propagation of updates and consistency management for several redundant data sources. Versioning is also important to detect possible update conflicts caused by optimistic replication. Cote uses version vectors which were described in Section 3.3.3. Version information is either explicitly or implicitly attached to all transferred data.

Update operations are different from read operations in that both the old and new file versions must be included with the file data. Consider a file eventually having a version history: $v_1 \rightarrow v_2 \rightarrow v_3$. A server is initially storing version v_1. Optimistic replication and lazy propagation allows for a client to have a newer version, v_2, in its cache. The client updates the file resulting to update $U(v_2 \rightarrow v_3)$ which is sent to the server specifying the new version v_3. The version vectors v_1 and v_3 are enough for the server to determine that this is a non-conflicting update. However, applying the update would lead to a broken file because the server does not yet know about the update $U(v_1 \rightarrow v_2)$ which should be applied first. By including the base version of the update, v_2, the client makes it possible for the server to detect this.

Version vectors may become large if a file is being updated by numerous nodes. Including a full version vector in each request and response involving file version information might cause significant overhead. When sequentially reading through a file (a common file system usage pattern) it is very likely that the version remains the same from one read operation to another. Redundant version vector transfers can be avoided by having the communicating nodes remember the last sent and received version of each file during the ongoing session. Then only the changed components of a vector need to be transferred. This is particularly useful because only one component of a version vector is updated for every new version of a file.

To not impose high memory requirements on communicating nodes the per-session *version history* can have a limited size (least recently used version information being purged as necessary). If a node receives a request with an incomplete version which can not be completed using the available version history then it responds with an error code indicating that the requester should try again with full version information. When sending a request a node can set a flag in the request indicating that it would like to receive full version information in the response.

When signing data the signature is always calculated over the data and full normalized version information. Correspondingly, signature is checked only after expanding the received version vectors. This way the possible indirect recipients receiving the data via intermediate peers can verify the signature even if the version vectors were encoded differently during the separate transfers.

5.3.5 Idempotent updates

A file system operation is idempotent if it produces the same result in the end whether it is performed one or more times. The idempotency allows for an easy recovery procedure of simply redoing an operation when it is not known if the operation was successfully completed. This could happen, for example, when a connection is lost after sending a request but before receiving a response.

Cote protocol has been designed so that updates are idempotent. Read operations are not strictly idempotent. They do not change the file system state but the result (the returned data) depends on the current file system state as seen by the responding node. However, this is not a problem because read operations are expected to return different data at different times and they are only repeated if the requester did not receive the earlier result in the first place.

Version information included in update requests is used to guarantee that repeating a successfully performed update does not change the file system state while returning the same result as the original update. Each update carries the version produced by applying the update. If the current version as stored by the server is equal to or dominates the version included in the update then the update has already been applied and the server just returns a response indicating success. Otherwise the server applies the update if possible (if the server has already applied all preceding updates and there is no update conflict).

Update conflicts affect idempotency because conflict resolution can change the file system state at the server making a previously possible update impossible or vice versa. However, no update is ever applied more than once. The update can be considered to have initially reached the server only after the file system state changed. The possible different result is not a problem because the fact

that the update is being repeated means that the requester did not see the earlier result or the requester actually expects the earlier result to change from failure to success.

5.4 Peer-to-peer caching

The core problem of peer-to-peer caching is to find peers that can most effectively deliver the desired data. It is not enough to find some peer that carries the data. To benefit from peer-to-peer caching a client needs to find peers that can collectively deliver data faster than the known servers. Sometimes the direct server connection is actually the best option.

Cote does not use a system-wide peer-to-peer overlay network. Instead remote peers are discovered with the help of the servers. This section gives a more detailed description of how the peers are discovered in Cote and how the distributed caching works. Some heuristics are presented to determine when to use peer-to-peer caching and for choosing good peers to interact with.

5.4.1 Discovering peers

To participate in peer-to-peer caching a client must be able to discover other client nodes, or peers. However, finding a random peer does not help if the peer does not have any cached data that might be valuable to the client. Cote clients are very independent in a sense that they do not have any obligations to store particular data. The downside is that efficient peer-to-peer lookup algorithms, such as Chord [54], can not be used to directly locate specific data. Instead, Cote clients are looking for suitable peers by broadcasting and with the help of the servers.

Clients in the same local network are good candidates for peer-to-peer caching. Low latency and high bandwidth of the network minimizes the overhead and enables efficient sharing of data. Being in the same local network the nodes might also be spatially close to each other. They are possibly part of the same workgroup under a single administrative domain. The supposition is that such nodes are more likely to be accessing the same files and thus benefit from sharing of cached data.

A client can discover peers in the same local network by broadcasting (or by multicasting). Each client willing to participate in peer-to-peer caching is listening to a predetermined default port. The client doing a discovery will broadcast a message specifying its node identifier and contact information (the port it is listening to) together with a list of files it has recently accessed or is about to access. Other clients having parts of the listed files in their cache will respond by sending their identifiers and contact information. The discovery process can be repeated periodically to discover new potential peers.

While broadcasting can be used to discover nodes in the same local network it has the potential to omit many good peers in neighboring networks. Additionally, if the servers are slow and no useful local peers exist then it might be reasonable to cooperate with peers in other more remote networks. This is made possible by server-assisted discovery.

The server-assisted node discovery in Cote has been mostly inspired by BitTorrent trackers which make it possible for BitTorrent peers to discover each

other [13]. Cote servers act like trackers for each individual file they are hosting. For each file a server maintains a list of clients that have recently accessed the file together with last access timestamps. A client can then query the server for a list of peers that have accessed the files the client is currently accessing. Such peers are potential candidates for effective sharing of cached data.

Node queries and lists can be efficiently piggybacked on data transfer requests and responses. A client can decide how often to query for node lists, possibly according to a user-specified upper limit for overhead traffic. To minimize the burden on the server the node lists are kept in volatile limited size memory area. The server purges records of least recent access to make space for new records.

Servers do not store only the node identifier (or its hash) for clients accessing files. A client may tell the server its public IP address and the port it is listening to. The server stores this contact information along with the node identifier hash and passes it on to other clients querying for node lists. Alternatively a client may refuse to publish its contact information causing the server not to record the access at all and "hiding" the client from others. Clients not accepting or not able to accept incoming connections should hide themselves from others. The contact information returned by a server might be outdated but it is only being used as a hint to find other nodes so this does not matter.

The downside of the design is that it requires client-server communication for the client to discover suitable remote peers. If the servers are overloaded or are not available then it is impossible for the clients to find new remote peers. However, they can continue peer-to-peer caching with the peers they already know about. Local peers can always be discovered which enables ad hoc networking without servers.

