A Java Network File System for Network Computers

Michael John Radwin

5 May 1997

Abstract

We introduce and discuss the Java Network File System (JNFS), a network file system for Network Computers (NCs). JNFS works on all NC–*compliant* NC devices, provides authentication and authorization support, works with other file systems such as NFS and NTFS, and offers reasonable performance.

1 Background and Motivation

Network Computers, or NCs, are "low–cost, easy–to–use network computing devices" [1]. The concept of the NC was introduced last year to address the costs and complexities of owning and maintaining personal computers (PCs). The two central tenets of the NC are

> ... to be cheaper and easier to maintain and use than PCs, and to provide access to corporate networks and the Internet. They are cheaper because they don't provide local storage, and lack the processing and video capabilities (as well as the other bells and whistles) that are standard on today's PCs. They are easier to maintain because servers control the software management, including downloading applications as they are needed, primarily using Sun's platform independent Java [6].

1.1 Network Computer Reference Profile

The Network Computer Reference Profile (NCRP) is a set of guidelines offered by Apple, IBM, Netscape, Oracle and Sun as a "common denominator of popular and widely used features and functions" [1]. It defines minimum hardware requirements and software protocols necessary to achieve NC–*compliance*. Among the required protocols and software are TCP/IP and the Java Application Environment.

The NCRP specification proposes that Sun's Network File System (NFS) be optionally used as a file system for NC devices: "NCs which do not implement a distributed file system need not implement this protocol" [1]. It does, however, mandate support of the TCP/IP–based Hypertext Transfer Protocol (HTTP) to enable Web browsing and the File Transfer Protocol (FTP) for file exchange.

1.2 NFS on Network Computers

1.2.1 NFS Support on Low End NCs

Many of today's low–end NCs come with no NFS support or only provide it as an option. HDS Network Systems, for example, makes a family of NC–compliant devices called the @workStation. In its most basic configuration, the @workStation has no file system support, although it does have a full Java implementation. More expensive @workStation configurations include full NFS support. Boundless Technology's Network Computer XL and XLC are configured similarly; Java support comes on all computers and NFS is optional.

1.2.2 NFS Support on High End NCs

Other vendors are committed to providing NFS client support. IBM's Network Station, for example, comes standard with NFS. Sun's JavaStation runs the JavaOS [10], which supports NFS at the kernel level. Wyse technology's WinTerm 4000 runs on top of the JavaOS, so it would support NFS as well.

1.2.3 Advantages of NFS

In addition to its widespread acceptance, platform–independence, and reliability [3], NFS has proven to be very efficient. Version 3 of NFS provides numerous performance enhancements over Version 2. Moreover, NFS can be implemented at the kernel level, which is substantially more efficient than a user–level daemon [9].

1.3 File System Alternatives for Network Computers

There are several alternatives to NFS that provide some form of file system support on an NC. We briefly discuss each below.

1.3.1 File Transfer Protocol

The FTP protocol [11] was designed for file transfers using TCP/IP. The model of use described by the protocol's architects expects that file transfers occur between a host file system and a client file system. Since NCs do not have a local file system (although they may optionally have local secondary storage for caching), the FTP protocol does not fit this model well.

1.3.2 Hypertext Transfer Protocol

The HTTP/1.0 protocol [2] was designed for transfer of hypertext and multimedia files from a server to a web browser and, to a lesser extent, transfer of data from a web browser to a web server. HTTP/1.0 specifies a GET operation that transfers a file from a server to the browser. It also defines a POST operation which is designed for annotations, posting messages, submitting form contents, and appending entries to databases. This POST operation may be extended beyond its intended use to allow FTP–like transfer of files from the client to the server. HTTP/1.0 also provides a weak challenge–response authentication protocol (WWW–Authenticate) that requires that passwords be sent in clear text over the network.

The newer HTTP/1.1 protocol [4] introduces a new PUT operation that is designed for transfer and storage of an enclosed file from the client to the server, and a DELETE operation that requests that a file on the server be removed. It also introduces the notion of persistent HTTP connections and other mechanisms that improve efficiency. HTTP/1.1 provides another weak challenge–response protocol (Digest) that does not require clear text passwords [5]. In sum, HTTP/1.1 provides a more reasonable model for a file system than HTTP/1.0, although it is not widely deployed among today's web servers.

