CapaFS: A globally accessible file system

Jude Thaddeus Regan

A dissertation submitted to the University of Dublin,

in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

1999

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed: ____________________

Jude Thaddeus Regan

17/9/99

Permission to lend and/or copy

I agree that Trinity College Library may lend or copy this dissertation upon request.

Signed: ____________________

Jude Thaddeus Regan

1999

Acknowledgments

I would like to thank Christian Jensen for all his help and support throughout the year, my Parents, TJ, Louise and The Butcher's Anvil for putting up with me.

Summary

We have designed and implemented a reference implementation of CapaFS, a global, decentralised file system that allows users to collaborate with other users anywhere in the world, with no prior arrangements or connections. The system uses filenames as sparse capabilities to name and grant access to files on remote servers. Users can share files by exchanging capability filenames with parties that they trust. Possession of a capability filename is the necessary and sufficient proof of authority to perform the operations, authorised by the capability on the file it names. CapaFS does not need to establish trust between client and server, it only needs to verify the validity of the CapaFS capability filename.

By using CapaFS we can demonstrate a globally accessable file system with a uniform namespace, that does not need any central authority to manage the namespace. We show that the capability filenames used in CapaFS are a secure and practical solution to sharing resources. We also prove the reference implementation of CapaFS can offer reasonable performance when compared with systems like NFS. The results of CapaFS show that the cost of software encryption can be tolerated in large networks like the Internet, where the cost of encryption is a small percentage of the total delay. CapaFS offers new functionality to users which sucessfully promotes the sharing of files and collaboration on the Internet, without imposing significant costs in terms of overhead and performance.

Table of contents

1 INTRODUCTION 4
1.1 Motivation for a globally accessible file system 5
1.1.1 Problems with centralised control 6
1.1.2 Certification Authorities 6
1.1.3 Crossing administrative Boundaries 7
1.2 CapaFS Name space 7
1.2.1 Host-based Naming 8
1.2.2 User-centred Naming 8
1.2.3 Global Naming 9
1.2.4 Global Access 9
1.3 Overview of the CapaFS Architecture 10
1.3.1 Kernel Space 11
1.3.2 User Space 11

2 STATE OF THE ART 13
2.1 WebFS 14
2.2 SFS 15
2.3 Echo FS 16
2.4 AFS 17
2.5 Truffles 18

3 DESIGN 20
3.1 Design Goals 20
3.2 Client-Server model 21
3.3 Capabilities 22
3.3.1 The Principle of least privilege 22
3.3.2 Capability properties 23
3.3.3 Sparse Capabilities 24
3.3.4 Capabilities in other systems 24
3.3.5 Format of CapaFS capability filenames 25
3.4 Encryption Design 26
3.4.1 Why Use RSA Encryption? 26
3.4.2 CapaFS Security 26
3.4.3 Difficulty of breaking RSA 28
3.4.4 The Mathematics behind of RSA Encryption 30
4 IMPLEMENTATION 32

4.1 CapaFSKeys 33
4.2 CapaFSFile 34
4.3 CapaFSLIB 36
4.3.1 The client open operation 36
4.3.2 The client close operation 37
4.3.3 The client read operation 38
4.3.4 The client write operation 38
4.3.5 The client lseek operation 39
4.3.6 The client fcntl operation 40
4.4 CapaFSMOD 40
4.4.1 Virtual File System 42
4.4.2 The CapaFS VFS Superblock 44
4.4.3 CapaFS VFS inode 45
4.4.4 Registering CapaFS with the VFS 46
4.4.5 Mounting CapaFS 47
4.5 CapaFS server 48
4.5.1 Server Open 49
4.5.2 Server Close 51
4.5.3 Server Read 52
4.5.4 Server write 53
4.5.5 Server lseek 53
4.5.6 Server fcntl 53
4.6 CapaFSH 54

5 EVALUATION 55
5.1 Internet Test Domain 56
5.2 Differences between CapaFS and NFS 56
5.3 Encryption Performance 57
5.4 RPC Performance 59
5.5 Performance evaluation of CapaFS and NFS 59

6 CONCLUSION 62
6.1 Where does CapaFS fit in? 63
6.2 Related Work 64
6.3 Future Work 65
6.4 Conclusion 66
References 67

Tables and illustrative material

Figure 1.1: Sites sharing files using CapaFS. 10

Figure 1.2: Overview of CapaFS Architecture 12

Figure 2.1: Summary of related file system properties 19

Figure 3.1: CapaFS Protocol Stack on Client and Server 21

Figure 3.2: Format of CapaFS capability filenames 25

Figure 4.1: Creating a CapaFS capability filename 32

Figure 4.2: The process of creating a CapaFS capability filename 35

Figure 4.3: CapaFS kernel module framework 41

Figure 4.4: Logical view of the Virtual File System 42

Figure 4.5: Registered file systems 46

Figure 4.6: A registered and mounted CapaFS file system 47

Figure 4.7: Format of Linux password file 49

Figure 4.8: Overview of the CapaFS open command 51

Figure 5.1: Decrypting capability filenames of varying strengths 58

Figure 5.2: Comparison of the various RPC libraries 59

Figure 5.3: Timing of remote test cases 60

Figure 5.4: Performance of CapaFS and NFS over the Internet 61

Figure 6.1: Where does CapaFS fit in? 64

Figure 6.2: CapaFS future work 65

Abbreviations

ACL	Access Control List
ADSL	Asymmetric Digital Subscriber Line
AFS	Andrew File System
CA	Certification Authority
CPU	Central Processing Unit
DNS	Domain Name Server
EOF	End Of File
FTP	File Transfer Protocol
HTTP	Hyper-Text Transfer Protocol
IP	Internet Protocol
LAN	Local Area Network
LIP	Large Integer Package
NFS	Network File System
OID	Object ID
PEM	Privacy Enhanced Mail
PPP	Point-to-Point Protocol
RPC	Remote Procedure Call
SFS	Secure File System
TCP	Transmission Control Protocol
TTP	Trusted-Third-Party
UID	User ID
UFO	User-level File Organizer
VFS	Virtual File System
WAN	Wide Area Network
XDR	eXternal Data Representation

1. INTRODUCTION

Today, businesses work in a global marketplace without time and geographic boundaries. International collaboration, Internet commerce, a mobile workforce and scarce resources are today's realities. The tie that binds this globally scattered business infrastructure is shared information. To remain competitive in the global marketplace, a company's access to this information must be fast and direct, regardless of where the information is stored or where it is being delivered. As a result, seamless, convenient shared access to information has never been more critical to supporting the business goals of an organisation. Engineers and scientists in large technical organisations need to operate with no boundaries. They need to obtain information any time, any place, on any computer platform. Although many businesses could see this as a threat, secure and safe sharing of resources will benefit any company.

Users benefit greatly from being able to work co-operatively, but they are often limited by constraints of geography, administrative boundaries, and the existing state of distributed systems. CapaFS allows geographically dispersed organisations continual access to the latest versions of files so they can share project data quickly and easily. Since much data is stored as files, the ability to share such data flexibly and securely will greatly facilitate the performance of co-operative work. Sharing data and setting up shared environments across networks like the Internet are difficult tasks. File-sharing for this environment must handle some difficult problems, such as limited trust between the sharing parties, failures of the communications media

between partners, difficulty of setting up the shared environment, and performance issues.

1.1 Motivation for a globally accessible file system

Most commercially available file systems allow users to share files using discretionary access control, which means that the specification of access rights is left to the discretion of the user who owns the file. However, in order to allow sharing fand enforce access control, the file system relies on a local database or password file of users in the system. This means that sharing is only allowed among users defined in this database. People collaborating across local-area networks usually work within the same organisation, limiting the incentive for malicious behaviour.

It is often useful for users to share files with remote or semi-anonymous users. Some examples of this type of sharing are: roaming mobile users who share files from their home site with the people they are visiting, or ad hoc workgroups that span administrative boundaries (either departments in a large company or companies in a virtual enterprise.). CapaFS facilitates collaboration by allowing users to grant remote or semi-anonymous users access to their files. An example of this is when two scientists with similar interests meet virtually on the Internet and want to collaborate on a project. The problem is, neither of their system administrators will create a user account for a semi-anonymous user. Using CapaFS, a scientist could create a capability filename, which references the resource to be shared and then simply give it to someone they trust.

1.1.1 Problems with centralised control

Central authorities have been trusted to coordinate secure collaboration on local area networks. This cannot be done on the Internet. Global distributed systems cannot rely on central authorities for security, without subjecting their users to rigid centralised control. The solution is a global accessible file system with no central authority, where nobody controls the namespace and you can grant access to remote machines that you trust. When dealing with shared resources, authentication is an essential part of any system. User accounts cannot be accessed without the proper authority. So whoever controls the authentication controls the creation of new users or resources. Giving this control to a central authority hinders deployment, complicates cross-administrative realm collaboration, excludes valid network resources, creates a single point of failure and prevents innovation. A centrally controlled name space hinders deployment because servers cannot become on-line until they get approval from the naming authority. Centralised control requires authentication before authorisation. We propose CapaFS, a decentralised file system with global authentication and flexible authorisation.

1.1.2 Certification Authorities

Certification authorities like Verisign, empose a form of centralised control. Systems, which must have a digital certificate to function, place their users under the control of the certificate authority. The deployment of secure HTTP has been restricted due to the problem of obtaining digital certificates from a central certificate authorities.

While most people should prefer to run secure web servers, the hassle of obtaining digital certificates from central authorities has kept most web servers from running secure HTTP. CapaFS provides a global network service with decentralised control of the name space. The growth of the World-Wide-Web in recent years shows the benefits and desire for decentralised control of shared resources.

1.1.3 Crossing Administrative Boundaries

Another form of centralised control results from systems, which penalize cross-administrative realm use. To facilitate collaboration, users of such systems form inconveniently large administrative realms, so anyone they may want to collaborate with should be in the same realm. Kerberos [1] authentication suffers from this problem. System administrators responsible for setting up user accounts often could not do so without the intervention of the Kerberos administrator. AFS [2] uses Kerberos for authentication. Administrators of AFS client machines must enumerate every single file server the client can talk to. A user of an AFS client cannot access a server the administrator did not include. CapaFS avoids the problem of centralised control by implementing a file system with a global name space and global access.

1.2 CapaFS Name space

When designing CapaFS we wanted to ensure objects in the system could be named and found easily without any penalty for cross-administrative realm use. There are many approaches to naming files in distributed systems including: host-based naming, which forces users to remember file locations, user-centered naming allows users to

choose what is in their name space and global naming ensures each filename has global meaning within the system.

1.2.1 Host-based Naming

Host-based naming or more generally context-dependent naming is used to identify files or directories on remote machines. NFS [3,4] and FTP [5] both use host-based naming. A problem with this approach is it requires the user to know the name of the host on which the desired object resides, in order to access it. Host-based naming is easy to implement but makes it difficult to locate and organise information.