5.4.2 Managing connections to other nodes

How Cote client maintains the set of established connections to other nodes has been strongly inspired by BitTorrent [13]. The differences are mostly due to fact that not all Cote clients are accessing the same file like the BitTorrent clients using the same tracker do. This makes it more challenging to find a useful peer.

A Cote client typically maintains TCP connections to several peers and servers simultaneously. All file system data is transferred over these connections. Establishing and maintaining a connection involves some overhead and binds some resources. Therefore the number of established connections has a predetermined upper limit and the connected peers and servers should be chosen so that they are able to serve the client efficiently. Because the sequence of future file accesses is not known in advance and the client does not know the cache contents of other peers, heuristics must be used to choose the most promising peers and servers.

A client may drop some existing connections and establish new connections either on demand or periodically. When a client needs to obtain data not available using the current set of established connections it drops the poorest performing connection in favor of a new server connection. The client may also broadcast a query to find new local peers providing the data but the rate of such broadcasts must be limited.

Additionally, a client periodically evaluates the quality of the established connections and drops the poorest performing connection to try new peers or

servers. Performance is monitored continuously and the measurements are evaluated over 30-second periods (configurable). The length of the period was rather arbitrarily chosen, mostly because it is what BitTorrent clients use [13].

Each client can and should also accept incoming connections from its peers. When periodically evaluating the established connections the client drops more than one connection if necessary to leave certain number of connection slots available for incoming connections. Newly established incoming connections are considered for periodic dropping only after they have been active for one full evaluation period.

If there are too many incoming connections over an evaluation period, leaving no connection slots available, then a client starts refusing connections. After a refused connection attempt and remote disconnect there is a period of time during which the peer should not try to connect to the client. A period of 5 minutes (configurable) is reasonable.

5.4.3 Choosing nodes for active cooperation

The heuristics suggested in this section are based on the strong temporal locality present in file access patterns, as evidenced by file system measurements [10], and the assumption of two clients likely accessing same files in the future if they have been previously accessing same files. Additionally, it is assumed that earlier performance of a peer or a server is a good indication of its future performance. The following heuristics can be used to evaluate both peers and servers.

In Cote an important measure is the *hit rate* of a node: the proportion of read operations that were successfully completed. The hit rate is used to estimate the future probability of the node having the data that the client will need. Dedicated servers should have a very high hit rate because a client knows beforehand which files a server should be storing.

Latency associated with operations is difficult to measure continuously because a connection can have several pending operations. On the other hand, if the connection is kept busy by the queued operations then the latency is not an issue. Instead of trying to measure latency and data throughput separately a client calculates the average time it takes to perform a successful operation from the start of the request to the end of the response. For queued operations the time between the end of the previous response and the end of the current response is used instead. This meter, *operation time*, automatically leaves out the latency when it is not that important of a factor.

Overall performance of a node is evaluated by dividing the hit rate with the average operation time. This value represents the estimated maximum rate of successfully performed operations. If there were no operations performed by the node during the evaluation period then the operation time and the hit rate from the previous period is used but the hit rate is halved. If a node has never performed any operations then it has zero overall performance rating.

For recently accessed files, a client remembers which peers have been carrying data from each file. This mapping is updated each time a page of a file is successfully downloaded from a peer or when a server response includes information about a peer which has accessed the same file. The least recently used entries are purged to keep the size of the files-to-peers mapping within predetermined limits.

When the client needs specific data it sends a request to a node that is known to recently have the data and has the highest performance rating. If a node has queued operations its performance is divided by the number of queued operations. This shares the load between well performing nodes when large data transfers are performed.

The above strategy as such would not give newly connected nodes a chance to prove their worthiness. Therefore, every 8th request (configurable) is sent to a random currently connected node with a timeout time similar to the one that would be used for the best performing candidate. In the worst case this wastes about 10% of the network capacity for redundant transfers. Such a randomly targeted request is also sent once before establishing a new connection if none of the connected nodes is known to carry the file. Servers are only being sent requests about files they are known to carry according to the volume header.

5.5 Consistency management

Cote is an optimistically replicated system. The updates are propagated optimistically between the servers, and clients may store updates locally delivering them only later to a server. Clients may also persistently cache data and share the cached data with their peers. Furthermore, a client can be downloading different pages of the same file from different nodes. All this makes consistency management rather challenging.

5.5.1 Enforcing causal ordering

Without any consistency control a user could have an extremely confusing view of a Cote file system. Different pages of a file could come from different sources and different file versions. Also the same page of a file could first come from one file version and then later from an older version.

To simplify consistency management, each Cote client is responsible for enforcing that the local users and applications see file versions in ascending order. This can be implemented using a *version cache*. A client stores the latest known version vector for each accessed file. Only data from this version or from its descendants is accepted by the client. Version cache is periodically written to a disk to make sure that the file system view remains familiar to the user after a restart. The size of the version cache can be limited but it should be big enough to hold version information for files that the user is likely to access again in a near future.

The version cache is also used to determine which branch of a file the client is following. A file version history can be branched because of conflicting updates. Figure 5.3 depicts a branched version history of a file. Only later versions from the same branch should be considered by the client. The selected branch can change when a user manually resolves a conflict, choosing to follow another branch.

To avoid unnecessary data transfers, a client can specify the desired base version when requesting data or discovering peers. Data is only accepted if the received version is equal to the base version or dominates it. If the responding node does not have a suitable file version available then it simply sends a failure response, thus avoiding a useless data transfer.

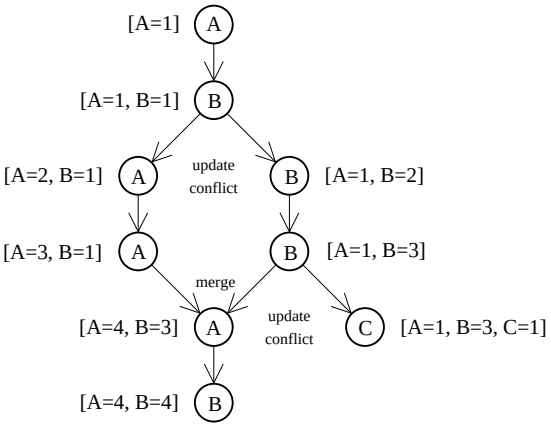

Figure 5.3: An example of a branched file version history. Circles represent individual versions of the file. Each circle includes the label of the producing node and the corresponding version vector is written next to the circle. Initially all nodes are using the same branch. Then an update conflict occurs between nodes A and B creating separate branches for each of them. Later the user of node A resolves the conflict by manually merging her changes to the branch used by node B. However, node C concurrently updates the file, leading to another update conflict and a version branch.

