AFS-3 Programmer's Reference: *File Server/Cache Manager* Interface

Edward R. Zayas

Transarc Corporation

Version 1.1 of 20 Aug 1991 9:38

FS-00-D162

Contents

Chapter 1

Overview

1.1 Introduction

1.1.1 The AFS 3.1 Distributed File System

AFS 3.1 is a distributed file system (DFS) designed to meet the following set of requirements:

- **Server-client model**: Permanent file storage for AFS is maintained by a collection of *file server* machines. This centralized storage is accessed by individuals running on *client* machines, which also serve as the computational engines for those users. A single machine may act as both an AFS file server and client simultaneously. However, file server machines are generally assumed to be housed in a secure environment, behind locked doors.

- **Scale**: Unlike other existing DFSs, AFS was designed with the specific goal of supporting a very large user community. Unlike the rule-of-thumb ratio of 20 client machines for every server machine (20:1) used by Sun Microsystem's NFS distributed file system [4][5], the AFS architecture aims at smoothly supporting client/server ratios more along the lines of 200:1 within a single installation.

  AFS also provides another, higher-level notion of scalability. Not only can each independently-administered AFS site, or *cell*, grow very large (on the order of tens of thousands of client machines), but individual cells may easily collaborate to form a single, unified file space composed of the union of the individual name spaces. Thus, users have the image of a single UNIX file system tree rooted at the

/afs directory on their machine. Access to files in this tree is performed with the standard UNIX commands, editors, and tools, regardless of a file's location.

These cells and the files they export may be geographically dispersed, thus requiring client machines to access remote file servers across network pathways varying widely in speed, latency, and reliability. The AFS architecture encourages this concept of a single, **wide-area file system**. As of this writing, the community AFS filespace includes sites spanning the continental United States and Hawaii, and also reaches overseas to various installations in Europe, Japan, and Australia.

- **Performance**: This is a critical consideration given the scalability and connectivity requirements described above. A high-performance system in the face of high client/server ratios and the existence of low-bandwidth, high-latency network connections as well as the normal high-speed ones is achieved by two major mechanisms:

  - **Caching**: Client machines make extensive use of caching techniques wherever possible. One important application of this methodology is that each client is required to maintain a cache of files it has accessed from AFS file servers, performing its operations exclusively on these local copies. This file cache is organized in a least-recently-used (LRU) fashion. Thus, each machine will build a local working set of objects being referenced by its users. As long as the cached images remain "current" (i.e., compatible with the central version stored at the file servers), operations may be performed on these files without further communication with the central servers. This results in significant reductions in network traffic and server loads, paving the way for the target client/server ratios.

    This file cache is typically located on the client's local hard disk, although a strictly in-memory cache is also supported. The disk cache has the advantage that its contents will survive crashes and reboots, with the expectation that the majority of cached objects will remain current. The local cache parameters, including the maximum number of blocks it may occupy on the local disk, may be changed on the fly. In order to avoid having the size of the client file cache become a limit on the length of an AFS file, caching is actually performed on **chunks** of the file. These chunks are typically 64 Kbytes in length, although the chunk size used by the client is settable when the client starts up.

  - **Callbacks**: The use of caches by the file system, as described above, raises the thorny issue of cache consistency. Each client must efficiently determine whether its cached file chunks are identical to the corresponding sections of the file as stored at the server machine before allowing a user to operate on those chunks. AFS employs the notion of a **callback** as the backbone of its cache consistency algorithm. When a server machine delivers one or more chunks of a file to a client, it also includes a callback "promise" that the client will be notified if any modifications are made to the data in the file. Thus, as long as the client machine is in possession of a callback for a file, it knows it is correctly synchronized with the centrally-stored version, and allows its users to operate on it as desired without any further interaction with the server. Before a file server stores a more recent version of a file on its own disks, it will first break all outstanding callbacks on this item. A callback will eventually time out, even if there are no changes to the file or directory it covers.

- **Location transparency**: The typical AFS user does not know which server or servers houses any of his or her files. In fact, the user's storage may be distributed among several servers. This location transparency also allows user data to be migrated between servers without users having to take corrective actions, or even becoming aware of the shift.