1.2.2 User-centered Naming

There is a huge amount of information on the Internet, the vast majority of which is irrelevant to a specific user. User-centred naming allows a user to choose what their name space consists of. Allowing users to select what remote objects are in their name space reduces clutter and disorganisation and increases relevancy. Jade [6] Prospero [7], and Amoeba [8] employ this approach. A problem with user-centred naming is that the same name may refer to different objects when used in different name spaces, making sharing difficult.

Jade, for example, creates a collection of small, per-user name spaces and a Jade filename has scope relative to a single logical name space. Amoeba capabilities uniquely identify an object by the server port and an OID local to that server. This reference is valid in all name-spaces. Prospero avoids this problem by supporting

closure. That is, every object has a name space associated with it and is resolved in that name space.

1.2.3 Global Naming

We choose global naming for CapaFS because it solves this problem by making all names part of a single name space. A user can refer to a CapaFS capability filename, and be confident that it will have the same meaning to every other user. Global naming allows the creation of links to remote objects, and without global naming users must be aware of the context in which a name is meaningful. CapaFS uses global naming as well as AFS, Locus and Sprite [9].

1.2.4 Global Access

Global access requires that files and directories must be accessible everywhere throughout the system, independent of location. In systems like NFS where the naming hierarchy differs on a per-host basis, not all files are accessible from all hosts. CapaFS ensures global naming and global access while preserving system integrity, through the use of capability file names. Capabilities define uniform semantics for system security and control. Possession of a capability filename is a necessary and sufficient proof of authority to perform the operations authorised by the capability on the file it names. CapaFS capability filenames are discussed in chapter 3.

Figure 1.1 shows how several sites might share CapaFS capability filenames among themselves. In this example four sites share three sets of files using the Internet for

transport. A user in TCD has access to files in MIT and in UL, while a user in MIT has access to files in UCD. Each relationship is separate from the others and does not depend on them.

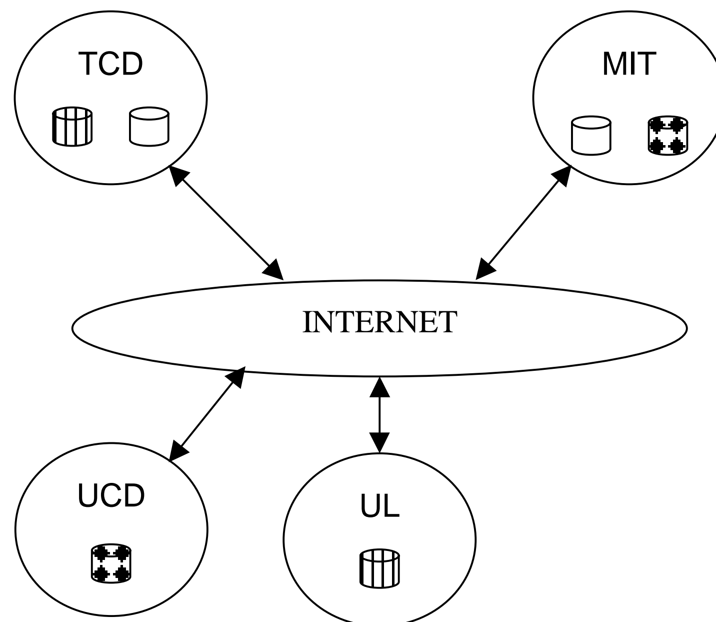

Figure 1.1: Sites sharing files using CapaFS.

1.3 Overview of the CapaFS Architecture

In order to transparently provide a global file system we needed to extend the functionality of the operating system to handle capability filenames. CapaFS allows users to customise their own environment to include links to remote resources, easily and quickly. In most systems today, accessing remote files involves mounting a file system which requires root privileges. When designing CapaFS we observed that most users will not have the capability to mount remote systems and decided to create

two prototypes of CapaFS. There were many alternate ways of extending operating system functionality, both in user and kernel space.

1.3.1 Kernel Space

Changing the operating system is the most straightforward approach. It involves modifying the actual operating system code and incorporating the desired functionality. This approach requires access to the source code and the privileges to install a new kernel. Sprite and Plan 9 both modify the operating system. A loadable kernel module, while similar to the above approach, allows the new functionality to be attached and removed on demand. These and any kernel modifications require super-user intervention. A prototype of CapaFS was implemented using this approach.

1.3.2 User Space

Another method of extending the operating systems functionality is using user-level libraries. Most applications do not directly access the operating system using system calls, but call library functions embedded in standard libraries. There are two classes of user libraries, static and dynamic. Applications must be re-linked to make use of the functionality stored in static libraries. Many systems make use of dynamic linking and shared libraries. These systems have several environment variables that can be used to specify alternate shared libraries, which contain new functionality but can still call functions in the standard library if needed. User-level libraries require no special privileges to install and use. The Jade[3] file system uses dynamically linked libraries

while Prospero was implemented using statically linked libraries. The implementation details of the CapaFS shared library and loadable module are discussed in chapter 4.

Figure 1.2 shows an overview of the ways CapaFS integrates into the operating system. Process A shows the standard interface before CapaFS is installed. With the CapaFS shared library installed, process B's calls are intercepted and either a CapaFS function or a standard library function is called. The CapaFS kernel module registers with the Virtual File System (VFS) and all file operation are redirected to the module. Details of the VFS architecture are left to chapter 4.

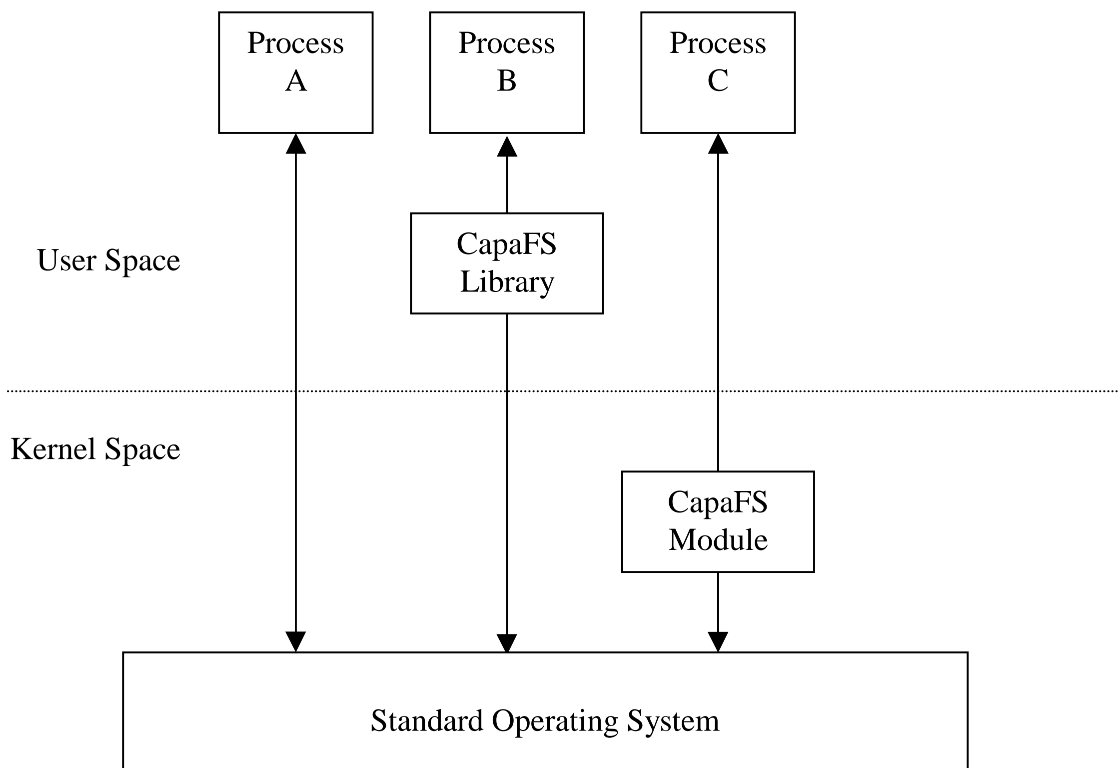

Figure 1.2: Overview of CapaFS Architecture

2. STATE OF THE ART

There has been a lot work done is distributed systems involving replication, caching, performance and security. The motivation for CapaFS is not increased security or performance but giving users the freedom to share files quickly and easily. We are looking at a number of distributed systems from the point of authentication and authorisation. Although all of these systems allow file sharing, we focus on how this is achieved and to what extent. We look at some of the problems and obstacles these file system have, in relation to file sharing, and how CapaFS may help to overcome them.

Although we focus solely on distributed file systems, each system is unique in its implementation of authentication and authorization features. WebFS [10,11] aims to provide a common platform for developing Internet application using HTTP. SFS [12,13] is a secure, global, decentralised file system, which uses self-certifying pathnames to gain access to remote servers. The Echo [14,15] distributed file system is designed around a truly global name space and uses certificate authorities and public-key encryption to ensure system security. The Andrew file system uses ACLs and Kerboeros for authentication and authorization. In AFS permissions are associated with directories, not individual files, to provide a level of abstraction for large systems. Finally the Truffles [16] file system provides a unique method of file sharing, with very little system administrator intervention. This is achieved using a secure form of electronic mail.

2.2 WebFS

WebFS is a global cache coherent file system that provides a common substrate for developing Internet applications. It aims to provide a solution for collaborative work and a file system for general purpose distributed computing. Instead of replacing HTTP the WebFS file system is built upon the existing HTTP protocol and uses it for transport. This allows existing URLs to be accessed through the file system while providing a number of cache coherence policies. WebFS provides authentication and security through a protocol layered upon HTTP. Public key cryptography is used to authenticate the identity of servers as well as clients.

In WebFS, ACLs associate a file with a list of X.509 certificates and permissions. Authentication is achieved using access control lists (ACLs). Individual users are identified by their certificates. Every file has a a list of pointers to X.509 certificates associated with it. Several files may point to the same bucket. So if a user's certificate occurs in the ACL for a file then the user can access the file. Authentication requires a hierarchy of certification authorities, which places the users of WebFS under the control of the certificate authorities. Another problem is HTTP must transfer the entire contents of a file before allowing any further operations to be performed upon a file. In addition to this, it has no seek mechanism and cannot perform reads at arbitrary offset in a file.

2.3 SFS

The secure file system is a global decentralised file system. It provides a single name space across all machines in the world while avoiding any form of centralised control. All user authentication in SFS is accomplished using public keys. SFS uses self-certifying pathnames to provide a global name space. A self-certifying pathname consists of a host name to connect to and a HostID which contains a cryptographic hash of the servers public key. The name of a SFS file system entirely suffices to certify its server. Users mount remote SFS file systems simply be referencing them.

A SFS client consists of the SFS client software and the user's authentication agent, which authenticates the user to the client software and eventually the SFS server.

Users of an SFS client machine, register with an authentication agent. The agent holds the users private keys. When a new file system is accessed the client contacts the agent on the user's behalf and the agent authenticates the user to the client. Users can access universally readable files if the authentication fails. First the client authenticates the server and then the server authenticates the client and user. Server authentication involves the server sending the client its public key. The client must verify this via a trusted third party (TTP). During authentication, two secure channels are established to encrypt all communication. SFS also has a read only protocol to support public data servers. It uses pre-computed digital signatures to verify the contents of files. SFS access control relies on user and group IDs, which change from one machine to another so users must have accounts on file servers to access any protected files. This means sharing two protected files involves a system administrator creating two new user accounts.

