Apple Filing Protocol Programming Guide

Networking > Mac OS X Server

2006-04-04

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Contents

Figures and Tables

Chapter 1 **Concepts 9**

Introduction

The Apple Filing Protocol (AFP) allows users of multiple computers to share files easily and efficiently over a network.

The *Apple Filing Protocol Programming Guide* describes the request blocks that an AFP client sends to an AFP server and the reply blocks that an AFP server sends to an AFP client in response to a request block.

Note that all values exchanged between an AFP client and an AFP server are sent over the network in network byte order.

Organization of This Document

This book contains the following chapters:

- "Concepts" (page 9) describes the concepts used in the AFP architecture.
- "Using Login Commands" (page 67) describes the commands used to open and close a connection with a file server.
- "Using Volume Commands" (page 69) describes the commands for interacting with a file server volume.
- "Using Directory Commands" (page 71) describes the commands for using directories.
- "Using File Commands" (page 73) describes the commands for working on files.
- "Using Combined Directory and File Commands" (page 75) describes commands that can be used on both files and directories.
- "Using Fork Commands" (page 77) describes the commands to interact with data forks.
- "Using Desktop Database Commands" (page 79) describes the commands to read and write information store in the server's desktop database.

See Also

Refer to the following reference document for AFP:

- *Apple Filing Protocol Reference*

The following sources provide additional information that may be of interested to AFP developers:

- *Inside AppleTalk*. Addison Wesley. ISBN 0-201-19257-8.
- *Applied Cryptography, Second Edition*, by Bruce Schneier. Specifically, the chapter on Diffie-Hellman Key Exchange.

Introduction

See Also

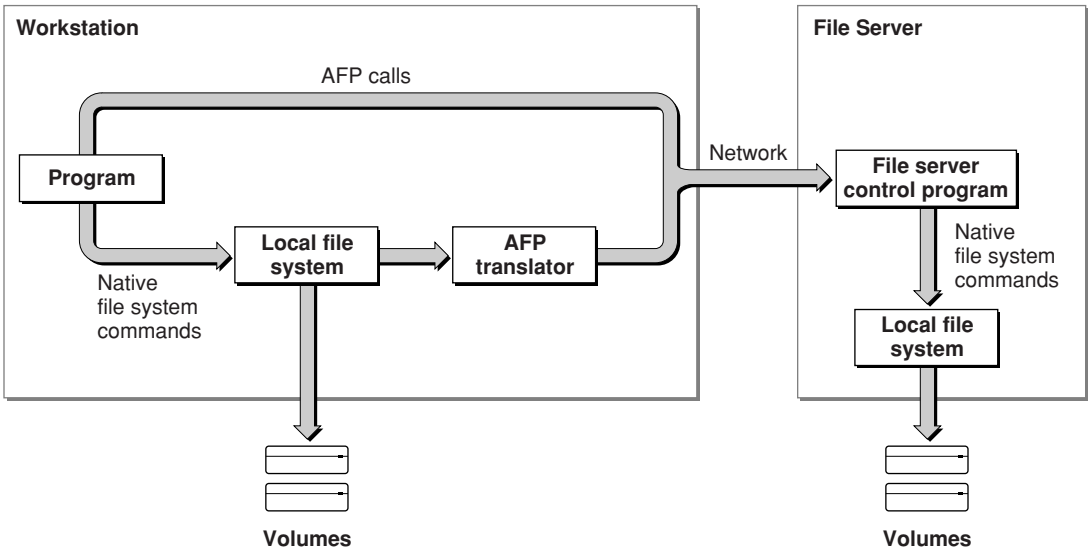
Concepts

File Access Model

This section introduces the file access model used by AFP to enable file sharing and discusses the components of AFP software.

Note: All values exchanged between an AFP client and an AFP server are sent over the network in network byte order.

Figure 1-1 AFP file access model



A program running in a local computer requests and manipulates files by using that computer's native file system commands. These commands manipulate files on disk or other memory resource that is physically connected to the local computer. Through AFP, a program can use the same native file system commands to manipulate files on a shared resource that resides on a remote computer (for example, a file server).

A program running on the local computer sends a command to the native file system. A data structure in local memory indicates whether the volume is managed by the native file system or by some external file system. The native file system discovers whether the requested file resides locally or remotely by looking at this data structure. If the data structure indicates an external file system, the native file system routes the command to the AFP translator.

The translator, as its name implies, translates the native commands into AFP commands and sends them through to the file server that manages the remote resource. The AFP translator is not defined in the AFP specification; it is up to the applications programmer to design it.

A program running on the local computer may also need to send AFP commands for which no equivalent command exists in the native file system. In this case, the AFP command is sent directly to the desired external file system, as shown in Figure 1-1. For example, user authentication might have to be handled through an interface written for that purpose.

AFP supports computers using Mac OS and personal computers using MS-DOS. AFP can be extended to support additional types of computers. Any implementation of AFP must take into account the capabilities of the networked computer's native file system and simulate its functionality in the shared environment. In other words, the shared file system should duplicate the characteristics of a local computer's file system. Simulating the functionality of each local computer's native file system becomes increasingly complex as different computer types share the same file server. Because each computer type has different characteristics in the way it manipulates files, the shared file system needs to possess the combined capabilities of all computers on the same network.

Three system components make up AFP:

- a file system structure, which is made up of resources (such as file servers, volumes, directories, files, and forks) that are addressable through the network. These resources are called AFP-file-system-visible entities. AFP specifies the relationship between these entities. For example, one directory can be the parent of another. For descriptions of AFP-file-system-visible entities, see "File System Structure" later in this chapter.
- AFP commands, which are the commands the local computer uses to manipulate the AFP file system structure. As mentioned earlier, the translator sends file system commands to the file server in the form of AFP commands, or the application running on the local computer can make AFP commands directly. Each AFP command is described in detail in the "Tasks" section of this document.
- algorithms associated with the commands, which specify the actions performed by the AFP commands.

Although this chapter distinguishes between local computers and file servers, AFP can support these two functions within the same node. However, AFP does not solve the concurrency problems that can arise when a computer is both an AFP client and an AFP server. The software on such combined nodes must be carefully designed to avoid potential conflicts.

AFP does not provide commands that support administration of the file server. Administrative functions, such as registering users and changing passwords, must be handled by separate network-administration software. Additional software must also be provided to add, remove, and find servers within the network.

File System Structure

This section describes the AFP file system structure and the parameters associated with its AFP-file-system-visible entities. These entities include the file server, its volumes, directories ("folders" in Mac OS terminology), files. and file forks. This section also describes the tree structure, called the volume catalog, which is a description of the relationships between directories and files.

By sending AFP commands, the AFP client can

- obtain information about the file server and other parts of the file system structure
- modify information that is obtained from a file server
- create and delete files and directories
- retrieve and store information within individual files

The following sections describe the file system structure's AFP-file-system-visible entities.

File Server

A file server is a computer with at least one large-capacity disk that allows other computers on the network to share information stored in it. AFP imposes no limit on the number of shared disks. Each disk attached to a file server usually contains one volume, although the disk may be subdivided into multiple volumes. Each volume appears as a separate entity to the AFP client.

A file server has a unique name and other identifying parameters. These parameters identify the server's machine type and number of attached volumes, as well as the AFP versions user authentication methods (UAMs) that the server supports. AFP file server parameters are listed Table 1-1.

Table 1-1 File server parameters

Parameter	Description
Server name	A string in Pascal format of up to 32 characters.
Server machine type	A string in Pascal format of up to 16 characters that describes the file server's hardware and software but has no significance to AFP.
Number of volumes	A two-byte integer.
AFP version strings	One or more strings of up to 16 characters each. For more information, see Table 1-12 (page 30).
UAM strings	One or more strings of up to 16 characters each. For more information, see Table 1-13 (page 30).
Server icon	A optional value of 256 bytes that is used to customize the appearance of server volumes on the Mac OS Desktop. It consists of a 32-by-32 bit (128 bytes) icon bitmap followed by a 32-by-32 bit (128 bytes) icon mask. The mask usually consists of the icon's outline filled with black (bits that are set). For more information about icons, refer to *Inside Mac OS X*.
Server signature	A 16-byte value that uniquely identifies a server used to prevent an AFP client from logging on to the same server twice.

Volumes

A file server can make one or more volumes visible to AFP clients. Each volume has parameters associated with it, as listed in Table 1-2 (page 12). To provide security at the volume level, the server can maintain an optional password parameter for any volume.

Table 1-2 Bit definitions for the Volume parameter

Constant and bit position	Parameter size	Description
`kFPVolAttributeBit` (bit 0)	Short	Volume attributes. See Table 1-3 (page 13) for details.
`kFPVolSignatureBit` (bit 1)	Two bytes	The volume signature identifies the volume type (flat, fixed Directory ID, or variable Directory ID). For details, see the section "Volume Types" (page 14).
`kFPVolCreateDateBit` (bit 2)	Four bytes	The date the server created the volume. This parameter cannot be modified by an AFP client.
`kFPVolModDateBit` (bit 3)	Four bytes	Updated by the server each time anything on the volume is modified. This parameter cannot be modified by an AFP client.
`kFPVolBackupDateBit` (bit 4)	Four bytes	Set by a backup program each time the volume's contents are backed up. When a volume is created, the Backup Date is set to 0x80000000 (the earliest representable date-time value).
`kFPVolIDBit` (bit 5)	Two bytes	For each session between the server and an AFP client, the server assigns a Volume ID to each of its volumes. This value is unique among the volumes of a given server for that session.
`kFPVolBytesFreeBit` (bit 6)	Four unsigned bytes	Total bytes free on volumes less than 4 GB in size. If a volume is more than 4 GB, the Bytes Free parameters may not reflect the actual value. In any case, Extended Bytes Total always reflects the correct value. This value is maintained by the server and cannot be modified by an AFP client.
`kFPVolBytesTotalBit` (bit 7)	Four unsigned bytes	Total bytes on volumes less than 4 GB in size. If a volume is more than 4 GB, the Bytes Total parameter may not reflect the actual value. In any case, Extended Bytes Total always reflects the correct value. This value is maintained by the server and cannot be modified by an AFP client.
`kFPVolNameBit` (bit 8)	A string of up to 2 7 characters	The volume name identifies a server volume to an AFP client user, so it must be unique among all volumes managed by the server. All eight-bit ASCII characters, except null (0x00) and colon (0x3A), are permitted in a volume name. This name is not used directly to specify files and directories on the volume. Instead, the AFP client sends an AFP command to obtain a particular volume identifier, which it then uses when sending subsequent AFP commands. For more information, see "Designating a Path to a CNode" (page 27).
`kFPVolExtBytes-FreeBit` (bit 9)	Eight unsigned bytes	Total bytes free on this volume. This value is maintained by the server and cannot be modified by an AFP client

Constant and bit position	Parameter size	Description
kFPVolExtBytesTotalBit (bit 10)	Eight unsigned bytes	Total bytes on this volume. This value is maintained by the server and cannot be modified by an AFP client.
kFPVolBlockSizeBit (bit 11)	Four bytes	The block allocation size.

The Attributes parameter for volumes provides additional information about the volume. Table 1-3 lists the bit definitions for the Attributes parameter for volumes.

Table 1-3 Bit definitions for the Volume Attributes parameter

Constant and bit position	Description
kReadOnly (bit 0)	If set, the volume is available for reading only.
kHasVolumePassword (bit 1)	If set, the volume has a volume password. Volume passwords were supported in prior versions of AFP; now the volume attributes reflect this information. This bit is the same as the HasPassword bit returned for each volume by FPGetSrvrParms.
kSupportsFileIDs (bit 2)	If set, the volume supports file IDs. In general, if file IDs are supported on one volume, they are supported on all volumes, but this bit allows the server to be more selective, if necessary.
kSupportsCatSearch (bit 3)	If set, the volume supports the FPCatSearch and FPCatSearchExt commands. Support for FPCatSearch and FPCatSearchExt is optional. This bit allows the server to make this capability available on a per-volume basis.
kSupportsBlankAccessPrivs (bit 4)	If set, the volume has a Supports Blank Access Privileges bit that, when set for a directory, causes the directory to inherit its access privileges from its parent directory.
kSupportsUnixPrivs (bit 5)	If set, the volume supports UNIX privileges.
kSupportsUTF8Names (bit 6)	If set, the volume supports UTF-8–encoded user names, group names, and pathnames.

Constant and bit position	Description
kNoNetworkUserIDs (bit 7)	If set, always map UNIX user IDs, group IDs and permissions to traditional User IDs, Group IDs and permissions. If not set, after logging into the server, an AFP client running on a UNIX-based machine should call getuid() to get the user's local user ID and send an FPGetUserInfo command to get the user's user ID from the server. If the user IDs match, the AFP client should call getpwuid() to get the user's local user name, which is returned in the pw_name field, and send an FPMapID command to get the user's user name from the server. If the user names match, the AFP client assumes both machines are operating from a common user directory, and displays UNIX permissions without mapping them. Showing UNIX user IDs, group IDs, and permissions is useful for home directory servers and other servers participating in a network user database. If the user IDs or user names do not match, or if the AFP client is not running on a UNIX-based machine, the AFP client should map UNIX user IDs, group IDs and permissions to traditional User IDs, Group IDs and permissions. This default behavior can be changed by settings on the server. The server can be forced to always set or to never set the kNoNetworkUserIDs bit.
kDefaultPrivs-FromParent (bit 8)	If set, directories inherit default privileges from the parent directory.
kNoExchangeFiles (bit 9)	If set, exchange files is not supported.
kSupportsExtAttrs (bit 10)	If set, the volume supports extended attributes.
kSupportsACLs (bit 11)	If set, the volume supports access control lists (ACLs).

Volume Types

An AFP volume is structured hierarchically. There are two types of hierarchical volumes: fixed and variable. A fixed Directory ID volume contains multiple directories, with each directory having its own permanent Directory ID that is assigned when the directory is created. The Directory ID is not used for any other directory during the lifetime of the volume, even if the directory to which it is assigned is later deleted.

A variable Directory ID volume also maintains the uniqueness of its Directory IDs but differs from a fixed Directory ID volume in that it does not associate a permanent Directory ID with each directory. For variable Directory ID volumes, the file server creates a unique Directory ID for a directory whenever the AFP client sends an FPOpenDir command. The file server then maintains this Directory ID until the client sends an FPCloseDir command or the AFP session terminates. A Directory ID obtained by sending an FPOpenDir command to a variable Directory ID volume must be used only for that session. If the Directory ID is stored and used to reference the directory in a later session, the results cannot be predicted: the command may fail, manipulate the wrong directory, or accidentally manipulate the correct directory.

Table 1-4 Volume types

Value	Description
1	Flat

Value	Description
2	Fixed Directory ID
3	Variable Directory ID (deprecated)

The volume types have the following support capabilities and constraints: Personal computers using MS-DOS can gain access to any type of server volume because the concept of Directory IDs is foreign to their file systems. However, Macintosh computers using the hierarchical file system (HFS) cannot directly use variable Directory ID volumes. Macintosh HFS volumes are fixed Directory ID volumes and hierarchical volumes on the file server can be handled by HFS only if they are fixed Directory ID volumes. Mac OS applications, such as the Finder, save Directory IDs and do no expect them to vary.

Note: AFP 3.*x* servers do not advertise support for variable Directory ID volumes, and AFP 3.*x* clients are not required to support variable Directory ID volumes.

Volume Catalog

The volume catalog is the structure that describes the branching tree arrangement of files and directories on fixed and variable Directory ID volumes. The catalog does not span multiple volumes; the AFP client sees a separate volume catalog for each server volume that is visible to AFP clients. Figure 1-2 (page 16) shows an example of a volume catalog and illustrates its elements.

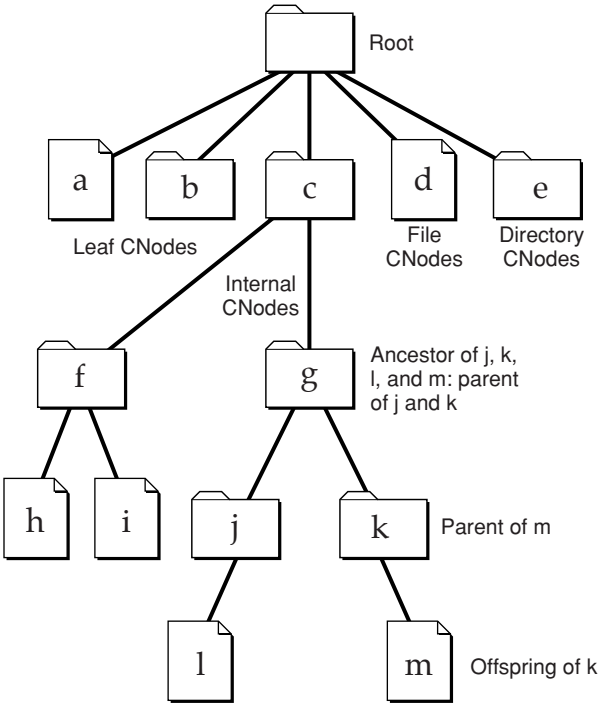
The volume catalog contains directories and files branching from a base directory known as the root. These directories and files are referred to as catalog nodes or CNodes (not to be confused with devices on a network, which are also known as nodes). Within the tree structure, CNodes can be positioned in two ways:

- at the end of a limb, in which case the CNode is called a leaf; a leaf CNode can be a file or an empty directory
- connected from above and below to other CNodes, which case the CNode is called an internal CNode. Internal CNodes are always directories

CNodes have a parent/offspring relationship. A given CNode is the offspring of the CNode above it in the catalog tree, and the higher CNode is considered its parent directory. Offspring are contained within the parent directory. The only CNode that does not have a parent directory is the root directory.

When an AFP command makes its way through the volume catalog, it can take only one shortest path from the root to a specific CNode. The CNodes along that path are said to be ancestors of the destination node, which in turn is called the descendent of each of its ancestors.

Figure 1-2 Volume catalog



Catalog Node Names

CNode names identify every directory and file in a volume catalog. Each directory and file has a Long Name, a Short Name, and may also have an AFPName.

