USENIX Association

Proceedings of the 10th USENIX Security Symposium

Washington, D.C., USA
August 13–17, 2001

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org

Capability File Names: Separating Authorisation from User Management in an Internet File System

Jude T. Regan[*] Christian D. Jensen
Department of Computer Science
Trinity College Dublin
juderegan@consultant.com Christian.Jensen@cs.tcd.ie

Abstract

The ability to access and share information over the Internet has introduced the need for new flexible, dynamic and fine-grained access control mechanisms. None of the current mechanisms for sharing information – distributed file systems and the web – offer adequate support for sharing in a large and highly dynamic group of users. Distributed file systems lack the ability to share information with unauthenticated users, and the web lacks fine grained access controls, i.e. the ability to grant individual users access to selected files.

In this paper we present Capability File Names, a new access control mechanism, in which self-certifying file names are used as sparse capabilities that allow a user ubiquitous access to his files and enables him to delegate this right to a dynamic group of remote users. Encoding the capaility in the file name has two major advantages: it is self-supporting and it ensures full compatablity with existing programs.

Capability file names have been implemented in a new file system called CapaFS. CapaFS separates user identification from authorisation, thus allowing users to share selected files with remote users without the intervention of a system administrator. The implementation of CapaFS is described and evaluated in this paper.

1 Introduction

The current suite of Internet protocols and applications [24, 28, 11] allows individuals and organizations convenient access to stored data from every corner of the globe. However, access to such data is typically read-only unless the user is identified on the server on which the data are stored. This prevents users from sharing selected files across organizational boundaries.

A distributed file system allows users read/write access to files stored on remote machines as if they were stored locally. This is a convenient abstraction because it hides the distribution of data from the user in the same way that remote procedure calls hide the distribution of processing. Moreover, most applications accept input from files, which makes distributed file systems the perfect medium for sharing among a heterogenous group of users on the Internet. A distributed file system must implement a flexible and scalable access control mechanism in order to serve different applications. Examples of such applications are mobile users, business-to-business (B2B) commerce and open software development projects.

Most distributed file systems [35, 5, 29, 34, 25, 22, 31, 2] 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 and enforce access control, the file system generally relies on a local database or password file of users in the system. Moreover, file systems are normally only "exported" to a designated set of machines.

Managing the sets of users that are authorized to access a set of files or the machines that the file system is exported to, is normally a privileged task performed by a small group of system administrators. This means that a third party has to intervene in order for two parties to collaborate. Inflexible local policies, encumbering mechanisms and overworked system administrators are often a major hindrance to collaboration.

As stated above, existing "discretionary" access control

[*] Jude Regan is currently employed as a Java Developer for Pfizer in Brussels, Belgium.

mechanisms limit the discretion of the user by imposing restrictions on what users are defined locally, what machines files can be exported to and what common authentication framework is required. We wish to extend discretionary access control mechanisms to the Internet, so that users may share their files with any remote user on any remote system without any limitation imposed by local system configuration.

The web provides a convenient medium for sharing of mainly read-only information among a large set of users. Access to information in a particular directory can be protected by passwords [6], thus providing a coarse grained access control mechanism. However, managing several groups with dynamic and overlapping access rights requires the owner to constantly copy, link, or move files among directories; this is not convenient.

We have thus identified the following four properties for a flexible and dynamic access control mechanism for sharing files on the Internet.

No Local Identification Users from different organizations must be able to collaborate, so the access control mechanism cannot rely on user identification in the local domain, e.g., a local data base of registered users.

User Autonomy Collaboration should be immediate, without the delays and hassle of asking a system administrator to define a new user, set up a new group or either export or import a particular file system.

Granularity The current mechanisms for sharing information on the Internet often require the information to be located in particular directories that are "exported" to the remote machines. Different directories may be exported to different sets of machines and with different access rights. Managing these different groups and access rights often requires a complex hierarchy of directories, which makes it difficult to maintain the information. It would be preferred if the user could leave the files where they are and share individual files directly with remote users.

Read/Write Sharing In order to support full collaboration, the access control mechanism must provide equally convenient read and write access to the shared information, although some files could be shared read-only.

Capability file names encode the access rights of the user into the name used by the client to access the file; the file name effectively becomes a capability for that file. The ability to present the capability file name to the file server is enough to allow the user to access the file with the rights encoded in the filename, so no identification is required. We have implemented a proto-type file server, called CapaFS, that acts as a proxy for a particular user and allows him to share his files with everybody on the Internet. The CapaFS server runs entirely in user space, so no intervention is required from the system administrators to set-up or run this service. Each capability file name encodes the access rights for a particular file on the server, this means that individual files are easily shared. The access rights encoded into the file name corresponds to the access rights of the server file system, which normally include both read and write access. An evaluation of the CapaFS proto-type shows that it meets all of the requirements listed above.

The rest of this paper is organized as follows: We start by examining related work on sharing in distributed file systems in section 2. Section 3 describes the principle behind capability filenames. Section 4 describes the design of CapaFS. We evaluate the implementation of CapaFS in Section 5. Some directions for future work are outlined in Section 6 and our conclusions are presented in Section 7.