2.4 Echo FS

The Echo file system is designed around a global name space. It allows secure global file access without global trust of the authentication root. Echo clients and servers attach to points in the global name space. The name space forms a hierarchy of trust, but clients do not need to go through the root of the trust hierarchy, in order to access servers with which they share a common prefix in the name space. In order for a user to access an object, the server must authenticate the identity of the user and the user may require server authentication.

Once both parties have authenticated each other, they create a secure channel to encrypt all traffic. When a server receives a request on a secure channel, there is a public key associated with it as well as a DNS name. All Echo files are protected by ACLs, which specify which users may perform what operations. Unlike other systems, access to objects is specified in terms of DNS names in ACLs, not public keys. Verifying the public key associated with a secure channel is not sufficient. The server must verify that the DNS name belongs to the user with the verified public key. This is accomplished through the use of certification authorities, which verify that a given public key belongs to a user with a given DNS name. Echo has no local name space and does not allow users to access new servers until a system administrator attaches the servers to a point in the name space.

2.5 AFS

The Andrew file system is designed around Vice and Venus Vice is a collection of dedicated servers running on a local-area network. Data sharing is provided by a process called Venus, which runs on each client workstation, and performs all file sharing on behalf on the client. Venus finds files in Vice, caches them locally and performs emulation of Unix file system semantics. Rather than trusting thousands of workstations, Andrew predicates security on the integrity of the much smaller number of Vice servers.

The security mechanisms in AFS include the logical and physical separation of servers and clients, support for secure communication at the RPC level, a distributed authentication service, a file-protection scheme that combines access control lists with Unix mode bits. ACLs in Andrew are associated with directories rather than files, because it provides an abstraction, which is useful in large systems.

The AFS name space consists of users and groups of users, which can authenticate themselves to a Vice server.   AFS-1 and AFS-2 support group inheritance, where a user's privileges become the cumulative privileges of all the groups it belonged to. AFS-3 uses Kerberos for authentication. Kerberos is an authentication system that provides evidence of a user or services identity. When a user logs into a workstation, their password is used to obtain tokens from an authentication server. The tokens, which are valid for twenty-five hours, are saved by Venus servers and used to establish secure connections to file servers on behalf of the user.

Despite an impressive list of security features, AFS client machines contain a fixed list of available servers that only a system administrator can update. So a user of an AFS client cannot access a server the administrator did not include. AFS uses Kerberos to encrypt all network traffic. Kerberos will not trust any file system on which a user does not have an account.

2.6 Truffles

The Truffles file system supports file sharing between arbitrary users from any sites connected to the Internet. It is an extension of the Ficus file system, which provides file sharing and replication. Truffles provides fine-grained access control and uses a secure form of electronic mail to establish the file sharing, authentication and encrypt network traffic to protect it from eavesdroppers. File sharing in Truffles is implemented using a somewhat awkward interface, and involves exchanging E-mail messages signed and encrypted with Privacy Enhanced Mail (PEM). So you need an E-mail address to share files.

PEM is an Internet standard that adds message authentication and confidentiality to E-mail. PEM authentication is done using RSA public key cryptography. A hash of the message to be sent, plus the original message is encrypted with the sender's private key. The hash is used by the receiver for message authentication. Since Truffles relies on using public keys to authenticate messages, it must make public keys available to users. Naming is achieved using X.500 and Truffles requires X.509

certificates for all users and servers. X.509 certificates are used to verify that a given X.500 name is associated with a given public key. This again involves a certification authority. A summary of the major findings of this chapter can be seen in Figure 2.1.

Properties	WebFS	AFS	ECHO	SFS	Truffles
Use Public Keys for authentication	YES	NO	YES	YES	YES
Uses ACLs	YES	YES	YES	NO	NO
X.509 Certificates	YES	NO	NO	NO	YES
Needs TTP	YES	NO	YES	YES	YES
Needs Administrator Intervention	YES	YES	YES	YES	NO

Figure 2.1: Summary of related file system properties

3. Design

3.1 Design Goals

When designing CapaFS we had the following goals:

- Global Naming - It must be possible for a user to reference a CapaFS filename and be confident that it has the same meaning to everyone in the system.

- Global Access - CapaFS files must be accessible everywhere in the system, independent of location and the semantics of the operations should not change.

- Security and Integrity – Users should not be able to modify CapaFS filenames either to obtain access to protected files or gain further permissions.

- Access Transparency – Users should use the same type of access mechanisms for both local and remote (CapaFS) resources.

- Low Administrator overhead – Configuration and installation of CapaFS should require little or no system administrator invention, to enforce easy of accessibility and promote sharing.

3.2 Client-Server model

In a traditional client-server distributed file system the server typically provides a name space, enforces file access permissions, and maps names and file offsets to disk blocks. CapaFS adopts the client-server approach, providing a global decentralised name space, with global access. Clients send file system commands such as open, read, and write across a network to be executed on the server. There are several advantages to the client-server approach for distributed file system design. Clients can access files transparently across a network using the same commands employed by the local file system, preserving binary compatibility. We achieve network hardware and protocol independence in CapaFS by using standard network protocol stacks, which include protocols such as RPC [21], XDR [22], and TCP/IP as shown in Figure 3.1. Network protocol independence helps insure portability by separating the network hardware from the distributed file system software.

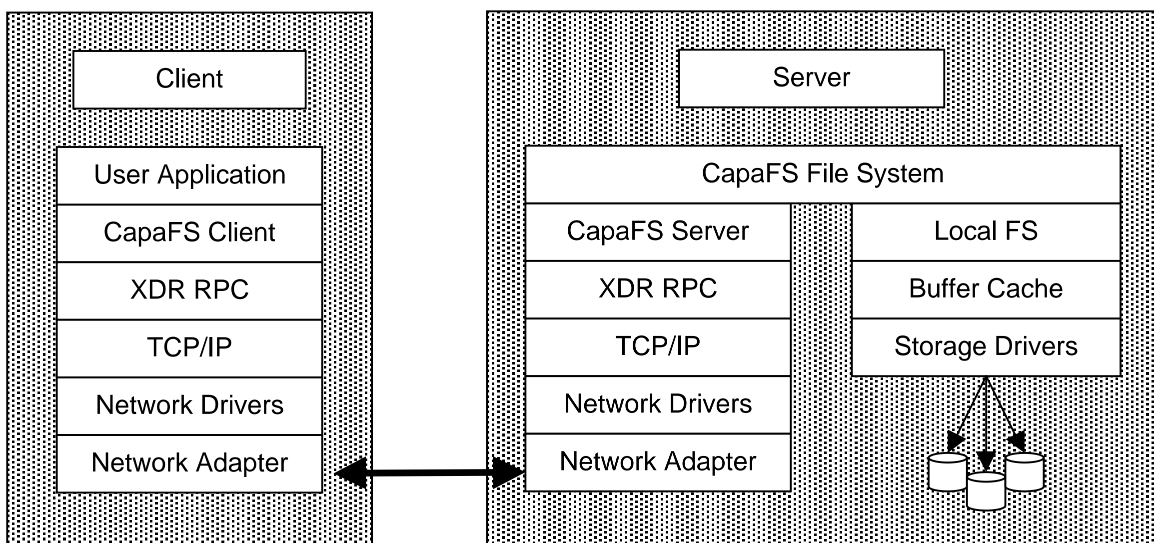

Figure 3.1: CapaFS Protocol Stack on Client and Server

3.3 Capabilities

There are two basic resource or more specifically file access mechanisms: capability based and access list-based control. Formally, ACLs may consist of {object, user, authority} triples, while a capability is a {object, type, authority} triple. ACLs have a distinct disadvantage. They are a static list of authorities rather than dynamic.

Capabilities can be view as unforgeable tickets specifying permissions, which allow their holder to access a named resource. A capability can be used to protect the object it references, while possession of a capability is a necessary and sufficient proof to authority to perform the operations authorised by the capability on the object that it names.

3.3.1 The Principle of least privilege

Capabilities allow each program to be restricted to exactly the set of authority it requires, adhering to the principle of least privilege. The principle of least privilege states that an application should have access to exactly those services and objects that are necessary to their operation. This minimises the failure scope of the application, simplifying security arguments and enforcing encapsulation. In theory capability systems should be faster and simpler than equivalent systems based on ACLs, because they do not have to traverse a list to examine the users identity and determine access rights.

3.3.2 Capability properties

A capability consists of:

- A trusted object identifier which specifies a unique object .
- An identifier which contains additional information redundancy for protection.

Capabilities can not only be used to name an object, but also to protect them and define what operations can be performed on them. Capabilities have the following properties:

- Possession of a capability gives a user authority to access the relevant object.

- They allow objects to be shared – all users that have capabilities to a given object, can share this object. Although there may be several capabilities for the same object, they may allow different permissions to each holder. The same capability held by different users will give each user the same access.

- Capabilities must be unforgeable and protected from disclosure.

3.3.3 Sparse Capabilities

There are three types of capabilities, tagged, partitioned and sparse. Tagging is a computer-architecture-oriented technique, which involves adding a number of bits to each memory area so a distinction can be made between data and capabilities.

Partitioned capabilities are stored in a special area separately from any data and can only be accessed by the system. This separation is to preserve the integrity of the capabilities. This technique is again, computer-architecture-oriented. The capabilities used in CapaFS are sparse. The key to sparse capabilities is that they do not need to be distinguished from data, either by tagging or partitioning. They are simply bit strings which can be stored on any medium.

3.3.4 Capabilities in other systems

In the Amoeba distributed object operating system, encrypted sparse capabilities are used to protect message ports. Amoeba capabilities include the port of the server that manages the object; an object id which is only useful to the server that manages the object; access rights specifying what operations are permitted on the object and a random number to protect the capability from forgery. The random field makes Amoeba capabilities sparse.

Capabilities or self-certifying pathnames in SFS consist of a location and a host id. The location is the address of the server, which manages the object. The host id is a one way hash, using the server's public key, of the client's public key and hostname. The SFS server must prove possession of the corresponding private key to decrypt the host id. The server then exchanges public keys with the client and user and then all traffic between them is secure. SFS capabilities are protected using public-key encryption, ensuring only the owner of a corresponding private key can access the capability.

3.3.5 Format of CapaFS capability filenames

Capabilities are used in many centralised systems, and stored in the protected system kernel. This is fine for centralised systems, but distributed systems cannot trust the kernels of user machines to protect the capabilities. Because knowledge of a capability conveys right to access the specified object or file, capabilities must be protected from theft and disclosure. The capabilities in CapaFS are protected from forgery and exposure using RSA encryption. This makes CapaFS capabilities sparse. The format of CapaFS capability filenames is shown in figure 3.2.