- **Reliability**: The crash of a server machine in any distributed file system will cause the information it hosts to become unavailable to the user community. The same effect is caused when server and client machines are isolated across a network partition. AFS addresses this situation by allowing data to be **replicated** across two or more servers in a read-only fashion. If the client machine loses contact with a particular server from which it is attempting to fetch data, it hunts among the remaining machines hosting replicas, looking for one that is still in operation. This search is performed without the user's knowledge or intervention, smoothly masking outages whenever possible. Each client machine will automatically perform periodic probes of machines on its list of known servers, updating its internal records concerning their status. Consequently, server machines may enter and exit the pool without administrator intervention.

  Replication also applies to the various databases employed by the AFS server processes. These system databases are read/write replicated with a single synchronization site at any instant. If a synchronization site is lost due to failure, the remaining database sites elect a new synchronization site automatically without operator intervention.

- **Security**: A production file system, especially one which allows and encourages transparent access between administrative domains, must be conscious of security issues. AFS considers the server machines as "trusted", being kept behind locked doors and only directly manipulated by administrators. On the other hand, client machines are, by definition, assumed to exist in inherently insecure environments. These client machines are recognized to be fully accessible to their users, making AFS servers open to attacks mounted by possibly modified hardware, operating systems, and software from its clients.

To provide credible file system security, AFS employs an authentication system based on the Kerberos facility developed by Project Athena at MIT [6][7]. Users operating from client machines are required to interact with *Authentication Server* agents running on the secure server machines to generate secure **tokens** of identity. These tokens express the user's identity in an encrypted fashion, and are stored in the kernel of the client machine. When the user attempts to fetch or store files, the server may *challenge* the user to verify his or her identity. This challenge, hidden from the user and handled entirely by the RPC layer, will transmit this token to the file server involved in the operation. The server machine, upon decoding the token and thus discovering the user's true identity, will allow the caller to perform the operation if permitted.

- **Access control**: The standard UNIX access control mechanism associates *mode bits* with every file and directory, applying them based on the user's numerical identifier and the user's membership in various groups. AFS has augmented this traditional access control mechanism with *Access Control Lists (ACLs)*. Every AFS directory has an associated ACL which defines the *principals* or parties that may operate on all files contained in the directory, and which operations these principals may perform. Rights granted by these ACLs include read, write, delete, lookup, insert (create new files, but don't overwrite old files), and administer (change the ACL). Principals on these ACLs include individual users and groups of users. These groups may be defined by AFS users without administrative intervention. AFS ACLs provide for much finer-grained access control for its files.

- **Administrability**: Any system with the scaling goals of AFS must pay close attention to its ease of administration. The task of running an AFS installation is facilitated via the following mechanisms:

  - **Pervasive RPC interfaces**: Access to AFS server agents is performed mostly via RPC interfaces. Thus, servers may be queried and operated upon regardless of their location. In combination with the security system outlined above, even administrative functions such as instigating backups, reconfiguring server machines, and stopping and restarting servers may be performed by an administrator sitting in front of *any* AFS-capable machine, as long as the administrator holds the proper tokens.
  - **Replication**: As AFS supports read-only replication for user data and read-write replication for system databases, much of the system reconfiguration work in light of failures is performed transparently and without human intervention. Administrators thus typically have more time to respond to many common failure situations.
  - **Data mobility**: Improved and balanced utilization of disk resources is facilitated by the fact that AFS supports transparent relocation of user data

between partitions on a single file server machine or between two different machines. In a situation where a machine must be brought down for an extended period, all its storage may be migrated to other servers so that users may continue their work completely unaffected.