2 Related Work

In the following, we examine a number of networked or distributed file systems that address some of the issues that we are trying to solve.

We present a short survey of existing access control mechanisms, before examining the distributed file systems.

2.1 Access Control Mechanisms

The access rights of all principals in the system are normally encoded in an access control matrix [20], either in the form of access control lists (ACLs) or capabilities.

An ACL is associated with every resource in the system. It lists all the users who are authorized to access the resource along with their access rights. Strong authentication mechanisms, such as Kerberos [18] or X.509 [27], allow ACLs to be used in a networked environment. The ACL relies on the authenticated identity of the user, which requires the identity of the user to be known before access can be granted. Moreover, ACLs do not scale very well, because it is difficult to delegate the right to delegate, i.e., the right to modify the ACL. This means that ACLs in large organizations have to be organized into a hierarchy where the right to delegate access rights, within a part of the organization, corresponds to the right to modify the ACL in the corresponding branch of the ACL hierarchy. Thus, the hierarchy of ACLs mustreflect the structure of the organization in which they are used. [9]

A capability is an unforgeable token that identifies a resource and lists the access rights granted to the holder of the capability. Anyone holding a copy of the capability may access the resource with the access rights specified by the capability. Capabilities can be stored in data structures in user space and copied or transferred among users. This makes it easy for users to create new adhoc work groups and to distribute access rights to that group. Because knowledge of a capability conveys right to access the specified object or file, capabilities must be protected from theft and disclosure, e.g., by encrypting part of the capability.

2.2 Amoeba File Server

Amoeba is a distributed object oriented operating system designed for a network of closely connected machines. Amoeba uses sparse capabilities to protect all objects in the system including files [38]. Capabilities are stored in data structures in a process' address space and can be exchanged freely among processes.

The Amoeba file service consists of two distinct servers: the directory server and the Bullet file server [32]. The directory server maps human-chosen ASCII names to capabilities used by the system. The Bullet file server implements the required functionality to create, read and write files. The create operation returns a capability that must be used by subsequent read and write operations. This capability can be stored in the directory service for later retrieval.

The sparse capability model implemented by the Amoeba file service has inspired the access control model used in capability file names. An important difference between the two systems is that capability file names are designed to ensure compatibility with existing applications, while Amoeba is not.

2.3 NFS

In NFS [34, 37], the server trusts the identification performed by the client. Clients and servers coordinate their user identifiers. File systems are explicitly exported to a designated set of clients. In order to access a file on the server, the user must be defined as a user on the server machine and the client machine must be added to the export list of the server. Both of these operations are privileged, i.e., require the intervention of a system administrator.

2.4 AFS

AFS [16, 35, 36, 10] is probably the most widely used wide-area file system. AFS mounts all remote file systems under a single directory (`/afs`). The set of remote directories available under the global mount point is managed locally and changes require the privileges of a system administrator.

AFS uses Kerberos [18] to authenticate users, which penalizes use across administrative boundaries. To facilitate collaboration, users of such systems form inconveniently large administrative realms, so anyone they may want to collaborate with will be in the same realm. Kerberos authentication suffers from this problem [4]. System administrators responsible for setting up user accounts often could not do so without the intervention of the Kerberos administrator. 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.

2.5 SFS

SFS [12, 22, 21] is a global decentralized file system. Like AFS it uses a global mount point to provide a single name space across all machines in the world while avoiding centralized control. Public-key cryptography is used to authenticate all entities in the system.

SFS introduces self-certifying pathnames to name files in this global name space. A self-certifying pathname consists of the file server's *location*, e.g., a host name or an IP address, and a *HostID* that tells the client how to certify a secure channel to the server.[1] The self-

[1] The HostID is actually a cryptographic hash of the server's location and its public key.

	No Local Identification	User Autonomy	Granularity	Read/Write Sharing
Amoeba	YES	YES	FILE	YES
NFS	NO	NO	FILE	YES
AFS	NO	NO	DIRECTORY	YES
SFS	NO	NO	FILE	YES
Truffles	NO	YES/NO	VOLUME	YES
WebFS	NO	YES/NO	FILE	YES

Table 1: Comparison of the surveyed file systems

certifying pathname entirely suffices to name and certify the file server.

SFS relies on a trusted third party to authenticate the user and the client machine. This means that collaboration is only possible if client and server have a common root for their certification authorities.

It is important to note that SFS self-certifying file names are used for authentication, not for authorization. Access control in SFS relies on user and group IDs, so the intervention of a system administrator with special privileges is required to create an account before a remote user can access files on the server. The authentication framework introduced with self-certifying file names is complementary to the authorization framework introduced with capability file names.