/CAPAFS	*MACHINE NAME*	*UID*	*FILE NAME*	*PERMISSIONS*
Plain Text			Encrypted	

Figure 3.2: Format of CapaFS capability filenames

3.4 Encryption Design

3.4.1 Why Use RSA Encryption?

There's no denying safety is an important issue to everyone involved with the Internet. At present, the Internet is the world's largest computer network. It is commonly assumed that if one wants to be sure a public key belongs to the person he hopes it does, he must use an identity certificate issued by a trusted Certification Authority (CA). There are many methods for establishing identity without using certificates from trusted certification authorities. The relationship between verifier and subject guides the choice of method. CapaFS uses RSA encryption and does not need digital certificates or certification authorities to function.

RSA encryption and authentication take place without any sharing of private keys. Each person uses only other people's public keys and his or her own private key. Anyone can send an encrypted message or verify a signed message, using only public keys, but only someone in possession of the correct private key can decrypt or sign a message. The larger the number of key bits used in the encryption process, the more difficult it is to successfully decrypt.

3.4.2 CapaFS Security

Millions of users are connected to the Internet. Thus, it has become an easy way to share information between users and/or organisations. Because of the ease of accessing to the Internet, it causes a new sort of problems concerning privacy and

security. The problems related to CapaFS are categorised into three kinds of problem: confidentiality, authentication and integrity.

Confidentiality

Confidentiality refers to communication where only the intended recipients can determine what was sent and nobody else. This is the basis of the privacy on any sort of computer network, including the Internet. It provides basic mechanisms for many solutions of security problems. Most confidentiality services commonly use a cryptography mechanism. CapaFS provides confidentiality of capability filenames by encrypting them so the actual filename and permissions cannot be disclosed, without the corresponding private key.

Authentication

Authentication in CapaFS, is achieved through the uses of capability filenames. A user authenticates itself to a CapaFS server, by proving possession of a valid capability filename. Possession of a capability is the only proof needed to authenticate a user to a server in CapaFS. For this reason, CapaFS filenames must be only given to parties that we trust. Many file systems use an elaborate dialogue between client and server to authenticate each other and establish trust. CapaFS does not need to establish trust between client and server, it only needs to verify the validity of a CapaFS filename. This is because, if a user possesses a valid capability filename, they must be a party we trust, because we only give capability filenames to parties we trust. This is what makes CapaFS unique.

Integrity

Integrity is the property to ensure that a CapaFS filename was not modified after the server gave it to the client. If the capability has been modified, the CapaFS server will not be able to decrypt the filename and permissions and access will be denied.
To create or change a capability filename, possession of the corresponding private key is required.

3.4.3 Difficulty of breaking RSA:

In CapaFS we use a variable key-size which we currently set at 256-bit, a good balance between speed and security, similar to Netscape which uses 256-bit RSA. RSA can be thought to be safer than DES because key lengths are variable. If a 256-bit modulus is not safe (RSA estimates that 512-bit can be broken for a $8.2 million effort), we can just make it longer. Of course longer keys make the encryption process slower which slows the open function in CapaFS, but increases in computer speed which make it easier to break encrypted data, also make it easier to encrypt data to a higher standard.

In order to break RSA encryption, you have to factor Mod, from the public key. So breaking RSA is as difficult as factoring. The existence of RSA cryptosystems has sparked a lot of research into factoring, and better factoring algorithms have been discovered recently; the fastest known factoring algorithm at this writing is the ’number field sieve’ which can factor any number in an amount of time not predictable. A general rule of thumb is, every ten bits added to the length of a number

(a number over 150 bits) makes it about ten times harder to factor. According to a special study report on Internet security [17], most experts believe that RSA will be secure "forever" with a key length of 2048 bits or more, unless some fundamentally new factoring algorithm is discovered. They state “Assuming that a 'nand' (the fundamental one-bit computing operation) requires at least one electron volt to perform, There is not enough energy in this universe to have a more than infinitesimal probability of factoring a 2048 bit modulus”. Long key lengths, however, are not suitable for file security, except for extremely sensitive individual documents.

The security therefore rests on the difficulty of factoring large numbers. There could be a major breakthrough in factoring, which would render all RSA encryption visible. This is thought to be unlikely, but what is needed is a formal mathematical proof, otherwise it is simply that no one has found a better method. Similarly a formal mathematical proof is required to show the equivalence between RSA and factoring, otherwise there could be a breakthrough in cracking RSA through an entirely different route. Again though highly unlikely, but this is not the same as a clearly demonstrated impossibility. The RSA source code has published which has been subjected to intense scrutiny. No one has yet found a flaw.

A flaw that was found in an earlier version was that the random number generator that produces the key was not as unpredictable as previously thought. This has been fixed. CapaFSKeys is responsible for the creation of both public and private keys in CapaFS. To avoid the aforementioned problem, when creating keys, users must enter a pass

phrase, which is used to initialise the random number generator. The values of each character entered and the amount of time taken to enter the pass phrase are used to create a random seed. The same pass phrase will not give the same keys.

3.4. The Mathematics behind of RSA Encryption

Ron Rivest, Adi Shamir, and Leonard Adleman invented the RSA algorithm in 1978. It is not illegal to transport this mathematical description out of the U.S. and Canada, but it is illegal to export an implementation of this cipher from the U.S. or Canada.
US Federal regulations forbid the export of encryption software that uses keys more than 40-bits.

The General algorithm is as follows:

1. Find P and Q, two large prime numbers.

2. Choose E such that E is less than PQ, and such that E and (P-1)(Q-1) are relatively prime, which means they have no prime factors in common. E does not have to be prime, but it must be odd. (P-1)(Q-1) can't be prime because it's an even number.

3. Compute D such that (DE - 1) is evenly divisible by (P-1)(Q-1).
   You can also write this as DE = 1 mod (P-1)(Q-1), and call D the multiplicative inverse of E.

4. The encryption function is encrypt (T) = (T^E) mod PQ, where T is the plaintext (a positive integer) and ^ indicates exponentiation.

5. The decryption function is decrypt(C) = (C^D) mod PQ, where C is the ciphertext (a positive integer) and ^ indicates exponentiation.

Your public key is the pair (PQ, E). Your private key is the number D, which is kept in the file capafsprivate in a users home directory. The private key must not be revealed to anyone. The product PQ is the modulus, often called N. E is the public exponent. D is the secret exponent. You can publish your public key freely, because there are no known easy methods of calculating D, P, or Q given only (PQ, E) (your public key).

4. IMPLEMENTATION

The CapaFS prototype implementation consists of the following:

- CapaFSKeys - Creates a public private key pair
- CapaFSFile - Produces a CapaFS capability filename using the public key created from CapaFSKeys
- CapaFSLIB - Shared library wrapper which adds glibc operations for
- CapaFSH - Creates a new shell with shared library wrapper installed
- CapaFSMOD - Loadable kernel module which allows CapaFS file systems to be mounted
- CapaFSS - CapaFS server which handles all remote requests from CapaFSLIB or CapaFSMOD

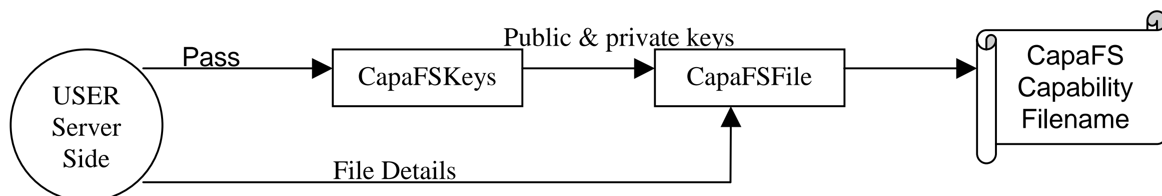

Figure 4.1: Creating a CapaFS capability filename

Figure 4.1 shows how a CapaFS capability filename is created. The user server side; refers to the user who owns the files and who has an account on the machine where the files reside. The user executes CapaFSKeys and enters a pass phrase, which then creates a public and private key. The keys are then used as input to CapaFSFile, in addition to file details entered by the user. The output of CapaFSFile is a CapaFS

capability filename, which must only be given to parties you trust. We now discuss the implementation of CapaFSKeys and CapaFSFile which are both essential in creating the capability filenames used in CapaFS.

4.1 CapaFSKeys

CapaFSKeys creates two files, one containing a public key and the other a private key using a standard RSA implementation. CapaFSKeys takes two arguments, which specify the name of the files to be created. The defaults are capafspublic and capafsprivate. To initialise the random number generator a random pass phrase must be entered. The characters of the pass phrase are accumulated and multiplied by the amount of time taken to enter the phrase. This gives us a random seed to initialise the random number generator. The seed must be random to ensure the chance of creating a public/private key pair twice is very small. The public/private key pair is created as follows:

- Generate two random prime numbers P and Q, of a specified bit length (256-bit)
- Let N equal the product of P and Q.
- Use the Euler Totient function on N which is equal to (P-1).(Q-1) and call this PHI.
- Choose the smallest odd number E, such that E and PHI have no common factors. The smallest number is chosen for speed reasons and is obtained by finding the first number, which has a greatest common dominator (GCD) of one with PHI. We now have the public key, which consists of E and N.
- We find the private key D by using the extended Elucian algorithm on E and PHI.

At this stage we have created a public key (E, N) and a private key (D, N). The public key is written to the file capafspublic and the private key to capafsprivate, unless others file names where given as command line arguments. The files are created in the current directory and capafsprivate must be copied to the users home directory on the machine where the CapaFS server is running.

4.2 CapaFSFile

CapaFSFile takes two inputs namely the public key of the user and the details of the CapaFS filename to be created. To create a capability filename CapaFS needs to be supplied with the hostname or IP address of the machine where the files reside. CapaFSFile automatically detects the UID of the user and this is stored in the capability also. The name of the remote files must also be supplied along with the permissions. The permissions are standard Unix permissions: read, write and execute. Only User permissions are needed, as Group and Other permissions are irrelevant because possession of a valid capability is the only method of accessing files in the CapaFS file system. The name of the symbolic link, which points to our CapaFS capability filename, must also be supplied.

Once all the file details are provided, the public key (E, N) is read from the file capafspublic and this is then used to encrypt the filename and permissions as follows:

- The characters of the filename are converted into very long integers.

- Each one is raised to the power of E mod N to create the encrypted cipher-text, which is used in the capability filename.
- Each encrypted cipher-text is stored and joined to the next to create one large string of cipher-text, which contains both the filename and permissions.

