WebNFS Developer's Guide

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A.

Part No: 805–5501–10
October 1998

Contents

CHAPTER 1

Introduction to the WebNFS Software Development Kit

The WebNFS™ Software Development Kit (SDK) was developed to solve the problem of remote file access for Java™ applications. It does this by delivering a fully functional, platform-independent file access method, with the same user interface for both local and remote file access.

SDK Features

The SDK is composed of

- The Extended Filesystem API
- WebNFS classes

The Extended Filesystem API

The extended filesystem API classes provide a common interface for multiple filesystem types. Also, they allow for dynamic loading of filesystem implementations. The API also provides a means to access file and filesystem specific information. Further information about these classes can be found in Chapter 3, and the `javadoc` html files.

Using the Extended Filesystem API

The API includes classes similar to java.io.*, which have been extended to include file access. Sample code is included in the `zip` file.

WebNFS Classes

The WebNFS classes make it possible to establish connections to an NFS™ version 2 or version 3 server using TCP or UDP. The classes also provide consistent naming with NFS URLs. Further information about these classes can be found in Chapter 3, and the `javadoc` html files.

Dependencies

Java Development Kit (JDK™) 1.1 must be installed in order for this product to function.

Technical References

- * RFC1094 NFS™: Network File System Protocol Specification
  http://ds.internic.net/rfc/rfc1094.txt
- * RFC2224 NFS URL Scheme
  http://ds.internic.net/rfc/rfc2224.txt
- * RFC1813 NFS Version 3 Protocol Specification
  http://ds.internic.net/rfc/rfc1813.txt
- * RFC2054 WebNFS Client Specification
  http://ds.internic.net/rfc/rfc2054.txt
- * RFC2055 WebNFS Server Specification
  http://ds.internic.net/rfc/rfc2055.txt
- * WebNFS White Paper
  http://www.sun.com/webnfs/webnfs.html

CHAPTER 2

Extended Filesystem API

The Extended Filesystem API provides a common interface for multiple filesystem types. Also, it allows for dynamic loading of filesystem implementations. The API includes a set of classes similar to those in the `java.io` API, and therefore provides a familiar programmatic means of accessing files, either locally or remotely. The API also provides a means to access file-and filesystem-specific information.

Requirements

The model requires the following:

- The extensions must be filesystem-neutral.
- The model must support access to multiple filesystems.
- The model must be extensible to support custom features of new filesystems.
- The model must support filesystem identification and naming through URLs.
- The model must provide `java.io.File` type access to developers to facilitate migration of existing applications.

Overview

The extended filesystem provides extensibility through the use of:

- An XFileAccessor interface that provides a common set of access methods that are required for all filesystems.

- An XFileExtensionAccessor abstract class that provides direct access to filesystem-specific features.
- The capability for automatic runtime selection of the file access mechanism.
- Support for multiple file namespaces through URL naming.

The API defines two means of access to network files and filesystems:

- An XFile class that supports naming of files with URLs such as the type described in Internet RFC 2224. Here users would specify URLs like "nfs://oaktree.edu/archie". Developers of other filesystems would register their own URL schemes and implement filesystem accessor classes to support a chosen filesystem type, such as "smb://www.microsoft.com/index.html". The XFile class also provides a default "native" URL scheme using java.io.* which has a file-naming syntax that varies according to the operating system that supports the Java Runtime Interface, such as Win95, UNIX, and VMS.
- Access to data through the XFileInputStream, XFileOutputStream, XRandomAccessFile, XFileReader and XFileWriter classes that are similar to current `java.io.File` classes.

The API supports three categories of applications:

- Web-based applications, specifically browsers that deal with URLs extensively.
- Traditional file-based applications that must extend their access to the network and still use the style of access provided by `java.io.File` classes.
- Mixed mode applications that use both remote and distributed filesystems.

Architecture

The following picture shows the extended filesystem architecture. The top layer of the architecture mirrors the standard `java.io.File*` programmatic interfaces. Under this layer is a set of XFileAccessors. There is one XFileAccessor interface implementation for each filesystem being supported. Currently only NFS and native filesystems are supported. The accessors implement the XFileAccessor interface. They are responsible for all aspects of file access. The API classes are responsible for multiplexing between the different filesystems as requests are made from the application.