Long Names and Short Names correspond in two of the native file systems that AFP supports: the Mac OS refers to files and directories by Long Names; MS-DOS computers use Short Names.

AFPNames are encoded in conformance to the Unicode standard (UTF-8), which uses 16-bits to encode more than 65,000 characters. To keep character coding simple and efficient, the Unicode Standard assigns each character a unique numeric value and name. To help when converting from UTF-8 to other script systems, AFPNames begin with a four-byte text encoding hint that specifies the script that was originally used to compose the name. The text encoding hint is followed by a two-byte length field specifying the length of the UTF-8 encoded name that follows.

The header file, `TextCommon.h`, for the Text Encoding Conversion Manager defines the constants for the text encoding hint:

```
enum {
    kTextEncodingMacRoman = 0,
    kTextEncodingMacJapanese = 1,
    kTextEncodingMacChineseTrad = 2,
    kTextEncodingMacKorean = 3,
    kTextEncodingMacArabic = 4,
    kTextEncodingMacHebrew = 5,
    kTextEncodingMacGreek = 6,
    kTextEncodingMacCyrillic = 7,
    kTextEncodingMacDevanagari = 9,
```

```
    kTextEncodingMacGurmukhi = 10,
    kTextEncodingMacGujarati = 11,
    kTextEncodingMacOriya = 12,
    kTextEncodingMacBengali = 13,
    kTextEncodingMacTamil = 14,
    kTextEncodingMacTelugu = 15,
    kTextEncodingMacKannada = 16,
    kTextEncodingMacMalayalam = 17,
    kTextEncodingMacSinhalese = 18,
    kTextEncodingMacBurmese = 19,
    kTextEncodingMacKhmer = 20,
    kTextEncodingMacThai = 21,
    kTextEncodingMacLaotian = 22,
    kTextEncodingMacGeorgian = 23,
    kTextEncodingMacArmenian = 24,
    kTextEncodingMacChineseSimp = 25,
    kTextEncodingMacTibetan = 26,
    kTextEncodingMacMongolian = 27,
    kTextEncodingMacEthiopic = 28,
    kTextEncodingMacCentralEurRoman = 29,
    kTextEncodingMacVietnamese = 30,
    kTextEncodingMacExtArabic = 31,
    kTextEncodingMacSymbol = 33,
    kTextEncodingMacDingbats = 34,
    kTextEncodingMacTurkish = 35,
    kTextEncodingMacCroatian = 36,
    kTextEncodingMacIcelandic = 37,
    kTextEncodingMacRomanian = 38,
    kTextEncodingMacCeltic = 39,
    kTextEncodingMacGaelic = 40,
    kTextEncodingMacKeyboardGlyphs = 41,
    kTextEncodingMacUnicode = 126,
    kTextEncodingMacFarsi = 140,
    kTextEncodingMacUkrainian = 152,
    kTextEncodingMacInuit = 236,
    kTextEncodingMacVT100 = 252,
    kTextEncodingMacHFS = 255,
    kTextEncodingUnicodeDefault = 256,
    kTextEncodingUnicodeV1_1 = 257,
    kTextEncodingISO10646_1993 = 257,
    kTextEncodingUnicodeV2_0 = 259,
    kTextEncodingUnicodeV2_1 = 259,
    kTextEncodingUnicodeV3_0 = 260,
    kTextEncodingISOLatin1 = 513,
    kTextEncodingISOLatin2 = 514,
    kTextEncodingISOLatin3 = 515,
    kTextEncodingISOLatin4 = 516,
    kTextEncodingISOLatinCyrillic = 517,
    kTextEncodingISOLatinArabic = 518,
    kTextEncodingISOLatinGreek = 519,
    kTextEncodingISOLatinHebrew = 520,
    kTextEncodingISOLatin5 = 521,
    kTextEncodingISOLatin6 = 522,
    kTextEncodingISOLatin7 = 525,
    kTextEncodingISOLatin8 = 526,
    kTextEncodingISOLatin9 = 527,
    kTextEncodingDOSLatinUS = 1024,
    kTextEncodingDOSGreek = 1029,
```

```
    kTextEncodingDOSBalticRim = 1030,
    kTextEncodingDOSLatin1 = 1040,
    kTextEncodingDOSGreek1 = 1041,
    kTextEncodingDOSLatin2 = 1042,
    kTextEncodingDOSCyrillic = 1043,
    kTextEncodingDOSTurkish = 1044,
    kTextEncodingDOSPortuguese = 1045,
    kTextEncodingDOSIcelandic = 1046,
    kTextEncodingDOSHebrew = 1047,
    kTextEncodingDOSCanadianFrench = 1048,
    kTextEncodingDOSArabic = 1049,
    kTextEncodingDOSNordic = 1050,
    kTextEncodingDOSRussian = 1051,
    kTextEncodingDOSGreek2 = 1052,
    kTextEncodingDOSThai = 1053,
    kTextEncodingDOSJapanese = 1056,
    kTextEncodingDOSChineseSimplif = 1057,
    kTextEncodingDOSKorean = 1058,
    kTextEncodingDOSChineseTrad = 1059,
    kTextEncodingWindowsLatin1 = 1280,
    kTextEncodingWindowsANSI = 1280,
    kTextEncodingWindowsLatin2 = 1281,
    kTextEncodingWindowsCyrillic = 1282,
    kTextEncodingWindowsGreek = 1283,
    kTextEncodingWindowsLatin5 = 1284,
    kTextEncodingWindowsHebrew = 1285,
    kTextEncodingWindowsArabic = 1286,
    kTextEncodingWindowsBalticRim = 1287,
    kTextEncodingWindowsVietnamese = 1288,
    kTextEncodingWindowsKoreanJohab = 1296,
    kTextEncodingUS_ASCII = 1536,
    kTextEncodingJIS_X0201_76 = 1568,
    kTextEncodingJIS_X0208_83 = 1569,
    kTextEncodingJIS_X0208_90 = 1570
};
```

To allow dissimilar computers to share resources, the file server provides CNode names in all three formats. When creating or renaming files and directories, the user provides a name consistent with the native file system. The server then uses an algorithm to generate the other name (Long or Short). This section describes the rules for forming CNode names and the algorithm used for creating and maintaining dual names.

The syntax for forming AFP Long Names is the same as the naming syntax used by the Macintosh HFS, with one exception: Null (0x00) is not a permissible character in AFP Long Names. Otherwise, the mapping of character code to character is the same for AFP as it is for Mac OS X. [See "Character Encoding" (page 64).] AFP Long Names are made up of at most 31 characters; valid characters are any printable ASCII code except colon (0x3A) and null (0x00). The volume name, and by inference the root's Long Name, cannot be longer than 27 bytes.

The syntax for forming AFP Short Names is the same as the naming syntax used by MS-DOS, which is more restrictive than the naming syntax used in the Mac OS: Names may be up to eight alphanumeric characters, optionally followed by a period (0x2E) and a one-to-three character alphanumeric character extension.

To ensure that a CNode can be uniquely specified by either name, AFP defines the following rules:

- no two offspring of a given directory can have the same Short Name or the same Long Name.
- a Short Name can match a Long Name if they both belong to the same file or directory.

Therefore, either name, Long or Short, uniquely identifies CNodes within the same directory.

AFP naming rules are such that any MS-DOS name can be used directly as a CNode Short Name, and any Mac OS X name can be used as a Long Name. The file server generates the other name for each CNode, deriving it from the first name specified and matching the second name as closely as possible. The Long Name format is a superset of the Short Name format. The name management algorithm mandates that whenever a CNode is created or renamed with a Short Name, the Long Name will always match. Deriving a Short Name from a Long Name is not so simple, and AFP does not stipulate an exact algorithm for this derivation. Therefore, different servers may create Short Names differently.

When a CNode is created, the caller supplies the node's name and a name type that indicates whether the name is a Long or Short Name. The server then checks the name to verify that the name conforms to the accepted format. The algorithm that follows describes how servers assign Short and Long Names to a CNode (referred to as an object of this algorithm).

```
IF name type is Short or name is in Short format
THEN check for new name in list of Short Names
    IF name already exists
    THEN return ObjectExists result
    ELSE set object's Short and Long Names to new name

ELSE { name type is Long OR name is in Long format }
    check for new name in list of Long Names
    IF name already exists
    THEN return ObjectExists result
    ELSE set object's Long Name to new name
        derive Short Name from Long Name
```

This algorithm is used for renaming as well as for creating new names. When a user renames an object, its other name is changed using the above algorithm.

One limitation of this algorithm is that it does not prevent a user from specifying a Long Name that matches the Short Name generated by the file server for another file. A server-generated Short Name is normally not visible to an user that sees only Long Names. If a user inadvertently specifies a Long Name that matches a Short Name, the command fails and the server returns a kFPObjectErr.

For example, for a file created with the Long Name "MacFileLongName", a file server can generate a Short Name of "MacFile". When the user tries to create a new file with the Long Name "MacFile" in the same directory, the command fails because the above algorithm stipulates that the Long Name and the Short Name would both have to be set to "MacFile".

Note: The root directory of a volume catalog represents the volume, and the root's Long Name is the same as the volume name. The volume may also have a UTF-8–encoded name. The volume has a Short Name, which is the Short Name of the root directory, but AFP does not allow its use. Neither the root nor the volume can be deleted or renamed through AFP.

If an AFP client creates a file having a UTF-8–encoded name, the file server is required to generate a valid Long Name and a valid Short Name for the file. The algorithm for generating Long and Short Names for a file having a UTF-8–encoded file name is beyond the scope of this specification.

Directories and Files

Directories and files are stored in volumes and constitute the next level of the file system structure visible to AFP clients. As was shown in Figure 1-2 (page 16), directories branch to files and other directories. Each directory has an identifier through which it and its offspring can be addressed. Therefore, directories can be thought of as logically containing their offspring directories and files with the parameters described below.

Directory IDs

Each directory in the volume catalog is identified by a four-byte long integer known as its Directory ID. Because two directories on the same volume cannot have the same Directory ID, the Directory ID uniquely identifies a directory within a volume.

Within the volume catalog, as mentioned earlier, directories have ancestor, parent, and offspring relationships with each other. The Directory ID of a CNode's parent is called the CNode's Parent ID.

A CNode can have only one parent, so a given CNode has an unique Parent ID. However, a CNode can have several ancestor directory identifiers, one for each ancestor. The parent directory is considered an ancestor.

Directory IDs from 1 to 16 are reserved. The Directory ID of the root is always 2. The root's Parent ID is always 1. (The root does not really have a parent; this value is returned only if an AFP command asks for the root's Parent ID.) Zero (0) is not a valid Directory ID.

Directory Parameters

For each directory, the server must maintain the parameters listed in Table 1-5. The parameters are obtained by calling `FPGetFileDirParms` and specifying in the `DirBitMap` parameter the directory parameters that are to be obtained. Some directory parameters can be set by calling `FPSetDirParms` or `FPSetFileDirParms`.

Table 1-5 Definitions for the Directory bitmap

Constant and bit position	Parameter size	Description
`kFPAttributeBit` (bit 0)	Two bytes	Additional information about the directory. For details, see Table 1-6 (page 21).
`kFPParentDirIDBit` (bit 1)	Four bytes	The unique identifier of the directory's parent directory.
`kFPCreateDateBit` (bit 2)	Four bytes	Date the directory was created. For more details, see the section "Date-Time Values" (page 25).
`kFPModDateBit` (bit 3)	Four bytes	Date the directory was most recently modified. For more details, see the section "Date-Time Values" (page 25).
`kFPBackupDateBit` (bit 4)	Four bytes	Date the directory was most recently backed up. For more details, see the section "Date-Time Values" (page 25).
`kFPFinderInfoBit` (bit 5)	32 bytes	Accompanies directories that are used by computers with HFS. This parameter is maintained by the AFP client and is not examined by AFP.

Constant and bit position	Parameter size	Description
kFPLongNameBit (bit 6)	String of up to 32 characters	The directory's Long Name. For details, see "Catalog Node Names" (page 16).
kFPShortNameBit (bit 7)	String of up to 12 characters	The directory's Short Name. For details, see "Catalog Node Names" (page 16).
kFPNodeIDBit (bit 8)	Four bytes	The directory's unique identifier.
kFPOffspringCountBit (bit 9)	Four bytes	Number of files and directories contained by the directory.
kFPOwnerIDBit (bit 10)	Four bytes	A User ID that uniquely identifies the owner of the directory. Starting with AFP 2.0, a directory can be owned by a group. For more information about the Owner ID parameter, see the section "Directory Access Controls" (page 54).
kFPGroupIDBit (bit 11)	Four bytes	A number that uniquely identifies the group affiliation of the directory. Starting with AFP 2.0, the Group ID can be a User ID. For more information about the Group ID parameter, see the section "Directory Access Controls" (page 54).
kFPAccessRightsBit (bit 12)	Four bytes	A bitmap that describes the access rights for the directory's owner, group affiliation, and Everyone. This bitmap also includes the UARights Summary byte.
kFPUTF8NameBit (bit 13)	AFPName	The directory's UTF-8–encoded name. For information about this encoding, see the discussion on the Unicode standard in the section "Catalog Node Names" (page 16).
kFPUnixPrivsBit (bit 15)	16 bytes	If the directory has UNIX privileges, they are stored in an FPUnixPrivs structure.

The Attributes parameter for directories provides additional information about the directory. Table 1-6 lists the bit definitions for the Attributes parameter for directories.

Table 1-6 Bit definitions for the directory Attributes parameter

Bit	Description
Invisible (bit 0)	The directory should not be made visible to the user.
IsExpFolder (bit 1)	The directory is a share point. This directory, and all directories within it, indicate to the user that access privileges are valid (for example, by displaying tabbed folders or drop-box folder icons or by enabling the Sharing menu item). None of the directories outside the shared (exported) area show access privileges on local computers, although they may still have valid access privilege information that only an administrator can see or modify. This bit is a read only bit. It cannot be set by FPSetFileDirParms.

Bit	Description
System (bit 2)	The directory is a system directory; the definition of "system directory" is left to the local computer.
Mounted (bit 3)	The directory is mounted by a user who is not an administrator. The icon for such a folder indicates to the user of the local computer that this directory is a share point, and that a remote user currently has it mounted. This bit is a read only bit. It cannot be set by `FPSetFileDirParms`.
InExpFolder (bit 4)	The directory is in a shared area. This directory, and all directories within this directory, indicate to the user that access privileges are valid. This directory cannot be shared because a share point cannot exist within another share point. This bit is a read only bit. It cannot be set by `FPSetFileDirParms`.
BackupNeeded (bit 6)	The directory needs to be backed up. This bit is set whenever the directory's modification date-time is modified.
RenameInhibit (bit 7)	The directory cannot be renamed.
DeleteInhibit (bit 8)	The directory cannot be deleted.
Set/Clear (bit 15)	When used in conjunction with the `FPSetFileDirParms` command, indicates whether the specified attributes are to be set (1) or cleared (0). It is not possible to set some attributes and clear other attributes in the same call.

No specific bit exists to inhibit moving a directory, but directory movement is constrained by the RenameInhibit bit when a directory is moved or moved and renamed.

Access Rights (a four-byte quantity) contains the read, write, and search access privileges corresponding to the directory's owner, group, and Everyone. The upper byte of the Access Rights parameter is the User Access Rights Summary byte, which indicates the privileges the current user of the AFP client has to this directory. Table 1-7 lists the bit definitions for the Access Rights parameter for directories.

Table 1-7 Bit definitions for the directory Access Rights parameter

Bit	Description
(bit 0)	Set if the directory's owner has search access to this directory.
(bit 1)	Set if the directory's owner has read access to this directory.
(bit 2)	Set if the directory's owner has write access to this directory.
(bit 8)	Set if the directory's group has search access to this directory.
(bit 9)	Set if the directory's group has read access to this directory.
(bit 10)	Set if the directory's group has write access to this directory.
(bit 16)	Set if Everyone has search access to this directory.
(bit 17)	Set if Everyone has read access to this directory.

Bit	Description
(bit 18)	Set if Everyone has write access to this directory.
(bit 24)	Set if the user has search access to this directory.
(bit 25)	Set if the user has read access to this directory.
(bit 26)	Set if the user has write access to this directory.
Blank Access Privileges (bit 28)	This directory has blank access privileges and has the same access privileges as the directory enclosing it.
(bit 31)	Set if the user is the owner of the directory. It is also set if the directory is not owned by a registered user.

An `FPUnixPrivs` structure is used to return UNIX privileges if a file or directory resides on a volume that supports UNIX privileges. The `FPUnixPrivs` structure is defined as

```
struct FPUnixPrivs {
    unsigned long uid;
    unsigned long gid;
    unsigned long permissions;
    unsigned long ua_permissions;
};
```

where

- `uid` is the user ID of the file or directory's owner
- `gid` is the group ID of the file or directory's owner
- `permissions` is the setting of the file or directory's permission bits
- `ua_permissions` is the user's access rights to the file or directory; bit 31 is set if the user is the owner of the file or directory

File Parameters

For each file, the server must maintain the parameters listed in Table 1-8. The parameters are obtained by calling `FPGetFileDirParms` and specifying in the `FileBitmap`parameter the file parameters that are to be obtained, by calling `FPResolveID` and specifying the file's File ID, or by calling `FPGetForkParms`. Some file parameters can be set by sending `FPSetFileParms`, `FPSetFileDirParms`, and `FPSetForkParms` commands.

Table 1-8 Definitions for the File bitmap

Constant and bit position	Parameter size	Description
`kFPAttributeBit` (bit 0)	Two bytes	Additional information about the file. For details, see Table 1-9 (page 25).
`KFPParentDirIDBit` (bit 1)	Four bytes	The unique identifier for the file's parent directory.