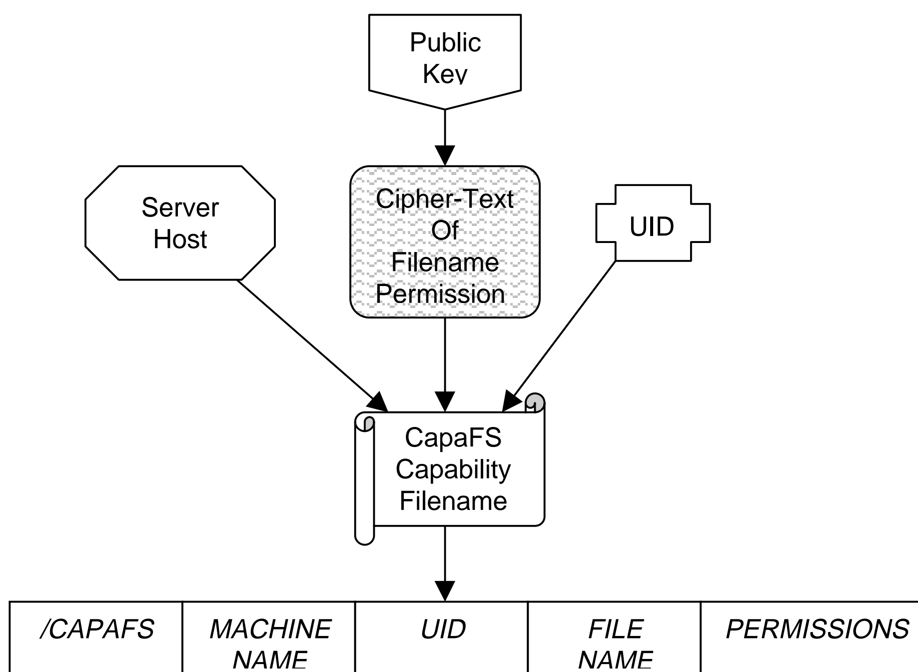

Figure 4.2: The process of creating a CapaFS capability filename

We now have all the necessary ingredients to create a CapaFS capability filename as shown in figure 4.2. A symbolic link is created which points to a CapaFS capability filename which always begins with the string "/CAPAFS". The combination of this and the hostname and UID specify a unique entity which can be access globally.

4.3 CapaFSLIB

The CapaFS shared library CapaFSLIB is used by a client to add the necessary functionality the handle CapaFS capability filename to the operating system. The CapaFS shared library replaces the standard shared C library, libc. The following file operations are wrapped: open, close, read, write, lseek, fcntl. We now discuss the implementation of each of these operations on the client side in CapaFSLIB.

4.3.1 The client open operation

The open operation opens a file on a remote host and returns a handle to the opened file. It takes a filename, flags and mode as arguments. The filename is checked to see if it is a symbolic link. If so, the destination to where the link points is checked to see if it is a valid CapaFS capability filename. If the filename is not a CapaFS filename then the standard libc open operation is called. Once we have a valid CapaFS filename it is parsed and the hostname of the server, along with the UID and encrypted filename and permissions is extracted. Using a remote procedure call (RPC) we create a connection to the remote host, specifying the program and version of the CapaFS server. RPC uses the Transmission Control Protocol (TCP) for transport instead of the User Datagram Protocol (UDP), because TCP ensures all RPC data packets will reach their destination and in the correct order. This is necessary over long distance networks, and the Internet, where timeouts and dropped packets are a regular occurrence. Once a connection is established between the client and server, the RPC open operation is called, which has the following arguments:

- Filename – the file to be opened

- Flags – specify if the file opened for reading, writing or both
- Mode – specify the permissions if a new file is created

On successful completion, open will return a valid file descriptor, which can be used to access the file specified in the capability, with the given permissions. The connection to the server is closed and the file details must be stored, in order to recognise a valid CapaFS file handle.

The file details are stored in a fileHandle structure within linked-list implementation as follows:

- File handle - the CapaFS file handle returned to the client
- Machine - the client machine name
- Next - a pointer to the next fileHandle

4.3.2 The client close operation

The close operation closes a CapaFS file descriptor so it no longer refers to any file and may be reused. This operation takes a file descriptor as an argument. A connection to the remote server is established and the RPC close operation is invoked, passing the file descriptor to the server. Once a file is closed its file handle may be reused. For this to occur the file handle must be found and removed from the linked list.

4.3.3 The client read operation

The read operation reads a number of bytes from a specified file on a remote host and returns them into a user buffer. The read arguments are a file descriptor, which specify the file to read, the address of the buffer to store the data and the number of bytes to read. Once a connection is established between client and server, the read RPC call is invoked with the arguments:

- File handle- the file descriptor of the file to read
- Count - the number of bytes to read.

The RPC read call returns two arguments, the status of the call and the data read from the file. The data is returned in blocks of eight kilobytes (8K), to ensure performance and compatibility with applications, which are already, optimised for the Network File System block size of 8K. When the data is successfully returned it is copied into the user buffer and the status returned.

4.3.4 The client write operation

The write operation writes a number of bytes to a specified file on a remote host. Write takes a file descriptor which specifies the remote file, a user buffer with the data to write to the file and a count of the number of bytes to written. The write operation connects to the remote host and calls the write RPC. On successful completion, write returns the number of bytes written to the file or an error. The remote write operation takes the following arguments:

- File handle- the file descriptor of the file to read

- Count - the number of bytes to read.
- Buffer - the user buffer containing the data to be written

4.3.5 The client lseek operation

The lseek operation moves the position of the file pointer for a given file. The pointer can be moved relatively from the current position or absolutely from the beginning or end of the file. Lseek takes a file descriptor and an offset, which is the number of bytes to move the file pointer. A number to indicate whether to move the file pointer from the beginning of the file, the end or the current position is also needed. The lseek operation connects to the remote CapaFS server and calls the lseek RPC with the following arguments:

- File handle- the file descriptor of the file, whose file pointer is to be moved
- Offset - the number of bytes to move.
- Whence - whether to move the file pointer relative to the start, end or the current position.

The CapaFSLIB client could remember the position of the file pointer locally, without the need of a RPC, if it got the size of the file in question. This approach was then eliminated due to the fact that in a shared file system like CapaFS a file may be shared for reading and writing among many users, so the file size could be constantly changing. The lseek operation allows the file position to be set beyond the end of the actual file end-of-file (EOF), so the file size must be known. This means that the file size would have to be constantly checked so this was abandoned in favour of a remote

lseek operation. On successful completion the position of the file pointer relative to the beginning of the file, is returned.

4.3.6 The client fcntl operation

The fcntl operation is used to manipulate a file descriptor. It takes two parameters namely the file descriptor and an operation to be performed. The commands vary from duplicating file descriptors, reading/setting flags and locking. The client connects to the server and calls the fcntl RPC with the following arguments:

- File handle- the file descriptor of the file to perform the command upon
- Command - the command to be performed

The return value of fcntl depends on the command issued. It can range from a new file descriptor to the value of flags.

4.4 CapaFSMOD

CapaFSMOD is a loadable kernel module for the Linux operating system. It allows CapaFS capability filenames to be mounted and access under a special mount point.

A kernel module is simply an object file containing routines and data to load into the running kernel. When loaded, the module resides in the kernel's address space and executes entirely within the context of the kernel. A module must have an init_module() and cleanup_module() functions. When writing a kernel module, all of the precautions and conventions used when writing kernel code must be observed.

The module can be loaded on demand using the standard insmod command, or removed using rmmod. CapaFSMOD registers CapaFS with the Virutal File System

(VFS) and then all file operations performed on a CapaFS mount point, are redirected to the VFS code in CapaFSMOD.

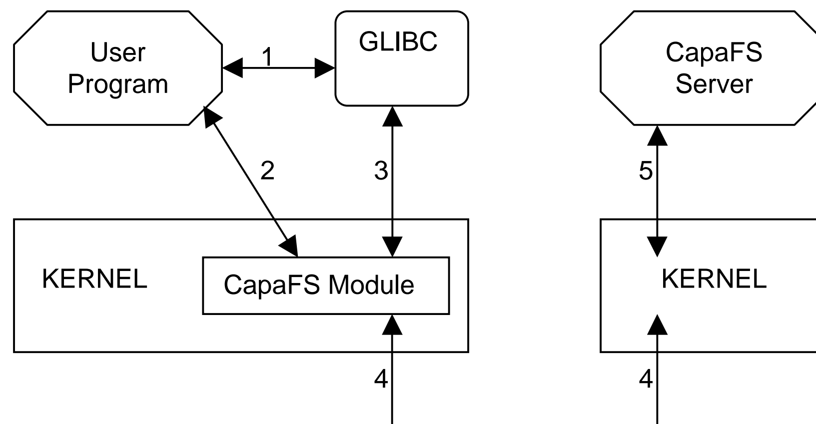

Figure 4.3: CapaFS kernel module framework

The framework of the CapaFS module is shown in figure 4.3. A user application can invoke either the standard library functions (1) or directly call a kernel system call (2). The standard library in turns makes a system call (3). The CapaFS module connects to the remote host (4) and then to the CapaFS server in user space (5).

4.4.1 Virtual File System

The Linux kernel contains a Virtual File System (VFS) layer, which is used during system, calls on files. The VFS is an indirection layer, which handles the file, oriented system calls and calls the necessary functions in the physical file system code to do the

I/O. This indirection mechanism is used to allow the integration and the use of several file system types. When a process issues a file oriented system call, the kernel calls a function contained in the VFS. This function handles the structure independent manipulations and redirects the call to a function contained in the physical file system code, which is responsible for handling the structure dependent operations. Figure 4.4 shows the relationship between the Linux kernel's VFS and it's real file systems.

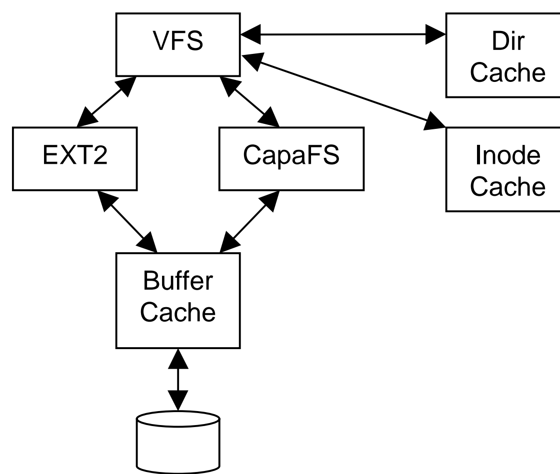

Figure 4.4: Logical view of the Virtual File System

The VFS must manage all of the different file systems that are mounted. This is done by maintaining data structures, which describe each file system. The VFS describes a file system in terms of superblocks and inodes. To initialise itself, a file system must register with the VFS. File system modules are loaded as the system needs them, so CapaFSMOD is only loaded when a CapaFS file system is mounted. When a file system is mounted the VFS must read its superblock. Each file systems superblock read routine must work out the file systems topology and map information into a VFS

superblock structure. The VFS keeps a list of the mounted file systems and their superblocks. Each superblock has a list of pointers to functions that perform particular routines. So the superblock representing a mounted CapaFS file system contains a pointer to the CapaFS specific read inode routine. Each VFS superblock contains a pointer to the first inode on the file system, which would be /CAPAFS in the CapaFS file system.

As every file or directory in the system is represented by a VFS inode, a number of inodes will be accessed frequently. These inodes are kept in the inode cache. The action of reading an inode causes it to be put into the inode cache and further accesses to the inode keep it in the cache, while less used VFS inodes get removed from the cache. There is also a VFS buffer cache which all file systems use to cache data buffers. The advantage of this is it makes CapaFS and all other Linux file systems independent from the underlying media and device drivers. The VFS also keeps a cache of directory lookups so that the inodes for frequently used directories can be accessed quickly.