TABLE 2–1 Extended Filesystem Architecture

Java Application			
com.sun.xfile.*			
nfs:	cifs:	file:	native:

URL Naming

Every filesystem type is identified by a URL of the general form described in RFC 1738. The extended filesystem uses the scheme name in the URL to provide for the automatic selection of a filesystem. For example, the NFS filesystem has the scheme name of "nfs". When presented with a filename string, the extended filesystem checks for a URL scheme followed by a colon at the beginning of the string. If it finds one, it uses the filesystem accessor class associated with that URL scheme. If it does not find one, it defaults to the "native" scheme for access through `java.io.*`.

URL Names and Native Names

Filenames are assumed to be absolute URLs or relative URLs as determined by a context. If a filename string begins with a valid URL scheme name followed by a colon, then the name is associated with the filesystem type indicated by the URL scheme name. For instance, a filename string of the form "nfs://hostname/path" is associated with the filesystem that has the scheme name of "nfs". If the filename has no colon-separated prefix, or the scheme is not recognized then the name is assumed to belong to the default "native" scheme, that is, a URL prefix of "native:" is assumed and the name is handled by `java.io.*`. The two-argument constructor for XFile takes an XFile object and a filename:

```
XFile(XFile dir, String filename).
```

The filename argument is interpreted according to the context set by the `dir` argument. The `dir` argument is used as a base URL and the name is evaluated as a relative URL. If the filename is an absolute URL, then the `dir`is ignored. If the filename is a relative URL, then it is combined with the base URL to form the name of the new XFile object. The rules that describe the evaluation of relative URLs are described in RFC 1808.

Filesystem Selection

Filesystems are selected using these rules:

1. If the `XFile(String filename)`constructor is used and the filename has a prepended URL scheme, the filesystem associated with that scheme is used; otherwise, the native filesystem is used.
2. If the `XFile(XFile dir, String filename)` constructor is used, the filename is evaluated as a relative URL to the `dir` XFile.

Filesystem Registration

The filesystem registration mechanism is dynamic. The classes will search for a filesystem factory based on the filesystem name specified in the URL. This factory produces XFileAccessors. The name of the filesystem must be a unique URL scheme.

Examples of File Name Usage and Filesystem Selection

```
XFile f1 = new XFile("nfs://hostname/directory1");

// NFS file based on an absolute URL

XFile f2 = new XFile("file.txt");

// not a URL, so defaults to the current directory of the "native" filesystem

XFile f4 = new XFile(f1,"text.html");

// evaluated relative to the base URL of f1 - which is "nfs".
```

One point to be made from these examples is that the validity of a constructed XFile is made when the XFile is used, not when it is constructed. This is consistent with the `java.io.File`class.

Direct Access to Filesystem-Specific Features

A reference to the object implementing a filesystem's specific features can be determined through the class interface. The interface allows developers to directly

call these methods. By doing this, you are increasing the risk of writing code that works only for one filesystem type so its use is discouraged. To obtain access to these features directly, use the public method `XFile.getExtensionAccessor()`. Once you have obtained this handle, you can access its methods directly. Refer to the filesystem implementor's documentation for class description details.

The following example shows how a method in the NFS ExtensionAccessor is used to set the user's authentication credentials using the NFS PCNFSD protocol:

```
import java.io.*;
import com.sun.xfile.*;
import com.sun.nfs.*;
public class nfslogin {
   public static void main(String av[])
   {
      try {
         XFile xf = new XFile(av[0]);
         nfsXFileExtensionAccessor nfsx =
            (nfsXFileExtensionAccessor) xf.getExtensionAccessor();

         if (! nfsx.loginPCNFSD("pcnfsdsrv", "bob", "-passwd-")) {
            System.out.println("login failed");
            return;
         }
         if (xf.canRead())
            System.out.println("Read permission OK");
         else
            System.out.println("No Read permission");
      } catch (Exception e) {
         System.out.println(e.toString());
         e.printStackTrace(System.out);
      }
   }
}
```

Class Descriptions

This section describes the additional methods that are not currently part of the `java.io` classes. For example, the first entry, called Xfile, describes the methods

that are in the `com.sun.xfile.XFile` class but not in the `java.io.File` class. All `com.sun.xfile` classes that have a counterpart in the `java.io` classes are generally a superset of the `java.io` classes in the methods provided, not including constructors. File descriptors are not supported and constructors that use a `java.io.File*` class now use a `com.sun.xfile.XFile*` class. Also shown are the few new interfaces and classes defined by the extended filesystem. A complete description of the API can be found in the `javadoc` files included in the release.