  - **Automated "nanny" services**: Each file server machine runs a *BOS Server* process, which assists in the machine's administration. This server is responsible for monitoring the health of the AFS agents under its care, bringing them up in the proper order after a system reboot, answering requests as to their status and restarting them when they fail. It also accepts commands to start, suspend, or resume these processes, and install new server binaries. Accessible via an RPC interface, this supervisory process relieves administrators of some oversight responsibilities and also allows them to perform their duties from any machine running AFS, regardless of location or geographic distance from the targeted file server machine.
  - **On-line backup**: Backups may be performed on the data stored by the AFS file server machines without bringing those machines down for the duration. Copy-on-write "snapshots" are taken of the data to be preserved, and tape backup is performed from these clones. One added benefit is that these backup clones are on-line and accessible by users. Thus, if someone accidentally deletes a file that is contained in their last snapshot, they may simply copy its contents as of the time the snapshot was taken back into their active workspace. This facility also serves to improve the administrability of the system, greatly reducing the number of requests to restore data from tape.
  - **On-line help**: The set of provided program tools used to interact with the active AFS agents are self-documenting in that they will accept command-line requests for help, displaying descriptive text in response.
  - **Statistics**: Each AFS agent facilitates collection of statistical data on its performance, configuration, and status via its RPC interface. Thus, the system is easy to monitor. One tool that takes advantage of this facility is the scout program. Scout polls file server machines periodically, displaying usage statistics, current disk capacities, and whether the server is unavailable. Administrators monitoring this information can thus quickly react to correct overcrowded disks and machine crashes.

- **Coexistence**: Many organizations currently employ other distributed file systems, most notably NFS. AFS was designed to run simultaneously with other DFSs without interfering in their operation. In fact, an NFS-AFS translator agent exists that allows pure-NFS client machines to transparently access files in the AFS community.

- **Portability**: Because AFS is implemented using the standard **VFS** and **vnode** interfaces pioneered and advanced by Sun Microsystems, AFS is easily portable between different platforms from a single vendor or from different vendors.

1.1.2 Scope of this Document

This document is a member of a documentation suite providing specifications of the operations and interfaces offered by the various AFS servers and agents. Specifically, this document will focus on two of these system agents:

- *File Server*: This AFS entity is responsible for providing a central disk repository for a particular set of files and for making these files accessible to properly-authorized users running on client machines. The *File Server* is implemented as a user-space process
- *Cache Manager*: This code, running within the kernel of an AFS client machine, is a user's representative in communicating with the *File Servers*, fetching files back and forth into the local cache as needed. The *Cache Manager* also keeps information as to the composition of its own cell as well as the other AFS cells in existence. It resolves file references and operations, determining the proper *File Server* (or group of *File Servers*) that may satisfy the request. In addition, it is also a reliable repository for the user's authentication information, holding on to their tokens and wielding them as necessary when challenged.

1.1.3 Related Documents

The full AFS specification suite of documents is listed below:

- *AFS-3 Programmer's Reference: Architectural Overview*: This paper provides an architectual overview of the AFS distributed file system, describing the full set of servers and agents in a coherent way, illustrating their relationships to each other and examining their interactions.
- *AFS-3 Programmer's Reference:Volume Server/Volume Location Server Interface*: This document describes the services through which "containers" of related user data are located and managed.
- *AFS-3 Programmer's Reference: Protection Server Interface*: This paper describes the server responsible for providing two-way mappings between printable user

names and their internal AFS identifiers. The *Protection Server* also allows users to create, destroy, and manipulate "groups" of users, which are suitable for placement on ACLs.

- *AFS-3 Programmer's Reference: BOS Server Interface*: This paper explicates the "nanny" service described above, which assists in the administrability of the AFS environment.
- *AFS-3 Programmer's Reference: Specification for the Rx Remote Procedure Call Facility*: This document specifies the design and operation of the remote procedure call and lightweight process packages used by AFS.

In addition to these papers, the AFS 3.1 product is delivered with its own user, administrator, installation, and command reference documents.

1.2 Basic Concepts

To properly understand AFS operation, specifically the tasks and objectives of the *File Server* and *Cache Manager*, it is necessary to introduce and explain the following concepts:

- **Cell**: A cell is the set of server and client machines operated by an administratively independent organization. The cell administrators make decisions concerning such issues as server deployment and configuration, user backup schedules, and replication strategies on their own hardware and disk storage completely independently from those implemented by other cell administrators regarding their own domains. Every client machine belongs to exactly one cell, and uses that information to determine the set of database servers it uses to locate system resources and generate authentication information.
- **Volume**: AFS disk partitions do not directly host individual user files or directories. Rather, connected subtrees of the system's directory structure are placed into containers called **volumes**. Volumes vary in size dynamically as objects are inserted, overwritten, and deleted. Each volume has an associated **quota**, or maximum permissible storage. A single UNIX disk partition may host one or more volumes, and in fact may host as many volumes as physically fit in the storage space. However, a practical maximum is 3,500 volumes per disk partition, since this is the highest number currently handled by the *salvager* program. The *salvager* is run on occasions where the volume structures on disk are inconsistent, repairing the damage. A compile-time constant within the *salvager* imposes the above

limit, causing it to refuse to repair any inconsistent partition with more than 3,500 volumes.