4.4.2 The CapaFS VFS Superblock

When a CapaFS file system that is mounted, it is represented by the VFS using a superblock structure. The superblock contains:

- Device - the device identifier for the block device, which is to be, used CapaFS has no block device.

- Inode pointers - the mounted inode pointer points to the first inode in the system which is /CAPAFS.
- Blocksize - The block size of CapaFS is 8K
- Superblock routines - A pointer to a set of superblock operations for CapaFS.
- File System type - A pointer to the mounted file system's file_system_type data structure
- File System specific – A pointer to information needed by CapaFS.

The superblock routines include:

- read_inode – passed an inode structure with the inode number initialised by the VFS to indicate which CapaFS inode to read.
- write_inode – called by VFS to write a CapaFS inode
- put_inode – called when a CapaFS inode is removed from the inode cache
- delete_inode – to delete an inode
- notify_change – used when an inodes attributes are changed
- put_super – called when CapaFS is unmounted
- write_super – used to write the superblock to disk
- statfs – called when the VFS needs to get the file system statistics
- remount_fs – used to remount CapaFS
- clean_inode – used to clear an inode.

4.4.3 CapaFS VFS inode

The read_inode method fills in the i_op field, which is a pointer to an inode_operations structure, which describes the methods that can be performed on individual CapaFS inodes. The information in each VFS inode is obtained from the underlying file system by file specific routines. VFS inodes exist only in the kernel's memory and are kept in the VFS inode cache as long as they are accessed frequently. In CapaFSMOD the VFS inodes contain the following fields:

- Device – device identifier of CapaFS
- Inode number – a unique inode number within the CapaFS file system. The device identifier and inode number are unique within the VFS.
- Mode – specifies if a file or directory and access rights
- User ids – owner identifier
- Times – creation, modification and write times
- Block Size – data is manipulated in 8K blocks
- Inode operations – list of pointers to functions which perform operations on inodes
- Count – the number of references to this CapaFS VFS inode. A count of zero means the inode can be discarded.
- Lock – used to indicate that the inode is in use, like reading the inode
- Dirty – indicates if the inode has been written

4.4.4 Registering CapaFS with the VFS

When CapaFSMOD is loaded it registers itself with the VFS and unregisters itself when it is unloaded. This is done using the file_system_type structure. The VFS keeps a list of the registered file systems using this structure. When a request is made to mount a CapaFS file system, the VFS calls the superblock routine to mount the file system as shown in Figure 4.5. The directory entry for the mount point will then be updated to point to the root inode for CapaFS.

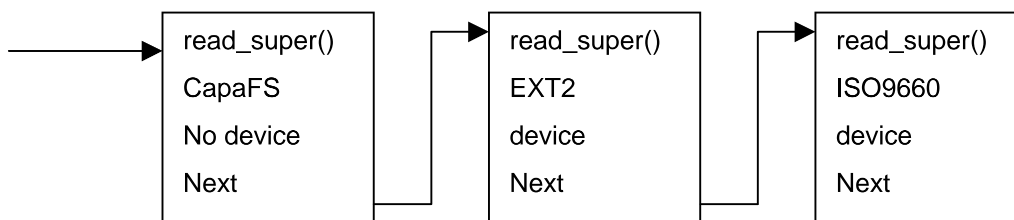

Figure 4.5 Registered file systems

The file_system_type structure contains the following information:

- Superblock – the read_super() routine the VFS calls to mount a CapaFS file system
- File system name – name of the file system: CapaFS
- Device needed – specifies whether a physical device is needed. CapaFS does not need a physical block device.

4.4.5 Mounting CapaFS

The mount command does not know which file systems the kernel supports. It passes the VFS the name of the file system, the physical block device that contains the file

system and the mount point where the new file system will be mounted. The name of the CapaFS capability filename must be specified using the –o flag. The VFS must first find the correct file_system_type structure and then invoke the associated routine to read the CapaFS superblock. If CapaFSMOD is not loaded, the kernel can demand load the module using a kernel daemon. Each mounted file system is described by a vfsmount structure on the vfsmntlist. It holds the device number, mount point and pointer to the read_super() function, which reads the CapaFS superblock as shown in Figure 4.6. This is similar to the file_system_type structure described earlier, but is from the mount point of view, not the VFS.

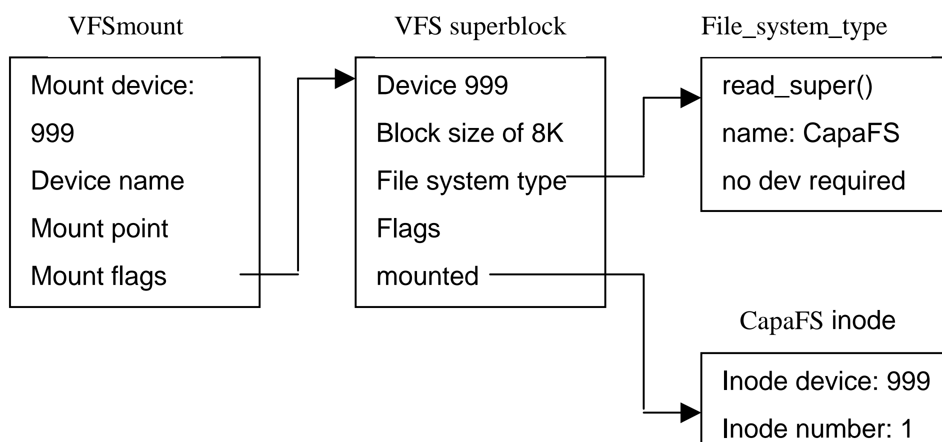

Figure 4.6: A registered and mounted CapaFS file system

Once the VFS structures are filled with the correct data, the kernel module has the same code to handle capability filenames, as the shared library discussed earlier. The kernel module has a performance advantage over the shared library and no matter what functions are called the VFS implementation ensures all operations can be

performed on CapaFS mounted directories. The only disadvantage is root privileges are required to mount a CapaFS file system, as with most other file systems.

4.5 CapaFS server

CapaFS servers can be set up easily without any system administrator intervention. The CapaFS server runs in user space and can be started by any user in the system. The server runs with the privileges of the user who starts it, to ensure that the server is restricted to only the files that this user can access. This means the server has access to exactly those services and files that are necessary for its operation, following the principle of least privilege. The underlying operating system enforces this restriction. This maintains the integrity of the system, and ensures a CapaFS server cannot be hijacked to gain access to another users or system files, thus providing no extra threat to system administrators.

A user who wants to share their files with others starts the CapaFS server. Once a user has created capability filenames and given them to parties they trust, they can start a CapaFS server which will handle requests from remote clients and allow sharing of the specified files with the allowed permissions. The CapaFS server will handle requests from both the CapaFSLIB shared library wrapper and CapaFSMOD the loadable kernel module, as the RPC specification for both is equivalent. They only differ in how they add functionality to the operating system to handle capability filenames. The remote procedure calls of the server are now described.

4.5.1 Server Open

The server RPC open operation receives a CapaFS capability filename, flags and mode as arguments. The capability is checked to see if it is a valid capability filename. Then it is parsed and the user id (UID) and cipher-text is extracted. Once the server has the UID, it can use it to find the corresponding private key to decrypt the cipher-text. The server opens the password file and searches for the UID. Once the UID is found the users home directory is also there. The format of the Linux password file is shown if Figure 4.7.

Login Name	Encrypted password	UID	GID	Real Name	Home DIR	Login shell

Figure 4.7: Format of Linux password file

The private key to decrypt the cipher-text can now be found in the home directory corresponding to the UID. The CapaFS server ensures a user's private key must be in their home directory, to enforce security and to prevent the server accessing dummy keys forged by interlopers. The CapaFS server does not accesses any files outside a users home directory, excluding the password file. Obviously, if the capability filename refers to files outside the home directory, they will be accessed regardless, but the user who started the server must have permissions to access them. The cipher-text is decrypted using the private key to reveal a filename and permissions.

The allowable permissions in CapaFS are Read, Write and Execute. There is no need or sense in having permissions for groups or others. This is because the CapaFS

capability filename contains the permissions, and the only way to access files in CapaFS is to prove possession of a valid capability, which already has permissions associated with it. This also allows encapsulation of permissions and objects, without the notion of a user. There are many options available when opening a file, which may contradict with the permissions specified in the CapaFS capability filename. For example a capability filename that grants read only permissions, cannot be open for writing. Once the permissions are checked, the server opens the file with the specified arguments. The file descriptor is saved along with other file details so as to recognise and associate a CapaFS file handle with an actual file. When a new file is opened successfully its file details are added to the list. The details of the file are stored in a linked list of fileDetails structures containing:

- Open arguments – contains the arguments used to open the file, which include the flags and mode.
- File handle – the file descriptor
- File position – the current position of the file pointer

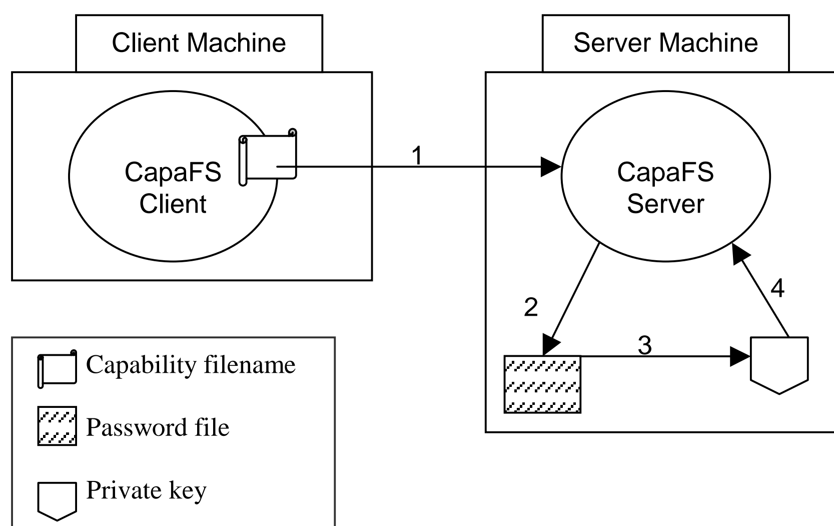

Figure 4.8: Overview of the CapaFS open command

A general overview of the server open function is given in figure 4.8:

1. A open command is performed on a CapaFS capability filename
2. The server uses the UID as a key to search the password file for the corresponding home directory.
3. The private key is read from the users home directory
4. The cipher-text in the capability filename is decrypted using the private key

4.5.2 Server Close

The servers close operation closes a CapaFS file descriptor so it no longer refers to any file and may be reused. The servers close operation is passed a file descriptor, to which it must find the associated file details and remote them from the list, so the file descriptor may be reused later.

4.5.3 Server Read

The read operation reads a number of bytes from a specified file on a remote host and returns them into a user buffer. The servers read arguments are a file descriptor which specifies the file to read, the address of the buffer to store the data and the number of bytes to read. The server must first relate a file descriptor to the details of the file, by searching the list of details using the corresponding file handle as a key. Once found the file is opened with the specified mode and flags obtained from the file details. The file pointer is then moved to its last position, which was also found in the file details.