2.6 Truffles

Truffles [31] is a distributed file system that attempts to make file sharing between users in different administrative domains both simple and secure. It uses the Ficus file system [15] to offer replication and sharing of files and Privacy Enhanced Mail (PEM) as a secure transport mechanism. Truffles allows users to share volumes (subsets of the entire file system) with little intervention from the system administrators once the volumes have been defined. Ficus replicates volumes using an optimistic one-copy policy and all systems sharing the volume may hold their own replica. However, file sharing relationships and file data transfer rely on Privacy Enhanced Mail (PEM) which requires a nonempty intersection between the public key infrastructures that each user belongs to; the slow adaptation of secure HTTP shows how this limits collaboration among users.

2.7 WebFS

WebFS [39, 2] is a global file system that uses HTTP [11] as the transport protocol between client and file server. The advantage of this approach is that existing URLs can be used as file names and accessed through the file system.

Authentication of both clients and servers are based on X.509 certificates [3]. WebFS maintains an ACL for each file consisting of the X.509 certificate and permissions for each authorised user. Users may therefore share their files with anyone who has a certificate *from a certification authority (CA) trusted by the local domain*. Managing the set of trusted CAs requires special privileges. Thus, authorisation in WebFS relies on the hierarchy of certification authorities, which places the users of WebFS under the control of these certification authorities.

2.8 Summary

A summary the surveyed systems is presented in Table 1.

NFS and AFS are very similar. System administrators are required to enumerate all exported file systems and all machines with remote access. With NFS, all machines form part of the trusted computing base, while AFS supports sharing among machines in different administrative domains. SFS allows free choice of authentication mechanism, but authorization relies on local ACLs. Both Truffles and WebFS allows dynamic sharing of files among users who trust the same certification authority, but it is impossible to share files with users without a recognized certificate. All the surveyed file systems provide read/write access and fine granularity, so they appear not to be real issues. However, one of the most popular media for information sharing, the web, provides little support for either.

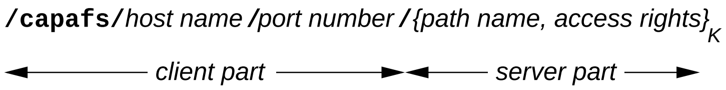


Figure 1: Basic capability file name

3 Capability File Names

The goal of capability file names is to allow users belonging to one or more organizations, to set up ad-hoc work groups without limitations imposed by the system or the intervention of their system administrators. Each user in the group should be allowed to share selected files with other members of the group, without compromising his remaining files, nor the files of other users on his system.

3.1 Basic Capability File Names

The basic capability file name consists of two parts, the "client part" that allows the client machine to identify the server and the "server part" which allows the server to identify the file and encodes the access rights. The server part is encrypted by the server to protect it from tampering. The structure of a capability file name is shown in Figure 1.

The client part of the capability file name consists of a prefix ("/capafs/") that identifies it as a capability file name. It also contains the host name of the machine running the server and the port number used by the server.

The server part consists of the absolute path name of the file on the server and the permissions (-rwx) granted to the client presenting the capability filename. The server part is encrypted (with key K) to protect it from tampering.

This design is similar to the sparse capabilities used in Amoeba (cf. 2.2). However, sparse capabilities are physical objects that are supported by the system and manipulated explicitly by programs. Capability files names are virtual objects that require no special support from the underlying system and they are manipulated as ordinary file names by programs. This difference is best illustrated by analogy with plane tickets. Traditional capabilities are equivalent to paper tickets (data structures) that have to be acquired and presented at the check-in desk (the file server) in order to grant access to the flight (the file). Capability file names are equivalent to confirmation numbers used when tickets are bought on the Web. Knowledge of a valid confirmation number (capability file name) is proven to the check-in desk in order to grant access to the flight.[2]

3.1.1 Creation of a Capability File Name

Capability file names are created on the server using a separate program. This program takes the host name and the port number that the server is using, the path name of the file and the permissions to be encoded into the capability file name as parameters. It then reads the encryption key from a file stored in the user's home directory and encrypts the server part with this key. It then creates a string by concatenating the prefix with the host name, the port number and the server part and returns it in a string to the remote user. Only the server needs the ability to decrypt the server part of the capability file name, so a fast symmetric cipher may be used. However, a number of possible extensions rely on public-key cryptography, so the server part could also be encrypted using the server's private-key. This reduces the number of keys that the server has to maintain, but it also reveals the contents of the server part of the capability file name.

3.1.2 Using Capability File Names

Each user, who wishes to share his files, must start a server to act as a proxy for remote file operations. This server is described in greater detail in section 4.4.

In order to prevent disclosure of the capability file names and to ensure the confidentiality and integrity of transferred file data, all communication between client and server must use a secure channel. This channel must encrypt all communication, but need not authenticate the end-points to each other, e.g., a fast symmetric encryption algorithm may be used with the Diffie-Hellman key exchange [7]. This solution is vulnerable to the man-in-the-middle attack, an extension that solves this problem is proposed in Section 3.2.

[2] This analogy breaks if two people present the same confirmation number to the check-in desk. The file server grants both users access to the file, while only one person can physically board the plane.

3.1.3 Delegation of a Capability File Name

Secure delegation of capability file names is orthogonal to the mechanism described in this paper, however it is important that the capability file name is protected from disclosure while in transit.