Volumes serve many purposes within AFS. First, they reduce the number of objects with which an administrator must be concerned, since operations are normally performed on an entire volume at once (and thus on all files and directories contained within the volume). In addition, volumes are the unit of replication, data mobility between servers, and backup. Disk utilization may be balanced by transparently moving volumes between partitions.

- **Mount Point**: The connected subtrees contained within individual volumes stored at AFS file server machines are "glued" to their proper places in the file space defined by a site, forming a single, apparently seamless UNIX tree. These attachment points are referred to as **mount points**. Mount points are persistent objects, implemented as symbolic links whose contents obey a stylized format. Thus, AFS mount points differ from NFS-style *mounts*. In the NFS environment, the user dynamically mounts entire remote disk partitions using any desired name. These mounts do not survive client restarts, and do not insure a uniform namespace between different machines.

  As a *Cache Manager* resolves an AFS pathname as part of a file system operation initiated by a user process, it recognizes mount points and takes special action to resolve them. The *Cache Manager* consults the appropriate *Volume Location Server* to discover the *File Server* (or set of *File Servers*) hosting the indicated volume. This location information is cached, and the *Cache Manager* then proceeds to contact the listed *File Server*(s) in turn until one is found that responds with the contents of the volume's root directory. Once mapped to a real file system object, the pathname resolution proceeds to the next component.

- **Database Server**: A set of AFS databases is required for the proper functioning of the system. Each database may be replicated across two or more file server machines. Access to these databases is mediated by a database server process running at each replication site. One site is declared to be the synchronization site, the sole location accepting requests to modify the databases. All other sites are read-only with respect to the set of AFS users. When the synchronization site receives an update to its database, it immediately distributes it to the other sites. Should a synchronization site go down through either a hard failure or a network partition, the remaining sites will automatically elect a new synchronization site if they form a **quorum**, or majority. This insures that multiple synchronization sites do not become active in the network partition scenario.

  The classes of AFS database servers are listed below:

  - *Authentication Server*: This server maintains the authentication database used to generate tokens of identity.

- *Protection Server*: This server maintains mappings between human-readable user account names and their internal numerical AFS identifiers. It also manages the creation, manipulation, and update of user-defined groups suitable for use on ACLs.
- *Volume Location Server*: This server exports information concerning the location of the individual volumes housed within the cell.

1.3 Document Layout

Following this introduction and overview, Chapter 2 describes the architecture of the *File Server* process design. Similarly, Chapter 3 describes the architecture of the in-kernel *Cache Manager* agent. Following these architectural examinations, Chapter 4 provides a set of basic coding definitions common to both the AFS *File Server* and *Cache Manager*, required to properly understand the interface specifications which follow. Chapter 5 then proceeds to specify the various *File Server* interfaces. The myriad *Cache Manager* interfaces are presented in Chapter 6, thus completing the document.

Chapter 2

File Server **Architecture**

2.1 Overview

The AFS *File Server* is a user-level process that presides over the raw disk partitions on which it supports one or more volumes. It provides "half" of the fundamental service of the system, namely exporting and regimenting access to the user data entrusted to it. The *Cache Manager* provides the other half, acting on behalf of its human users to locate and access the files stored on the file server machines.

This chapter examines the structure of the *File Server* process. First, the set of AFS agents with which it must interact are discussed. Next, the threading structure of the server is examined. Some details of its handling of the race conditions created by the callback mechanism are then presented. This is followed by a discussion of the read-only volume synchronization mechanism. This functionality is used in each RPC interface call and intended to detect new releases of read-only volumes. *File Servers* do *not* generate callbacks for objects residing in read-only volumes, so this synchronization information is used to implement a "whole-volume" callback. Finally, the fact that the *File Server* may drop certain information recorded about the *Cache Managers* with which it has communicated and yet guarantee correctness of operation is explored.