The file is then read into a buffer, which is subsequently copied into a user buffer on the client side, and returned to the client user. The number of bytes read is returned to the client, and the current file position is saved in the file details for the next operation.

4.5.4 Server write

The server write operation writes a number of bytes to a file specified by a file descriptor. Write accepts a file descriptor, which specifies the file, a user buffer with the data to write to the file and a count of the number of bytes to written. The write operation must relate the file handle to file details, similarly to the read operation. Once this is achieved the data in the buffer is written to the specified file. Again the number of bytes written is returned and the file positioned saved for the next call.

4.5.5 Server lseek

The server lseek operation moves the position of the file pointer for a given file. The pointer can be moved relatively from the current position or absolutely from the beginning or end of the file. Lseek takes a file descriptor and an offset, which is the number of bytes to move the file pointer. A number to indicate whether to move the file pointer from the beginning of the file, the end or the current position is also needed. The file handle is used to get the file details. The file is then opened using the mode and flags in the file details. The file pointer for the file is also updates.

Then the lseek is performed upon the file and the file pointer position is saved. The number of bytes the file pointer is from the beginning of the file is returned.

4.5.6 Server fcntl

The server fcntl operation is used to manipulate a file descriptor. It takes two parameters namely the file descriptor and an operation to be performed. The commands vary from duplicating file descriptors, reading/setting flags and locking.

The fcntl operation uses the file handle to obtain the file details. The file details are then used to open the file and move its file pointer to it last position. Then the fcntl function is called and the command is executed upon the file descriptor. The return value of fcntl depends on the command issued. It can range from a new file descriptor to the value of flags.

4.6 CapaFSH

CapaFSH is a simple shell script to facilitate easy of use and installation of CapaFSLIB. The shared library wrapper must be installed to function. CapaFSH creates a new shell with the LD_PRELOAD environment variable set to the shared library wrapper, so CapaFSLIB can be started and stopped on demand.

5. EVALUATION

We will first discuss the CapaFS design choices and how they affect performance and then compare our CapaFS prototype to the Network File System. When designing CapaFS we decided to create two client implementations, one using a user space shared library and the other a loadable kernel module. The kernel module CapaFSMOD uses the VFS layer of indirection, to handle all file requests for CapaFS. The VFS has the ability to not only cache recently used inodes in its inode cache, but also has a directory cache for recently and frequently accessed directories. The shared library implementation has no caching at all. This is because the shared library wrapper was the first prototype implemented and was designed to evaluate the functionality of the CapaFS file system. The library runs at user-level and requires no kernel modifications. Compared with the kernel-module the user-level approach has the following advantages. First, it is easy to experiment and to examine different design options. Second, debugging user-level processes is much easier than kernel-level mechanisms because ordinary-debugging tools can be used, unlike a kernel implementation. Thirdly, portability among heterogeneous operating systems is easier. The obvious disadvantage of this method is that performance will be degraded by the user-level approach.

Since both the shared library and kernel module interacts with the same user-level server, the obvious choice of evaluation is the kernel module. At this moment in time we cannot evaluate the kernel module. This is due to the problem of obtaining suitable documentation of the new Linux RPC kernel support, which has been

released in Linux kernel 2.2.x. The kernel module is fully functional is all aspects but one, the RPC interface has not yet been defined, and therefore cannot be evaluated, even though it includes file and directory caching and the shared library does not, and because of this all test will not be on cached data.

5.1 Internet Test Domain

Since CapaFS is an Internet file system, it is irrelevant evaluating it on the Local Area Network (LAN), because it will not be used in that domain. CapaFS must be evaluated in it proposed domain, so tests are based solely on the performance of CapaFS over Wide Area Networks (WAN). The most widely used sharing file system is the Network File System (NFS). NFS has certain limitations that CapaFS does not. Setting up a NFS relationship is a heavyweight operation and requires substantial system administrator intervention on both the client and server sides. Despite this, NFS is the obvious file system to compare with CapaFS. The results presented here are not a final evaluation of CapaFS, but an evaluation to date. Work is on going to complete the CapaFSMOD prototype, which will have superior performance when compared with the user-level shared library. Despite all of this, the shared libraries' performance is acceptable. Most CapaFS operations are close to their NFS counterpart except one: the open operation.

5.2 Differences between CapaFS and NFS

There are several factors, which might differentiate CapaFS performance from NFS performance. Firstly, CapaFS is much more CPU intensive than NFS. This is

because of the RSA public key encryption and integrity checks performed on CapaFS capability filenames. The open operation involves the user-level server reading a public key and decrypting the cipher-text in the capability, to reveal a filename and permissions. The overhead of decrypting the capability is proportional to the bit-size of the encryption used. We use a 256-bit encryption, to ensure longevity and security of CapaFS capabilities. The encryption size of CapaFS can be simply changed and then recompiled for a faster implementation. Although this is mean the capabilities are not as secure as before, which may be acceptable for systems will discard capabilities after a short period.

5.3 Encryption Performance

The Large Integer Package (LIP) [19] was used to handle the large integers in CapaFS. LIP is written entirely in ANSI C and is easily portable. To increase performance, LIP is compiled twice. The initial object file is created and used to run the LIP timer program, which creates a header file. The header file is then used in the second compilation, to select better compile options giving much improved performance. Standard RSA encryption is used in CapaFS, to provide security and integrity of the capability filenames.

CapaFS was designed to be portable and adaptable. The strength of the encryption can be changed easily. This is done by changing a constant in CapaFSKeys. CapaFSKeys is responsible for creating the public and private key pair. Once you create keys of a certain length using CapaFSKeys, the rest of the CapaFS file system

can handle them. The decryption of the capability filename is the performance barrier in CapaFS. We currently use 256-bit encryption, but we have also tried weaker and stronger variations. Anything less than 256-bit is decrypted almost immediately but is also too small to be considered. Figure 5.1 shows the times to decrypt a standard capability in CapaFS.

Encryption Strength	Time to decrypt
128-bit	< 1
256-bit	2
512-bit	8
1024-bit	58

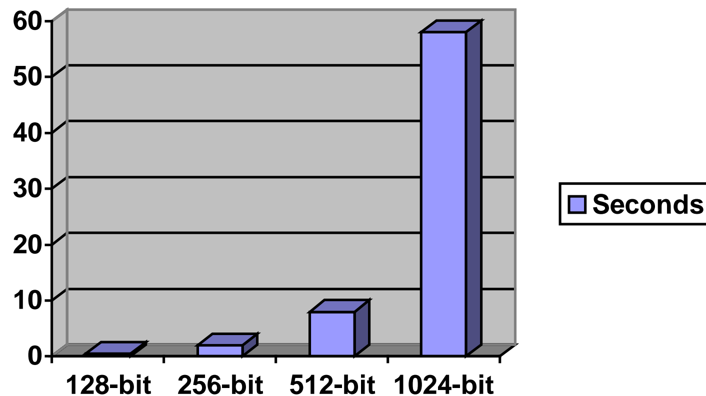

Figure 5.1: Decrypting capability filenames of varying strengths

As you can see from the graph, the amount of time to decrypt a capability depends entirely on the bit size of the keys used to create it. While 128-bit, 256-bit and even 512-bit have acceptable times, the 1024-bit decryption took almost one minute to complete. This strength of encryption is obvious unacceptable for CapaFS. Given the

times above and the difficulty of factoring large prime numbers, a key size of 256 bits is a good balance between performance and strength.

5.4 RPC Performance

CapaFS is implemented using the standard Linux RPC calls. All data in CapaFS is stored in opaque data types, and is not marshalled by external data representation (XDR), but all arguments from the client to the server are marshalled using XDR.

Figure 5.2 shows how Linux RPC compares with other RPC implementations. Note that asynchronous RPC (ARPC) can handle a number of outstanding requests simultaneously which gives it much greater performance in the long run.

RPC implementation	100 byte
SUN RPC	0.22 ms
ARPC	0.23 ms
XARPC	0.25 ms
LINUX RPC	0.23 ms

Figure 5.2: Comparison of the various RPC libraries

5.5 Performance evaluation of CapaFS and NFS

The performance of CapaFS and NFS is now compared and discussed. The tests of both CapaFS and NFS were performed using two standard Pentium machines running RedHat Linux with 32MD of memory. The client machine accesses the server over a number of networks including a point-to-point protocol (PPP) link from the client to the Internet and then across a asymmetric digital subscriber line (ADSL) on to a LAN

where the server resides. This mix of heterogeneous networks is to show CapaFS's performance and robustness over the Internet and other networks.

The client and server were set up on different machines with a medium load and each machine running in standard multi-user mode. We envisage this is a typical situation were CapaFS would be used. The tests covered the following operations for both CapaFS and NFS:

- Opening or lookup of a remote file
- Reading 100K from a remote file
- Writing 100K to a remote file

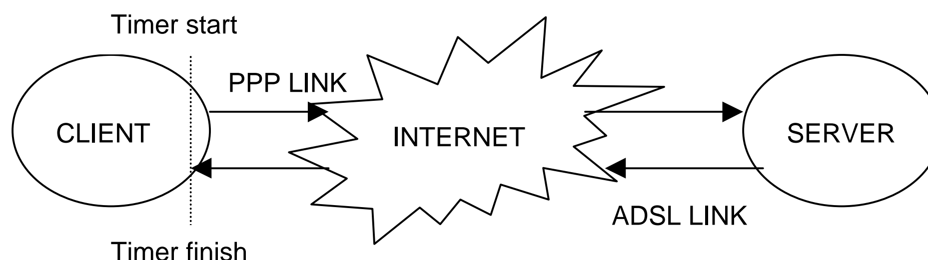

Figure 5.3: Timing of remote test cases

All tests are timed from the point the operation is invoked, until the point when a result is returned, as shown in figure 5.3. In addition to this, all tests are run from the client side, accessing files on the server side as usual. The performance results of these tests are given in figure 5.4. This shows that the performance of the CapaFS shared library is comparable to NFS for all fundamental operations excluding the open

command. This is as expected, due to the nature of decrypting the capability filename. One point to note is, once a capability file is opened using CapaFS, a file handle is returned, which ensures access to the permissions the capability grants without the need to go through the decryption process again.

	CapaFS (sec)	NFS (sec)
Open (Lookup for NFS)	153	131
Read 100K	271	264
Write 100K	277	274

Figure 5.4: Performance of CapaFS and NFS over the Internet

This shows that the performance of CapaFS is similar to NFS over the Internet. The performance penalty of the user-level shared library is not significant over long haul networks like the Internet. We feel the small loss in performance is an acceptable penalty when the new functionality CapaFS adds is taken into consideration. While the results here show CapaFSLIB is similar in performance to a user-level NFS implementation, CapaFSMOD, the kernel module implementation should give improved performance. We predict that it will have similar performance to an in-kernel NFS server, despite the server is in user-space. This is because the performance advantage a kernel-space server has over a user-space server in minimal, when put in the context of return-trip times over large networks like the Internet.