- XFile - Counterpart Is java.io.File

  public XFile(String url) - class constructor

  public XFile(XFile dir, String relurl) - class constructor

- XFileOutputStream - Counterpart Is FileOutputStream
- XFileOutputStream - Counterpart Is FileOutputStream

  public XFileOutputStream(XFile file) - class constructor

- XFileInputStream - Counterpart Is FileInputStream

  public XFileInputStream(XFile file) - class constructor

- XRandomAccessFile - Counterpart Is RandomAccessFile

  public XRandomAccessFile(XFile file, String mode) -class constructor

- XFilenameFilter - Counterpart Is FilenameFilter

  public abstract boolean accept(XFile dir, String name) - filter based on file spec

- XFileReader - Counterpart Is FileReader

  XFileReader(XFile file)

- XFileWriter- Counterpart Is FileWriter

  XFileWriter(XFile file, String access)

File Interface Examples

Extending existing programs to network filesystems should be straightforward using the extensions provided. You simply replace declarations of type java.io.FileXYZ with the counterpart java.io.XFileXYZ. Here are a few simple examples of applications that use the APIs.

CODE EXAMPLE 2–1 Using Streams Interface

```
import java.io.*;
import java.net.*;
import com.sun.xfile.*;

XFile xf = new XFile("file.txt");
if (xf.isFile()) {
    System.out.println("file is file");
}
if (xf.isDirectory()) {
    System.out.println("file is directory");
}
XFileInputStream nfis = null;
nfis = new XFileInputStream(xf);
for (int count = 0; ; count++) {
    int val = (byte) nfis.read();
    if (val == -1)
        break;
    System.out.write(val);
}
System.out.println("read " + count + " bytes ");
```

CODE EXAMPLE 2–2 Using XRandomAccessFile Interface

```
import java.io.*;
import java.net.*;
import com.sun.xfile.*;
// create connection to host
XFile xf = new XFile("nfs://ian/simple.html");
XRandomAccessFile xraf = new XRandomAccessFile(xf,"r");
int count = 10;
xraf.seek(count);
```

```
System.out.println("Value at position " + count
    + " is " + (byte)xraf.read());
System.out.println("Current file position is "
    + xraf.getFilePointer());
xraf.seek(0);
for (count = 0 ; count < 100 ; count++) {
    int val = xraf.read();
    if (val == -1)
        break;
    System.out.print(val);
}
```

CHAPTER 3

NFS Classes for the Extended Filesystem

This is the first implementation of remote file system access for Java applications that provides 100% Pure Java™ compatibility. Although there are already Java applications that access remote files through the URLConnection class of java.net, the access is generally read-only and limited to whole file transfer as supported by the underlying protocols: HTTP and FTP. There is no provision for the same kind of file access that applications currently enjoy through java.io classes.

Through the extended file system API and the NFS client, applications can perform the full range of file operations on a remote NFS server.

NFS URL

The NFS URL is a global, universal name for naming files or directories on any NFS server. The general form is similar to that of other URL schemes: nfs://server/path" where the scheme name is "nfs" and the server name is the name of any NFS server. The path is a slash-separated path that names an NFS-exported file on the server. The words "global" and "universal" are not simply buzzwords: "global" means that the URL can refer to a file or directory on any NFS server in the world whether the server is an IBM mainframe, a UNIX server, or a Windows NT server running NFS. The word "universal" means that the URL always has the same syntax whether the client is a Macintosh, Windows 95 laptop, or VAX VMS Minicomputer. These properties are important for Java applications; Java applications must have global access via the Internet and to be "100% Pure Java" they must run unchanged across all Java platforms.

The NFS URL Scheme is described in RFC 2224.

Connecting to the Server

When an application first tries to access a file named by an NFS URL through the extended file system, the NFS client will negotiate a connection with the server. First the client will attempt to make a TCP connection since TCP is the preferred transport protocol on the Internet for conveying large volumes of data. In addition, corporate firewalls can be configured to allow NFS TCP connections to Internet servers on port 2049. If the server rejects the TCP connection, the client will then use the UDP protocol. UDP performs as well as TCP on local area networks but is not as well suited to Wide Area Networks and the Internet.