2.2 Interactions

By far the most frequent partner in *File Server* interactions is the set of *Cache Managers* actively fetching and storing chunks of data files for which the *File Server* provides central storage facilities. The *File Server* also periodically probes the *Cache Managers* recorded in its tables with which it has recently dealt, determining if they are still active or whether their records might be garbage-collected.

There are two other server entities with which the *File Server* interacts, namely the *Protection Server* and the *BOS Server*. Given a fetch or store request generated by a *Cache Manager*, the *File Server* needs to determine if the caller is authorized to perform the given operation. An important step in this process is to determine what is referred to as the caller's **Current Protection Subdomain**, or **CPS**. A user's CPS is a list of principals, beginning with the user's internal identifier, followed by the the numerical identifiers for all groups to which the user belongs. Once this CPS information is determined, the *File Server* scans the ACL controlling access to the file system object in question. If it finds that the ACL contains an entry specifying a principal with the appropriate rights which also appears in the user's CPS, then the operation is cleared. Otherwise, it is rejected and a protection violation is reported to the *Cache Manager* for ultimate reflection back to the caller.

The *BOS Server* performs administrative operations on the *File Server* process. Thus, their interactions are quite one-sided, and always initiated by the *BOS Server*. The *BOS Server* does not utilize the *File Server*'s RPC interface, but rather generates UNIX signals to achieve the desired effect.

2.3 Threading

The *File Server* is organized as a multi-threaded server. Its threaded behavior within a single UNIX process is achieved by use of the **LWP lightweight process** facility, as described in detail in the companion "AFS-3 Programmer's Reference: Specification for the *Rx* Remote Procedure Call Facility" document. The various threads utilized by the *File Server* are described below:

- **WorkerLWP**: This lightweight process sleeps until a request to execute one of the RPC interface functions arrives. It pulls the relevant information out of the request, including any incoming data delivered as part of the request, and then executes the server stub routine to carry out the operation. The thread finishes its current activation by feeding the return code and any output data back through the RPC channel back to the calling *Cache Manager*. The *File Server* initialization sequence specifies that at least three but no more than six of these `WorkerLWP` threads are to exist at any one time. It is currently not possible to configure the *File Server* process with a different number of `WorkerLWP` threads.

- **FiveMinuteCheckLWP**: This thread runs every five minutes, performing such housekeeping chores as cleaning up timed-out callbacks, setting disk usage statistics, and executing the special handling required by certain AIX implementations. Generally, this thread performs activities that do not take unbounded time to accomplish and do not block the thread. If reassurance is required, `FiveMinuteCheckLWP` can also be told to print out a banner message to the machine's console every so often, stating that the *File Server* process is still running. This is not strictly necessary and an artifact from earlier versions, as the *File Server*'s status is now easily accessible at any time through the *BOS Server* running on its machine.
- **HostCheckLWP**: This thread, also activated every five minutes, performs periodic checking of the status of *Cache Manager*s that have been previously contacted and thus appear in this *File Server*'s internal tables. It generates *RXAFSCB_Probe()* calls from the *Cache Manager* interface, and may find itself suspended for an arbitrary amount of time when it enounters unreachable *Cache Manager*s.

2.4 Callback Race Conditions

Callbacks serve to implement the efficient AFS cache consistency mechanism, as described in Section 1.1.1. Because of the asynchronous nature of callback generation and the multi-threaded operation and organization of both the *File Server* and *Cache Manager*, race conditions can arise in their use. As an example, consider the case of a client machine fetching a chunk of file *X*. The *File Server* thread activated to carry out the operation ships the contents of the chunk and the callback information over to the requesting *Cache Manager*. Before the corresponding *Cache Manager* thread involved in the exchange can be scheduled, another request arrives at the *File Server*, this time storing a modified image of the same chunk from file *X*. Another worker thread comes to life and completes processing of this second request, including execution of an *RXAFSCB_CallBack()* to the *Cache Manager* who still hasn't picked up on the results of its fetch operation. If the *Cache Manager* blindly honors the *RXAFSCB_CallBack()* operation first and then proceeds to process the fetch, it will wind up believing it has a callback on *X* when in reality it is out of sync with the central copy on the *File Server*. To resolve the above class of callback race condition, the *Cache Manager* effectively doublechecks the callback information received from *File Server* calls, making sure they haven't already been nullified by other file system activity.