Constant and bit position	Parameter size	Description
`kFPCreateDateBit` (bit 2)	Four bytes	Date the file was created. For more details, see the section "Date-Time Values" (page 25).
`kFPModDateBit` (bit 3)	Four bytes	Date the file was most recently modified. For more details, see the section "Date-Time Values" (page 25).
`kFPBackupDateBit` (bit 4)	Four bytes	Date the file was most recently backed up. For more details, see the section "Date-Time Values" (page 25).
`kFPFinderInfoBit` (bit 5)	32 bytes	Accompanies files that are used by computers with HFS. This parameter is maintained by the AFP client and is not examined by AFP.
`kFPLongNameBit` (bit 6)	String of up to 32 characters	The file's Long Name. For details, see "Catalog Node Names" (page 16).
`kFPShortNameBit` (bit 7)	String of up to 12 characters	The file's Short Name. For details, see "Catalog Node Names" (page 16).
`kFPNodeIDBit` (bit 8)	Four bytes	The file's unique number obtained by the file server from the File Manager's PBGetCatInfo call.
`kFPDataForkLenBit` (bit 9)	Four-byte unsigned integer	The current length of the file's data fork. If the data fork's length is greater than 4 GB, specifying this bit returns the actual length of the data fork. If the data fork's length is greater than 4 GB, specifying this bit returns 4 GB.
`kFPRsrcForkLenBit` (bit 10)	Four-byte unsigned integer	The current length of the file's resource fork. If the resource fork's length is greater than 4 GB, specifying this bit returns the actual length of the resource fork. If the resource fork's length is greater than 4 GB, specifying this bit returns 4 GB.
`kFPExtDataForkLenBit` (bit 11)	Eight-byte unsigned integer	The current length of the file's data fork.
`kFPLaunchLimitBit` (bit 12)		
`kFPUTF8NameBit` (bit 13)	AFPName	The file's name in UTF-8 format.
`kFPExtRsrcForkLenBit` (bit 14)	Eight-byte unsigned integer	The current length of the file's resource fork.
`kFPUnixPrivsBit` (bit 15)	16 bytes	If the file has UNIX privileges, they are returned in an `FPUnixPrivs` structure.

The file number is a unique number associated with each file on the volume. This number is purely informative; AFP does not allow the specification of a file by its file number.

The Attributes parameter for files provides additional information about the file. Table 1-9 lists the bit definitions for the Attributes parameter for files.

Table 1-9 Bit definitions for file Attributes parameter

Constant and bit position	Description
`kFPInvisibleBit` (bit 0)	File should not be made visible to the user.
`kFPMultiUserBit` (bit 1)	File is an application that has been written for simultaneous use by more than one user.
`kFPSystemBit` (bit 2)	File is a system file.
`kFPDAlreadyOpenBit` (bit 3)	File's data fork is currently open by a user.
`kFPRAlreadyOpenBit` (bit 4)	File's resource fork is currently open by a user.
`kFPWriteInhibitBit` (bit 5)	User cannot write to the file's forks.
`kFPBackUpNeededBit` (bit 6)	File needs to be backed up.
`kFPRenameInhibitBit` (bit 7)	File cannot be renamed.
`kFPDeleteInhibitBit` (bit 8)	File cannot be deleted.
`kFPCopyProtectBit` (bit 10)	File should not be copied.
`kFPSetClearBit` (bit 15)	When used in conjunction with the `FPSetFileDirParms` command, indicates whether the specified attributes are to be set (1) or cleared (0). It is not possible to set some attributes and clear other attributes in the same call.

No specific bit exists to inhibit moving a file, but file movement is constrained by the RenameInhibit bit only when a file is moved and renamed, not when it is simply moved.

The Finder will not copy a file whose CopyProtect bit is set. An attempt to copy the file using the FPCopyFile request will in an error. This bit may be read, but not set, using AFP. It is to be set by some administrative program, whose specification is beyond the scope of this document.

The BackupNeeded bit is set whenever the file's modification date-time changes.

The data fork length and resource fork length are equal to the number of bytes in the corresponding fork.

The creation, backup, and modification date-time parameters are described next.

Date-Time Values

All date-time quantities used by AFP specify values of the server's clock. These values correspond to the number of seconds measured from 12:00 am on January 1, 2000 in Greenwich Mean Time (GMT). AFP represents date-time values with four-byte signed integers. One of the AFP commands allows the AFP client to obtain the current value of the server's clock. At login time, the AFP client should read this value (*s*) and the value of the AFP client's clock (*w*) and computer the offset between these values (*s* - *w*). All subsequent

date-time values read from the server should be adjusted by adding this offset to the date-time. This adjustment will correct for differences between the two clocks and will ensure that all computers see a consistent time base.

The creation date-time of a directory or a file is set to the server's system clock when the file or directory is created. The backup date-time is set by backup programs. When a file or directory is created, the server sets the backup date-time to 0x80000000, which is the earliest representable time.

The server changes the modification date-time of a directory each time the directory's contents are modified. Therefore, any of the following actions will cause the server to assign a new modification date to the directory: renaming the directory; creating or deleting a CNode in the directory; moving the directory; changing its access privileges, Finder Info, or changing the Invisible attributes of one of its offspring.

An AFP client with the appropriate privileges can set the creation and modification date-time parameters to any value.

File Forks

A file consists of two forks: a data fork and a resource fork. The bytes in a file fork are sequentially numbered starting with 0. The data fork is an unstructured finite sequence of bytes. The resource fork is used to hold Mac OS resources, such as icons and drivers, and a data structure for mapping them within the fork. AFP is designed to consider both forks as finite-length byte sequences; however, AFP contains no rules relating to the structure of the resource fork. For more information about resource forks, refer to *Inside Mac OS X*.

Either or both forks of a given file can be empty. Non-Mac OS AFP clients that need only one file fork must use the data fork. Files created by a computer with an MS-DOS operating system will have an empty resource fork because a resource fork is unintelligible to that operating system. Consequently, an MS-DOS computer that has gained access to a server file created by a Macintosh may not be aware of the existence of the file's resource fork.

Although AFP allows the creation of MS-DOS applications that can understand and manipulate resource forks, such applications would have to preserve the internal structure of the forks. Mac OS computers expect a specific format in the resource fork of any file, so AFP clients of computers that cannot manage the internal structure of the resource fork should never alter the contents of a resource fork.

To read from or write to the contents of a file's data or resource fork, the AFP client first sends a command to open the particular fork of the file, creating an access path to that file fork. The access path is not be confused with the paths and pathnames described in the next section.

Once the AFP client creates this access path, all subsequent read and write commands refer to it for the duration of the session. For each access path, the server maintains the parameters listed in Table 1-10.

Table 1-10 Access path parameters

Parameter	Description
`OForkRefNum`	Two bytes (0 is invalid)
`AccessMode`	Two-byte bitmap
`Flag`	Bit 7 of a one-byte value

The OForkRefNum parameter uniquely identifies the access path among all access paths within a given session. The AccessMode parameter indicates to the server whether this access path allows reading or writing. It is maintained by the server and is inaccessible to clients of AFP. The Flag parameter indicates to the server that the access path belongs to the data or the resource fork.

In addition to the above parameters, the server must provide a way to gain access to the parameters of the file to which an open fork belongs. For details, see the FPGetForkParms command in the Reference section.

Designating a Path to a CNode

In order to perform any action on a CNode, the AFP client must designate a path to the CNode. AFP provides rules for specifying a path to any CNode in the volume catalog. A CNode (file or directory) can be unambiguously specified to the server by the identifiers shown in Figure 1-3.

Figure 1-3 CNode specification

The Volume ID specifies the volume on which the destination CNode resides. The Directory ID can belong to the destination CNode (if the CNode is a directory) or to any one of its ancestor directories, up to and including the root directory and the root's parent directory.

An AFP pathname is formatted as a Pascal string (one length byte followed by the number of characters specified by the length byte) or a UTF-8 string (a four-byte text encoding hint followed by two length bytes followed by the number of characters specified by the length bytes). An AFP pathname is made up of CNode names, concatenated with intervening null-byte separators. Each element of a pathname must be the name of a directory, except for the last one, which can be the name of a directory or a file.

The elements of a pathname can be Long or Short Names. However, a given pathname cannot contain a mixture of Long and Short Names. A path type byte, which indicates whether the elements of the pathname are all Short or all Long Names, is associated with each pathname. A pathname consisting of Short Names and a path type of 1. A pathname consisting of Long Names has a path type of 2.

An AFP pathname that consists of Long or Short Names can be up to 255 characters long. The length of an AFP pathname that consists of UTF-8–encoded names is virtually unlimited. A single null byte as the length byte indicates that no pathname is supplied. Because the length byte is included at the beginning of the string, each pathname element (CNode name) does not include a length indicator.

The syntax of an AFP pathname follows this paragraph. The asterisk (*) represents a sequence of zero or more of the preceding elements of the pathname; the plus (+) represents a sequence of one or more of the preceding elements; <Sep> represents the separators in the pathname; the vertical bar (|) is an OR operator; and the term on the left side of the ::= symbol is defined as the term(s) on the right side.

```
<Sep> ::= <null-byte>+
<Pathname> ::= empty-string |
    <Sep>*<CNode name>(<Sep><Pathname>*
```

The syntax represents a concatenation of CNode names separated by one or more null bytes. Pathnames can also start or end with a string of null bytes.

A pathname can be used to traverse the volume catalog in any direction. The pathname syntax allows paths either to descend from a particular CNode through its offspring or to ascend from a CNode to its ancestors. In either case, the directory that is the starting point of this path is defined separately from the pathname by its Directory ID. The first element of the pathname is an offspring of the starting point of the directory. The pathname must be parsed from left to right to obtain each element that is used as the next node on the path.

To descend through a volume, a valid pathname must proceed in order from parent to offspring. A single null-byte separator preceding this first element is ignored.

To ascend through a volume, a valid pathname must proceed from a particular CNode to its ancestor. To ascend one level in the catalog tree, two consecutive null bytes should follow the offspring CNode name. To ascend two levels in the catalog tree, three consecutive null bytes are used as the separator, and so on.

A particular volume may descend and ascend through the volume catalog. Because of this, many valid pathnames may refer to the same CNode.

A complete path specification can take a number of forms. The table that follows summarizes the different kinds of path specifications that can be used to traverse the volume catalog illustrated in Figure 1-4 (page 28). A zero in square brackets `[0]` represents a null-byte separator.

The descriptors and examples that follow refer to this table and the corresponding volume catalog illustrated in Figure 1-4. To simplify these examples, the CNodes in this catalog are named *a* through *j*, except the root, which is named *x*. The path type is ignored in this example. The letter *v* represents the volume's two-byte Volume ID. Lines connect the CNodes; the unconnected lines indicate that other CNodes in this volume are not shown.

Figure 1-4 Example of a volume catalog

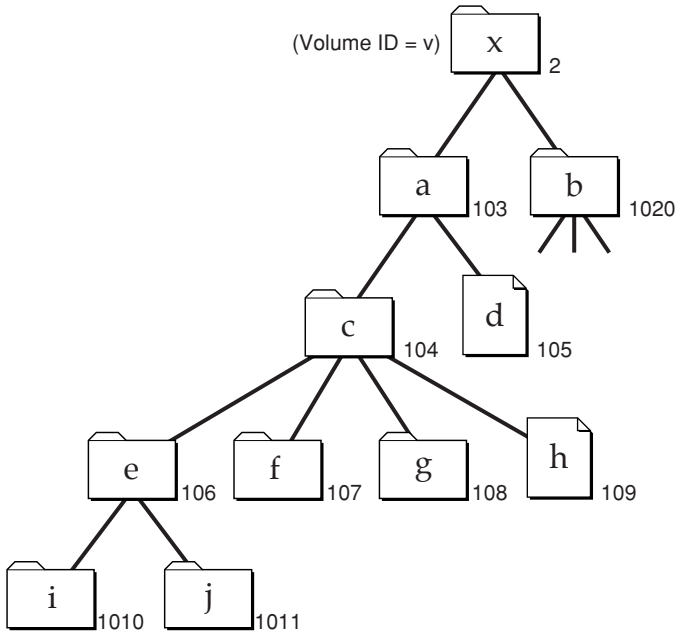


Table 1-11 provides the Volume ID, Directory ID, and pathname for some sample path specifications in Figure 1-4 (page 28).

Table 1-11 Sample path specifications

Example	Volume ID	Directory IDs	Pathname
First	v	2	a[0]c[0]e[0]j[o]
Second	v	104	e[0]j
Third	v	106	[0]j
Fourth	v	106	j
Fifth	v	106	[0]
Sixth	v	104	e[0][0]g[0][0]h
Seventh	v	104	e[0][0][0]
Eighth	v	1	x[0]a[0]c[0]h

The first example of a path specification in Table 1-11 contains the Volume ID, the root directory's Directory ID, which is always 2, and a pathname. In this case, the pathname must contain the names of all of the destination file's ancestors except the root, and it must end with the name of the file itself. The single trailing null byte is ignored.

The second example contains the Volume ID, the Directory ID of an ancestor, and a pathname.

The third example is essentially the same as the second example. The single leading null byte is ignored.

In the fourth example, the Directory ID is the Parent ID of the destination file. In this case, the pathname need contain only the name of the destination file itself.

The fifth example illustrates another way to uniquely specify a descending path to a directory. It includes the CNode's Volume ID, its Directory ID, and a null pathname. This path specification is used to specify the directory *e*.

The sixth example illustrates a descending path. The first CNode in the pathname is the offspring of the starting point Directory ID. Then the pathname ascends though *e*'s parent (c) down to directory *g*, backup to *g*'s parent (*c*), and down again to *h*.

The seventh shows an ascending pathname that starts at directory *c* (whose Directory ID is 104), moves down to *e*, and then ascends to *e*'s parent's parent (*a*).

The eighth example is a special case in which the starting point of the path is Directory ID 1, the parent of the root. The first name of the pathname must be the volume name or root directory name corresponding to Volume ID *v*; beyond that, pathname traversal is performed as in the other examples.

AFP Login

To make use of any resource managed by a file server, the AFP client must first log in to the server. This section provides an overview of the AFP login process.

After a user selects an AFP server to log in to, the AFP client sends the `FPGetSrvrInfo` command to request information about that server. The server returns information that includes

- the AFP versions the server supports
- the user authentication methods (UAMs) the server supports
- the server's machine type
- the server's name
- the server's network address
- the server's signature
- whether the server supports optional functionality, such as reconnect, Open Directory, `FPCopyFile`, `FPChangePassword`, saving passwords, and server notifications

During the AFP login process, the AFP client tells the server which AFP version the client will use to establish the connection and which UAM it will use to authenticate the user.

Each AFP version is uniquely described by a string of up to 16 characters called the AFP version string. The AFP version strings for the AFP versions supported by AFP 3.*x* are listed in Table 1-12 (page 30).

Table 1-12 AFP version strings

AFP version	AFP version string
AFP 2.1	AFPVersion 2.1
AFP 2.2	AFP2.2
AFP 2.3	AFP2.3
AFP 3.0	AFPX03
AFP 3.1	AFP3.1
AFP 3.2	AFP3.2

The UAMs supported by AFP 3.*x* and their corresponding strings are listed in Table 1-13 (page 30).

Table 1-13 AFP UAM strings

String	UAM name
`No User Authent`	No User Authentication UAM. For details, see "No User Authentication" (page 34).
`Cleartxt Passwrd`	Cleartext Password. For details, see "Cleartext Password" (page 35).
`Randnum Exchange`	Random Number Exchange. For details, see "Random Number Exchange" (page 36).
`2-Way Randnum`	Two-Way Random Number Exchange. For details, see "Two-Way Random Number Exchange" (page 38).

String	UAM name
DHCAST128	Diffie-Hellman Key Exchange. Allows the client to send a password of up to 64 bytes to the server through a strongly encrypted "tunnel." This type of encryption is useful for servers that require the use of cleartext password. For details, see "Diffie-Hellman Key Exchange" (page 39).
DHX2	Diffie-Hellman Key Exchange 2. Allows the client to send a password of up to 256 bytes to the server through a strongly encrypted "tunnel." This type of encryption is useful for servers that require the use of cleartext password. For details, see "Diffie-Hellman Key Exchange 2" (page 44).
Client Krb v2	Kerberos. Allows the client to use Kerberos v4 and Kerberos v5 tickets to authenticate a user.
Recon1	The Reconnect UAM. Allows the client to use the FPLoginExt command to reconnect using a reconnect token (also known as a *credential*) containing all of the information required to authenticate.

The prospective AFP client initiates the login process by sending an FPLogin or an FPLoginExt command to the server. Both commands include the AFP version string and the UAM string that the client has selected. Depending on the selected UAM method, the FPLogin or FPLoginExt command may include user login information (such as a user name or password), or a subsequent FPLoginCont command may include such information. The sending of additional FPLoginCont commands may be required to complete user authentication, as described in the next section, "File Server Security" (page 34).

If the UAM succeeds, an AFP session between the AFP client and the server begins.

As mentioned earlier, in addition to the AFP and UAM versions that the server supports, the FPGetSrvrInfo command returns a Flags parameter whose bits provide additional information about the server that is useful to an AFP client. The bits of the Flags parameter are listed in Table 1-14.

Table 1-14 Bit definitions for the Flags parameter returned by FPGetSrvrInfo

Constant and bit position	Description
kSupportsCopyFile (bit 0)	Set if the server supports the FPCopyFile command.
kSupportsChgPwd (bit 1)	Set if the server supports the FPChangePassword command.
kDontAllowSavePwd (bit 2)	Set if the client should not allow the user to save his or her password for volumes mounted at system startup. The item-selection dialog box may still allow the user to save his or her name. However, when this bit is set, the button offering that option is not displayed.
kSupportsSrvrMsg (bit 3)	Set if the server supports the FPGetSrvrMsg command.
kSrvrSig (bit 4)	Server supports server signatures, which is a 16-byte number that uniquely identifies the server. An AFP client should use the server signature to ensure that it does not log in to the same server multiple times. Preventing multiple logins is important when the server is configured for multihoming.

Constant and bit position	Description
kSupportsTCP (bit 5)	Set if the server supports TCP/IP.
kSupportsSrvrNotify (bit 6)	Set if the server supports server notifications.
kSupportsReconnect (bit 7)	Set if the server supports the FPGetSessionToken and FPDisconnectOldSession commands.
kSupportsDirServices (bit 8)	Set if the server supports Open Directory.
kSupportsUTF8SrvrName (bit 9)	Set if the server supports server names in UTF-8 encoding.
kSupportsUUIDs	Set if the servers supports Universal Unique Identifiers (UUIDs).