The NFS client incorporates the latest WebNFS technology described in RFC 2054. It connects directly with the NFS server and attempts to locate the requested file or directory using a "public filehandle" and multi-component lookup". This is a very efficient way to connect to the server since it avoids additional steps needed by conventional NFS clients to "mount" the file system using the NFS MOUNT protocol.

If the NFS server does not support WebNFS connection, the client will attempt to connect using the MOUNT protocol. Since the MOUNT protocol does not use a well-known network port, it cannot easily transit a firewall to connect to Internet servers, but it can be used to connect to local Intranet NFS servers. When accessing files on a server that does not support WebNFS connections, the URL may need an extra slash in the pathname, for instance if UNIX NFS clients normally use the name "server:/path" in their mount command then the equivalent URL for the NFS client will be nfs://server//path". Whether or not the extra slash needs to be used is not an issue for the user to figure out. It's the responsibility of whoever distributes the URL to determine the correct path depending on whether the server is a WebNFS server or a MOUNT-only server.

The server administrator has little to do since the NFS client has much in common with other NFS clients. If the server supports the WebNFS service then the public filehandle may be associated with a particular directory by including the "public" option in the share command. The pathname in the NFS URL is relative to the directory with the public filehandle, for instance, if the server exports the directory "/export" with the "public" option then the file "/export/this/file" is named by the NFS URL "nfs://server/this/file".

On Solaris™ 7 servers, the public filehandle has a default location at the system root, from this location an NFS URL can name a file or directory on any of its exported file systems.

NFS Versions

The NFS client implements both versions 2 and 3 of the NFS protocol. Version 2 is by far the most widely supported version of NFS. Version 3 features many improvements particularly in performance. To an end-user or application developer the version of NFS that is being used is unimportant, the NFS client negotiates the NFS version automatically with the server. Because version 3 has many performance improvements the client will exercise a preference for that version of the protocol. It will use version 2 only if the server does not support version 3.

TCP or UDP ?

The NFS protocol is transport independent. It will run over a stream-oriented protocol like TCP or a datagram protocol like UDP. Historically, there are more UDP implementations of the NFS protocol because early implementations of TCP were notoriously slow. In recent years the performance of UDP and TCP implementations on local area networks is comparable and on wide area networks and the Internet there is a distinct preference for the congestion control and reliability of TCP. Additionally, it is much easier to control TCP connections at a firewall than UDP.

The NFS client will first attempt to use TCP to connect to any NFS server whether it be a local server or across the world on the Internet. Only if the server rejects the TCP connection attempt will the NFS client use UDP.

Reliability

More than any other distributed file system protocol, the NFS protocol is known for its reliability and data safety. The NFS version 2 protocol was notorious for slow write speed. The NFS guarantee of data safety required the server to store the data from each write request to disk before replying to the client. Although the "synchronous write" requirement imposed a performance trade-off, the client was assured that a server crash would never lose its data. An NFS server crash or network outage will never result in corrupted or half-written files. Version 3 preserves the data safety guarantee while allowing the server to improve write performance through asynchronous writes. The NFS client uses the improved write technique of version 3 to deliver excellent write performance with data safety.

If an NFS server crashes or the network connection is lost, the NFS client will persist in attempting to restore the connection and continue where it left off. If a TCP connection is broken for any reason, the client will re-establish the connection. If a UDP request or reply is lost, the client will re-transmit the request until the operation succeeds using an exponential backoff on the timeout to avoid overloading the server with retransmissions. Since NFS servers are designed to be stateless, the server need do nothing on recovery other than serve new NFS requests that may include retransmitted requests that were not completed before the crash.

This reliability has tangible benefits for users who are used to the low bandwidth access to busy Internet servers. With protocols like HTTP or FTP a lost connection usually means that a file transfer must begin over and applications that use these protocols will receive an error. These problems are transparent to applications that use the NFS classes through the extended file system. Any pending read or write will block until it completes successfully. In the case of a file transfer over NFS it means that the file transfer will resume automatically from where it left off. This blocking behavior and persistence need not be inconvenient to an interactive user with limited patience; a Java application can implement a "stop" or "abort" button that kills a blocked thread.