1.3.3 WebNFS

The WebNFS protocol [3] is described as "the filesystem for the World Wide Web". WebNFS, however, is not a substitute for NFS. Instead, WebNFS is both a URL scheme for specifying filenames in a web browser, and a small set of extensions to NFS that allow it to be used more efficiently over the Internet. The NCRP does not mandate or suggest WebNFS support, so it does not work with minimally NC–compliant devices.

2 The Java Network File System

In this paper we introduce the Java Network File System (JNFS), a network file system for NCs. We offer this alternative for two reasons. First, since the NCRP does not require a file system and lower–end NCs do not provide one, JNFS fills this need. Second, since JNFS provides a network file system interface to a host's native file system, all NCs can access file systems other than NFS.

2.1 Design Goals

We had several goals in mind when we designed JNFS. First and foremost, the file system client had to run on all network computers. Since a number of vendors are shipping several different types of NCs, it would be ideal to develop a single client that works on all of them. Second, the file system had to provide some form of authentication and access control to ensure that unauthorized users could not gain access to files. In addition, the file server had to be interoperable with other types of file systems, and performance had to be reasonably good. Our implementation of JNFS achieved many of these goals:

- Because the NCRP mandates support of the Java Application Environment [1], a Java implementation of JNFS ensures that it will work on all NC–compliant devices.
- Good security is provided by both an authentication protocol based on digital signatures, and an Access Control List implementation for granting access to files.
- Interoperability with other file systems is provided because JNFS runs on top of a native file system. Thus, JNFS can provide access to files served over both local file systems such as NTFS or a UNIX local file system, as well as network file systems such as NFS or DFS.
- Performance is slower than NFS, but reasonable for a Java application. On average, JNFS file transfers take about 8 times as long as analogous NFS transfers.

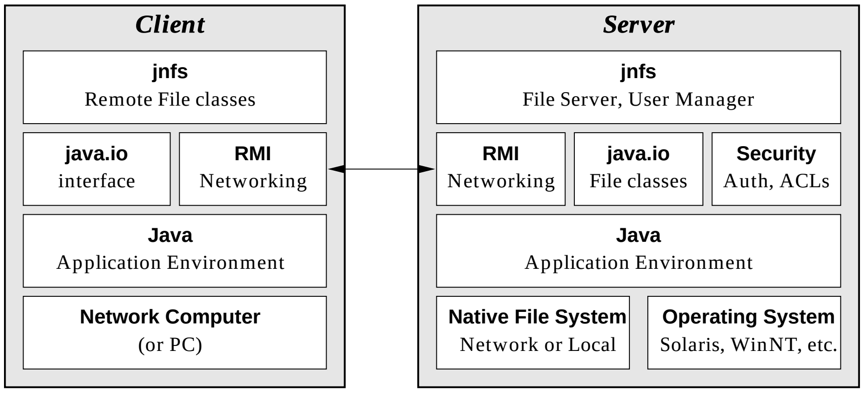

Figure 1: An overview of the JNFS architecture. Both the client and server are implemented in Java and communicate using RMI. The client runs on any NC or PC with the Java Application Environment; the server runs as a user-level daemon over a native file system and operating system.

3 Architecture

Both the JNFS client and server were developed with Sun's Java Development Kit 1.1. The client will run on both any NC–compliant device as well as PCs with the Java support. The server is also written Java for ease of portability. An overview of the architecture is presented in Figure 1.

3.1 File Server

The JNFS server runs as a user-level daemon as the privileged user (for example, *root* on UNIX, or any user with the *Administrative Privilege* on Windows NT). It serves out files from the underlying file system(s) used by the host operating system. Thus, if the server's operating system provides support for distributed file systems such as NFS, JNFS will serve those files out as well.

JNFS also employs a small number of `native` method calls to obtain file permission information from the underlying file system. The authorization policy is discussed in detail in Section 4.2.

3.2 Network Communication

All communication between the JNFS client and server is achieved with the Java Remote Method Invocation [8] mechanism. Remote Method Invocation (RMI) is a framework for "distributed Java–to–Java applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts."

In many respects, RMI is like Remote Procedure Call (RPC), an abstraction built on top of IP that provides a type-safe mechanism for calling functions remotely. Like RPC, RMI provides connection establishment and shutdown, and automatic marshaling and un–marshaling of arguments and return values. RMI, like the newer RPCs (Microsoft and DCE, for example), is also well–integrated with exception–handling mechanisms; exceptions thrown while invoking a remote method on the server side are thrown across the network to the client.