6. CONCLUSION

We have designed and implemented CapaFS, a global, decentralised file system that allows users to collaborate with other users anywhere in the world, with no prior arrangements or connections. The system uses filenames as sparse capabilities to name and grant access to files on remote servers. Users can share files without the intervention of system administrators, by exchanging capability filenames with parties that they trust. Unlike other systems, CapaFS does not use need to authenticate the client machine to the server. A client must simply prove possession of a valid capability filename, which is the necessary and sufficient proof of authority to perform the operations, authorised by the capability on the file it names. CapaFS does not need to establish trust between client and server, it only needs to verify the validity of the CapaFS filename.

CapaFS may be used successfully in many different environments, to provide previously unavailable functionality. Roaming mobile users who share files from their home site with the people they are visiting, is one setting. CapaFS may also be used in large businesses, to cross administrative boundaries or company boundaries in a virtual enterprise. People who work with semi-anonymous users over the Internet and collaborate on projects together, may use CapaFS to facilitate and promote sharing.

A decentralised file system with global authentication and flexible authorisation can free users from many of the problems, which have developed due to increased security

and centralised control. CapaFS solves some of the problems these central authorities cause. They subject their users to rigid centralised control, making their users rely on the central authorities, creating a single point of failure. Distributed systems should not rely on centralised control. User accounts cannot be accessed without the proper authority, so whoever controls the authentication controls the creation of new users or resources. CapaFS distributes control through-out the entire file system, where nobody controls the name-space, or access to it. Digital certificates and certificate authorities empose another form of centralised control.

6.1 Where does CapaFS fit in?

At the time of writing, NFS is the de-facto standard network file system. It is used throughout the world on a variety of hetrogeneous platforms. It would be advantateous to consider that CapaFS would replace NFS. CapaFS is not designed to replace NFS by any means. NFS and CapaFS may easily co-exist in the same system, with little difficulty. We imagine an environment where both file systems work together, where NFS will be used to access remote files on the local area network (LAN), while CapaFS will used to access files on the wide area network (WAN) or cross administravive boundaries, as shown in Figure 6.1. Short CapaFS capabilities could be created for roaming mobile users, who will only want to access their files for a few days, while NFS could be used for stationary desktop users.

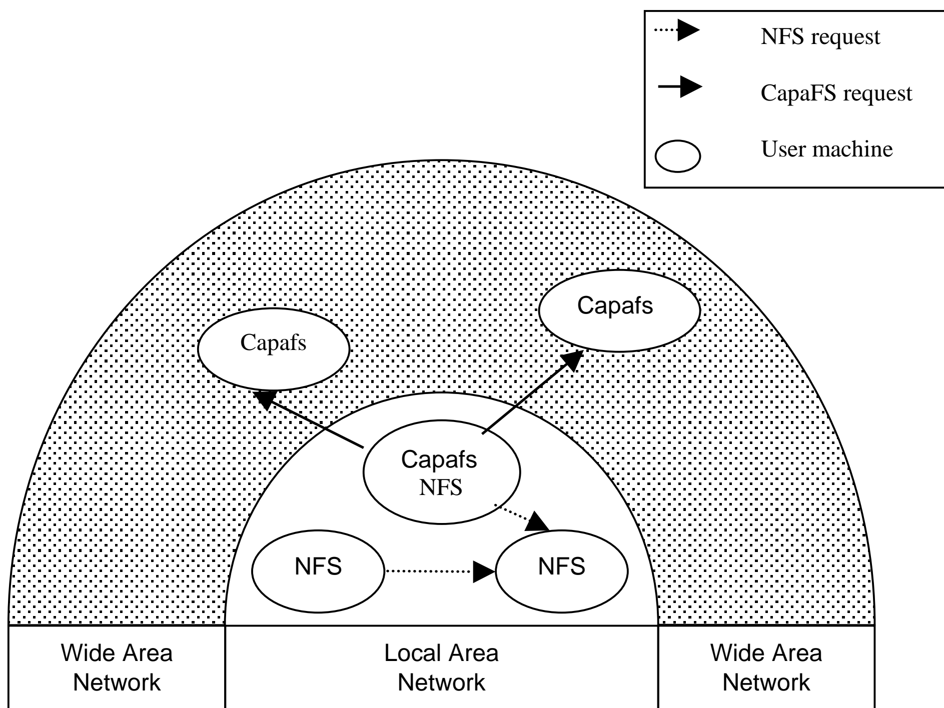

Figure 6.1: Where does CapaFS fit in?

6.2 Related Work

A number of previous file systems have provided a global name space, decentralised control or file sharing across administrative boundaries. One of the first to provide global name space across the wide area was AFS. A user of an AFS client cannot access a server unless the system administrator did not include it in the list of accessible machines. UFO [20] provides access to a read-only HTTP name space and read-write access using FTP. The Truffles service is an extension of the Ficus file system and provides fine-grained access control without any system administrator intervention. Truffles relies on centralised certification authorities to name users.

WebFS implements a network file system unsing HTTP as transport. This allows existing URLs to be accessed through the standard file system. The SFS file system uses self-certifying pathnames to access remote servers. SFS access control relies on user and group IDs, which change from one machine to another so users must have accounts on file servers to access any protected files.

6.3 Futher Work

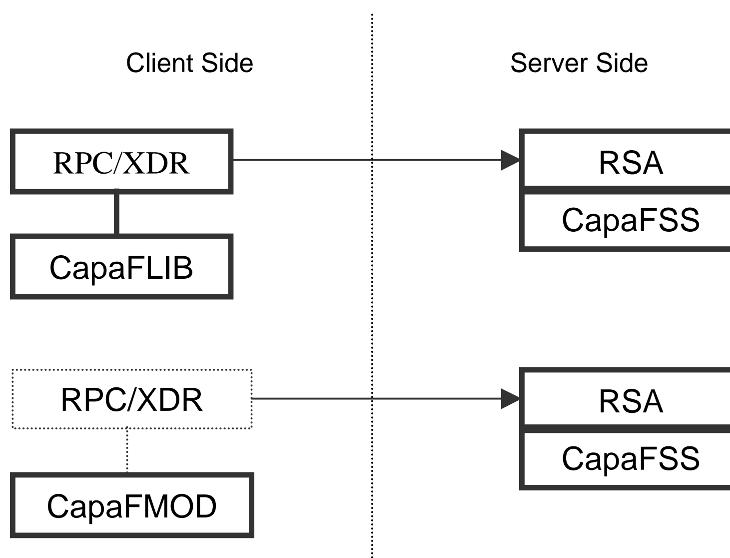

Figure 6.2: CapaFS future work

The main goal of the future work of CapaFS is to release a stable version of the CapaFSMOD loadable module. The only area of the module which is not implemented is the RPC specification as shown in Figure 6.2. Others areas of improvement include using a asynchronous RPC library which can handle multiple outstanding requests, and increase performance. The CapaFSLIB shared library

wrapper should port to most Unix platforms with little trouble. Although the kernel module uses the VFS layer of indirection, porting this to a platform which supports it, like Solaris, would still require some re-engineering. The CapaFS user-space server could also be ported with ease. We achieved network hardware and protocol independence in CapaFS by using standard network protocol stacks and protocols such as RPC, XDR, and TCP/IP. An important issue in distributed file system is the ability to work on a number of platforms and promote sharing amount them.

6.4 Conclusion

CapaFS demonstrates a globally accessable file system with a uniform namespace that does not need any central authority to manage the namespace. Capability filenames are a secure and practical solution to sharing resources. A portable, easy to install reference implementation shows that CapaFS can offer reasonable performance when comapred with systems like NFS. It also shows that the cost of software encryption can be tolerated in large networks like the Internet, where the cost of encryption is a small percentage of the total delay. CapaFS offers new functionality to users which sucessfully promotes the sharing of files and collaboration on the Internet, without imposing significant costs in terms of overhead and performance.

REFERENCES

[1] Steven M. bellovin, Michael merritt. Limitations of the Kerberos authentication System. In the 1990 October issue of Computer Communications Review, October 1990.

[2] Everhart, C. F. Conventions for names in the service directory in the AFS Distributed File System. In Technical report, Transarc Corporation, March 1990

[3] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun network filesystem. In proceedings of the summer 1985 USENIX, pages 119-130. USENIX, 1985.

[4] Sun Microsystems, Inc. NFS: Network file system protocol specification. RFC 1094, Network Working Group 1989.

[5] Postel, J. and Reynolds J. File Transfer Protocol (FTP). Request For Comments 959, USC Information Sciences Institute, Marina del Ray, Calif., October 1985

[6] Rao, H. C. The Jade File System. PhD thesis, University of Arizona 1991.

[7] B. Clifford Neuman. The Prospro File System: A Global File System Based on the Virtual System Model. In proceedings of the May 1192 workshop on File Systems, May 1992.

[8] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp. Experiences with the Amoeba Distributed Operating System. From

Department of Mathematics and Computer Science, Vrije University, Netherlands.

[9] John Ousterhout, Andrew Cherenson, Frederick Douglis, Michael Nelson, and Brent Welch. The Sprite Network Operating System. From Department of Electronic Engineering and Computer Sciences, University of California, Novermber 1987.

[10] Eshwar Belani, Alex Thornton and Min Zhou. Authentication and security in WebFS. From http://www.cs.berkeley.edu/WebOS/security.ps, January 1997

[11] Amin M. Vahdat, Paul C. Eastha, and Thomas E. Anderson. WebFS: A global cache coherent file system. From http://www.cs.berkeley.edu/~vahdat/webfs/webfs.html, December 1996.

[12] David Mazieres. Security and decentralised control in the SFS distributed file system. Master's thesis, Massachusetts Institute of Technology, August 1997.

[13] David Mazieres and M. Frans Kaashoek. Escaping the Evils of Centralised Control with self-certifying pathnames. From MIT Laboratory for Computer Science, 1997.

[14] Andrew D. Birrell, Andy Hisgen, Chuck jerian, Timothy Mann, and Garret Swart. The Echo distributed file system. Technical Report 111, Digital Systems Research Center, September 1993.

[15] Morrie Gasser, Andy Goldstein, Charlie Kaufman, and Butler Lampson. The Digital Distributed System Security Architecture. In proceedings of the National Computer Security Conference, 1989.

[16] Peter Reiher, Jr. Thomas Page, Gerald Popek, Jeff Cook, and Stephen Crocker. Truffles – a secure service for widespread file sharing. In proceedings of the PSRG Workshop on Network and Distributed System Security, 1993

[17] Ray Dillinger. The RSA Algorithm: countdown to 20 Sept 2000. Nov 1995-Jan 1996.

[19] Arjen K Lenstra, Paul Leyland. Large Integer Package, July 1997

[20] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. Ufo: A Personal Global File System Based on User-Level Extensions to the Operating System. In Proceedings of the 1997 USENIX Technical Conference, Anaheim, CA, January 1997.

[21] R. Srinivasan. RPC: Remote procedure call protocol specification version 2. RFC 1831, Network Working Group, August 1995.

[22] R. Srinivasan. XDR: External data representation standard. RFC 1832, Network Working Group, August 1995.