In short, the NFS protocol and the NFS client implementation of it are very reliable.

Security

NFS servers currently control access to their files using "trusted host" security. The server trusts that the client machines will construct a valid credential that correctly identifies the individual requesting file access. Generally the server identifies the list of trusted client machines with an access list that explicitly lists the names of the client machines, or identifies a "netgroup" that lists the names of trusted clients or other netgroups. If the client machine is "trusted" by the server then the NFS client will be able to access the server like any other NFS client.

Since the Java Runtime has no concept of the user’s identity the NFS client cannot automatically construct a credential that identifies the user with a UID or GID value since these values may be meaningless on platforms that do not support them (Macintosh and Windows machines do not require the user to log in). Instead, the client constructs a "nobody" credential that uses a UID and GID value of 60001 which the server identifies as user "nobody". Since file access permissions on public or Internet archive servers are often liberal in allowing read access to anyone, the "nobody" identity is of no consequence. However, if the user wishes to access private files in his or her own home directory or to create files with his or her identity assigned to the owner of the file then a credential with the user’s UID and GID must be used.

The NFS client uses the same technique to acquire a valid user credential as PC-NFS clients - it uses the network PCNFSD service. The PCNFSD service is commonly installed wherever PC-NFS clients are present. It need not be installed on every NFS server, only one server in the network is sufficient. The NFS client provides a loginPCNFSD() method in that can be called through the NFS XFileExtensionAccessor class. This method takes the user's login name, password and the name of a host running the PCNFSD service and passes this information to the PCNFSD server (the password is not encrypted but it is obscured by exclusive ORing with a bit pattern so that the password is not disclosed to casual network sniffers). If the login name and password are valid the PCNFSD server returns the user's UID and GID which the NFS classes then use in subsequent requests to the NFS server. Here is an example of the a Java application setting the user's credential:

```
import sun.xfile.*;
import sun.nfs.*;

public class pcnfsd {

    public static void main(String av[])
    {
        try {
            XFile xf = new XFile(av[0]);

           nfsXFileExtensionAccessor nfsx =
                (nfsXFileExtensionAccessor) xf.getExtensionAccessor();

            if (! nfsx.loginPCNFSD("pcnfsdsrv", "jane", "-passwd-")) {
                System.out.println("login failed");
                return;
            }

            if (xf.canRead())
                System.out.println("Read permission OK");
            else
                System.out.println("No Read permission");

        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Of course a more realistic application would likely obtain the parameters for the call to loginPCNFSD() from a dialog box with a protected text field for the user's password.

The "trusted host" security has several shortcomings, not the least of which is that it is not particularly secure. A determined network sniffer could monitor PCNFSD calls and obtain passwords, a malicious client could masquerade as a "trusted" host by spoofing the source IP address, or the trusted host itself could be compromised in which case an intruder could spoof the credential of anyone. In addition, on an Internet scale it would be impractical to maintain access lists for all client machines. Even then, it is common for users to access the server from unpredictable IP

addresses either because the client is a "kiosk" machine used by many different users, or because the source IP address is dynamically allocated by a DHCP server.

For this reason there is work underway in the IETF to define more secure NFS authentication. The secure RPC working group has devised a new framework for secure RPC called "RPCSEC_GSS" (RFC 2203)which is based on the IETF's GSS-API, an open-ended framework that facilitates "plug-in" security schemes that can provide secure authentication, data integrity and data privacy for NFS traffic on the network. SunSoft™ is currently working on a Solaris plug-in for Kerberos v5 which will also be available in a future delivery. The RPCSEC_GSS mechanism will also support public key based security schemes that are more suitable for Internet use.

Network Performance

A common criticism of Java applications is that they are "too slow". The often-quoted figure is "20 to 40 times" slower than equivalent compiled C or C++ applications. However, it is strongly associated with networks and in a network context Java applications suffer zero speed penalty. In just the last 10 years CPU speed has increased tenfold, yet the speed of network access through modems has barely doubled and many of us are still using the same 10 Mb/sec Ethernets we were using 10 years ago (though now switched). The consequence of this is that machines are much faster than the networks they communicate with. Java applications that make use of the network spend most of their runtime waiting for the network.