Reconnecting Sessions

If an AFP session is disconnected due, for example, a network outage, but the AFP client still has the required information, the AFP client can reconnect the session.

Clients that use the Reconnect UAM, described in "Reconnect" (page 50), follow these steps to reconnect:

1. Log in successfully by calling FPLoginExt using a UAM that provides a session key.
2. Call FPGetSessionToken to get a token, specifying the Type parameters as kRecon1Login (5).
3. Periodically call FPGetSessionToken with the Type parameter set to kReconn1Refresh (7) to refresh the token before it expires.
4. If a disconnect occurs, call FPLoginExt to log in again, specifying the Reconnect UAM as the UAM, and passing the current token obtained by calling FPGetSessionToken in step 2 or 3. The reconnect token contains all of the user name and password information required for the server to authenticate the client, so logging in again does not require the client to repeat the authentication steps that took place in step 1.
5. If the login in step 4 completes successfully, call FPDisconnectOldSession and pass the token obtained in step 2. If the server can find the previous session identified by the token, it will transfer all the previous session's open files and locked resources to the new session and return a result code of kFPNoErr.

Clients that don't use the Reconnect UAM follow these steps to reconnect:

1. Log in successfully by calling FPLogin or FPLoginExt.
2. Call FPGetSessionToken to get a token.
3. If a disconnect occurs, log in again using the same UAM, user name and password that were used in step 1.
4. Call FPDisconnectOldSession and pass the token obtained in step 2. If the server can find the previous session identified by the token, it will transfer all the previous session's open files and locked resources to the new session and return a result code of kFPNoErr.

In either case, if the reconnect fails for any reason, the client must start over by logging in again as in step 1. The client should call `FPDisconnectOldSession` and send the current token to tell the server that it can free resources that were locked up by the earlier session.

Note: The `FPDisconnectOldSession` command will fail if the server cannot find the previous session or if the AFP client does not use same information to log in again. For security reasons, the server also fails all reconnects if the user originally logged in as the Guest user.

If the server returns a result code other than `kFPNoErr`, the AFP client can attempt to reopen its files. If the files were previously opened without Deny Modes and the AFP client did not apply byte range locks, the client should be able to reopen those files. In this case, reconnect is also deemed successful. If the reconnect is not successful, an AFP client can take the steps described in the next section, "Recovering From a System Crash."

Recovering From a System Crash

If an AFP session is disconnected and the client reconnect information is lost due to a local system crash, the AFP client will not be able to reconnect the session. If the server allows reconnect, any files that were left open on the remote server when the local system crashed will be saved but will not be available for opening until the reconnect timeout expires. This also applies to the case where a sleeping AFP client fails to wake up or crashes and the server is saving the information until the sleep timeout expires.

To tell the server to close files left open by an old session and disconnect that session, an AFP client that supports AFP 3.1 and later can create and save a unique client-defined identifier and use the `FPGetSessionToken` command to send it the server. The client must do this before the local system crashes, for example, as part of its login sequence. When it receives the identifier, the server associates the identifier with the current session.

Later, if the local system crashes and is restarted, the AFP client can log in and send the `FPGetSessionToken` command again, this time telling the server to look for a session having the specified identifier. If the server finds such a session, it closes the files that are associated with it, frees any other associated resources, and disconnects the old session.

Note: For security purposes, before disconnecting the old session, the server verifies that the same login information was used to log in from the same system. For security purposes, the server also fails `FPGetSessionToken` if the user originally logged in as the Guest user.

Disconnect Timers

In previous versions of AFP, there was only one timer for determining whether a disconnect had occurred. With 10.2.x, there are two timers for determining disconnections:

- Active timer, which is set to 60 seconds by default
- Idle timer, which is set to 120 seconds by default

If the client has an outstanding request to the server and has not received any data (including tickles) from the server, the client waits until the active timer expires before assuming that a disconnect has occurred.

If the client has no outstanding requests to the server, the client waits until the idle timer expires before assuming that a disconnect has occurred.

Special considerations arise when the system is awakening from sleep:

- If the system is on a LAN, the active timer is set to (activeTimer / 4).
- If the system is on a WAN, the active timer is set to (activeTimer / 2).

If the client is connected to an AFP 2.x server, the Active time and the Idle timer are both set to 120 seconds.

Note: The `check_afp.app` plist contains an option that can be set to disallow idle sleep if an AFP volume is mounted.

In all situations, after a disconnect, if the server supports reconnect, reconnect is started.

File Server Security

Information stored in a shared resource needs protection from unauthorized users. The role of file server security is to provide varying amounts and kinds of protection, depending on what users feel is necessary.

AFP provides security in three ways:

- user authentication when the user logs in to the server
- an optional volume-level password when the user first attempts to gain access to a volume
- directory access controls

User Authentication Methods

AFP provides the capability for servers and AFP clients to use a variety of methods to authenticate users when they log in. Five user authentication methods are defined: no user authentication, cleartext password, random number exchange, two-way random number exchange, and Diffie-Hellman Key Exchange. Some UAMs are also used to change a password after the user logs in.

The AFP client indicates its choice of UAM by giving the server a UAM string. These strings are intended to be case-insensitive and diacritical-sensitive.

Some UAMs require additional user authentication information to be passed to the server in the `FPLogin` or `FPLoginExt` command. The following sections describe the UAMs and the kinds of information they require.

No User Authentication

The No User Authentication UAM requires no authentication information. When the `FPLogin` and `FPLoginExt` commands use the No User Authentication UAM, there is no *UserAuthInfo* parameter. The corresponding case-insensitive UAM protocol name for the No User Authentication UAM is `No User Authent`. The No User Authentication UAM is used when a user logs on as the Guest user.

In order to implement the directory access controls described later in this section, the server must assign a User ID and a Group ID to the user for the session.

User ID numbers and Group ID numbers are assigned from the same pool of numbers. In addition, starting with AFP 2.1, AFP servers must assign zero to users who log in as the Guest user and 1 to the Administrator/Owner.

In this UAM, the server assigns to the user "Everyone" access privileges for every directory in every server volume. "Everyone" access privileges are described in the section "Directory Access Controls" (page 54).

Cleartext Password

The Cleartext Password UAM transmits the user name and password to the server as cleartext (that is, not encoded). The protocol name for the Cleartext Password UAM is `Cleartxt Passwrd`.

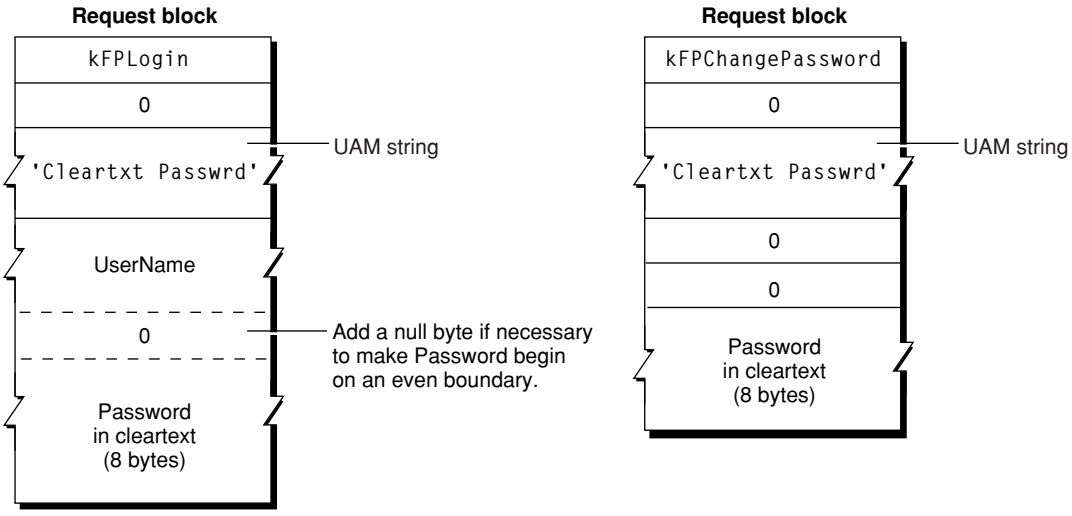
For the `FPLogin` command, the `UserAuthInfo` parameter consists of a user name (which is a string of up to 255 Macintosh Roman characters) followed by the user's password (up to 8 bytes). For the `FPLoginExt` command, the `UserAuthInfo` parameter consists of a user name (which is a string of up to 255 Unicode characters) followed by the user's password (up to 8 bytes). To ensure that the user's password is aligned on an even byte boundary in the packet, the AFP client may have to insert a null byte (0x00) between the user name and the password. If the user provides a password that is shorter than 8 bytes, it must be padded at the end with null bytes to make the password eight bytes long. The permissible set of characters in passwords consists of all 8-bit ASCII characters.

User name comparison is case-insensitive, but password comparison is case-sensitive for this UAM. If there is a user of the specified name whose password matches the password supplied by the caller of `FPLogin` or `FPLoginExt`, the user has been authenticated and the login succeeds. If the passwords do not match, a `kFPUserNotAuth` result code is returned.

The Cleartext Password UAM should be used by AFP clients only if the intervening network is secure against eavesdropping. Otherwise, the password information can be read from `FPLogin` or `FPLoginExt` command packets by anyone listening on the network.

Figure 1-5 (page 36) shows the request block for calling `FPChangePassword` when using the Cleartext Password UAM.

Figure 1-5 Request block when using the Cleartext Password UAM



Random Number Exchange

In environments in which the network is not secure against eavesdropping, Random Number Exchange is a more secure user authentication method. The protocol name for this UAM is `Randum Exchange`. With Random Number Exchange, the user's password is never sent over the network and cannot be picked up by eavesdropping. Deriving the password from information sent over the network is as difficult as breaking a DES-encrypted password.

With the Random Number Exchange UAM, only the user name is sent in the UserAuthInfo parameter of the `FPLogin` or `FPLoginExt` command. If the user name is valid, the server generates an eight-byte random number and sends it back to the AFP client, along with an ID number and a `kFPAuthContinue` result code. The `kFPAuthContinue` result code indicates that all is well at this point, but the user has not yet been authenticated. The AFP client then encrypts the random number with the user's password and sends the result to the server in the `UserAuthInfo` parameter of the `FPLoginCont` command along with the ID number returned earlier by the server in the reply block from the `FPLogin` or `FPLoginExt` command. The server uses the ID number to associate an earlier `FPLogin` or `FPLoginExt` command with this call to `FPLoginCont`. The server looks up the password for that user and uses it as a key to encrypt the same random number. If the two encrypted numbers match, the user has been authenticated and the login succeeds. Otherwise, the server returns a `kFPUserNotAuth` result code.

The following steps explain the Random Number Exchange UAM in greater detail:

1. The AFP client sends the `FPLogin` or `FPLoginExt` command block with the UAM string and the `UserAuthInfo` parameter containing the user name string. For `FPLogin`, the user name can be up to 255 Macintosh Roman characters long; for `FPLoginExt`, the user name can be up to 255 Unicode characters long.

2. Upon receiving this command block, the server examines its user database to determine whether the user name is valid. User name comparison is case-insensitive and diacritical-sensitive.

3. If the server does not find the user name in the user database, it sends an error code to the AFP client indicating that the user name is not valid and denies the login request. If the server finds the name in the user database, it generates an eight-byte random number and sends it to the AFP client, along with an ID number and an `kFPAuthContinue` result code. The `kFPAuthContinue` result code indicates that all is well at this point, but the user is not yet authenticated.

4. Both the AFP client and the server use the National Institute of Standards and Technology Data Encryption Standard (NIST DES) algorithm to encrypt the random number. The user's case-sensitive password is applied as the encryption key to generate an eight-byte value. The server applies the same algorithm to the password it finds associated with the user name in its database.

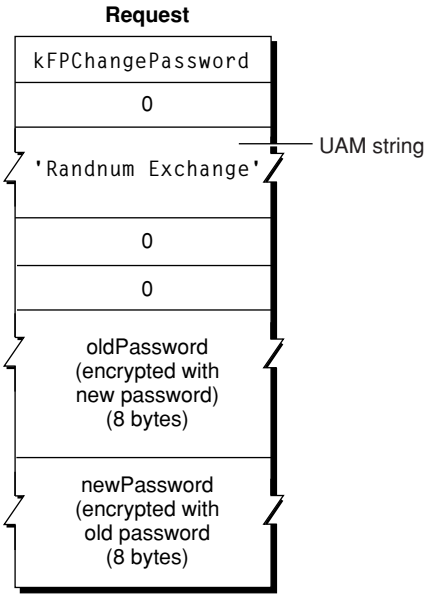
5. The AFP client sends the encrypted value to the server in the `UserAuthInfo` parameter of the `FPLoginCont` command, along with the ID number it received from the server. The server uses the ID number to associate a previous `FPLogin` or `FPLoginExt` command with its corresponding `FPLoginCont` command.

6. The server compares the AFP client's encrypted value with the encrypted value obtained using the password from its user database. If the two encrypted values match, the authentication process is complete and the login succeeds. The server returns a result code of `kFPNoErr` to the AFP client. If the two encrypted values do not match, the server returns the `kFPUserNotAuth` result code.

Note: The Random Number Exchange UAM uses eight-byte passwords consisting of eight-bit ASCII characters. The NIST DES algorithm ignores the low-order bit of each byte thereby using only 56 bits of the 64-bit password. The result is that in passwords, "0" is equivalent to "1", "b" is equivalent to "c", and so on.

Figure 1-6 (page 37) shows the request block for calling `FPChangePassword` when using the Random Number Exchange UAM.

Figure 1-6 Request block when using the Random Number Exchange UAM to change a password



Two-Way Random Number Exchange

With the Two-Way Random Number Exchange UAM, the user is authenticated to the server and the server is authenticated to the user, which guards against spoofing (that is, using a fake server to get passwords or data). This method uses the same steps as the Random Number Exchange UAM with three additional steps. The corresponding UAM string is `2-Way Randum`.

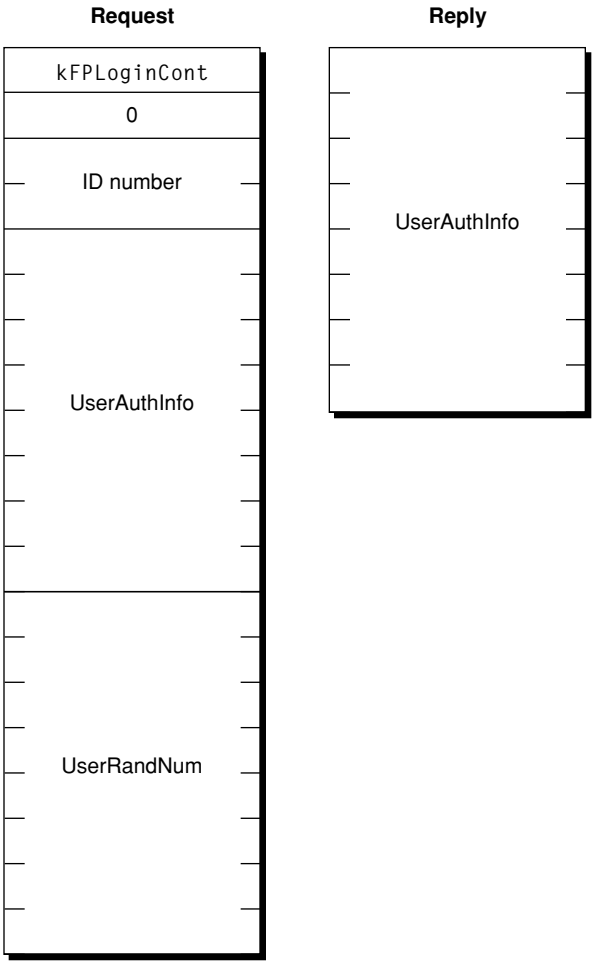
Like the Random Number Exchange UAM, the Two-Way Random Number Exchange UAM starts when the client sends the `FPLogin` or `FPLoginExt` request to the server that includes the user's user name. If the server finds the user name in the user name database, the server returns an ID number, an eight-byte random number, and a result code of `kFPAuthContinue`. The client then encodes the random number with the user's password and sends the encoded number and the ID number to the server in an `FPLoginCont` request. If the encoded password matches the server's copy of the random number encoded by the server's copy of the password, the client is authenticated and `kFPNoErr` is returned.

The additional steps of the Two-Way Random Number Exchange are

1. The client sends to the server an `FPLoginCont` request that includes a second eight-byte random number.
2. The server encodes the second eight-byte random number with it's copy of the user's password from the user database and returns the encoded random number in the `FPLoginCont` reply block.
3. The client encodes the random number with the user's password and compares it with the encoded random number from the server. If they match, the server is also authenticated.

Figure 1-7 (page 39) shows the request and reply block formats for the `FPLoginCont` command when the Two-Way Random Number Exchange UAM is used.

Figure 1-7 Request and reply blocks for Two-Way Random Number Exchange



The Two-Way Random Number Exchange UAM is not available for use with the `FPChangePassword` command. Instead, the Random Number Exchange UAM should be used to change a password. A user who has already logged in using the Two-Way Random Number Exchange UAM and who is changing his or her password has already authenticated the server, so there is no need to authenticate the server again by using the Two-Way Random Number Exchange UAM.

Note: With the Two-Way Random Number Exchange UAM, each password byte is shifted left one bit before it is used to encrypt the random number. This shifting causes the password's high-order bit to be ignored by the NIST DES algorithm instead of the low-order bit as with the Random Number Exchange UAM. Two values are still accepted for each byte of the password. However, the two values will not be adjacent in ASCII space and so will probably not be adjacent alphabetically. For example, "0" will match "□" "7" will match "", and so on.

Diffie-Hellman Key Exchange

Diffie-Hellman Key Exchange (DHX) is an implementation of the Diffie-Hellman Key Agreement Protocol using the SSLeay/OpenSSL implementation of CAST 128 in CBC mode. The UAM protocol name for DHX is `'DHCAST128'`.