3.1.4 Persistence of a Capability File Name

Capability file names are not persistent in themselves; they are simply names (i.e., character strings) that are lost when the client terminates or if the client fails; server failure does not affect the validity of a capability file name. However, a capability file name can be stored on stable storage or serve as the source of a symbolic link, thus making it persistent. Using a symbolic link to point to a capability file name allows its holder to assign a meaningful name to the remote file, although this name only has local significance.

3.1.5 Revocation of a Capability File Name

In order to allow revocation of capability file names, the server must maintain a capability revocation list (CRL) of all capability file names that have been revoked. The CRL grows with time and searching through it may become prohibitive. One solution is to limit the time that a capability file name is valid (i.e., include a timeout value in the server part). As the timeout is only used on the server, client and server clocks do not have to be synchronized. However, new versions of the capability file names must be acquired when the capability file name expires.

Another solution is to mark files that have a revoked capability file name associated with them (e.g., changing the files meta-data, such as the inode number), the CRL is only searched if the file is marked. The CRL is still needed because there may be number of valid operations that change the meta data, e.g., restoring a file from backups changes the inode number.

3.2 Capability File Names with Server Authentication

The secure communication channel discussed in the basic scheme above suffers from the man-in-the-middle attack, where a third party intercepts the initial message from the client and sets up connections to both client and server. The man-in-the-middle relays all messages between client and server and knows both keys. We therefore need an authentication mechanism that escapes the problems of centralized control. In order to authenticate the server, its public-key (SPuK) is added to the client part of the capability file name. In this case the server's private-key (SPrK) is used to encrypt the server part of the capability file name. The structure of a capability file name with server authentication is shown in Figure 2(a).

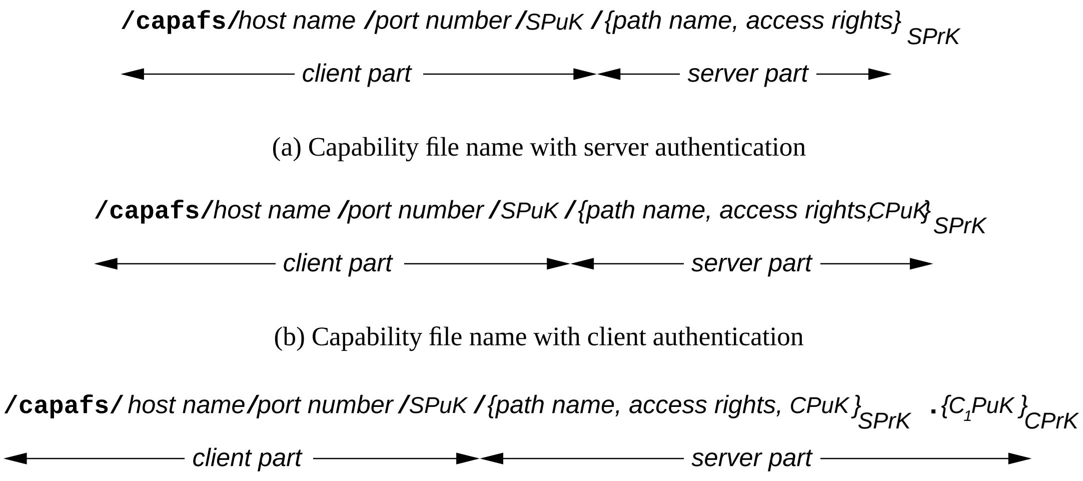
The server is authenticated in the following way before the client sends the capability file name to the server. The client selects a session key to be used by the secure channel. The session key is encrypted with the server's public-key and sent to the server. The server responds with a message encoded with the session key, which proves its possession of the server's secret key and thereby authenticates the server. The client is implicitly authenticated as belonging to the set of authorized user, when the server receives the capability file name. This authentication protocol is very similar to the protocol used in SFS [23].

3.3 Capability File Names with Client Authentication

As mentioned above, the client is implicitly authenticated through the possession of the capability file name. However, knowing the identity of the client "enables the monitoring, mediating, and recording of capability propagations to enforce security policies including the $\star$-property in the Bell-LaPadula model" [14]. Moreover, it introduces accountability into the system. The structure of a capability file name with client (and server) authentication is shown in Figure 2(b).

The identity of the client is used on the server side and should be included in the server part *before* the capability file name is given to the client.

The authentication of the client follows the scheme described in Section 3.2. After the client has received the first message with the session key from the server, it signs the capability file name with his private-key and sends it to the server. It is important to note that the authentication of the client does not necessarily depend on his physical identity; the key-pair used could be created explicitly for this capability file name.