However, since RPC is procedure–based, it does not fit well in the object–oriented model that Java provides. RMI provides RPC–like functionality for Java at the object level. Another difference is that RPC may be used over several different protocols, including TCP or UDP; RMI runs only over TCP or HTTP.

3.3 Remote File Classes

At the core of JNFS is a client–side class library that mirrors the functionality of the standard Java file classes in the `java.io` package. Table 1 presents an overview of the file classes in the `java.io` package and their JNFS equivalents. Each of these JNFS client classes inherits from either its `java.io` counterpart or from an ancestor of its `java.io` counterpart.

The JNFS client classes transmit both file information and blocks of data across RMI. We describe in greater detail the purpose and functionality of the JNFS client classes below.

FileServer Serves three main functions: it provides a factory interface to the remote file classes, allows remote administration of file system users to the system administrator, and provides the UserManager interface for remote user management (see Section 3.4 for a discussion of the user management classes).

Clients obtain a reference to the FileServer with bootstrap Naming service and authenticate themselves using the security protocol outlined in

package `java.io`	package `jnfs`	*purpose*
File	RemoteFile	pathname abstraction, file attributes
FileInputStream	RemoteFileInputStream	read–only input stream
FileOutputStream	RemoteFileOutputStream	write–only output stream
RandomAccessFile	RemoteRandomAccessFile	read/write, block–oriented random access
FileReader	RemoteFileReader	character file input
FileWriter	RemoteFileWriter	character file output
	FileServer	gives back filehandles, provides user info

Table 1: `java.io` file classes and their JNFS equivalents.

Section 4. Once authenticated, a client may request file handles from the server using the *getFile()*, *getFileInputStream()*, *getFileOutputStream()*, and *getRandomAccessFile()* methods. The FileServerHandle convenience class automates most of this process by performing the bootstrap lookup, and caching and refreshing authentication tokens.

Since the FileServer interface extends the UserManager interface, the file server also has a complete set of user management functions. These classes are discussed in Section 3.4.

RemoteFile A remote interface to a File object on the FileServer, this class is a descendant of the File class. Methods not overridden by RemoteFile are implemented in terms of remote calls if they rely on file system information or locally if they rely only on parsing (such as pathname manipulations).

Once a user has received a handle to an instance of the RemoteFile class, it may check attributes such as: whether the file *exists*, is *readable*, is *writable*, is a *directory or a file*, time last *modified*, *length*, *list* the contents of a directory, and *get* ACL *information.*

The RemoteFile class also allows the user to perform several destructive operations, such as: *delete* a file or directory, *make a directory*, *rename* a file or directory, and *set* ACL *information.*

RemoteFileInputStream An interface to a FileInputStream on the File-Server, this class allows reading data from a file remotely. It is a descendant of the InputStream class from which all input streams (including FileInput-

Stream) descend.

For efficiency, this class uses a client–side helper thread to pre–fetch up to four blocks of 4096 bytes apiece. A *read()* operation will block until some data is available.

RemoteFileOutputStream An interface to a FileOutputStream on the FileServer, this class allows writing data from a file remotely. Complementing the RemoteFileInputStream class, this class is a descendant of OutputStream.

For efficiency, this class uses a client–side helper thread to send blocks to the server in the background. If more than four blocks of data are queued for delivery, a *write()* operation will block until space becomes available in the queue. Upon closing, it blocks until it has flushed all unwritten blocks to the server.

RemoteRandomAccessFile An interface to a RandomAccessFile on the FileServer, this class does not descend from RandomAccessFile but rather implements the DataInput and DataOutput interfaces and additionally provides the *getFilePointer()*, *seek()*, and *length()* functions of RandomAccessFile.

A file may be opened for either read–only or read–write access. All conversion of Java base types (such as *readDouble()* or *writeLong()*) are done on the client side; data is both fetched and sent across the network in blocks. For efficiency, this class keeps a buffer cache of recently used blocks, utilizing a Least Recently Used replacement policy. The cache also associates a dirty bit with each block, so if a block need not be returned to the server if it is never written to.

RemoteFileReader A convenience class for reading character files remotely. Since the Java specification [7] requires a 16–bit Unicode `char` type, Java differentiates between reading the `byte`s and characters of a file. The RemoteFileReader interprets a RemoteFileInputStream using the default character encoding, performing `byte`–to–`char` conversions.

RemoteFileWriter A convenience class for writing character files remotely. Complementing the RemoteFileReader class, this class does `char`–to–`byte` conversions on a RemoteFileOutputStream.