DHX is strong against packet sniffing attacks but vulnerable to active attacks such "Man in the Middle." There is no way for the client to verify that the server knows the password, so the server could easily be spoofed. There is some weakness in using fixed initialization vectors, p and g. DHX is useful when the server requires passwords in cleartext.

With DHX, the client and the server each generate a random number, Ra and Rb respectively, which serve as "private keys" for the session. The client and server use modulus exponentiation to derive "public keys", Ma and Mb, from the private keys and exchange them. The client combines Ra and Mb, and the server combines Ma with Rb to generate identical session keys, K.

After the key exchange is complete, a key verification phase follows. Each side generates a random number (nonce), encrypts it with the session key, and sends it to the other side. Each side takes the other's verifier, decrypts to get the nonce, modifies the nonce in a way that is known to both parties, encrypts it with the session key, and sends it back. The originator verifies that the nonce was modified as expected. Incrementing the nonce is a simple and effective way of modifying the verifier.

Table 1-15 lists the values used to calculate the content of messages exchanged between the client and server when the UAM is DHX.

Table 1-15 Variables and notation used by the DHX UAM

Value	Meaning
password	User password padded with nulls to 64 bytes.
username	Pascal string (pstring), padded to an even byte length.
ServerSig	Obtained from the server by sending `FPGetSrvrInfo` command.Due to a problem in the initial implementation, the ServerSig must be set to 16 bytes of 0x00 in message #2.
AFP Vers	Pascal string (pstring) denoting the version of the AFP protocol used for the session.
ID	A two-byte number used by the server to keep track of the login/change password request. The server may send any two-byte number, the client passes it back unchanged.
x^y	Raise x to the yth power.
nonce	A random number.
nonce + 1	Add one to the nonce.
Ra	32 byte (256 bits) random number used internally by the client.
Rb	32-byte (256 bit) random number used internally by the server.
p	16 byte (128 bit) prime number satisfying the property that (p - 1)/2 is also prime (called a Sophie Germain prime).
g	A small number that is primitive mod p.
Ma	g^Ra mod p (sent by the client to the server in the first message); 16 bytes.
Mb	g^Rb mod p (sent by the server to the client in the second message); 16 bytes.
K	Key = Mb^Ra mod p = Ma^Rb mod p.

Value	Meaning
(dataBytes, IV)K	Encrypt dataBytes using CAST 128 CBC using initialization vector (IV).
C2SIV	Client-to-server initialization vector.
S2CIV	Server-to-client initializaion vector.

For DHX, the constants p and g are defined as follows (MSB first):

```
UInt8 p = { 0xBA, 0x28, 0x73, 0xDF, 0xB0, 0x60, 0x57, 0xD4, 0x3F,  0x20, 0x24,
 0x74,  0x4C, 0xEE, 0xE7, 0x5B  };
UInt8 g = { 0x07 };
```

For DHX, the client-to-server (C2SIV) and server-to-client S2CIV) initialization vectors are defined as follows:

```
UInt8 C2SIV[] = { 0x4c, 0x57, 0x61, 0x6c, 0x6c, 0x61, 0x63, 0x65  };
Uint8 S2CIV[] = { 0x43, 0x4a, 0x61, 0x6c, 0x62, 0x65, 0x72, 0x74  };
```

Note: Numbers are encoded in network byte order, most significant byte (MSB) first.

Logging in Using DHX

The login sequence when using the DHX UAM consists of an exchange of the four messages shown in Table 1-16. In Table 1-16, the pipe symbol (|) is used to separate the elements that make up the message.

Table 1-16 Login sequence using DHX

Message	Sender/Receiver	Content
1	Client to server	| `FPLogin` (2 bytes) | AFP Vers | `'DHCAST128'` | username (padded) | Ma |
2	Server to client	| ID | Mb | (nonce, ServerSig, S2CIV)K | and a result code
3	Client to server	| `FPLoginCont` (2 bytes) | ID | (nonce + 1, password, C2SIV)K |
4	Server to client	A result code of `kFPNoErr` if authentication was successful

In response to Message 1, the server may return the following result codes (but it may delay sending some of these result codes until Message 4):

- `kFPBadUAM` — the server doesn't support the DHX UAM.
- `kFPBadVersNum` — the server doesn't support the requested AFP version.
- `kFPParamErr` — the user name is not valid.
- `kFPMiscErr` — the session is already authenticated.
- `kFPServerGoingDown` — the server is shutting down.
- `kFPUserAlreadyLoggedOnErr` — the server allows only one active session per user.
- `kFPAuthContinue` — the server is prepared to continue to login process.

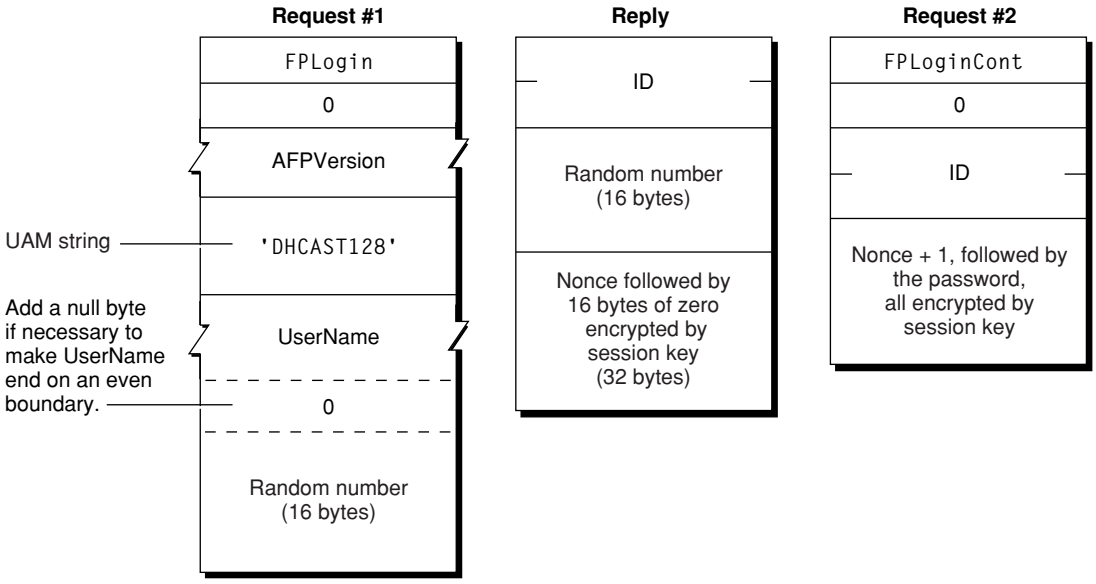
The server may delay sending some of the above result codes until the fourth message or may report a `kFPUserNotAuth` result as `kFPParamErr` to limit the amount of information disclosed to the client.

In response to Message 3, the server may return any of the following result codes:

- `kFPNoErr` — authentication was successful; the server decrypted the nonce/password and verified that the nonce was incremented properly and the password sent by the client matches the password on the server.
- `kFPUserNotAuth` — the password is incorrect.
- `kFPParamErr` — authentication failed and the server prefers not to indicate whether the user name or the password is invalid.
- `kFPPwdExpiredErr` — the user's password has expired.
- `kFPPwdNeedsChangeErr` — the user's password needs to be changed.

Figure 1-8 (page 42) shows the request and reply blocks for `FPLogin` when using the DHX UAM.

Figure 1-8 Request and reply blocks when using DHX with `FPLogin`



Changing Passwords Using DHX

There is no equivalent to `FPLoginCont` when changing a password, so the client has send the `FPChangePassword` command at least twice and use the ID to keep track of the state of the password-changing process. The ID first appears in Message 1 and is set to 2 bytes of 0x00. The server sends a non-zero value for ID in Message 2, and the client must copy it from Message 2 into Message 3. The key used to encrypt the old and new passwords is created in the same way as the key when logging in. The values of p and g are the same values that are used when logging in.

When using the DHX UAM, the password changing sequence consists of an exchange of at least four messages shown in Table 1-17. In Table 1-17, the pipe symbol (|) is used to separate the elements that make up the message.

Table 1-17 Password-changing sequence using DHX

Message	Sender/Receiver	Content
1	Client to server	| `FPChangePassword` (2 bytes) | '`DHCAST128`' | Username (padded) | ID (0x00 0x00) | Ma |
2	Server to client	| ID | Mb | (nonce, ServerSig, S2CIV)K | and a result code
3	Client to server	| `FPChangePassword` (2 bytes) | '`DHCAST128`' | Username (padded) | ID | (nonce + 1, newPassword, oldPassword, C2SIV)K |
4	Server to client	A result code of `kFPNoErr` if the password was changed

In response to Message 1, the server may return any of the following result codes (or may wait until it receives the second `FPChangePassword` command to return the first three result codes):

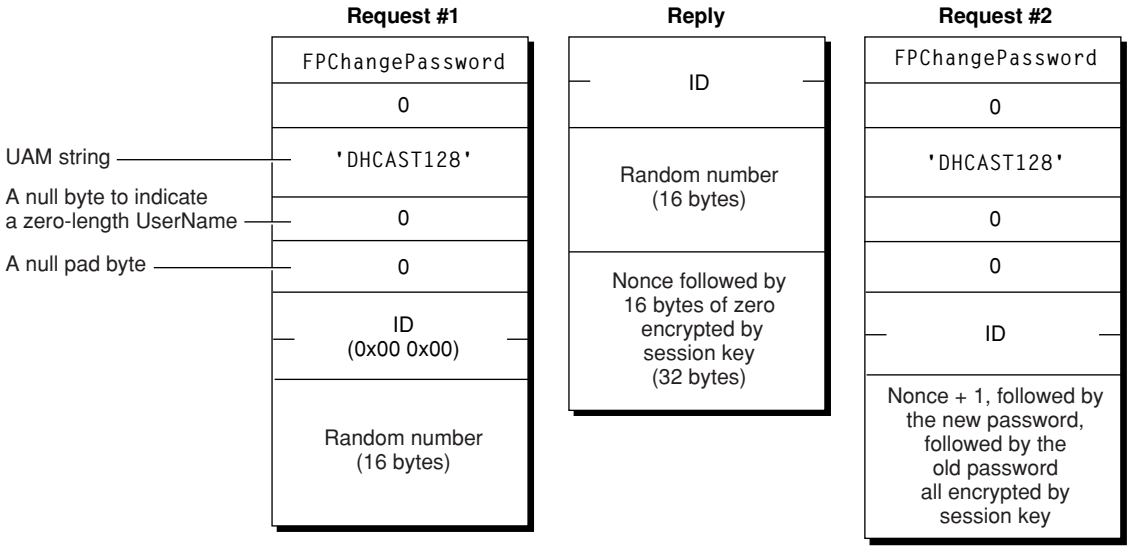
- `kFPBadUAM` —the server doesn't support DHX for changing passwords.
- `kFPParamErr` — the user name is not valid.
- `kFPServerGoingDown` — the server is shutting down.
- `kFPAuthContinue` — the server is prepared to continue the password-changing process.

In response to Message 3, the server may return any of the following result codes:

- `kFPNoErr` — the password was changed.
- `kFPUserNotAuth` — the old password is incorrect.
- `kFPParamErr` — to limit the amount of information released to the client.
- `kFPPwdPolicyErr` — the new password does not conform to the server's password policy.
- `kFPPwdSameErr` — the new password is the same as the old password.
- `kFPPwdTooShortErr` — the new password is too short.

Figure 1-9 (page 44) shows the request and reply blocks for calling `FPChangePassword` with the DHX UAM.

Figure 1-9 Request and reply blocks when using DHX with `FPChangePassword`



Diffie-Hellman Key Exchange 2

Diffie-Hellman Key Exchange 2 (DHX2) is an implementation of the Diffie-Hellman Key Agreement Protocol using the SSLeay/OpenSSL implementation of CAST 128 in CBC mode. The UAM protocol name for DHX2 is `'DHX2'`.

DHX2 differs from DHX in that DHX2 uses variable-sized prime (p) and generator (g) values, which allows servers to choose an appropriate level of security. The minimum size of the prime is increased to 512 bits to improve resistance to numerical methods of attack. In addition, unlike DHX, DHX2 does not use the server signature(ServerSig) in Message 2.

DHX2 is strong against packet sniffing attacks but vulnerable to active attacks such "Man in the Middle." There is no way for the client to verify that the server knows the password, so the server could easily be spoofed. There is some weakness in using fixed initialization vectors, p and g, which is alleviated by putting the random nonces first in the encrypted portions of the messages. DHX2 is useful when the server requires passwords in cleartext.

As with DHX, in DHX2 the client and server each generate a random number, Ra and Rb respectively, which serve as "private keys" for the session. The client and server use modulus exponentiation to derive "public keys", Ma and Mb, from the private keys and exchange them. The client combines Ra and Mb, and the server combines Ma with Rb to generate identical session keys, K.

After the key exchange is complete, a key verification phase follows. Each side generates a random number (nonce), encrypts it with the session key, and sends it to the other side. Each side takes the other's verifier, decrypts to get the nonce, modifies the nonce in a way that is known to both parties, encrypts it with the session key, and sends it back. The originator verifies that the nonce was modified as expected. Incrementing the nonce is a simple and effective way of modifying the verifier.

Table 1-18 lists the values used to calculate the content of messages exchanged between the client and server when the UAM is DHX2.

Table 1-18 Variables used by the DHX2 UAM

Value	Meaning
password	User password padded with nulls to 256 bytes.
username	Pascal string (pstring), padded to an even byte length.
AFP Vers	Pascal string (pstring) denoting the version of the AFP protocol used for the session.
ID	A two-byte number used by the server to keep track of the login/change password request. The server may send any two-byte number, the client passes it back unchanged.
ID + 1	The ID incremented by one.
clientNonce	A 16-byte random number used in the key verification portion of the exchange.
serverNonce	A 16-byte random number used in the key verification portion of the exchange.
clientNonce + 1	The clientNonce incremented by one.
MD5(data)	Take the MD5 hash of the data, which results in a 16-byte (128 bit) value.
p	A variable length prime number (at minimum 512 bits in size) satisfying the property that (p - 1)/2 is also a prime(called a Sophie Germain prime) sent by the server to the client. (Two byte length followed by data.)
g	A small number that is primitive mod p sent by the server to the client. (Four bytes.)
x^y	Raise x to the yth power.
Ra	An x bit random number used internally by the client.
Rb	An x bit random number used internally by the server.
Ma	g^Ra mod p (sent by the client to the server); the same number of bytes as p, padded with nulls at the MSB end.
Mb	g^Rb mod p (sent by the server to the client); the same number of bytes as p, padded with nulls at the MSB end.
x	The size of p in bits.
len	The size of p & Ma & Mb in bytes; a two-byte value.
K	Key = MD5(Mb^Ra mod p) = MD5(Ma^Rb mod p)
(dataBytes, IV)K	Encrypt dataBytes using CAST 128 CBC using initialization vector (IV)
C2SIV	Client-to-server initialization vector.
S2CIV	Server-to-client initialization vector.

For DHX2, the client-to-server (C2SIV) and server-to-client (S2CIV) initialization vectors are defined as follows:

```
UInt8 C2SIV[] = { 0x4c, 0x57, 0x61, 0x6c, 0x6c, 0x61, 0x63, 0x65  };
```

```
Uint8 S2CIV[] = { 0x43, 0x4a, 0x61, 0x6c, 0x62, 0x65, 0x72, 0x74  };
```

Note: Numbers are encoded in network byte order; most significant byte (MSB) first. The constants C2SIV and S2CIV have the same definitions in DHX and DHX2.

Logging In Using DHX2

When using the DHX2 UAM, the login sequence consists of an exchange of the six messages shown in Table 1-19. In Table 1-19, the pipe symbol (|) is used to separate the elements that make up the message.

Table 1-19 Login sequence using DHX2

Message	Sender/Receiver	Content
1	Client to server	| `FPLogin` (2 bytes) | AFP Vers | `'DHX2'` | Username (padded) |
2	Server to client	| ID | g | len | p | Mb | and a result code
3	Client to server	| `FPLoginCont` (2 bytes) | ID | Ma | (client nonce, C2SIV)K |
4	Server to client	| ID + 1 | (clientNonce + 1, serverNonce, S2CIV)K | and a result code
5	Client to server	| `FPLoginCont` (2 bytes) | ID + 1 | (serverNonce+1, password, C2SIV)K |
6	Server to client	A result code of `kFPNoErr` if authentication was successful

Some older implementations of Apple's AFP client add ten extra bytes to the end of the `FPLoginCont` packet (message five in Table 1-19). Similarly, two extra bytes are added to the end of message two in Table 1-19. Servers should ignore the presence and contents of these bytes.

In response to Message 1, the server may return the following result codes (but it may delay sending some of these result codes until Message 6):

- `kFPBadUAM` — the server doesn't support the DHX2 UAM.
- `kFPBadVersNum` — the server doesn't support the requested AFP version.
- `kFPParamErr` — the user name is not valid.
- `kFPMiscErr` — the session is already authenticated.
- `kFPServerGoingDown` — the server is shutting down.
- `kFPUserAlreadyLoggedOnErr` — the server allows only one active session per user.
- `kFPAuthContinue` — the server is prepared to continue to login process.

The server may delay sending some of the above result codes until the sixth message or may report a `kFPUserNotAuth` result as `kFPParamErr` to limit the amount of information disclosed to the client.

In response to the `FPLoginCont` command, the server may return any of the following result codes:

- `kFPNoErr` — authentication was successful; the server decrypted the nonce/password and verified that the nonce was incremented properly and the password sent by the client matches the password on the server

- kFPUserNotAuth — the password is incorrect
- kFPParamErr — authentication failed and the server prefers not to indicate whether the user name or the password is invalid
- kFPPwdExpiredErr — the user's password has expired
- kFPPwdNeedsChangeErr — the user's password needs to be changed

Changing Passwords Using DHX2

There is no equivalent to FPLoginCont when changing a password, so the client has send the FPChangePassword command at least twice and use the ID to keep track of the state of the password-changing process. The ID first appears in Message 1 and is set to 2 bytes of 0x00. The server sends a non-zero value for ID in Message 2, and the client must copy it from Message 2 into Message 3 as well as from Message 4 into Message 5. The key used to encrypt the old and new passwords is created in the same way as the key when logging in. The values of p and g are the same values that are used when logging in.