/capafs/host name /port number /SPuK /{path name, access rights} SPrK
client part
server part
/capafs/host name /port number /SPuK /{path name, access rights,CPuK} SPrK
client part
server part
/capafs/ host name/port number /SPuK /{path name, access rights, CPuK}SPrK .{C1PuK }CPrK
client part
server part

(a) Capability file name with server authentication

(b) Capability file name with client authentication

(c) Capability file name propagation chain

Figure 2: Extensions to the basic capability file names

3.4 Capability File Names with Propagation Limitation

The introduction of client authentication above, allows us to monitor the use of capability file names. The addition of delegation chains [1], allows the server to impose restrictions on the further delegation of a capability file name, e.g., restricting further delegation to a known set of recipients, restricting the right to delegate the capability file name to the original recipient or preventing delegation altogether. Moreover, it allows the server to implement different delegation policies for different files. This approach is similar to transfer certificates in CRISIS [3] or authorization proxies implemented on top of Kerberos [26].

The structure of a capability file name with propagation limitation is shown in Figure 2(c).

A user (C) who wish to further delegate a capability file name to another user (C_1), encodes the public-key of the recipient (C_1PuK) with his own private-key (CPrK) and appends it as an extension to the filename. Further delegation of this capability file name will add another extension, so a long delegation chain means that the filename will have many extensions. When the server receives the delegated capability filename, it first retrieves the public-key of the original recipient (CPuK). It uses this key to decrypt the public-key included in the first extension, this process is repeated until the final public-key is retrieved. This final key is then used to authenticate the requesting client, as described in Section 3.2.

3.5 Summary

Capability file names allow flexible and dynamic sharing of files among users in different organizations, without the intervention of system administrators in either organization. A number of extensions to the basic capability file name allow clients and servers to authenticate each other. This authentication does not rely on digital certificates to prove the physical identity of either party. Instead, the knowledge of a private-key is used to authenticate users. This allows collaboration without compromising the semi-anonymous nature of the Internet, i.e., a user may assume a virtual identity and associate it with a public/private key pair. The user may then use a grassroots mechanism, such as the PGP web of trust, to distribute the public-key associated with this virtual identity.

4 CapaFS

CapaFS [30] is a userlevel file system that implements basic capability file names. CapaFS uses AES [17, 8] for symmetric encryption and RSA [33] for asymmetric encryption of the server part of the capability file name. AES is also used to establish a secure channel between client and server. We currently use PGP [13] to provide confidentiality and authentication of capability file names distributed per email, i.e., we do not require a traditional hierarchical public key infrastructure.

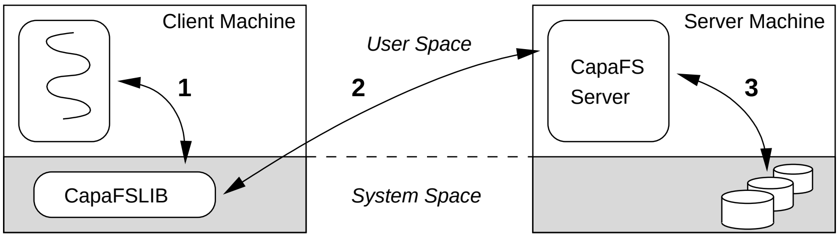


Figure 3: The CapaFS architecture

4.1 Overview

CapaFS consists of two parts: a shared library that replaces `libc` on the client machine and a user level file server. The architecture of CapaFS is illustrated in Figure 3.

The library intercepts all operations on files under `/capafs` (1) and redirects them to the designated server (2). If the operation is `open()`, the server verifies the capability file name and returns a file handle or an error to the client. All other operations use this file handle to access the file, [3] so the server only has to decrypt the secret part of the capability file name once. Clients with valid file handles can perform remote file operations using standard NFS semantics (3).

4.2 Creating Capability File Names

Two programs are used to create a capability file name: `CapaFSKeys` creates an encryption key, either a symmetric key or the private-key of a public/private key pair, and `CapaFSFile` produces a capability file name from the file's location, the server's port number, the remote user's access rights, and the encryption key created by `CapaFSKeys`.

It is important to note that CapaFS is not a distributed file system, i.e., all files are created locally, it simply provides remote users with access to locally stored files. The symbolic link mechanism described in Section 3.1.4 can be used to create the illusion of local and remote files stored in the same directory.

4.3 Capability File System Wrapper Library

The CapaFS shared library CapaFSLIB is used by a client to add the necessary functionality to 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` and `fcntl`.

CapaFSLIB can be installed in any directory as long as the shared library loader knows where to find it; this is normally achieved by setting a variable in the user's environment. This means that Capability File Names can be installed without any intervention from the system administrators, thus promoting user-to-user collaboration.

4.4 Capability File 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 user's or system files, thus providing no extra threat to systems.

A user who wants to share his files with others simply 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.

[3] The file handle is a large randomly chosen integer, thus effectively a sparse capability.

4.5 Granularity

The CapaFS server effectively acts as a proxy between the remote user and the local file system. Capability file names can be used to grant access to both individual files, and to all files in a directory.

Standard file system semantics apply to the remote operations on both files and directories with the following exceptions: file operations are executed with the intersection of the access rights of the user who started the server and the permissions encoded in the capability file name, and the right to access a file in a particular directory implies the right to search all higher level directories.

5 Evaluation

In the following we present an evaluation of the functionality, security and performance of the developed prototype.