5.5.2 Conflict resolution

The use of version cache means that two nodes having conflicting versions of a file never share the file data. This insulates the different branches of a version history from each other. However, it is important that such conflicts are brought to the attention of users and that they can be resolved. Otherwise different nodes eventually end up using very different versions of a file.

There are two ways to detect a version conflict. When sending a read request a client includes the current base version (if any) in the request. If the other party has a conflicting version it sends a response indicating that no suitable version was found and includes the local version in the response. Both sides then know about the conflict and also know the conflicting version.

Another way to detect a conflict is when client is sending an update to a server. A server only stores one version of a file at a time and rejects conflicting updates. When rejecting such an update the server includes the local version in the response message together with a status code indicating a version conflict.

When a client detects a version conflict, it may notify the user about it. Additionally, the client may persistently store information about the most recently detected alternative version branches. The user can then query the client for alternative version branches of a given file.

To manually resolve a file version conflict, a user makes a copy of the file and then instructs the client to follow another version branch. This is done simply by inserting the desired version in the version cache. Further reads of the file then return information from that branch of the version history. The user can then compare the copy and the file from the currently selected branch. If necessary

the user merges changes from one branch to the other or, alternatively, just picks the branch to be followed.

Version conflicts between servers are detected similarly. Upon detecting a conflict a server should notify the file owner if his e-mail address or other contact information is known to the server. Otherwise the notification is sent to the server administrator. The conflict can be examined similarly by instructing a client to follow one version or the other. The files can also be locally compared by server administrators. A user authorized to update a file can use a special operation to request a server to follow a different version branch. The server will then try to download a suitable version from other servers storing replicas of the file.

Chapter 6

Evaluation

Fully evaluating a distributed file system design involves numerous variables that must be accounted for. Performance and behavior depends on file access patterns, performance characteristics of the participating nodes, the network structure and other network characteristics, among other issues. In that sense the only way to get really qualified data on the performance of a file system is to fully implement it and to do measurements while actually using the file system in an intended way under real circumstances.

Implementing the described design would be beyond the scope of this thesis and reaching the point where the file system could be evaluated in a real large scale setup would require a major effort. Instead, the characteristics of the design are briefly compared to some implemented distributed file systems and evaluated in the scope of the initial requirements. Additionally, some potential performance bottlenecks of the design are studied in more detail. Finally, the results are analyzed. The objective is to find out whether the design has potential and should be developed further.

6.1 Comparison

To put the features in perspective, Table 6.1 shows a comparison of several different file system designs and the design presented in this thesis. NFS [12, 50], AFS [49] and Coda [48] represent client-server oriented file systems. Ficus [26] is a peer-to-peer file system where updates can be propagated through intermediate nodes and Ivy [38] uses shared peer-to-peer storage for file system data.

Cote is a hybrid design combining features of client-server and peer-to-peer architectures. Availability is maximized by using optimistic replication. This enables disconnected operation and ad hoc networking. However, Cote does not provide mechanisms for automatically resolving update conflicts caused by the optimistic approach. Nor does it include mechanisms for maintaining better consistency guarantees when good connectivity is available. The main feature explored by the Cote design is peer-to-peer caching which differentiates it from other distributed file systems.

It should be noted that at this point the Cote design does not cover implementation details and it is likely that some adjustments would be made to the overall design as part of the implementation process.

	NFS	AFS	Coda	Ficus	Ivy	Cote
System architecture (client-server, peer-to-peer)	c-s	c-s	c-s	p-t-p	p-t-p	mixed
Designed mainly for large scale usage		✓	✓	✓	✓	✓
Designed mainly for wide-area usage			✓	✓	✓	✓
Relies on trusted dedicated servers	✓	✓	✓			✓
Supports server/peer redundancy	✓ (NFSv4)	✓	✓	✓	✓	✓
Supports disconnected operation			✓	✓		✓
Supports ad hoc networking				✓		✓
Supports peer-to-peer caching						✓
Uses epidemic update propagation				✓		
Uses shared peer-to-peer storage					✓	
The consistency level offered (close-to-open, varying, weak)	c-t-o	c-t-o	var.	weak	var.	weak
Uses optimistic updates/replication			✓	✓	✓	✓
Automatic update conflict resolution	–	–		✓	✓	
Provides access to old file versions					✓	
Has been implemented	✓	✓	✓	✓	✓	
Widely used in production	✓	✓				–

Table 6.1: A feature comparison of several distributed file systems. Smaller check marks indicate that some support for the feature is available but not to the extent provided by other systems. Varying consistency level means that close-to-open consistency is provided when the system is fully connected but weaker semantics are applied in a partitioned network.

6.2 Fulfilling requirements

Chapter 4 set some requirements for the file system design. This section goes through the requirements evaluating how well the presented Cote design actually fulfills them.

6.2.1 Scalability

A distributed file system is scalable if it can effectively handle more users and more data by simply adding new computing and storage resources to the system. The amount of resources required to serve n users should be $O(n)$.

Adding more data to the system requires more storage capacity. Additionally, more data throughput capacity may be needed. For example, the growing performance of processors together with increased storage and memory capacity has led into bigger and bigger applications and data sets. This means that not only is the amount of stored data increasing but the data throughput is increasing as well.

Cote makes it possible to evenly distribute the data storage and thus also data transfers over several servers. Files are assigned to servers effectively randomly. The required assumptions are that the biggest individual files are significantly smaller than the total storage provided by one server and that the amount of data transferred for any individual file is insignificant compared to the aggregate amount of data transferred per server.

Let there be 2^n servers in the system, each covering $1/2^n$ of the volume data. The files are assigned to servers using a unique n-bit pattern for each server[1]. If the amount of data and data throughput is doubled, the pattern can be extended to $n+1$ bits, effectively halving the coverage of each server to $1/2^{n+1}$ of the volume thus maintaining the same average load per server. The number of required servers is doubled to 2^{n+1} which is in line with the doubling of data storage and data throughput requirements.

Adding new users to the system adds new activity. If the usage patterns remain the same then the aggregate number of file operations performed and the aggregate amount of data transferred grows linearly as new users are added. The previous example then applies also to doubling the number of users. If peer-to-peer caching is effective then less server resources needs to be added. It might also be feasible to double the replication factor instead of halving the volume coverage of each server. This doubles the update propagation traffic between the servers but increases availability.