2.5 Read-Only Volume Synchronization

The *File Server* issues a callback for each file chunk it delivers from a read-write volume, thus allowing *Cache Manager*s to efficiently synchronize their local caches with the authoritative *File Server* images. However, no callbacks are issued when data from read-only volumes is delivered to clients. Thus, it is possible for a new snapshot of the read-only volume to be propagated to the set of replication sites without *Cache Manager*s becoming aware of the event and marking the appropriate chunks in their caches as stale. Although the *Cache Manager* refreshes its volume version information periodically (once an hour), there is still a window where a *Cache Manager* will fail to notice that it has outdated chunks.

The **volume synchronization** mechanism was defined to close this window, resulting in what is nearly a "whole-volume" callback device for read-only volumes. Each *File Server* RPC interface function handling the transfer of file data is equipped with a parameter (*a_volSyncP*), which carries this volume synchronization information. This parameter is set to a non-zero value by the *File Server* exclusively when the data being fetched is coming from a read-only volume. Although the `struct AFSVolSync` defined in Section 5.1.2.2 passed via *a_volSyncP* consists of six longwords, only the first one is set. This leading longword carries the creation date of the read-only volume. The *Cache Manager* immediately compares the synchronization value stored in its cached volume information against the one just received. If they are identical, then the operation is free to complete, secure in the knowledge that all the information and files held from that volume are still current. A mismatch, though, indicates that every file chunk from this volume is potentially out of date, having come from a previous release of the read-only volume. In this case, the *Cache Manager* proceeds to mark every chunk from this volume as suspect. The next time the *Cache Manager* considers accessing any of these chunks, it first checks with the *File Server* it came from which the chunks were obtained to see if they are up to date.

2.6 Disposal of *Cache Manager* Records

Every *File Server*, when first starting up, will, by default, allocate enough space to record 20,000 callback promises (see Section 5.3 for how to override this default). Should the *File Server* fully populate its callback records, it will not allocate more, allowing its memory image to possibly grow in an unbounded fashion. Rather, the *File Server* chooses to break callbacks until it acquires a free record. All reachable *Cache Manager*s respond by marking their cache entries appropriately, preserving the consistency guarantee. In fact, a *File Server* may arbitrarily and unilaterally purge itself of all records associated with a particular *Cache Manager*. Such actions will reduce its performance (forcing these *Cache Manager*s to revalidate items cached from that *File Server*) without sacrificing correctness.

Chapter 3

Cache Manager **Architecture**

3.1 Overview

The AFS *Cache Manager* is a kernel-resident agent with the following duties and responsibilities:

- Users are to be given the illusion that files stored in the AFS distributed file system are in fact part of the local UNIX file system of their client machine. There are several areas in which this illusion is not fully realized:
  - **Semantics**: Full UNIX semantics are not maintained by the set of agents implementing the AFS distributed file system. The largest deviation involves the time when changes made to a file are seen by others who also have the file open. In AFS, modifications made to a cached copy of a file are not necessarily reflected immediately to the central copy (the one hosted by *File Server* disk storage), and thus to other cache sites. Rather, the changes are only guaranteed to be visible to others who simultaneously have their own cached copies open when the modifying process executes a UNIX *close()* operation on the file.

    This differs from the semantics expected from the single-machine, local UNIX environment, where writes performed on one open file descriptor are immediately visible to all processes reading the file via their own file descriptors. Thus, instead of the standard "last writer wins" behavior, users see "last closer wins" behavior on their AFS files. Incidentally, other DFSs, such as NFS, do not implement full UNIX semantics in this case either.

  - **Partial failures**: A panic experienced by a local, single-machine UNIX file system will, by definition, cause all local processes to terminate immediately. On the other hand, any hard or soft failure experienced by a *File Server* process or the machine upon which it is executing does not cause any of the *Cache Manager*s interacting with it to crash. Rather, the *Cache Manager*s will now have to reflect their failures in getting responses from the affected *File Server* back up to their callers. Network partitions also induce the same behavior. From the user's point of view, part of the file system tree has become inaccessible. In addition, certain system calls (e.g., *open()* and *read()*) may return unexpected failures to their users. Thus, certain coding practices that have become common amongst experienced (single-machine) UNIX programmers (e.g., not checking error codes from operations that "can't" fail) cause these programs to misbehave in the face of partial failures.