5.1 Functionality

The functionality of CapaFS has been evaluated through the following scenario. Two researchers are writing a paper for a scientific conference; one author is located in Belgium, the other in Ireland. The security policies of their respective organizations prevent either from easily obtaining an account on the other's system.

With CapaFS, any user can start a server on a machine with access to the Internet, thus allowing him to share his files with anybody without the intervention of the system administrator of his machine.

They decide to format the paper with the LaTeX [19] text preparation system. The first author writes an initial draft of the paper, creating all the required LaTeX source files in the process; a listing of the source directory is shown in Figure 4(a).[4] He then starts the CapaFS server, creates capability filenames, with read/write permission, for all LaTeX files and sends them to the second author over a secure channel.

The second author then creates a new local directory and symbolic links in that directory that point to the received capability file names. A listing of the second author's source directory is shown in Figure 4(b). The second author can now edit the source files directly and process them with LaTeX locally.

A separate capability file name is used for each file – instead of a directory capability file name – to improve the speed of formatting the document. A directory capability file name would mean that temporary files created by LaTeX would be created on the server. Instead, these files are created locally. The directory of the second author – after processing the source files with LaTeX – is shown in Figure 4(c).

In our example one author created all the files and distributed the required capability filenames to the other, but the system allows both users to start CapaFS servers and thus both users to create new files.

5.2 Security

We assume that an attacker has control over the network between client and server, but that the security of neither client nor server machine has been compromised.

An attacker who controls the network may attack a system by interception, interruption, modification and fabrication of messages.

Interception An eavesdropper reads the message as it passes on the network, thus compromising the confidentiality of the message. CapaFS ensures confidentiality by encrypting the communication between client and server.

Interruption Preventing the message from reaching its destination results in denial of service. This type of attack cannot be prevented without complete control over the network.

Fabrication Creation of new messages allows an attacker to masquerade as either client or server. Masquerading as the client requires him to know a capability file name, but they are only sent over secure channels which means that they cannot be known by outsiders. Masquerading as the server allows the attacker to learn capability file names as clients connect to it and send erroneous data to the client. In Section 3.2 we proposed a mechanism similar to the self certifying file names in SFS;

[4] Non-essential details have been deleted from the directory listings. In order to protect the local system, the host name and port number have been changed and the server part of the capability file name is truncated after 8 characters.

```
-rw-r--r--  [...deleted...] paper.bib
-rw-r--r--  [...deleted...] paper.tex
```

(a) Directory on the server

```
lrwxr-xr-x  [...deleted...] paper.bib -> /capafs/fs.dsg.cs.tcd.ie/9999/5be34dd[...deleted...]
lrwxr-xr-x  [...deleted...] paper.tex -> /capafs/fs.dsg.cs.tcd.ie/9999/715a9f3[...deleted...]
```

(b) Directory on the client

```
-rw-r--r--  [...deleted...] paper.aux
-rw-r--r--  [...deleted...] paper.bbl
lrwxr-xr-x  [...deleted...] paper.bib -> /capafs/fs.dsg.cs.tcd.ie/9999/5be34dd[...deleted...]
-rw-r--r--  [...deleted...] paper.blg
-rw-r--r--  [...deleted...] paper.dvi
-rw-r--r--  [...deleted...] paper.log
lrwxr-xr-x  [...deleted...] paper.tex -> /capafs/fs.dsg.cs.tcd.ie/9999/715a9f3[...deleted...]
```

(c) Directory on the client after processing the files

Figure 4: Typesetting a paper on a remote machine

this mechanism is not yet implemented. Adding the servers public key to the client part of the capability file name allows the client to authenticate the server.

Modification The integrity of file data is compromised if the message is modified between the client and the server. However, modified messages cannot go undetected because the client and the server communicate over an encrypted link.

Encryption of communication between client and server ensures confidentiality and integrity. Clients are explicitly not authenticated, because requests may arrive from any node, and knowledge of the capability file name is enough to grant access to the file. Authentication of the server is easily achieved by including the public key of the server in the capability file name as described in Section 3.2.

5.3 Performance

Performance was not of primary concern to us, but the capability file name mechanism is unlikely to be widely adopted if it introduces a very large overhead. We therefore decide to evaluate the performance of CapaFS. The performance of CapaFS is compared to NFS, a widely used networked file system. However, it is important to note that CapaFS is designed to provide functionality which NFS cannot provide, so the comparison should only be taken as an indication of the usability of CapaFS.

There are several factors, which might differentiate CapaFS performance from NFS performance. First of all CapaFS runs entirely in userspace, while NFS has been integrated into the operating system kernel. Second, CapaFS is more CPU intensive than NFS because of the RSA public key encryption and integrity checks performed on CapaFS capability filenames. The open operation requires the user-level server to decrypt the server part of the capability file name, in order to reveal the filename and permissions. The overhead of decrypting the capability is proportional to the bit-size of the encryption used. Finally, NFS has been fine-tuned for more than a decade, while CapaFS is a recent prototype implementation, e.g., none of the caching strategies used by NFS have currently been implemented in CapaFS.