However, there is one centralized "component": the volume owner. The volume owner must maintain the volume replication table in the volume header and to ensure trustworthiness of the server administrators. Additionally, the volume header must be stored by each participating node and its size grows when new servers are added to the system.

6.2.2 Peer-to-peer caching

Cote design enables peer-to-peer caching without obligating clients to store any file data as per the requirements. However, clients are choosing their active peers based on performance merits, so a client not caching any data or not sharing any cached data is unlikely to benefit from peer-to-peer caching. This serves to provide an incentive for the users to share their resources.

Cote can not use efficient decentralized peer lookup algorithms such as Chord [54] to locate file data because each peer is free to cache any data or not to cache anything. This causes a dependence on servers because clients rely on them to be able to find remote peers. On the other hand, even a large scale Cote system does not need continuous global connectivity to work as intended. Clients in an isolated network segment can operate normally as long as there is a server within the segment. Additionally, the server is likely to return pointers to peers in the same segment, minimizing the effect of disruption on peer-to-peer caching.

The efficiency of the peer-to-peer caching scheme designed for Cote depends on similarities between file access patterns of nodes. If it is a false assumption that two nodes that have accessed same files recently are likely to access same files soon then the described scheme is not efficient. To evaluate the efficiency at least a client implementation is needed. Trace data collected from real distributed file systems could also be used to simulate the efficiency of peer-to-peer caching.

[1] To obtain redundancy with replication factor f the number of servers is increased to $f2^n$ and a unique bit pattern is assigned to each group of f servers

6.2.3 Ad hoc networking without dedicated servers

Ad hoc networking without dedicated servers requires that the files to be accessed are kept in the client cache to make them available when servers are not reachable. A user can choose the files or they can be selected automatically. Algorithms for automatically choosing the files, or *hoarding*, have not been discussed in this thesis but they have been studied by Kistler [32, 31], for example.

Assuming the files to be accessed are persistently cached then a Cote client can serve the data to local peers even if there are no servers available. The clients can then perform updates and deliver the updated data to each other. The update propagation rate depends on the predetermined threshold for data age which forces a client to check the latest version available at a server or among the current peers. The same applies to update propagation via servers.

6.2.4 Hierarchical control and trust

Cote provides security mechanisms to support hierarchical control and trust. The number of components a user must trust is reduced by making it possible for a client to verify the original source of data even when receiving it indirectly via peer-to-peer caching. The servers are trusted based on the certification by the volume owner. Access control framework based on authorization certificates enables hierarchical delegation of authorization. It has also been modelled so that a client can check the validity of an update even when receiving it from a peer.

These features create a control and trust hierarchy with the volume owner at the top. Additionally, graft points make it possible for a volume under one administrative domain to reference other volume under different administration. In effect, the referencing volume is certifying the referenced volume. This allows for more flexibility in large scale systems.

The desired level of availability is achieved through redundant servers and by ensuring that the servers are properly managed when certifying them. Confidentiality is not as well protected as integrity and availability because it depends on clients properly performing access control checks when serving data. However, external security mechanisms such as file encryption can be used to ensure confidentiality.

6.3 Potential performance killers

Cote design uses some techniques that have the potential to limit the performance when compared to traditional straightforward client-server implementations. The efficiency of peer-to-peer caching is not a limiting factor in itself because a client can always use a dedicated server. However, supporting peer-to-peer caching scheme and optimistic replication requires some features that affect the performance of client-server operations as well.

Firstly, all data received from servers is signed. The signature is not only calculated over the file data but it also includes a timestamp. This means that a server must calculate a new signature for each page of data it delivers. Secondly, version vectors are used extensively to ensure consistency and to detect update conflicts. This requires frequent version vector comparisons and updates.

The performance effect of these features was evaluated to make sure that it is feasible to build a system based on them. Additionally, the effect of using one-way content hashes to generate volume and file identifiers was evaluated although calculating a content hash is a light operation compared to calculating a signature.

6.3.1 Performing the measurements

The following measurements were performed on a 3-year-old desktop computer having an AMD Athlon 1.2 GHz processor. Memory capacity was not a limiting factor because the data sets were chosen so that the test programs would operate within the limits of the available physical memory. Both tasks to be examined, data signing and version vector processing, are processor intensive. No disk or network activity was associated with the timed tests.

The computer was running a Debian GNU/Linux operating system. Standard included versions of development tools and utility libraries were used to implement short test programs for the measurements. Performance was evaluated by measuring the real elapsed time for each test round using `gettimeofday()` from within the test program. No other activity was taking place during the tests (except for the possible background activity by the kernel and the X environment).

All tests were repeated 10 times for each set of parameters. The results were averaged and their standard deviation was calculated to check that there were no significant anomalies due to possible background activity.

6.3.2 Data signing

The performance effect of data signing was measured by generating 16 megabytes of pseudo-random data (read from `/dev/urandom`) and then signing it page by page using different page sizes. The RSA public key algorithm with a 1024-bit key was used and the content hashes were calculated using MD5. These algorithms were chosen for the evaluation because they are well-known and commonly used for public key cryptography. Implementations for the algorithms were provided by the *crypto* library (Version 0.9.7e, Debian revision 2) from the OpenSSL project[2].

According to the Cote design, servers are signing lot of data (every page they deliver to clients) and clients have to verify signatures (when they receive cached data pages from their peers). The performance effect of data signing was evaluated by measuring the elapsed time for calculating and signing the per-page content hashes. Afterwards, these pages together with the signatures were written to a file and they were used as an input to a signature verification test. The effect of signature verification was evaluated by measuring the elapsed time for verifying the signatures.

Results are presented in Figure 6.1 in the form of data throughput achieved for the signing and signature verification processes. Content hash calculation is a light operation compared to generating or verifying signatures. Therefore the signing and the verification throughput increases when the page size grows (less signatures are needed for the same amount of data).