When using the DHX2 UAM, the password changing sequence consists of an exchange of at least six messages shown in Table 1-20. In Table 1-20, the pipe symbol (|) is used to separate the elements that make up the message.

Table 1-20 Password-changing sequence using DHX2

Message	Sender/Receiver	Content
1	Client to server	| FPChangePassword (2 bytes) | 'DHX2' | Username (padded) | ID (0x00 0x00) |
2	Server to client	| ID | g | len | p | Mb | and a result code
3	Client to server	| FPChangePassword (2 bytes) | 'DHX2' | Username (padded) | ID | Ma | (clientNonce, C2SIV)K |
4	Server to client	| ID+1 | (clientNonce+1, serverNonce, S2CIV)K | and a result code
5	Client to server	| FPChangePassword (2 bytes) | 'DHX2' | Username (padded) | ID+1 | (serverNonce+1, newPassword, oldPassword, C2SIV)K |
6	Server to client	A result code of kFPNoErr if the password was changed

In response to Message 1, the server may return kFPAuthContinue or any of the following result codes:

- kFPBadUAM —the server doesn't support DHX2 for changing passwords.
- kFPParamErr — the user name is not valid.
- kFPServerGoingDown — the server is shutting down.

In response to Message 3, the server may return kFPAuthContinue or any of the following result codes:

- kFPUserNotAuth — the old password is incorrect.
- kFPPwdPolicyErr — the new password does not conform to the server's password policy.
- kFPPwdSameErr — the new password is the same as the old password.

- kFPPwdTooShortErr — the new password is too short.

Kerberos

The AFP client learns whether a server supports the Kerberos UAM by examining the kSupportsDirServices bit in the Flags parameter returned by the FPGetSrvrInfo command. If that bit is set, a server that supports Kerberos UAM places its principal name in the DirectoryNames parameter returned by FPGetSrvrInfo.

The AFP client uses the principal name to determine if the server supports Kerberos v4 or v5.

Note: Mac OS X AFP servers only support Kerberos V5 authentication.

Then the client tries to get a service ticket from the server. If it cannot get a ticket, the client must use some other authentication method. If the client gets a service ticket, it can call FPLoginExt, providing the following values in the request block:

- two-byte Flags parameter
- AFP Version string
- UAM string (Client Krb v2)
- kFPUTF8Name (defined as 3)
- length of the user name that follows
- UTF-8–encoded user name
- kFPUTF8Name (defined as 3)
- length of the realm in that follows
- UTF-8–encoded realm

The server replies with a result code of kFPAuthContinue. The reply block contains a two-byte ID value.

If the client is using Kerberos v4, it calls FPLoginCont, providing the following values in the request block:

- UTF-8–encoded user name
- pad byte if one is necessary to force user name to end on an even boundary
- length of the ticket that follows
- ticket, created by KClientGetTicketForService()

The user is authenticated if the server returns a result code of kFPNoErr and a reply block consisting of a two-byte length parameter and an authenticator.

If the client is using Kerberos v5, it calls FPLoginCont, providing the following values in the request block:

- ID returned by FPLoginExt
- UTF-8–encoded user name
- pad byte if one is necessary to force user name to end on an even boundary

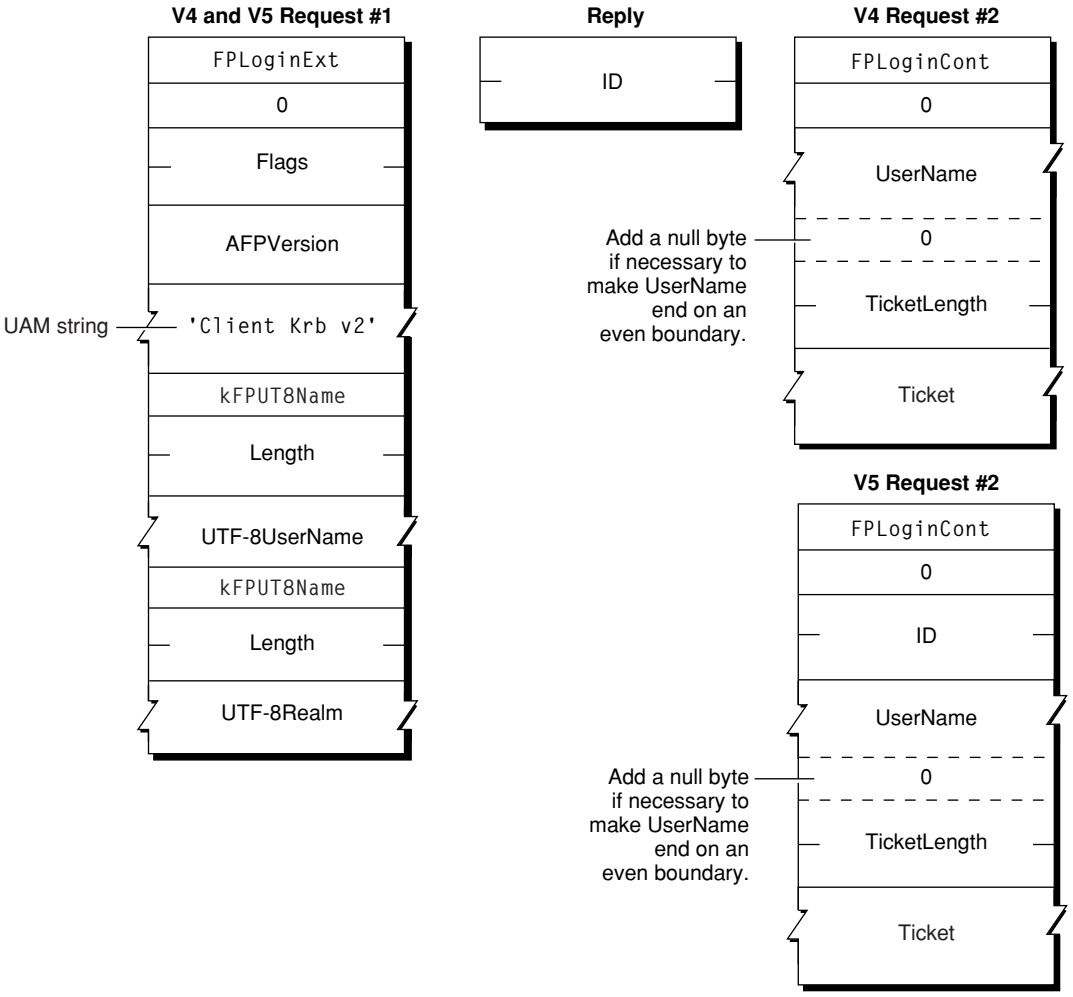
- length of the ticket that follows
- ticket, created by `gss_init_sec_context` with `GSS_C_MUTUAL_FLAG` and `GSS_C_REPLAY_FLAG` set and no channel bindings

The user is authenticated if the server returns a result code of `kFPNoErr` and a reply block consisting of a two-byte length parameter and an authenticator.

After the client receives the `FPLoginCont` reply packet, the client sends an `FPGetSessionToken` command with a type of `kGetKerberosSessionKey` (8) in order to get a random session key from the server. This session key is encrypted on the server using `gss_wrap()` and is decrypted on the client using `gss_unwrap()`. Note that the client may call `FPGetSessionToken` later on in order to get a disconnect token.

Figure 1-10 (page 49) shows the request and reply blocks for `FPLoginExt` and `FPLoginCont` when using the Kerberos UAM.

Figure 1-10 Request and reply blocks when using Kerberos with `FPLoginExt`



Reconnect

Unlike the other UAMs described in this section, which are used to log in to an AFP server, the Reconnect UAM is used only to reconnect to a server. The Reconnect UAM can be used when the original connection was made using a UAM that provides a session key, such as DHX, DHX2, and Kerberos. The UAM protocol name for the Reconnect UAM is `'Recon1'`.

The goals of the Reconnect UAM are:

- Store in a token returned by the `FPGetSessionToken` command all of the information required to reconnect, even if the server has been rebooted.
- Use only a secure hash function and a symmetric encryption algorithm.
- Provide mutual authentication to prove that the server to which the client is reconnecting is the same server that was originally authenticated.
- Ensure that a compromised session key or seed value will not compromise the long term server key.

Table 1-21 lists the variables used to calculate values for the Reconnect UAM.

Table 1-21 Variables used by the Reconnect UAM

Value	Size in Bytes	Meaning
k1	16	Initial session key returned by the UAM that was used to log in; known to both the client and the server at the time of reconnect.
ks	16	Long term server key.
s	8	Lamports hash seed.
n	4	Number of times to run the hash function.
m	4	Maximum number of times to run the hash function (m >= n).
clientNonce	8	A random number selected by the client nonce.
serverNonce	8	A random number selected by the server.
t1	4	Initial timestamp.
t2	4	Timestamp used when reconnecting.
t3	4	Time interval between the server's clock and the client's clock.
exp	4	Credential's expiration time.
now	4	Current time as known by the server or by the client.
user/domain		Username information that uniquely identifies the user.
sessionInfo		Information that uniquely identifies a session.

Value	Size in Bytes	Meaning
(data)key		Data encrypted using a symmetric encryption algorithm using key. (CBC mode)
(data)H		Data hashed with a secure hash function.
(data)H(n)		Data hashed n times with a secure hash function.
(data)HMAC(key)		Data signed by a keyed HMAC algorithm.
revocation list		List of hash value and time-to-live pairs. Pairs stay in the list until the time-to-live value has passed.
cred		(s, m, exp, t3, user/domain)ks

Table 1-22 describes common methods of attack and the ways in which the Reconnect UAM is protected from these attacks.

Table 1-22 Attacks on the Reconnect UAM

Attack	Defense
Man in the Middle	If the original UAM used to connect to the server was resistant to Man in the Middle attacks, nonce checks in message *a*, which require knowledge of *s*, should keep out the Man in the Middle.
Replay	The timestamp in message *a*, protected by the HMAC, and the credential revocation list should prevent simple replay attacks. Even if the attacker succeeds in controlling the clock on the server and manages to force a server restart, the attacker cannot log in because s is not known, so the challenge/response step cannot be performed successfully.
Reflection	This type of attack is thwarted by the use of chained nonces, by having the user information in the credential, and by having each message be non-symmetrical.
Interleaving	This type of attack is thwarted by the use of chained nonces.
Chosen Text	The server's key is not used to encrypt any data that is obtained from the client.
Forced Delay	Timestamps, key expiration and the use of the revocation list should thwart this type of attack.

Getting a Credential

After the client successfully logs in and mounts a remote volume, it calls `FPGetSessionToken`, setting the `Type` parameter to `kRecon1Login` (5), and sending to the server its initial timestamp (t1) encrypted with the session key (k1):

(t1)k1

As a result of the login process, the server also knows the session key (k1) and the sessionInfo for this session. The server also has a long term session key (ks).

The server uses t1 to compute the clock skew and determine an appropriate expiration time for the credential it is about to create. The server then generates a credential by concatenating the Lamports hash seed (s), the maximum number of times to run the hash function (m), the expiration time, the user/domain, and encrypting the concatenation using its long term session key (ks):

cred = (s, m, exp, t3, user/domain)ks

The server also computes (cred)H and stores the result in its revocation list. The server then uses the session key (k1) to encrypt a concatenation of the credential (cred), the Lamports hash seed (s), the maximum number of times to run the hash function (m), the expiration time (exp), and sessionInfo, and sends the result to the client. The formula for this calculation is:

(cred, s, m, exp, sessionInfo)k1

The client uses the session key (k1) to decrypt the result, obtaining the encrypted credential, the Lamports hash seed, the maximum number of times to run the hash function, the encrypted credential's expiration time, and the sessionInfo. The client is responsible for storing this information so that it can use it later.

Table 1-23 summarizes the exchange between client and server when getting a credential.

Table 1-23 Getting a credential

Message	Sender/Receiver	Content
1	Client to server	| `FPGetSessionToken` (2 bytes) | `kRecon1Login` | IDLength | (t1)k1 | ID |
2	Server to client	| (cred, s, m, exp, sessionInfo)k1 |

Refreshing the Credential

Before the credential expires, the client calls `FPGetSessionToken` again, setting the `Type` parameter to `kRecon1RefreshToken` (7) and sending to the server the initial timestamp (t1) and the current credential encrypted with the session key (k1). The formula for this calculating this value is:

(t1, cred)k1

Both the client and the server compute k2 using the following formula:

k2 = (cred, s)H

The server decrypts the value sent by the client. If the credential is valid, the server creates a new credential encrypted with the long term session key and a new expiration time, stores (cred')H on the revocation list, and returns the encrypted credential to the client along with a new Lamports hash seed, a new maximum number of times to run the hash function, and the sessionInfo, all encrypted by k2. The formula for creating this value is:

(cred', s', m', exp', sessionInfo)k2

The client uses k2 to decrypt the reply, obtaining the new credential, the new Lamports hash seed, the new maximum number of times to run the hash function, the new expiration and the sessionInfo. Before this credential expires, the client refreshes it again.

Table 1-24 summarizes the exchange between client and server when refreshing a credential.

Table 1-24 Refreshing a credential

Message	Sender/Receiver	Content
1	Client to server	| `FPGetSessionToken` (2 bytes) | `kRecon1RefreshToken` | IDLength| (t1, cred)k1 | ID |
2	Server to client	| (cred', s', m', exp', sessionInfo)k2 |

Using the Credential to Reconnect

If the connection to the server goes down for any reason, the client has the current credential, the Lamports hash seed (s), and the maximum number of times to run the hash (m).

The client logs back in using the `FPLoginExt` command, specifying `Recon1ReconnectLogin` as the UAM, and sending the following information to the server:

(cred, (s)H(n), n, t2, (clientNonce)[(s)H(n-1])HMAC(s)

The server uses its long term session key (ks) to decrypt cred. If decryption fails, the server fails the login attempt. It also retrieves s, m, exp, t3, and user/domain.

If the decryption succeeds, the server computes (cred)H and looks it up in the revocation list. If found, the credential has expired, so the server fails the login attempt.

If (cred)H is not found in the revocation list, the server checks exp, m >= n, and HMAC(s) user/domain. If any are invalid, the server fails the login attempt.

The server then computes and compares (s)H(n)' and (s)H(n1). If they don't match, the server fails the login attempt.

The server then decrypts and hashes clientNonce, chooses serverNonce, adds (cred)H, t3+now to the revocation list, and sends the following value to the client:

(serverNonce, (clientNonce)H)[(s)H(n-1)]

The client decrypts the value, verifies (clientNonce)H, and hashes serverNonce. The client uses the `FPLoginCont` command to send the following value to the server:

((serverNonce)H[(s)H(n-1)]

The server decrypts the value and verifies (serverNonce)H. If they don't match, the server fails the login attempt. If they match, the server replies to the client with a result code of `kFPNoErr`. The client is now logged in. Both the server and the client make the following calculation:

k1' = (clientNonce, serverNonce)H

The client calls `FPGetSessionToken` using k1' as the session key to get a new credential. It also calls `FPDisconnectOldSession` to tell the server to disconnect the old session and transfer is resources to the new session.

Table 1-25 summarizes the exchange between client and server when reconnecting.

Table 1-25 Reconnecting using the Recon1 UAM

Message	Sender/Receiver	Content
1	Client to server	| `FPLoginExt` (2 bytes) | Flags | AFP version | '`Recon1`' | UserNameType | UserName | PathType | Pathname | (cred, (s)H(n), n, t2, (clientNonce)[(s)H(n-1])HMAC(s) |
2	Server to client	|(serverNonce, (clientNonce)H)[(s)H(n-1)] | and a result code
3	Client to server	| `FPLoginCont` (2 bytes) | ID | ((serverNonce)H[(s)H(n-1)] |
4	Server to client	`kFPNoErr` or another result code indicating log in failure
5	Client to server if result is `kFPNoErr`	| `FPGetSessionToken` (2 bytes) | `kRecon1ReconnectLogin` | IDLength| (t1, cred)k1 | ID |

Volume Passwords

AFP provides an optional second-level of access control through volume passwords. A server can associate a fixed-length 8-character password with each volume it makes visible to AFP clients.

The AFP client can issue an `FPGetSrvrParms` command to the server to discover the names of each volume and to get an indication of whether each of them is password-protected.

To send AFP commands that refer to a server volume, the AFP client uses a volume identifier called the Volume ID. The AFP client obtains this ID by sending an `FPOpenVol` command to the server. This command contains the name of the volume as one of its parameters. If a password is associated with the volume, the command must also include the password as another parameter.

Volume passwords constitute a simple protection for servers that do not need to implement the directory access controls described in the next section. However, volume passwords are not as secure as directory access controls.

Directory Access Controls

Directory access controls provide the greatest degree of network security in AFP by access privileges to users. Once the user has logged in, access privileges allow users varying degrees of freedom for performing actions within the directory structure.

AFP defines three directory access privileges: search, read, and write:

- A user with *search* access to a directory can list the parameters of directories contained within the directory.
- A user with *read* access to a directory can list the parameters of files contained within the directory in addition to being able to read the contents of a file.
- A user with *write* access to a directory can modify the contents of a directory including the parameters of files and directories contained within the directory. Write access allows the user to ad and delete directories and files as well as modify the data contained within a file.

Each directory on a server volume has an owner and a group affiliation. Initially, the owner is the user who created the directory, although ownership of a directory may be transferred to another user. Only the owner of a directory can change its access privileges. The server uses a name of up to 31 characters and a four-byte ID number to represent owners of directories. Owner name and Owner ID are synonymous with User name and User ID.

The group affiliation is used to assign a different set of access privileges for the directory to a group of users. For each group, the server maintains a name of up to 31 characters, a four-byte ID number and a list of users belonging to the group. Assigning group access privileges to a directory gives those privileges to that set of users.

Each user may belong to any number of groups or to no group. One of the user's group affiliations may be designated as the user's primary group. This group will be assigned initially to each new directory created by the user. The directory's group affiliation may be removed or changed later by the owner of the directory.