3.4 User Management Classes

The JNFS client library also includes a set of classes for user management:

FileSystemUser Represents a user of the file system. It groups a user name with a PublicKey and provides an interface for maintaining collections of active Challenges and Tokens. Tokens and challenges used in the authentication protocol described in Section 4.1.2. The FileSystemUser is primarily used on the server side of JNFS, although it may be passed back and forth across the network.

FileSystemSigner Groups a user name with both a PublicKey and a PrivateKey. The class is named a "signer" because it has the ability to digitally sign Challenges in order to generate Responses. The FileSystemSigner class is used only on the client side of JNFS. Since it contains sensitive information (the user's PrivateKey), it should be stored in a secure place (preferably a smart card that can be inserted into the NC at login time).

UserTable Maintains a persistent collection of FileSystemUsers. The user table is read from and written to secondary storage by means of the *load()* and *save()* methods. Provides *get()*, *put()* and *remove()* operations for a particular user, and both *getUsers()* and *getUsernames()* to enumerate the collection of users.

The UserTable is used directly by the file server implementation and therefore performs no security checks. The FileServer class does not expose the UserTable class itself but instead presents the UserManager interface, discussed below.

UserManager A secure interface for managing users over the network. The UserManager allows any authentic FileSystemUser to view the complete table of users or look up an individual user by name (analogous to a world–readable `/etc/passwd` file on UNIX systems). The system administrator may additionally (once authenticated) invoke any of the methods of the UserTable class.

4 Security

Security in JNFS is discussed in two sections: authentication (confirming that a user is who they claim to be), and authorization (granting appropriate access to an authentic user).

4.1 Authentication

One important portion of security is authentication, for any access control policy is useless if it cannot be determined who is requesting permission to a file. Thus, some mechanism must be in place to authenticate users before file system access is granted. In general,

> Authentication is nothing more nor less than the determination by the authorized receiver(s), and perhaps the arbiter(s), that a particular message was most probably sent by the authorized transmitter under the existing authentication protocol and that it hasn't subsequently been altered or substituted for [13].

Below, we explore authentication as it is used in traditional network file systems and discuss our implementation of an authentication protocol for JNFS.

4.1.1 Traditional Network File System Authentication

Although NFS provides several options for authentication, a popular choice is the "UNIX style" authentication [9], in which "NFS servers accept client requests only if the client's network address appears in a list of trusted hosts." This scheme requires that the NFS file server trust the client's operating system.

The client's operating system authenticates a user at login time (usually by means of entering a password). Whenever that user requests access to a particular file, the local operating system determines if that user should be granted such access. If so, it requests the file from the NFS file server and then hands it off to the user.

If the security of the client's operating system is compromised, then the security of files served over NFS are compromised because NFS believes that the local operating system has properly authenticated the user.

4.1.2 A Challenge–Response Protocol with Digital Signatures

JNFS employs a more secure challenge–response mechanism for authentication. The protocol, outlined below, requires a user to use its digital signature to sign a random number to prove its authenticity. There are four stages to authentication:

1. *Initialization.* The user tells the server it wishes to be authenticated.

2. *Challenge.* The server generates a challenge and issues it to the user.

3. *Response.* The user signs the challenge and returns it to the server.

4. *Verification.* The server verifies that the signed version of the challenge matches the issued challenge and grants access to the client.

In the *initialization* stage, the user U tells JNFS that it would like access to the file system. The server confirms that U's name N_U appears in the user table and then proceeds to the *challenge* stage.

JNFS then generates a 64–bit random number R using the SecureRandom class in the `java.security` package. The server creates a *challenge* $\langle R, N_U \rangle$, records it, and issues it to U.

U receives the challenge and generates a *response* by signing $\langle R, N_U \rangle$ with its secret key $Priv_U$. Using the Digital Signature Algorithm [12], it creates a signature S_U for $\langle R, N_U \rangle$. The user returns $\langle R, N_U, S_U \rangle$ back to the server.

The JNFS server *verifies* the response by using U's public key Pub_U to interpret the signature S_U. If S_U is indeed a signed version of the challenge $\langle R, N_U \rangle$ that it had previously issued, the server accepts the user's credentials, since only U knows the private key $Priv_U$ used to produce the signature from the challenge. For efficiency, the server grants U a token (T_U) it will use on subsequent transactions until it expires. When T_U expires, U repeats the authentication process to obtain another token.