The performance of CapaFS and NFS is now compared and discussed. The tests of both CapaFS and NFS were performed using two 1GHz Pentium III machines running RedHat Linux with 256MB of memory. The two machines were connected with 100Mbit Ethernet through a 100Mbit switch.

The client and server were set up on different machines running in standard multi-user mode. The tests covered the following operations for both CapaFS and NFS: Opening or lookup of a remote file, reading a 1KB from a remote file, and writing a 1KB to a remote file.[5]

All tests are timed from the point the operation is invoked, until the point when a result is returned. In addition to this, all measurements are performed on the client side, accessing files on the server side as usual. The performance results of these tests are given in Table 2.

File operation	CapaFS	NFS
`open()`	1292 μs	159 μs
`read()`	117 μs	94 μs
`write()`	987 μs	8 μs

Table 2: Performance comparison of CapaFS and NFS

Our measurements show that CapaFS has an acceptable absolute performance, the most expensive operation (`open()`) costs little over a millisecond, so the cost of file system operations are dominated by communications costs in a wide area network.

CapaFS takes significantly longer than NFS to open a file (x10), but it is only called once when the file is initially opened (subsequent read and write operations use a file handle returned by the open call). The higher cost of the open command is to be expected, because the server part of the capability file name has to be decrypted. The cost of reading data is roughly equivalent in the two systems, while the cost of writing data to a remote file is significantly higher in CapaFS (x100). We attribute the big difference in write performance of NFS and CapaFS to NFS's use of asynchronous writes, which makes NFS significantly faster when writing data to a remote file. The asynchronous write strategy is acceptable on local area networks, where the probability of errors and partitions is low, but we do not believe that such optimizations are appropriate in a file system designed for use in wide area networks.

5.4 Summary

We have shown that CapaFS meets all of the requirements defined in Section 1.

No Identification CapaFS allows dynamic sharing of selected files, without identification of the remote user; the knowledge of the capability file name is enough to grant access. Neither of the users described in Section 5.1 holds an account on the other user's machine.

No Intervention Both CapaFSLIB and the CapaFS server runs in user space and were setup without the intervention of the system administrator. However, if every user is to run a CapaFS server, some support from system administrators would be needed to coordinate the local use of port numbers. We didn't experience problems with port number allocation during these experiments.

Fine Granularity Capability file names can be used to grant access to individual files, as well as directories.

Read/Write Access The measurements presented in Section 5.3 successfully read and wrote files across the Internet.

6 Future Work

The current implementation of CapaFS relies on a wrapper library on the client's machine. We would like to extend this with a file system implemented as a loadable kernel module. This allows us to implement efficient caching policies for remote files and decreases the overhead of context-switching between the user library and the kernel. Installation of the loadable kernel module for the file system requires the intervention of the system administrator, but this installation can be performed once for all users.

The propagation limitation mechanism outlined in Section 3.4 should also be implemented. This reduces the risk of sharing files by limiting the number of users authorized to delegate the capability file names.

7 Conclusions

In this paper we addressed the issue of sharing information in large open (Internet) distributed file systems.

We presented a new access control mechanism designed to facilitate sharing among dynamic groups of non-authenticated users. This design is implemented in CapaFS, a global and decentralized file system that allows

[5] Link-level encryption has been disabled in CapaFS in order to provide comparability with NFS.

users to collaborate with other users regardless of location and with no prior arrangements or intervention by system administrators. 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 has no need to authenticate the client machine to the server. A client must simply prove possession of a valid capability filename; this is necessary and sufficient proof of authority to perform the operations – encoded in 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.

Capability file names 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, may use CapaFS to facilitate and promote sharing.

A decentralized file system with global authentication and flexible authorization can free users from many of the problems that have developed due to increased security and centralized control.

Acknowledgements

The authors would like to thank our colleague Stefan Weber for his help with the performance evaluation presented in this paper, and also the paper's anonymous reviewers for their comments, which have helped us to improve the paper.

References

[1] T. Aura. Distributed access-rights management with delegation certificates. In J. Vitek and C.D. Jensen, editors, *Secure Internet Programming*, number 1603 in Lecture Notes in Computer Science LNCS, pages 211–235. Springer Verlag, 1999.

[2] E. Belani, A. Thornton, and M. Zhou. Authentication and security in WebFS, January 1997.

[3] E. Belani, A. Vahdat, T. Anderson, and M. Dahlin. The crisis wide area security architecture. In *Proceedings of the 7th USENIX Security Symposium*, pages 15–29, San Antonio, Texas, U.S.A., January 1998.

[4] S. M. Bellovin and M. Merrit. Limitations of the Kerberos authentication system. *Computer Communications Review*, 20(5):119–132, October 1990.

[5] A. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart. The Echo distributed file system. Technical Report 111, Digital Equipment Corp. Systems Research Center, 1993.

[6] K. Coar. *Using .htaccess Files with Apache*, 2000.

[7] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT–22(6):644–654, November 1976.