The term *Everyone* is used to indicate every user that is able to log in to the server. A directory may be assigned certain access privileges for Everyone that would be granted to a user who is neither the directory's owner nor a member of the group with which the directory is affiliated.

With each directory, the file server stores three access privileges bytes, which correspond to the owner of the directory, its group affiliation, and Everyone. Each of these bytes is a bitmap that encodes the access privileges (search, read, and write) that correspond to each category. The most significant bits of each access privileges byte must be zero.

To perform directory access control, AFP associates the five parameters shown in Table 1-26 with each directory.

Table 1-26 Directory access control parameters

Parameter	Size
Owner ID	Four bytes
Group ID	Four bytes
Owner access privileges	One byte
Group access privileges	One byte
Everyone access privileges	One byte

The Owner ID is the same as the owner's User ID. The Group ID is the ID number of the group with which the directory is affiliated, or zero. The file server maintains a one-to-one mapping between the Owner ID and the user name and between the Group ID and the group name. As a result, each name is associated with a unique ID. AFP includes commands that allow users to map IDs to names and names to IDs. Assignment of User IDs, Group IDs, and primary groups is an administrative function and is outside the scope of this protocol.

A Group ID of zero means that the directory has no group affiliation. The groups access privileges are ignored.

When a user logs on to a server, identifiers are retrieved from a user database maintained on the server. These identifiers include the User ID (a four-byte number unique among all server users) and one or more four-byte Group IDs, which indicate the user's group memberships. The exact number of group memberships is implementation-dependent. One of these Group IDs may represent the user's primary group.

The server must be able to derive what access privileges a particular user has to a certain directory. The user access privileges (UARights) contain a summary of the privileges, regardless of the category (Owner, Group, Everyone) from which they were obtained. In addition, the user access privileges contain a flag indicating whether the user owns the directory.

The server uses the following algorithm to extract user access privileges. The OR in this algorithm indicates inclusive OR operations.

```
UARights := Everyone's access rights;
clear UARights owner flag
If (Owner ID = 0) then
    set UARights own flag
If (User ID = Owner ID) then
    UARights := UARights OR owner's access privileges;
    set UARights owner flag
If (any of user's Group IDs = directory's Group ID) then
    UARights := UARights OR directory's group access privileges
```

An Owner ID of zero means that the directory is not owned or is owned by another user. The owner bit of the access privileges byte is always set for such a directory.

The access privileges required by the user to perform most file management functions are explained in the following paragraphs according to the symbols listed in Table 1-27.

Table 1-27 Access privilege notation

Symbol	Meaning
SA	Search access to all ancestors down to, but not including the parent directory
WA	Search or write access to all ancestors down to, but not including, the parent directory
SP	Search access to the parent directory
RP	Read access to the parent directory
WP	Write access to the parent directory

Almost all operations require *SA*. To perform any action within a given directory, the user must have permission to search every directory in the path from the root to the parent's parent directory. Access to files and directories within the parent directory is then determined by *SP*, *RP*, and *WP*.

Specific file management functions and the access privileges needed to perform them are listed in Table 1-28.

Table 1-28 File management functions and required privileges

Function	Required access privileges
Create a file or a directory	The user must have *WA* plus *WP*. A hard create (delete first of the file exists) requires the same privileges as deleting a file.

Function	Required access privileges
Enumerate a directory	To enumerate a directory is to list in numerical order the offspring of the directory and selected parameters of those offspring. The user must have search access to all directories down to but not necessarily including the directory being enumerated (*SA*). In addition, to view its directory offspring, the user must have search access to the directory being enumerated (*SP*). To view its file offspring, search access to the directory is not required, but the user must have read access to the directory (*RP*).
Delete a file	The user must have *SA*, *RP*, and *WP*. A file can be deleted only it if is not open at that time.
Delete a directory	The user must have *WA* plus *WP*. A hard create (delete first of the file exists) requires the same privileges as deleting a file.
Rename a file	To enumerate a directory is to list in numerical order the offspring of the directory and selected parameters of those offspring. The user must have search access to all directories down to but not necessarily including the directory being enumerated (*SA*). In addition, to view its directory offspring, the user must have search access to the directory being enumerated (*SP*). To view its file offspring, search access to the directory is not required, but the user must have read access to the directory (*RP*).
Rename a directory	The user must have *SA*, *RP*, and *WP*. A file can be deleted only it if is not open at that time.
Rename a file	The user must have *SA*, *SP*, and *WP*. A directory can be deleted only if it is empty
Rename a directory	The user must have *SA*, *RP*, and *WP*.
Read directory parameters	The user must have SA and SP.
Open a file for reading	A file's fork must be opened in read mode before its contents can be read. To open a file in read mode, the user must have *SA* and *RP*. Read mode and other access modes are described in the next section.
Open a file for writing	A file's fork must be opened in write mode in order to write to it. To open an empty fork for writing, the user must have *WA* and *WP*. (The empty fork must belong to a file that has both forks of zero length. To open an existing fork (when either fork is not empty) for writing, *SA*, *RP*, and *WP* are required.
Write file parameters	For an empty file (where both forks are zero length), the user must have *WA* plus *WP*. For a non-empty file (where one or both forks are not zero length), the user must have *SA*, *RP*, and *WP*.
Write directory parameters	For directories that contain offspring, the user must *SA*, *SP*, and *WP*. For directories that are empty, the user must have *WA* and *WP*.

Function	Required access privileges
Move a directory or a file	Through AFP, a directory or a file can be moved from its parent directory to a destination parent directory on the same volume. To move a directory, the use must have *SA* and *SP* to the source parent directory, *WA* to the destination parent directory, plus *WA* to both the source and the destination parent directories. To move a file, the user needs *SA* plus *RP* to the source parent directory, plus *WP* to both the source and the destination parent directories.
Modify a directory's privileges	A directory's Owner ID, Group ID, and the three access privileges bytes can be modified only if the user is the directory's owner and then only if the user has *WA* plus *WP* or *SP* access to the parent directory.
Copy a file (`FPCopyFile`)	To copy a file, on a single volume or across volumes managed by the server, the user must have *SA* plus *RP* access to the source parent and *WA* plus *WP* to the destination parent directory.

Inherited Access Privileges

AFP Version 2.1 and later supports inherited access privileges through the directory's Blank Access Privileges bit in the Directory bitmap. When the Blank Access Privileges bit is set for a directory, its other access privilege bits are ignored and the access privilege bits of the directory's parent apply to the directory, including the parent's group affiliation.

The Blank Access Privileges bit cannot be set for a directory that is a share point. Likewise, the Blank Access Privileges bit cannot be set for a volume root directory (Directory ID = 2) of a shared volume because it is always a share point for the administrator/owner.

Important: Inherited access privileges are useful because they cause access privileges to behave as users expect them to: When a directory with the Blank Access Privileges bit set is moved within the directory hierarchy, it always reflects the access privileges of the directory containing it. When the Blank Access Privileges bit is cleared, its current access privileges "stick" to that directory and remain unchanged no matter where the directory is moved. Therefore, although implementing inherited access privileges is optional, it is highly recommended that you include this feature in your AFP implementation as it has subtle human interface repercussions.

File Sharing Modes

AFP controls user access to shared files in two ways. The first, described in the previous section, provides security by controlling user access to specific directories. The second, described in this section, preserves data integrity by controlling a user's access to a file while it is being used by another user.

To control simultaneous file access, the file server must enforce synchronization rules. These rules prevent applications from damaging each other's files by modifying the same version simultaneously. These rules also prevent users from obtaining access to information while it is being changed.

Synchronization rules are built from the mode in which a first user and subsequent users open a file. AFP provides two classes of modes: access modes and deny modes.

Access and Deny Modes

Most file systems use a set of permissions to regulate the opening of files. This set includes permission to modify the contents of a file (read-write) and permission to see the file's contents (read only). In a stand-alone system, these two file-access modes are sufficient.

In the shared environment of a file server, this set of permissions, or access modes, is expanded. In addition to the expanded set of access modes, a set of restrictions is provided by deny modes.

A user application can specify an access mode and a deny mode when it opens a file on the file server. AFP supports the access modes: read, write, read-write, and none. None access allows no further access to the fork, except to close it, and may be useful in implementing synchronization. In addition to one of these access modes, the user indicates a deny mode to the server to specify which rights should be denied to others trying to open the fork while the first user has it open. Users that subsequently try to open that fork can be denied read, write, read-write, or none access.

A user sending an `FPOpenFork` command can be denied file access for the following reasons:

- The user does not possess the rights (as owner, group, or Everyone) to open the file with the requested access mode. A result code of `kFPAccessDenied` is returned.
- The fork is already open with a deny mode that prohibits the second user's requested access. For example, the first user opened the fork with a deny mode of DenyWrite, and the second user tries to open the for in the write mode. A `kFPDenyConflict` result code is returned to the second user.
- The fork is already open with an access mode that conflicts with the second user's requested deny mode. For example, the first user opened the fork for Write access and a deny mode of DenyNone. The second user tries to open the fork with a deny mode indicating DenyWrite. This request is not granted because the fork is already open for Write access. A `kFPDenyConflict` result code is returned to the second user.

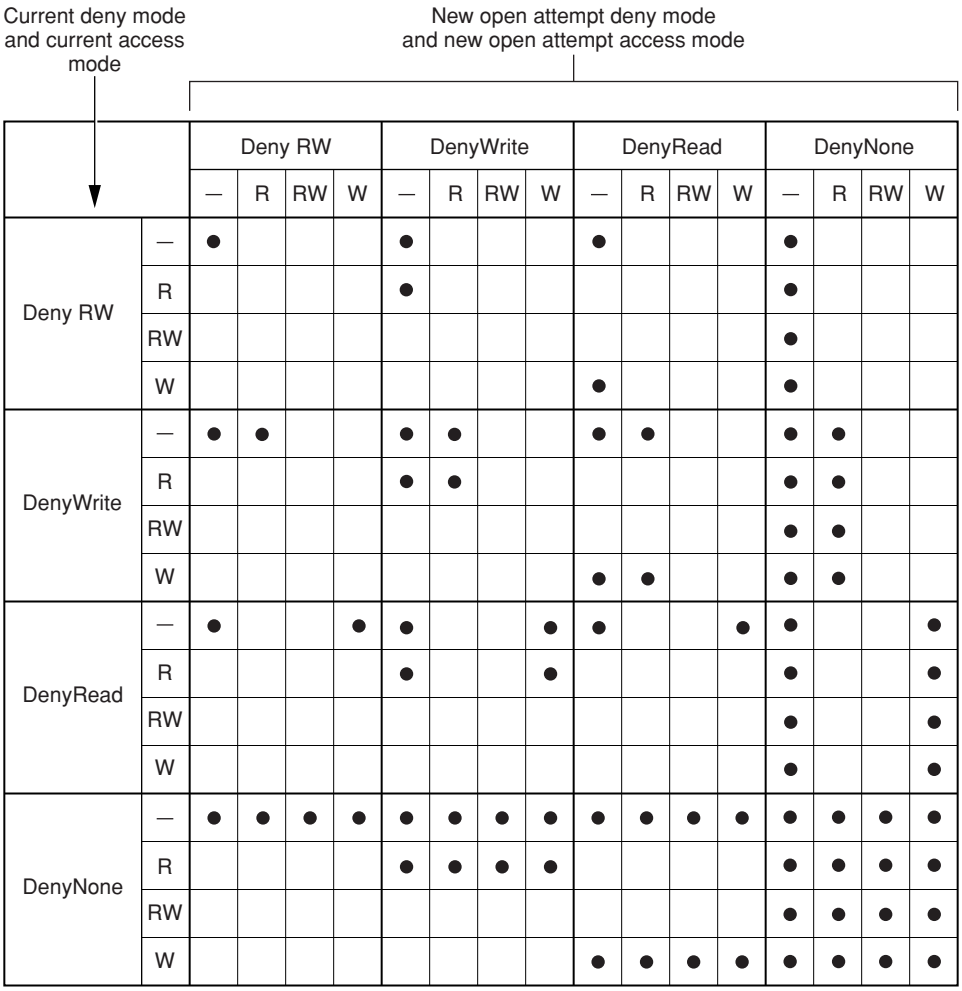
Deny modes are cumulative in that each successful opening of a fork combines its deny mode with previous deny modes. Therefore, if the first user opening a file specifies a deny mode of DenyRead, and the second user specifies DenyWrite, the fork's current deny mode is DenyRead-Write. DenyNone and DenyRead combine to form a current deny mode of DenyRead.

Similarly, access modes are cumulative. If the first user opening a file has Read access and the second has Write access, the current access mode is Read-Write.

Synchronization Rules

Synchronization rules, as previously discussed, allow or deny simultaneous access to a file fork. They are based on the current deny mode and current access mode of the fork and on the new deny and access modes being requested in a new `FPOpenFork` command. Synchronization rules are summarized in Figure 1-11 (page 60). A dot indicates that a new open command has succeeded; otherwise, it has failed.

Figure 1-11 Synchronization rules



		Deny RW				DenyWrite				DenyRead				DenyNone			
		—	R	RW	W	—	R	RW	W	—	R	RW	W	—	R	RW	W
Deny RW	—	●				●				●				●			
	R					●								●			
	RW													●			
	W									●				●			
DenyWrite	—	●	●			●	●			●	●			●	●		
	R					●	●							●	●		
	RW													●	●		
	W									●	●			●	●		
DenyRead	—	●			●	●			●	●			●	●			●
	R					●			●					●			●
	RW													●			●
	W													●			●
DenyNone	—	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	R					●	●	●	●					●	●	●	●
	RW													●	●	●	●
	W									●	●	●	●	●	●	●	●

Access Control Lists

This version of AFP includes support for access control lists (ACLs), which can be enabled on a per volume basis. The inheritance and multiple ownership capabilities of ACLs improve workflow in environments where files and directories require different owners at various phases of work. When ACLs are enabled, computers running Mac OS X are full-fledged peers on Windows networks, which promotes the adoption of XServe as an NT replacement.

Note: ACLs also eliminate the 16 group membership limit.

When ACLs are enabled for a volume, each file and directory has a security descriptor. A security descriptor includes:

- a set of flags, including flags for the discretionary and system ACL (described below), each indicating whether the ACL inherits the settings of the ACLs above it.

- an owner SID, similar to the UNIX file owner
- a primary group SID, similar to the UNIX file group owner
- a discretionary access control list (DACL) that specifies which permissions are granted or denied to which users or groups
- a system control list (SACL) that determines which file accesses by which users cause the access to be logged in a security log

Access control entries (ACEs) in the DACLs and SACLs contain the following information:

- a SID, specifying the user or group to which the ACE applies
- a set of flags, including inheritance flags (listed in Table 1-29) and a flag that applies only to SACL entries

Table 1-29 Inheritance flags

Flag	Description
`INHERITED_ACE`	Indicates whether the entry was inherited from a parent ACL.
`INHERIT_ONLY_ACE`	Indicates whether the entry exists only to be propagated to children and is used only when child objects are created or when that entry is changed. If set, the entry is not checked when access or audit checks are done.
`CONTAINER_INHERIT_ACE`	Indicates whether the entry should be inherited by directories below the object to which the entry applies.
`OBJECT_INHERIT_ACE`	Indicates whether the entry should be inherited by files below the object to which the entry applies.
`NO_PROPAGATE_-INHERIT_ACE`	Indicates, when the entry is copied to a child, whether the settings of the `CONTAINER_INHERIT_ACE` and `OBJECT_INHERIT_ACE` flags should be cleared, so that changes to the entry don't propagate to grandchildren or objects below grandchildren.

- a set of access right bits (listed in Table 1-30); for DACL entries, the access rights bits allow or deny permission; for SACL entries, the access rights bits specifying the types of accesses to be audited

Table 1-30 Access rights bits

Access right bit	Description
Generic access rights	The four high-order bits of the access mask format used by securable objects. Each securable object maps these bits to a set of its standard and object-specific access rights. For example, a file object maps the `GENERIC_READ` bit to the `READ_CONTROL` and `SYNCHRONIZE` standard access rights and to the `FILE_READ_DATA`, `FILE_READ_EA` and `FILE_READ_ATTRIBUTES` object-specific access rights.
`GENERIC_ALL`	Read, w rite, and execute access

Access right bit	Description
GENERIC_EXECUTE	Execute access, including FILE_READ_ATTRIBUTES, FILE_EXECUTE, and SYNCHRONIZE, all of which are described below.
GENERIC_READ	Read access, including FILE_READ_ATTRIBUTES, FILE_READ_DATA, READ_CONTROL, and SYNCHRONIZE, all of which are described below
GENERIC_WRITE	Write access, including FILE_APPEND_DATA, FILE_WRITE_ATTRIBUTES, FILE_WRITE_DATA, FILE_WRITE_EA, WRITE_CONTROL, and SYNCHRONIZE, all of which are described below.
Standard access rights	A set of standard access rights that correspond to operations common to most types of securable object. Constants defined for the standard access rights bits include the following:
DELETE	Right to delete the object
READ_CONTROL	Right to read the object's security descriptor, but not including information in the SACL
SYNCHRONIZE	Right for a thread to block until the object is in the "signaled state"
WRITE_DAC	Right to modify the DACL in the object's security descriptor
WRITE_OWNER	Right to change the object's owner in the object's security descriptor
File and directory access rights	
FILE_ADD_FILE	Right to create a file in a directory
FILE_ADD_SUBDIRECTORY	Right to create a directory in a directory
FILE_APPEND_DATA	Right to create a directory in a directory (when set for a directory) or to append data to a file (when set for a file)
FILE_DELETE_CHILD	Right to delete a directory and all the files it contains
FILE_EXECUTE	Right to execute a program
FILE_LIST_DIRECTORY	Right to list the contents of a directory
FILE_READ_ATTRIBUTES	Right to read a file's DOS attributes, including hidden, read-only, system, and archive attributes.
FILE_READ_DATA	Right to read data from a file or pipe (when set for a file or pipe), or to list the contents of a directory (when set for a directory)
FILE_READ_EA	Right to read an object's extended attributes
FILE_TRAVERSE	Right to traverse a directory; equivalent to FILE_EXECUTE
FILE_WRITE_ATTRIBUTES	Right to write a file's attributes.