4.1.3 Security of the Protocol

There are several points where the security of the authentication protocol can be attacked. We address each below:

- *An eavesdropper E observes the challenge $\langle R, N_U \rangle$ as it is sent over the network from the server to U.* In this scenario, E learns two pieces of information: the name of one of the file system users (N_U) and the random number R. This information is only useful to E if it can generate a signed version of the challenge.

To do so requires that either E have a copy of $Priv_U$ or that the Digital Signature Algorithm is insecure. Since the Digital Signature Algorithm uses the *computationally secure* [13] cryptographic hash function SHA-1 for message digests and the RawDSA algorithm for signing data, the protocol is as secure as these algorithms.

- *E intercepts the response $\langle R, N_U, S_U \rangle$ and sends it to the server, masquerading as U.* A protection against this "man–in–the–middle" attack is to encrypt all messages in the authentication protocol so E cannot interpret any of the communication.
- *E observes T_U as it is sent from the server to U.* Again, applying encryption to the authentication protocol avoids this man–in–the–middle attack.
- *E guesses T_U.* This attack is extremely unlikely. Since T_U is 64 bits long, the chances that E correctly guesses T_U are 1 in 2^{64}. The randomness of T_U is dependent on the quality of the random numbers that SecureRandom provides.

4.1.4 Implementation of Authentication

The `jnfs.security` package implements the authentication protocol outlined above. It provides an Authentication class that manages the various stages of the protocol. The Challenge, Response, and Token classes encapsulate the data required by the Authentication class.

4.2 Authorization

Once the user's authenticity is verified, JNFS determines if that user should is authorized to access a file. The `jnfs.security` package provides a PermissionGranter abstraction that determines if a particular FileSystemUser should be granted access to a file. It grants access to files by comparing Access Control Lists to rules specified in an access control policy.

4.2.1 Access Control Lists

Every file and directory in JNFS has an Access Control List (ACL) associated with it. Each ACL is composed of one or more owners and their positive and negative permissions, as well as any number of groups which also have some form of access to the file. The PermissionGranter provides both *getAccessControl()* and *setAccessControl()* methods to obtain and modify ACLs. Only an owner of a file or the JNFS administrator may change a file's ACL.

The PermissionGranter recognizes a small set of Permissions that could be granted to a user or group for any file or directory. These include:

- *Attributes.* Specifies permission to check attributes such as whether a file exists, is a file or directory, length, modification time, etc.
- *Delete.* Specifies permission to delete a file.
- *Execute.* Specifies permission to execute a file.
- *List.* Specifies permission to list a contents of a directory.
- *Read.* Specifies permission to read from a file.
- *Rename.* Specifies permission to rename a file.
- *Write.* Specifies permission to write to a file.

In order to check if a user should be granted a particular Permission to a file, the PermissionGranter provides the *checkPermission()* method. Unlike the *getAccessControl()* method which only returns the ACL for an individual file, *checkPermission()* implements whatever security policy the file system chooses to use. For example, in UNIX, a user may read a file only if that user is granted both the *Read* permission for that file as well as the *Execute* permission for every parent directory of that file.

4.2.2 Rule–Based Access Control Policy

Subclasses of PermissionGranter use a set of rules to implement the *checkPermission()* method appropriately. The UnixStylePermissionGranter, for example, recursively checks all parent directories for *Execute* permission before checking the actual file for the requested permission. It also maps Permissions to their equivalent UNIX mode bits counterparts. It performs several

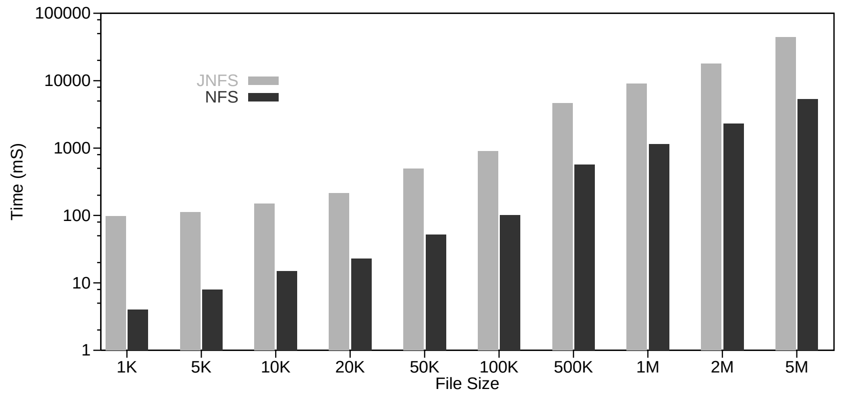

Figure 2: Read performance of the JNFS system vs. NFS on several different file sizes.

translations, such as mapping *List* to *Read*, and *Delete* to *Write* on the file's parent directory.