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

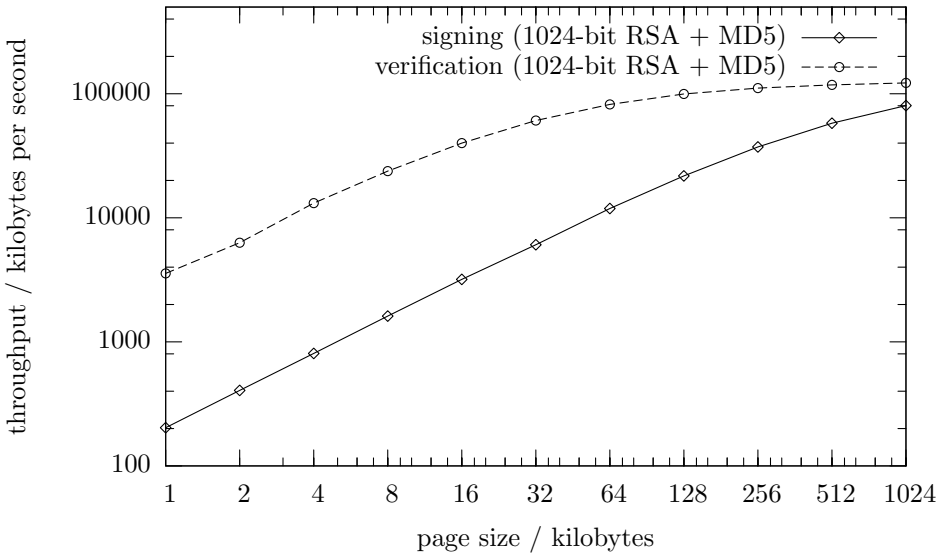

Figure 6.1: Data signing and signature verification performance. 16 megabytes of pseudo-random data was repeatedly signed and verified using different page sizes. The average data throughput is given for each page size.

```
#define NODE_ID_LENGTH MD5_DIGEST_LENGTH

typedef unsigned char vv_node_id[NODE_ID_LENGTH];
typedef unsigned long vv_node_time;

struct vv_entry {
    vv_node_id node_id;
    vv_node_time node_time;
};

struct vv_vector {
    unsigned int capacity;
    unsigned int length;
    struct vv_entry *entries;
};
```

Figure 6.2: The data structures used in the version vector implementation.

6.3.3 Version vector operations

Cote uses version vectors extensively to enforce consistency and to detect update conflicts. The performance effect of using version vectors was evaluated by implementing the basic version vector operations [45]. The data structures used are illustrated in Figure 6.2.

16-byte (128-bit) node identifiers were used to allow the use of MD5 hashes of node public keys as identifiers. This makes it possible to create a new Cote node with a globally unique and provable identifier without any coordination among the nodes. However, it also increases the size of the version vectors. The memory footprint sizes of the version vectors used in the measurements are shown in Figure 6.3.

The implemented operations include version vector updates and comparisons. Version vectors must be updated when new versions of files are created. If it is the first time a node updates the file then a new node-specific entry is added to the version vector. Otherwise the node-specific entry is updated to reflect the current timestamp value. Version vectors are compared to find out if a file version is equal to, a predecessor of, an ancestor of or in conflict with another version. These comparisons are done when receiving data and processing read or update requests.

When implementing the algorithm, version vector comparisons were favored over updates because they are more frequent. Entries within the vector were ordered by the node identifier making it possible to use binary search to locate a specific entry in $O(\log_2 n)$ time and to efficiently compare two vectors in $O(n)$ time.

Performance was evaluated for differently sized version vectors because the size of a version vector depends on how many nodes have updated the file in the past. Version vectors used in the measurements were created using pseudo-random data. A set of one million random vectors (or vector pairs for comparisons) was used when testing performance with 1-entry vectors, a quarter million vectors for testing performance with 4-entry vectors and so on.

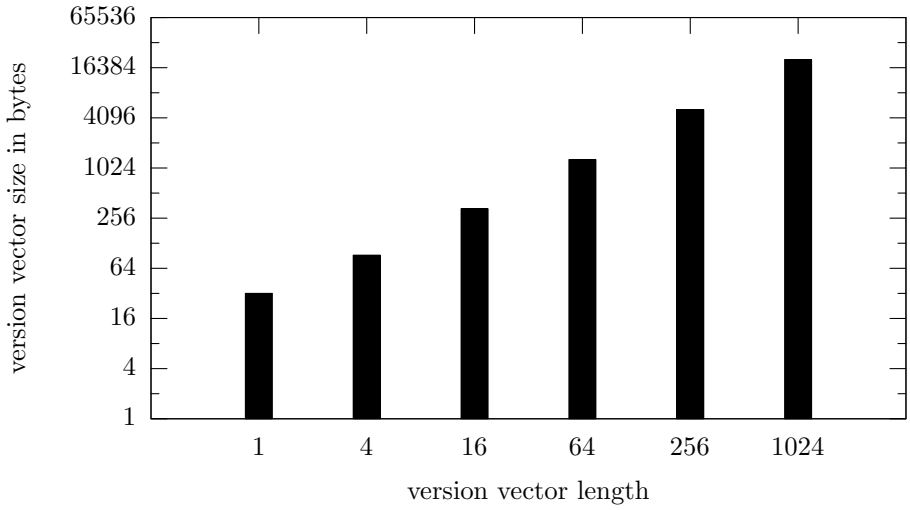

Figure 6.3: Memory footprints of the version vectors used in the measurements. The data structures used for the tests included 12 bytes for the version vector header and 20 bytes for each entry within the vector. A 16-byte MD5 hash of a node public key was used as the node identifier.

Figure 6.4 shows the results for version vector updates. Updates of an existing entry (a node has previously updated the file) and addition of new entries (a node is updating the file for the first time) were measured separately because they perform differently. The latter case requires that entries coming after the new node identifier are being pushed further, requiring a memory copy operation. This could have been avoided by using a more elegant data structure but the simple version was good enough for this performance analysis.

The notable 10-fold performance drop in entry additions when going from 256 entries to 1024 entries is probably due to memory architecture issues affecting memory copy operations (for example, running out of level 1 (L1) cache).

Comparisons were measured by comparing two random vectors and by comparing two equal vectors. In the former case the comparison produces a conflict rather quickly while the latter case completes only after fully comparing the two vectors entry by entry. The vectors to be compared were not stored next to each other in memory to make the comparison performance more realistic. These results are shown in Figure 6.5.

6.3.4 Identifier generation

Cote uses a one-way hash function to generate volume and file identifiers. The hash is calculated over a tuple containing a public key of the creating entity and a number identifying the object among the objects created by the entity.