The NFS client uses several techniques to use the network efficiently. First it caches NFS file attributes and data in memory to avoid unnecessary calls to the server. When the client connects to an NFS version 3 server it eliminates the 8k read and write limitation of version 2 and reads or writes data in large 32k blocks. Version 3 also allows the client to use safe, asynchronous writes that are several times faster than the synchronous writes required by version 2. Finally, the NFS client utilizes Java threads to implement read-ahead and write-behind. If the client detects that the application is reading a file sequentially then it will asynchronously issue read requests ahead of the current block of data in anticipation of the application's need. The use of a single block read-ahead can double the client's file reading throughput. Similarly, write-behind allows the client application to write blocks of data to the server without waiting for confirmation for each block of data. This has a similar effect on file write throughput.

Our experience with the NFS client is that it can read and write data at over 1 MB per second across a 10Mb Ethernet from a Sun Ultra 1 client.

Package Size

Implementations of the NFS protocol are assumed to be large. This misconception is perhaps due to NFS clients being integrated with the operating system along with their attendant administrative mount commands, automounters and the like. The NFS classes contain 70k of Java bytecode which reduces to 38k when delivered in a zip file. The classes that comprise the API contain 55k of bytecode (29k zipped).

Using NFS Classes through the Extended Filesystem

Although the intent of the API is to hide details of file system implementation behind a common API, there are some features of the underlying file system that should be considered.

Caching

The NFS client uses the same cache model as other NFS client implementations. File data and attributes are cached for a time interval depending on the frequency of update. If the file is updated frequently then the cache time will be as short as 3 seconds. However if the file is updated infrequently then the cache time may be as long as 60 seconds. If the file changes on the server during the time in which the cached copy is considered valid, then the change will not be noticed until the cache time has expired. This caching behavior should be considered when Java applications running in different Java virtual machines or on multiple network clients need to coordinate their activity around a common set of files.

The NFS client caches aggressively for good network performance. Currently the cache is open-ended: as new NFS files or directories are encountered, their filehandles and file attributes are cached. These cache entries are never released, so if a Java application touches a large number of files perhaps in a file tree walk, then the application may encounter an OutOfMemoryError. A future version of the client will manage the cache within a bounded amount of memory.

Buffering

When using the java.io classes either directly or as a "native" file system under the extended file system, there is no attempt to buffer data explicitly unless the BufferedInputStream or BufferedOutputStream classes are used. Data is buffered implicitly by the underlying file system.

The NFS classes access the network directly through the Java Socket interface. To provide acceptable performance the NFS classes buffer all reads and writes. The buffer size varies from 8k for NFS version 2 servers to 32 for NFS version 3 servers though the actual buffer size is invisible to the Java application itself. The Java application must be diligent in calling the close() method when writing to an OutputStream or XRandomAccessFile to ensure that a partially filled buffer is written to the NFS server before the application exits. There is nothing in the Java Runtime that will cause partially written buffers to be flushed automatically at application exit.

Symbolic Links

Although the NFS protocol supports the use of symbolic links, neither the java.io classes nor the API acknowledge their existence. The NFS classes make symbolic links transparent to the application by following links automatically in the same way that PC-NFS clients do. If a Java application attempts to access an NFS URL that references a symbolic link, the NFS classes will follow the link (and any further links) and return the attributes of the file or directory that is referenced by the link. A Java application cannot create a symbolic link through the API though a future release of the NFS XFileExtensionAccessor will support this feature.

File Attributes