Since JNFS runs on top of a native file system, it adapts ACLs to the native file system's conception of file permissions. Since Java does not provide a platform–independent mechanism for generating ACLs from the information provided by operating system, the UnixNativePermissionGranter class provides this information by making `native` calls to *stat()*, *getpwuid()*, and *getgrgid()*. Similarly, in order to convert ACLs to UNIX permission bits, it makes `native` calls to *chmod()* and *chown()*. An NTNativePermissionGranter class would make calls to the Win32 equivalents to these functions.

5 Performance

Performance in our simulations was reasonable for a Java application, but not nearly as good as NFS. On average, JNFS file transfers take about 8 times as long as analogous NFS transfers on a Sun SparcStation 10 running Solaris 2.5. We present a comparison of JNFS and NFS read speeds on various different file sizes in Figure 2. There are several factors that lead to worse performance in our simulation:

- The JNFS server runs as a user-level daemon over a native file system. Since it does not run as part of the kernel, it incurs a large expense in traps to the kernel. Similarly, the client also runs in user space, which further worsens performance.

- Since the Java byte–codes are interpreted instead of executed natively, actual execution of the file system code is much slower. On an NC, NFS would be slower because it would be implemented in Java instead of a compiled language.

- The RMI networking protocol has a large overhead in comparison with NFS's UDP–based implementation. Although many of the JNFS functions transfer a block of data across the network, the RMI service still marshals and un–marshals this data, resulting in more copying than necessary.

6 Conclusions

The JNFS system demonstrates the viability of a Java–based network file system. Although JNFS cannot complete with a native NFS implementation for performance, it fills a niche left by the omission of a network file system in the Network Computer Reference Profile.

The strength of JNFS lies in a more secure challenge–response protocol for authentication and interoperability with native file systems. Both low– and upper–end NC users can benefit from these capabilities: low–end NCs gain support for a network file system and upper–end NCs gain support for network file systems other than NFS.

Acknowledgments

We are grateful to Professor Tom Doeppner of Brown University, for his advice and guidance throughout this project.

References

[1] Apple, IBM, Netscape, Oracle, and Sun. Network computer reference profile, 1996. `http://www.nc.ihost.com/nc_ref_profile.html`.

[2] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol — HTTP/1.0, 19 February 1996. `http://ds.internic.net/rfc/-rfc1945.txt`.

[3] Brent Callaghan. *WebNFS: The Filesystem for the World Wide Web*, 3 May 1996. `http://www.sun.com/solaris/networking/webnfs/-webnfs.html`.

[4] R. Fielding et al. Hypertext transfer protocol — HTTP/1.1, January 1997. `http://ds.internic.net/rfc/rfc2068.txt`.

[5] J. Franks et al. An extension to HTTP: Digest access authentication, January 1997. `http://ds.internic.net/rfc/rfc2069.txt`.

[6] John Gartner. Network computer market takes shape. *TechTools*, 30 April 1997. `http://www.techweb.com/tools/netpc/netpc.html`.

[7] James Gosling and Henry McGilton. *The Java Language Environment: A White Paper*. JavaSoft Inc., May 1996. `http://java.sun.com/docs/-language_environment/`.

[8] JavaSoft Inc. Remote method invocation specification, May 1997. `http://java.sun.com/products/jdk/1.1/docs/guide/rmi/spec/-rmiTOC.doc.html`.

[9] SunSoft Inc. The NFS distributed file service, March 1995. `http://-www.sun.com/solaris/desktop/nfs.html`.

[10] Peter Madany. *JavaOS: A Standalone Java Environment*, 1997. `http://java.sun.com/products/javaos/javaos.white.html`.

[11] J. Postel and J. Reynolds. File transfer protocol, October 1985. `http://ds.internic.net/rfc/rfc959.txt`.

[12] Benjamin Renaud. *Java Cryptography Architecture API Specification and Reference*. JavaSoft Inc., 1997. `http://java.sun.com/products/-JDK/1.1/docs/guide/security/CryptoSpec.html`.

[13] Gustavus J. Simmons. A survey of information authentication. In Gustavus J. Simmons, editor, *Contemporary Cryptology: The Science of Information Integrity*, pages 379–419. IEEE Press, 1991.