Identifier generation performance was measured by generating identifiers based on a 1024-bit (128-byte) key and a value obtained from a 64-bit counter

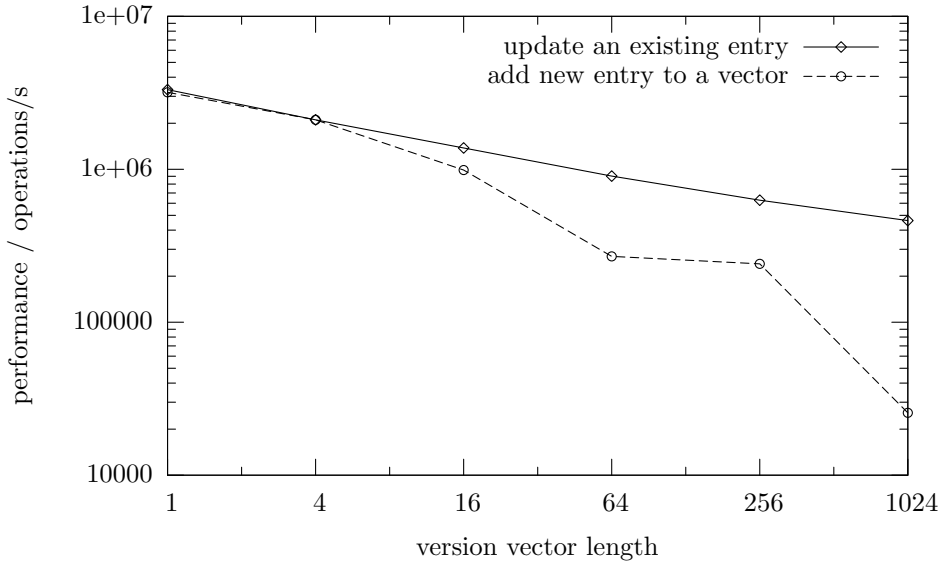

Figure 6.4: Version vector update performance. Updates of existing entries and additions of new entries were measured separately.

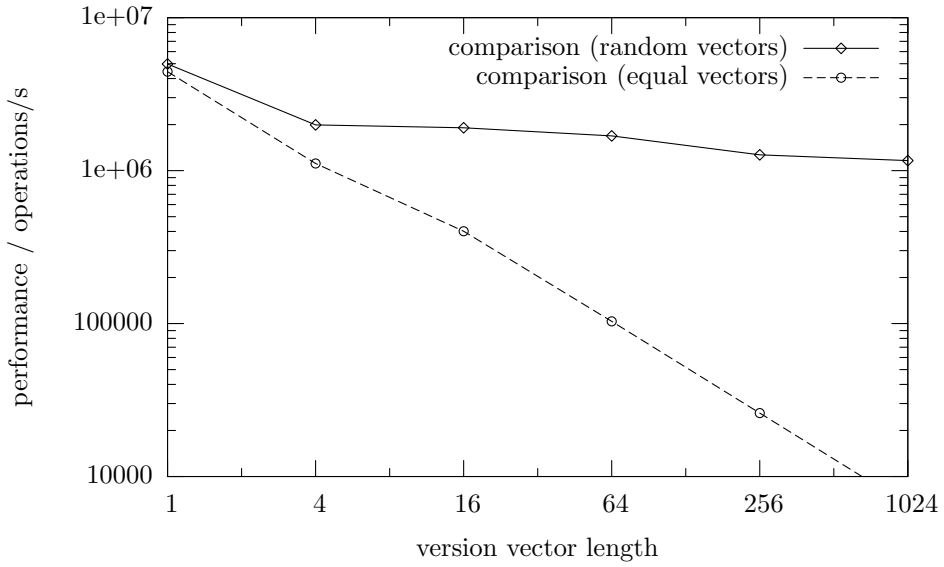

Figure 6.5: Version vector comparison performance. Comparisons of two random vectors and two equal vectors were measured separately.

which was advanced after each generated identifier. MD5 was used as the hashing algorithm and the implementation was provided by the same *crypto* library used for the data signing measurements above. The elapsed time for generating one million different identifiers was measured.

The result is that, on average, more than 509 000 identifiers were generated per second.

6.4 Analysis

The previous evaluation showed that the Cote design fulfills the core requirements set for it, at least in theory. This section will analyze the results and measurements further to get more insight into the weaknesses of the design and to determine if the potential performance killing features are a real problem or not.

6.4.1 Weaknesses and possible improvements

There are several aspects of the design that could be improved. The most obvious weaknesses are related to the characteristics of the chosen architecture, the nature of optimistic replication and the security mechanisms used. Many of the issues are due to compromises that need to be done and solving them might introduce new weaknesses.

Scalability

Cote depends on trusted well-managed server environments. The volume owner must ensure that the servers and their administrators are trustworthy. This requires increasing effort as more servers are needed. Thus, while the technical aspects of Cote are scalable according to the standard definition [15], manageability of the volume sets an upper limit to its size. On the other hand, a large volume is likely to have an organization as the volume owner, instead of a single person.

In this regard all file systems that depend on dedicated trusted server environments have limited scalability compared to fully peer-to-peer systems like Ivy. It is a compromise that allows for better control.

Automated tools could be implemented to streamline the maintenance procedures. Cote also provides a way to graft volumes onto each other, providing scalability by dividing a large file system into more manageable components. A volume does not need to be big. Given the right tools it is feasible to even have a separate personal volume for each user [49]. This makes it also possible to control quota at the volume level [49].

Update propagation

Due to the optimistic nature of replication Cote is not well suited for situations where several nodes are updating the same files intensively. The update propagation based on cache timeouts can also cause problems when trying to cooperate in real time. Users might need to manually trigger a cache refresh to see the recent remote updates.

One way to avoid these problems would be to use update callbacks like AFS does [49]. However, applying the approach as is to peer-to-peer caching would require that peers allocate more resources to remember the pages they have served to each other. If a client is fetching pages of the same file from multiple sources then it would also require multiple callbacks to keep the cached data up-to-date. This might limit the efficiency of the suggested peer-to-peer caching scheme.

Callbacks can not be efficiently used during disconnected operation and ad hoc networking because the original source of data is often not reachable. However, similar mechanisms could be used among the peers forming the ad hoc network. Haddock-FS [8], a file system specifically targeted for ad hoc networking, provides adaptive consistency semantics by enforcing a single-writer multiple-readers token scheme within a group of peers when high connectivity is available. Cote could be extended to support adaptive consistency.

Access control

In Cote, data confidentiality depends on the clients performing access control checks correctly when serving data to their peers. It is reasonable to question whether the clients can be trusted to perform access control checks. However, it would be impossible to prevent a malicious user from distributing data that he has already obtained, anyway. A real risk is that a client implementation has a security bug allowing unauthorized access, especially if there are various client implementations available from different sources.

Applying *cryptographic access control* [27] would solve the confidentiality issues. Cryptographic access control would also reduce the need to trust the servers because it would make it impossible for a malicious server to breach confidentiality or integrity of data. However, it does not prevent removal of data or updates being ignored so the servers must still be trusted to provide a reliable persistent storage.