[8] Federal Information Processing Standard *Draft*. *Advanced Encryption Standard (AES)*. National Institute of Standards and Technology, 2001.

[9] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. Technical Report 2693, Network Working Group, IETF, September 1999.

[10] C. F. Everhart. Conventions for names in the service directory in the AFS distributed file system. Technical report, Transarc Corporation, March 1990.

[11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. Request for Comments (RFC) 2616, Network Working Group, IETF, 1999.

[12] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *Proceedigs of the 4th Symposium on Operating Systems Design and Implementation*, pages 181–196, San Diego, California, U.S.A., October 2000.

[13] S. Garfinkel. *PGP: Pretty Good Privacy*. O'Reilly & Associates, Inc., 1994.

[14] L. Gong. A secure identity–based capability system. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 56–63, Oakland, California, U.S.A., May 1989.

[15] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page Jr., G.J.Popek, and D. Rothmeier. Implementation of the Ficus replicated file system. In *Proceedings of the Summer USENIX Conference*, pages 63–71, June 1990.

[16] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, m. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[17] V. Rijmen J. Daemen. The block cipher rijndael. In J.-J. Quisquater and B. Schneier, editors, *Smart Card Research and Applications*, Lecture Notes in Computer Science (LNCS) 1820, pages 288–296. Springer-Verlag, 2000.

[18] J. Kohl and C. Neuman. The Kerberos network authentication service (v5). Request for Comments (RFC) 1510, Network Working Group, IETF, September 1993.

[19] L. Lamport. *LaTeX – A Document Preparation System – User's Guide*. Addison-Wesley, 1985.

[20] B. W. Lampson. Protection. In *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems*, pages 437–443, March 1971. reprinted in *Operating Systems Review, 8, 1* January 1974 pages 18–24.

[21] D. Mazières. Security and decentralised control in the SFS distributed file system. Master's thesis, MIT Laboratory of Computer Science, 1997.

[22] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the 17th Symposium on Operating Systems Principles*, pages 124–139, Kiawah Island, S.C., U.S.A., 1999.

[23] David Mazières and M. Frans Kaashoek. Escaping the evils of centralized control with self-certifying pathnames. In *Proceedings of the 8th ACM SIGOPS European workshop: Support for composing distributed applications*, pages 118–125, Sintra, Portugal, September 1998.

[24] N. J. Neigus. File transfer protocol for the ARPA network. Request for Comments (RFC) 542, Bolt Beranek and Newman, Inc., August 1973.

[25] B. C. Neuman. Prospero: A tool for organizing internet resources. *Electronic Networking: Research, Applications and Policy*, 5(4):30–37, 1992.

[26] B. C. Neuman. Proxy-based authorization and accounting for distributed systems. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 283–291, Pittsburgh, Pennsylvania, U.S.A., May 1993.

[27] Telecommunication Standardization Sector of ITU. *Information Technology — Opens Systems Interconnection — The Directory: Authentication Framework*. Number X.509 in ITU–T Recomandation. International Telecomunication Union, November 1993. Standard international ISO/IEC 9594–8 : 1995 (E).

[28] J. B. Postel. Simple mail transfer protocol. Request for Comments (RFC) 821, Information Sciences Institute, University of Southern California, August 1982.

[29] H. C. Rao and L. L. Peterson. Accessing files in an internet: The JADE file system. *IEEE Transactions on Software Engineering*, 19(6):613–624, June 1993.

[30] J. Regan. Capafs: A globally accessible file system. Department Technical Report TCD-CS-1999-70, Department of Computer Science, Trinity College Dublin, 1999.

[31] P. Reiner, T. Page Jr., G. Popek, J. Cook, and S. Crocker. Truffles – a secure service for widespread file sharing. In *Proceedings of the Privacy and Security Research Group Workshop on Network and Distributed System Security*, 1994.

[32] R. van Renesse, A. S. Tanenbaum, and A. Wilschut. The design of a high-performance file server. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 22–27, Newport Beach, california, U.S.A., June 1989.

[33] R. L. Rivest, A. Shamir, and L. Adleman. On a method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[34] R. Sandberg, D. Goldberg, Kleinman S, D. Walsh, and B. Lyon. Design and implementation of the Sun Network File System. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130, Portland, Oregon, U.S.A., June 1985.

[35] M. Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions on Computer Systems*, 7(3):247–280, 1989.

[36] M. Satyanarayanan. Scalable, secure and highly available file access in a distributed workstation environment. *IEEE Computer*, pages 9–21, May 1990.

[37] Sun Microsystems Inc. NFS: Network file system protocol specification. Request for Comments (RFC) 1094, Network Working Group, March 1989.

[38] A. S. Tanenbaum, S. J. Mullender, and R. van Renesse. Using sparse capabilities in a distributed operating system. In *Proceedings of the 6th International Conference in Computing Systems*, pages 558–563, June 1986.

[39] A. Vahdat, P. Eastham, and T. Anderson. Webfs: A global cache coherent file system. Department of Computer Science, UC Berkeley, Technical Draft, 1996.