The java.io and the extended file system APIs support a limited set of file attributes that are considered common across many file system types. For instance all file systems are assumed to support some notion of file size, modification time, access control for read or write and filetypes with common behaviors like directories and "flat" files. The NFS protocol supports some file attributes that are not in the "common" set like the file owner and group (UID and GID), creation time, last access time, and UNIX™-style permission bits. While these attributes are not accessible through the API they will be available through the XFileExtensionAccessor in a subsequent release of the NFS classes.

File Accessibility

The XFile methods canRead() and canWrite() are implemented within the NFS classes with code that checks the permission bits of the file attributes against the user's UID. While this technique is used by all NFS version 2 clients, it may sometimes return misleading result if file access is controlled on the server with an Access Control List (ACL) since the effects of the ACL cannot always be represented by an equivalent set of permission bits. Version 3 supports an ACCESS request that allows the client to request the server do the access check subject to the ACL. The result from ACCESS is always an accurate indication of the file access permitted the user. The NFS client does not currently implement the ACCESS check but will support it in the next release.

File Truncation

The NFS protocol allows a client to control the size of a file by setting the file size attribute. For instance a file can be truncated by setting the file size to zero. The API allows Java applications to obtain the file size through XFile.length() but there is no method that allows the length to be set. This may be a future addition.

NFS Exception Information

Many of the I/O methods in the API will throw an IOException for any condition that causes the operation to fail. The underlying NFS classes throw a subclass of IOException called NFSException that conveys more detailed information about the failure in an error code within the Exception object. For instance, the NFS exception will indicate whether a write failed because the file system is exported read-only, or because there is no space left on the disk. While an application that knows it is accessing an NFS file can cast the IOException to an NFSException and obtain the NFS-specific error code, there is yet no file system independent mechanism that will allow an application to obtain a common set of error codes for all file system types. The NFS error codes will be documented in the next release of the NFS classes.

References

- T. Berners-Lee, L. Masinter, M. McCahill, "Uniform Resource Locators (URL)," RFC-1738, December 1994.
  `http://ds.internic.net/rfc/rfc1738.txt`
- B. Callaghan, "NFS URL Specification," RFC-2444, October 1996.
  `http://ds.internic.net/rfc/rfc2224.txt`

- B. Callaghan, "WebNFS Client Specification," RFC-2054, October 1996.
  `http://ds.internic.net/rfc/rfc2054.txt`
- B. Callaghan, "WebNFS Client Specification," RFC-2054, October 1996.
  `http://ds.internic.net/rfc/rfc2054.txt`
- B. Callaghan, "WebNFS Server Specification," RFC-2055, October 1996.
  `http://ds.internic.net/rfc/rfc2055.txt`
- A. Chiu, "Authentication Mechanisms for ONC RPC," Internet-Draft: draft-ietf-oncrpc-auth-04.txt, October 1997.
  `ftp://ftp.ietf.org/internet-drafts/draft-ietf-oncrpc-auth-04.txt`
- M. Eisler, A. Chiu, L. Ling, "RPCSEC_GSS Protocol Specification," RFC-2203, September 1997.
  `http://ds.internic.net/rfc/rfc2203.txt`
- R. Fielding, "Relative Uniform Resource Locators," RFC-1808, June 1995
  `http://ds.internic.net/rfc/rfc1808.txt`
- Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, B. Lyon, "Design and Implementation of the Sun Network Filesystem," USENIX Conference Proceedings, USENIX Association, Berkeley, CA, Summer 1985. The basic paper describing the SunOS implementation of the NFS version 2 protocol, and discusses the goals, protocol specification and trade-offs. R. Srinivasan, "Binding Protocols for ONC RPC Version 2," RFC-1833, August 1995.
  `http://ds.internic.net/rfc/rfc1833.txt`
- R. Srinivasan, "RPC: Remote Procedure Call Protocol Specification Version 2," RFC-1831, August 1995.
  `http://ds.internic.net/rfc/rfc1831.txt`
- R. Srinivasan, "XDR: External Data Representation Standard," RFC-1832, August 1995.
  `http://ds.internic.net/rfc/rfc1832.txt`
- Sun Microsystems, Inc.®, "Network Filesystem Specification," RFC-1094, DDN Network Information Center, SRI International, Menlo Park, CA. NFS version 2 protocol specification.
  `http://ds.internic.net/rfc/rfc1094.txt`
- Sun Microsystems, Inc., "NFS Version 3 Protocol Specification," RFC-1813, DDN Network Information Center, SRI International, Menlo Park, CA.
  `http://ds.internic.net/rfc/rfc1813.txt`
- X/Open Company, Ltd.®, X/Open CAE Specification: Protocols for X/Open Internetworking: (PC)NFS, Developer's Specification, X/Open Company, Ltd., Apex Plaza, Forbury Road, Reading Berkshire, RG1 1AX, United Kingdom, 1991. This is a reference for PC-NFS and accompanying protocols.
- X/Open Company, Ltd., X/Open CAE Specification: Protocols for X/Open Internetworking: XNFS, X/Open Company, Ltd., Apex Plaza, Forbury Road, Reading Berkshire, RG1 1AX, United Kingdom, 1991. This is an indispensable reference for the NFS and accompanying protocols, including the Lock Manager and the Portmapper.