6.4.2 Overhead

It can be expected that implementing peer-to-peer caching in a way that does not compromise security necessarily involves some overhead compared to a straightforward client-server architecture. Without implementation it is difficult to estimate whether the possible benefits compensate for this additional overhead. Therefore the overhead is analyzed by comparing it to traditional client-server systems, without assuming anything about savings obtained through peer-to-peer caching.

This analysis is based on the earlier measurements performed on a desktop computer. It is worth noting that overhead is even more significant for portable devices because it translates directly to larger power requirements.

Data signing

Data signing is a significant burden to the servers. It can be made less of an issue by using large pages but there are practical limits for a page size. Using too large pages wastes network bandwidth and disk space because whole pages are always transferred and stored. 4 KiB pages are commonly used in distributed

file systems targeted for local area networks but in Cote this page size is out of question because of the involved overhead. However, considering that a low-end desktop computer could in theory almost saturate a 100 Mbps[3] network link using 64 KiB pages, data signing overhead does not appear to be an invincible obstacle.

The Digital Signature Standard (DSS) which uses the DSA and SHA-1 algorithms might be a better option for Cote because it is known to perform better for generating the signatures (but worse for verifying them). This would reduce server load.

Data signing overhead can also be reduced by using coarser granularity for timestamps. For example, if the timestamps have one hour granularity then a server can cache and reuse a signature if the same data block is to be served repeatedly during the hour.

Version vectors

Reasonably sized version vectors are not a performance killer considering processor usage. Quite contrary, they perform well even when using 128-bit node identifiers. However, the version vectors must also be stored and transferred. If a page size of 4 kilobytes would be used then a 64-entry version vector would already cause 24% overhead for data transfers. This potential overhead is alleviated by only transferring changed vector components over the course of a session as described in Section 5.3.4.

For a long-lived file being updated by numerous nodes the size of the version vector might become an issue. Applying a pruning algorithm like that suggested by Saito [45] would require that no separate version branches older than some upper time limit exists. This would require that all conflicts are resolved within a predetermined time. It is not straightforward to implement such a scheme in Cote but with adjustments, it might be doable.

Identifier generation

Generating file identifiers using a one-way hash algorithm is not a bottleneck in Cote. It causes some overhead but the overhead is insignificant compared to signing the file creation requests before delivering them to a server.

[3] 100 Mbps is about 12200 kilobytes per second.

Chapter 7

Conclusion

A distributed file system is a convenient way to share data among a large group of people [49] and to enable user mobility [26, 48]. It can be integrated transparently to the file system interface making the remote data as easily available as local data. Most of the problems involved with designing and implementing distributed file systems can be directly attributed to the distributed nature [15] of the system.

There has been a lot of research on the different aspects of distributed file systems [34, 47]. Recent research has focused on large scale, even global scale [43], systems and on mobile clients [8]. Peer-to-peer semantics have been applied to utilize the aggregate processing power, storage capacity and network capacity of the participating computers [1, 17, 38]. Nevertheless, the well-known and widely adopted distributed file systems are still based on the traditional client-server architecture [12, 50, 49].

This thesis first explored the theory and practical problems related to the distributed file systems in general. Chapter 2 gave an overview of the various issues and some rationale for developing distributed file systems in the first place. Ensuring file system consistency is one of the most challenging problems. It was the focus of Chapter 3.

A new file system design was presented in Chapter 5 based on the use case and requirements set forth in Chapter 4. The design relies in many ways on prior research and related work presented in earlier chapters. It provides the manageability of dedicated server environments while allowing workstations to cooperate in a peer-to-peer fashion. The design combines these two partly conflicting approaches by taking the idea of peer-to-peer caching [53, 51, 22] from web serving and applying it to distributed file systems.

It was demonstrated that such a hybrid system can be designed. The evaluation in Chapter 6 showed that the design fulfills the core requirements. The overhead of potential performance killing features was measured. Data signing was found to cause significant overhead which can be alleviated by using large pages and coarser timestamps.

The actual performance of the proposed peer-to-peer caching scheme is highly dependant on file system usage characteristics and file access patterns. A fully featured implementation would be needed to make realistic measurements about the performance effects of the caching scheme. The potential of peer-to-peer caching should be explored further.

7.1 Further work

The development of mobile devices and wireless communication technologies is likely to drive the development of distributed file systems towards systems supporting ad hoc networking and peer-to-peer cooperation. Haddock-FS [8] is an example of a recently developed file system targeted specifically for ad hoc networks and mobile devices. The varying level of connectivity in such environments has also motivated research on adaptive consistency models [16, 18] and epidemic update propagation schemes [19].

The development of large scale file systems will also continue. While some designs have been suggested for maintenance-free global storage [43] they have yet to be fully implemented and deployed. It will be interesting to follow the development of file systems based on distributed peer-to-peer storage [1, 3, 17, 38, 44]. Further research topics are also being brought up due to the security aspects of untrusted large scale systems. These include decentralized methods for access control [27] and authentication of users [30].

I hope I will have a chance to further develop and improve the design presented in this thesis. A prototype implementation is needed to get some real data about the performance characteristics of the suggested peer-to-peer caching scheme. The prototype is also required to develop better heuristics for choosing the peers to interact with.

The presented design could be used to implement a scalable client-server file system without applying any client level peer-to-peer semantics. This might be a good starting point for exploring the implementation details and to gradually implement required functionality.

Another approach would be to first develop a version for use in trusted local networks. Many security mechanisms could be left out from such a version. This would make it possible to experiment with peer-to-peer caching without having to worry about the overhead caused by data signing.

My intention is to make possible further documentation, plans and source code available at the Cote file system web site: `http://www.cotefs.org/`.

Bibliography

[1] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, Massachusetts, USA, December 2002.

[2] Rafael Alonso and Henry F. Korth. Database system issues in nomadic computing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 388–392, Washington, D.C., USA, May 1993.

[3] Mattias Amnefelt and Johanna Svenningsson. Keso — a scalable, reliable and secure read/write peer-to-peer file system. Master's thesis, Royal Institute of Technology, Stockholm, Sweden, May 2004.

[4] Ross J. Anderson. The Eternity service. In *Proceedings of Pragocrypt '96*, pages 242–252, Prague, Czech Republic, 9–10 1996. CTU Publishing.

[5] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 109–126, Copper Mountain, Colorado, USA, December 1995.

[6] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 198–212, Pacific Grove, California, USA, October 1991.

[7] Gerco Ballintijn, Maarten van Steen, and Andrew S. Tanenbaum. Characterizing Internet performance to support wide-area application development. *Operating Systems Review*, 34(4):41–47, October 2000.