Access right bit	Description
FILE_WRITE_DATA	Right to write to a file (when set for a file) or create a file in a directory (when set for a directory); when applied to a directory, this bit is equivalent to FILE_ADD_FILE.
FILE_WRITE_EA	Right to write extended attributes

An ACL can have a mixture of explicitly set and inherited ACEs. When a file or directory is created, ACEs are copied to the new object in the following order:

1. Explicit ACL entries that deny an SID certain rights
2. Explicit ACL entries that grant an SID certain rights
3. Inherited ACL entries that deny an SID certain rights
4. Inherited ACL entries that grant an SID certain rights

Inherited entries are placed in order in which they are inherited. ACEs inherited from the parent come first, then entries inherited from the grandparent (that is, that the parent inherited and passed on), and so on. As ACEs are processed from first to last, explicit entries override entries inherited from further up the tree.

Inheritance occurs when the object is created and at the time an ACL for a directory is changed, and does not occur at the time that an object is moved into the directory tree. When a folder or file is moved within the volume, its ACL is also moved without change and without updating inherited permissions. Instead, the ACL is updated the next time its permissions are changed, which forces the parent to propagate its permissions.

ACEs in which the CONTAINER_INHERIT_ACE bit or the OBJECT_INHERIT_ACE bit is not set are not copied.

ACEs in which the CONTAINER_INHERIT_ACE bit is set are copied when a directory is created, but not when a file is created. The INHERIT_ONLY_ACE bit is cleared.

ACEs in which the OBJECT_INHERIT_ACE are copied when a file or a directory is created. If copied to a file, the INHERIT_ONLY_ACE bit is cleared. If copied to a directory, the INHERIT_ONLY_ACE bit is set. The intention is to allow directories to give one set of permissions to subdirectories and another set of permissions to files.

The INHERITED_ACE bit is set on all ACEs that are copied.

If the NO_PROPAGATE_INHERIT_ACE bit is set on the entry being copied, the CONTAINER_INHERIT_ACE and OBJECT_INHERIT_ACE bits are cleared in the copy.

When ACLs are enabled for a volume, they are mapped to effective owner, group, and everyone UNIX permissions.

When accessing remote volumes for which ACL is enabled, use the FPAccess command to determine whether the client has access to the file or directory, and use the FPGetACL command to get the ACLs for a file or directory, and the FPSetACL command to set the ACLs for a file or directory.

Desktop Database

For file server volumes, AFP provides an interface that replaces the Finder's direct use of the Desktop file. This interface is necessary because the Desktop file was designed for a standalone environment and could not be shared by multiple users. The AFP interface to the Desktop database replaces the Desktop file and can be used transparently for both local and remote volumes.

The Desktop database is used by the file server to hold information needed specifically by the Finder to build its unique user interface, in which icons are used to represent objects on a disk volume. To create certain parts of this interface, the Finder uses the Desktop database to perform three functions:

- to associate documents and applications with particular icons and store the icon bitmaps
- to locate the corresponding application when a user opens a document
- to hold text comments associated with files and directories

Macintosh applications usually contain an icon that is to be displayed for the application itself as well as other icons to be displayed for documents that the application creates. These icons are stored in the application's resource fork and in the Desktop database. The Desktop database associates these icons with each file's creator (the `fdCreator` field in the FInfo record) and the type (the `fdType` field in the FInfo record), which are stored as part of the file's Finder information.

The Finder allows a Mac OS user to open a document, that is, to select a file and implicitly start the application that created the file. To do this, the Desktop database maintains a mapping between the file creator and a list of the locations of each application that has that file creator associated with it. This mapping is referred to as an APPL mapping because all Macintosh applications have a file creator of `'APPL'`. The Finder obtains the first item in the list and tries to start the application. If for some reason the application cannot be started (for example, if it is currently in use), the Finder will obtain the next application from the Desktop database's list and try that one. This list is dynamically filtered to present to the Finder only those applications for which the AFP client has the proper access rights.

The Desktop database is also a repository for the text of comments associated with files and directories on the volume. The Finder will make calls to the Desktop database to read or write these comments, which can be viewed and modified by selecting the Get Info item in the Finder's File menu. Comments are completely uninterpreted by the Desktop database.

For more information about the Finder and the use of the Desktop file, refer to *Inside Mac OS X*.

Character Encoding

If the server and the sharepoint support UTF-8 names, the AFP server and client send and receive decomposed UTF-8. However, characters in the range of U2000 to U2FFF, UFE30 to UFE4F, and U2F800 to U2FA1F are not decomposed. For complex characters, Unicode 3.2-based tables are used. For additional information, see http://developer.apple.com/technotes/tn/tn1150.html#UnicodeSubtleties and the Unicode specifications.

For Macintosh Roman, AFP utilizes character string entity names that can be composed of any 8-bit character. Character representations are exactly the same as those used by the Mac OS and are shown in Figure 1-12 (page 65).

Note: The information in this section applies only to Macintosh Roman character representations and does not apply to Unicode character representations.

Figure 1-12 AFP character set mapping

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	SPACE	0	@	P	‘	p				∞		—		
1	SOH	DC1	!	1	A	Q	a	q			°	±	`			
2	STX	DC2	"	2	B	R	b	r			¢	≤	´			
3	ETX	DC3	#	3	C	S	c	s			£	≥	√			
4	EOT	DC4	$	4	D	T	d	t			/	·	~			
5	ENQ	NAK	%	5	E	U	e	u			•	μ	≈			
6	ACK	SYN	&	6	F	V	f	v			ƒ	δ	Δ	÷		
7	BEL	ETB	’	7	G	W	g	w			§	Σ	·	◊		
8	BS	CAN	(	8	H	X	h	x			¤	∏	¨			
9	HT	EM	)	9	I	Y	i	y			'	π				
A	LF	SUB	*	:	J	Z	j	z			“	∫	˝			
B	VT	ESC	+	;	K	[	k	{			“	”	¸			
C	FF	FS	,	<	L	\	l	|		‹	…	Ã			
D	CR	GS	-	=	M	]	m	}			≠	Ω	Õ			
E	SO	RS	.	>	N	^	n	~			fi		˛			
F	SI	US	/	?	O	_	o	DLE			fl	¿	ˇ			

Throughout AFP, character string comparison is done in a case-insensitive manner (that is, K = k), and it must also be done in a diacritical-sensitive manner (that is, e é).

The mapping in Figure 1-12 shows the rules for uppercase equivalence of characters in AFP. Any character that does not appear in this table has no uppercase equivalent in AFP and therefore can only match itself. Note that this mapping does not exactly conform to the standards used in all human languages. In certain languages, the uppercase equivalent of e is E; in other languages (and in AFP), it is E´.

Using Login Commands

An AFP client uses the following commands to get information about a file server and to open and close a session with it:

- `FPGetSrvrInfo`
- `FPGetAuthMethods`
- `FPLogin` and `FPLoginExt`
- `FPLoginCont`
- `FPGetSrvrParms`
- `FPGetSessionToken`
- `FPDisconnectOldSession`
- `FPLogout`
- `FPMapID`
- `FPMapName`
- `FPChangePassword`
- `FPGetUserInfo`

The AFP client sends the `FPGetSrvrInfo` command to obtain server information. The `FPGetSrvrInfo` command returns server information including the following server parameters: server name, machine type, AFP version strings, UAM strings, volume icon and mask, a bitmap of flags, and optionally, a list of available Open Directory names. For descriptions of server parameters, see `FPGetSrvrInfo` in the Reference section.

From the lists of AFP versions and UAMs that the server supports, the AFP client selects the highest AFP version and the most secure UAM that the AFP client supports. To establish a session with the file server, the AFP client includes the strings for the selected AFP version and UAM in an `FPLogin` or `FPLoginExt` command.

When calling `FPLoginExt`, the AFP client must specify the user name in UTF-8 encoding and specify the Open Directory domain in which the user can be found. (A user name specified in UTF-8 encoding is the same as a AFPName file name, except that there is no text encoding hint.) Before calling `FPLoginExt`, the AFP client may first send an `FPGetAuthMethods` command to get the authentication methods that the Directory Service domain supports.

In response to the `FPLogin` or `FPLoginExt` command, the server performs user authentication. Depending on the selected UAM, the entire user authentication process can involve one or more `FPLoginCont` commands to complete the authentication process with the server. A session is established between the file server and the AFP client when the authentication process completes successfully.

After a session is established, the AFP client must obtain a list of the server's volumes. To obtain the list, the AFP client makes an `FPGetSrvrParms` command, which returns the number of volumes shared by the server, the names of the volumes, and whether they are password-protected.

The `FPGetSessionToken` command gets a reconnect token that the AFP client may later use if the session is disconnected unintentionally. In the case of an unintentional disconnection, the AFP client logs in again using the same user and authentication information that it used to log in previously, re-establishes the state of the connection, and sends an `FPDisconnectOldSession` command that passes the reconnect token to the server to tell it to release resources associated with the disconnected session.

When the AFP client user no longer needs to communicate with the server, the AFP client issues an `FPLogout` command to terminate the session.

The `FPMapID` and `FPMapName` commands are used for directory access control. The `FPMapID` command obtains the user or group name corresponding to a given User or Group ID. The `FPMapName` command converts a user or group name to the corresponding User or Group ID.

The `FPChangePassword` command changes a user's password.

The `FPGetUserInfo` command retrieves information about a user.

The `FPGetSrvrMsg` command retrieves log in and server messages from the server.

Using Volume Commands

AFP provides the following volume-level commands:

- `FPOpenVol`
- `FPCloseVol`
- `FPGetVolParms`
- `FPSetVolParms`
- `FPFlush`
- `FPCatSearch` and `FPCatSearchExt`

After obtaining the volume names through the `FPGetSrvrParms` command, the AFP client sends an `FPOpenVol` command for each volume to which it wants to gain access. If a volume has a password, it must be supplied at this time. The command returns the requested volume parameters, including the volume identifier, *VolumeID*.

The volume identifier is used in all subsequent commands to identify the volume for which the commands apply and remains valid until the session is terminated by calling `FPLogout` or the volume is closed by calling `FPVolClose`.

After obtaining the volume's volume identifier, the AFP client can obtain the volume's parameters by calling `FPGetVolParms`. The AFP client can also change the volume's parameters by calling `FPSetVolParms`.

The `FPFlush` command requests that the server flush (write to disk) any data associated with a particular volume.

The `FPCatSearch` and `FPCatSearchExt` commands search a volume for files that match specified criteria. The `FPCatSearchExt` command differs from the `FPCatSearch` command in that `FPCatSearchExt` is prepared to handle the larger values that may be returned for searches on volumes greater than 4 GB in size.

Using Directory Commands

AFP provides these commands for working on directories:

- FPSetDirParms
- FPOpenDir
- FPCloseDir
- FPEnumerate, FPEnumerateExt, and FPEnumerateExt2
- FPCreateDir

The FPSetDirParms command allows the AFP client to modify a directory's parameters. To obtain a directory's parameters from the file server, the AFP client uses the FPGetFileDirParms command, which is described in the section "Using Combined Directory and File Commands" (page 75).

On variable Directory ID volumes, the AFP client uses the FPOpenDir command to open a directory on and retrieve its Directory ID. The Directory ID is used in subsequent commands to enumerate the directory or to obtain access to its offspring. For variable Directory ID volumes, the FPOpenDir command is the only way to retrieve the Directory ID. Calling FPGetFileDirParms, FPEnumerate, FPEnumerateExt, or FPEnumerateExt2 to retrieve the Directory ID on such volumes causes an error to be returned.

On a fixed Directory ID volume, calling FPGetFileDirParms, FPEnumerate, FPEnumerateExt, or FPEnumerateExt2 is the preferred way to obtain a Directory ID, although calling FPOpenDir also works.

The AFP client can close directories on variable Directory ID volumes by sending the FPCloseDir command, which invalidates the corresponding Directory ID.

The AFP client uses the FPEnumerate, FPEnumerateExt, and FPEnumerateExt2 commands to list, or enumerate, the files and directories contained within a specified directory. In reply to this command, the server returns a list of directory or file parameters corresponding to those offspring. The FPEnumerateExt command differs from the FPEnumerate command in that the FPEnumerateExt command is prepared to handle larger values that may be returned when volumes are larger than 4 GB in size. The FPEnumerateExt2 command differs from the FPEnumerate command in that the StartIndex and MaxReplySize components to the FPEnumerateExt2 command are longs, allowing you to specify larger values than can be specified by the FPEnumerate and FPEnumerateExt commands.

Directories are created by the FPCreateDir command.

Using File Commands

AFP provides these commands for working on files:

- `FPSetFileDirParms`
- `FPCreateFile`
- `FPCopyFile`
- `FPCreateID`
- `FPDeleteID`
- `FPResolveID`
- `FPExchangeFiles`

The AFP client uses the `FPSetFileParms` command to modify a specified file's parameters, the `FPCreateFile` command to create a file, and the `FPCopyFile` command to copy a file that exists on a volume managed by a server to any other volume managed by that server. To obtain a specified file's parameters, the AFP client uses the `FPGetFileDirParms` command, discussed in the next section.

The `FPCreateID` command creates a unique File ID for an existing file, and `FPDeleteID` removes a File ID.

The `FPResolveID` command uses a File ID to retrieve information about a file.

The `FPExchangeFiles` command preserves existing file IDs when an application performs a Save or a Save As operation.

Using Combined Directory and File Commands

AFP provides five commands that operate on both files and directories:

- `FPGetFileDirParms`
- `FPSetFileDirParms`
- `FPRename`
- `FPDelete`
- `FPMoveAndRename`

The AFP client uses the `FPGetFileDirParms` command to retrieve the parameters associated with a given file or directory. When it uses this command, the AFP client does not need specify whether the CNode is a file or directory; the file server indicates the CNode's type in response to this command.

The `FPSetFileDirParms` command is used to set the parameters of a file or directory. When the AFP client uses this command, it need not specify whether the object is a file or directory. This command allows the AFP client to set only those parameters that are common to both types of CNodes.

The `FPRename` command is used to rename files and directories.

The `FPDelete` command is used to delete a file or directory. A file can be deleted only if it is not open; a directory can be deleted only if it is empty.

The `FPMoveandRename` command is used to move a file or a directory from one parent directory to another on the same volume. The moved CNode can renamed at the same time.

Using Fork Commands

AFP provides these fork-level commands:

- FPGetForkParms
- FPSetForkParms
- FPOpenFork
- FPRead and FPReadExt
- FPWrite and FPWriteExt
- FPFlushFork
- FPByteRangeLock and FPByteRangeLockExt
- FPCloseFork

The AFP client uses the FPGetForkParms command to read a fork's parameters.

The FPSetForkParms command is used to modify a fork's parameters.

The FPOpenFork command is used to open either fork of an existing file. This command returns an open fork reference number, which is used in subsequent commands for this open fork.

The FPRead and FPReadExt commands are used to read the contents of the fork. The FPReadExt command differs from the FPRead command in that the FPReadExt command is prepared to handle large values that may be returned for volumes greater than 4 GB in size.

The FPWrite and FPWriteExt commands are used to write to a fork. The FPWriteExt command differs from the FPWrite command in that the FPWriteExt command is prepared to handle the large values that are required for writing to volumes greater than 4 GB in size.

The FPFlushFork command is used to request that server write to disk any of the fork's data that is in the server's internal buffers.

The FPByteRangeLock and FPByteRangeLockExt commands are used to lock ranges of bytes in the fork. The FPByteRangeLockExt command differs from the FPByteRangeLock command in that the FPByteRangeLockExt command is prepared to handle large values that are required for locking ranges on volumes greater than 4 GB in size. Locks allow multiple users to share a file's open fork. Locking a range of bytes prevents other AFP clients from reading or writing data within the specified range. If an AFP client locks a byte range, that range is reserved for exclusive manipulation by the client that placed the lock.

The FPCloseFork command is used to close an open fork. This command invalidates the open fork reference number that was assigned when the fork was opened.

Using Desktop Database Commands

An AFP client uses the following commands to read and write information stored in the server's Desktop database:

- `FPOpenDT`
- `FPCloseDT`
- `FPAddIcon`
- `FPGetIcon`
- `FPGetIconInfo`
- `FPAddAPPL`
- `FPRemoveAPPL`
- `FPGetAPPL`
- `FPAddComment`
- `FPRemoveComment`
- `FPGetComment`

Before any other Desktop database commands can be sent, the AFP client must send an `FPOpenDT` command. This command returns a reference number to be used in all subsequent commands on the Desktop database.

When access to the Desktop database is no longer needed, the AFP client makes an `FPCloseDT` command.

`FPAddIcon` adds a new icon to the Desktop database, and `FPGetIcon` retrieves the bitmap for a given icon as specified by its file creator and type. `FPGetIconInfo` retrieves a description of an icon. This command can be used to determine the set of icons associated with a given application. Successive `FPGetIconInfo` commands return information on all icons associated with a given file creator.

`FPAddAPPL` adds an APPL mapping for the specified application and its file creator. `FPRemoveAPPL` removes the specified application from the list of APPL mappings corresponding to its file creator. It is the AFP client's responsibility to add and remove APPL mappings for applications that are added to or removed from the volume, respectively. For applications that are moved or renamed, the AFP client should remove the old APPL mapping before the operation and add a new APPL mapping with the updated information after the operation has been completed successfully.

`FPGetAPPL` returns the next APPL mapping in the Desktop database's list of applications that correspond to a given file creator.

`FPAddComment` stores a comment string associated with a particular file or directory on the volume. When adding a comment for a file or directory that already has an associated comment, the existing comment is replaced.

`FPRemoveComment` removes the comment associated with a particular file or directory. `FPGetComment` retrieves the comment associated with a particular file or directory.

Document Revision History

This table describes the changes to *Apple Filing Protocol Programming Guide*.

Date	Notes
2006-04-04	Moved reference documentation to become a seperate document.
2005-06-04	Fixed idle timer information.
2005-05-12	Updated for AFP version 3.2.

Document Revision History