[8] João Barreto and Paulo Ferreira. A replicated file system for resource constrained mobile devices. In *Proceedings of the IADIS International Conference on Applied Computing*, Lisbon, Portugal, March 2004.

[9] T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifiers (URI): Generic Syntax.* RFC2396, IETF, August 1998.

[10] Mathew Blaze. *Caching in large-scale distributed file systems.* PhD thesis, Princeton University, Princeton, New Jersey, USA, January 1993.

[11] D. R. Brownbridge, L. F. Marshall, and B. Randell. The Newcastle Connection. *Software-Practice and Experience*, 12:1147–1162, 1982.

[12] B. Callaghan, B. Pawlowski, and P. Staubach. *NFS version 3 protocol specification.* RFC1813, IETF, June 1995.

[13] Bram Cohen. Incentives build robustness in BitTorrent. Presented in Workshop on Economics of Peer-to-Peer Systems, June 2003. `http://bittorrent.com/bittorrentecon.pdf` (March 2005).

[14] Brian Cornell, Peter A. Dinda, and Fabián E. Bustamante. Wayback: A user-level versioning file system for Linux. In *Proceedings of the USENIX 2004*, pages 19–28, Boston, Massachusetts, USA, 6–7 2004.

[15] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: Concepts and design.* Pearson Education, 3 edition, 2001.

[16] Simon Cuce and Arkady Zaslavsky. Supporting multiple consistency models within a mobility enabled file system using a component based framework. *Mobile Networks and Applications*, 8(4):317–326, August 2003.

[17] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 202–215, Banff, Canada, October 2001.

[18] Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. PRACTI replication for large-scale systems. Technical Report UTCS-04-28, University of Texas at Austin, Austin, Texas, USA, June 2004.

[19] Anwitaman Datta, Manfred Hauswirth, and Karl Aberer. Updates in highly unreliable, replicated peer-to-peer systems. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003.

[20] Roger Dingledine, Michael J. Freedman, and David Molnar. The Free Haven Project: Distributed anonymous storage service. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, pages 67–95, Berkeley, California, USA, July 2000.

[21] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylönen. *SPKI Certificate Theory.* RFC2693, IETF, September 1999.

[22] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with coral. In *Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation*, San Francisco, California, USA, March 2004.

[23] Kevin Fu, M. Frans Kaashoek, and David Maziéres. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 181–196, San Diego, California, USA, October 2000.

[24] Andrew Goldberg and Peter Yianilos. Towards an Archival Intermemory. In *In Proceedings of the IEEE International Forum on Research and Technology Advances in Digital Libraries*, pages 147–156, Santa Barbara, California, USA, April 1998.

[25] Björn Grönvall, Assar Westlund, and Stephen Pink. The design of a multicast-based distributed file system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 251–264, New Orleans, Louisiana, USA, February 1999.

[26] Richard G. Guy. *Ficus: A very large scale reliable distributed file system.* PhD thesis, University of California, Los Angeles, California, USA, June 1991. Also available as UCLA Technical Report CSD-910018.

[27] Anthony Harrington and Christian Jensen. Cryptographic access control in a distributed file system. In *Proceedings of the 8th ACM Symposium on Access Control Models and Technologies*, pages 158–165, Como, Italy, June 2003.

[28] Gandalf Hernandez. Secure digital archiving. Master's thesis, Royal Institute of Technology, Stockholm, Sweden, 2004.

[29] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[30] Michael Kaminsky, George Savvides, David Mazières, and M. Frans Kaashoek. Decentralized user authentication in a global file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, New York, USA, October 2003.

[31] J. J. Kistler. *Disconnected operation in a distributed file system.* PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, May 1993.

[32] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[33] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, Massachusetts, USA, October 1996.

[34] Eliezer Levy and Abraham Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22(4):321–374, December 1990.

[35] David Mazières. *Self-certifying file system.* PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, May 2000.

[36] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, pages 124–139, Kiawah Island, South Carolina, USA, December 1999.

[37] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 174–187, Banff, Canada, October 2001.

[38] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, Massachusetts, USA, December 2002.

[39] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 15–24, Orcas Island, Washington, USA, December 1985.

[40] Philippe Owezarski and Nicolas Larrieu. An analysis of Internet traffic characteristics and related issues. Technical Report N°03496, Laboratoire d'Analyse et d'Architecture des Systèmes, Centre National de la Recherce Scientifique, Toulouse, France, November 2003.

[41] T. W. Page, Jr, R. G. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, and G. J. Popek. Perspectives on optimistically replicated peer-to-peer filing. *Software Practice and Experience*, 28(2):155–180, February 1998.

[42] Peter Reiher, John Heidemann, David Ratner, Greg Skinner, and Gerald Popek. Resolving file conflicts in the Ficus file system. In *Proceedings of the Summer USENIX Conference*, pages 183–195, June 1994.

[43] Sean Rhea, Chris Wells, Patrick Eaton, Dennis Geels, Ben Zhao, Hakim Weatherspoon, and John Kubiatowicz. Maintenance-free global data storage. *IEEE Internet Computing*, 5(5):40–49, September 2001.

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

[45] Yasushi Saito. Unilateral version vector pruning using loosely synchronized clocks. Technical Report HPL-2002-51, Hewlett-Packard Laboratories, Palo Alto, California, USA, March 2002.

[46] Yasushi Saito and Marc Shapiro. Optimistic replication. Technical Report MSR-TR-2003-60, Microsoft Research, September 2003. Will be superceded by article to appear in ACM Computing Surveys.

[47] M. Satyanarayanan. A survey of distributed file systems. Technical Report CMU-CS-89-116, Carnegie-Mellon University, Pittsburgh, Philadelphia, USA, February 1989.

[48] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[49] Mahadev Satyanarayanan. Scalable, secure, and highly available distributed file access. *Computer*, 23(5):9–21, May 1990.

[50] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. *NFS version 4 protocol*. RFC3010, IETF, December 2000.

[51] Tyron Stading, Petros Maniatis, and Mary Baker. Peer-to-peer caching schemes to address flash crowds. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, Cambridge, Massachusetts, USA, March 2002.

[52] William Stallings. *Operating Systems*, chapter 11. Prentice-Hall International Inc., second edition, 1995.

[53] Angelos Stavrou, Dan Rubenstein, and Sambit Sahu. A lightweight, robust P2P system to handle flash crowds. Technical Report EE020321-1, Columbia University, New York, New York, USA, February 2002.

[54] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, pages 149–160, San Diego, California, USA, August 2001.

[55] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 224–237, October 1997.

[56] Werner Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 93–109, Charleston, South Carolina, USA, December 1999.