  To support this transparent access paradigm, the *Cache Manager* proceeds to:

  - Intercept all standard UNIX operations directed towards AFS objects, mapping them to references aimed at the corresponding copies in the local cache.
  - Keep a synchronized local cache of AFS files referenced by the client machine's users. If the chunks involved in an operation reading data from an object are either stale or do not exist in the local cache, then they must be fetched from the *File Server*(s) on which they reside. This may require a query to the volume location service in order to locate the place(s) of residence. Authentication challenges from *File Server*s needing to verify the caller's identity are handled by the *Cache Manager*, and the chunk is then incorporated into the cache.
  - Upon receipt of a UNIX close, all dirty chunks belonging to the object will be flushed back to the appropriate *File Server*.
  - Callback deliveries and withdrawals from *File Server*s must be processed, keeping the local cache in close synchrony with the state of affairs at the central store.

- Interfaces are also be provided for those principals who wish to perform AFS-specific operations, such as Access Control List (ACL) manipulations or changes to the *Cache Manager*'s configuration.

This chapter takes a tour of the *Cache Manager*'s architecture, and examines how it supports these roles and responsibilities. First, the set of AFS agents with which it must interact are discussed. Next, some of the *Cache Manager*'s implementation and interface choices are examined. Finally, the server's ability to arbitrarily dispose of callback information without affecting the correctness of the cache consistency algorithm is explained.

3.2 Interactions

The main AFS agent interacting with a *Cache Manager* is the *File Server*. The most common operation performed by the *Cache Manager* is to act as its users' agent in fetching and storing files to and from the centralized repositories. Related to this activity, a *Cache Manager* must be prepared to answer queries from a *File Server* concerning its health. It must also be able to accept callback revocation notices generated by *File Servers*. Since the *Cache Manager* not only engages in data transfer but must also determine where the data is located in the first place, it also directs inquiries to *Volume Location Server* agents. There must also be an interface allowing direct interactions with both common and administrative users. Certain AFS-specific operations must be made available to these parties. In addition, administrative users may desire to dynamically reconfigure the *Cache Manager*. For example, information about a newly-created cell may be added without restarting the client's machine.

3.3 Implementation Techniques

The above roles and behaviors for the *Cache Manager* influenced the implementation choices and methods used to construct it, along with the desire to maximize portability. This section begins by showing how the VFS/vnode interface, pioneered and standardized by Sun Microsystems, provides not only the necessary fine-grain access to user file system operations, but also facilitates *Cache Manager* ports to new hardware and operating system platforms. Next, the use of UNIX system calls is examined. Finally, the threading structure employed is described.

3.3.1 VFS Interface

As mentioned above, Sun Microsystems has introduced and propagated an important concept in the file system world, that of the Virtual File System (VFS) interface. This abstraction defines a core collection of file system functions which cover all operations required for users to manipulate their data. System calls are written in terms of these standardized routines. Also, the associated *vnode* concept generalizes the original UNIX *inode* idea and provides hooks for differing underlying environments. Thus, to port a system to a new hardware platform, the system programmers have only to construct implementations of this base array of functions consistent with the new underlying machine.

The VFS abstraction also allows multiple file systems (e.g., vanilla UNIX, DOS, NFS, and

AFS) to coexist on the same machine without interference. Thus, to make a machine AFS-capable, a system designer first extends the base vnode structure in well-defined ways in order to store AFS-specific operations with each file description. Then, the base function array is coded so that calls upon the proper AFS agents are made to accomplish each function's standard objectives. In effect, the *Cache Manager* consists of code that interprets the standard set of UNIX operations imported through this interface and executes the AFS protocols to carry them out.

3.3.2 System Calls

As mentioned above, many UNIX system calls are implemented in terms of the base function array of vnode-oriented operations. In addition, one existing system call has been modified and two new system calls have been added to perform AFS-specific operations apart from the *Cache Manager*'s UNIX "emulation" activities. The standard *ioctl()* system call has been augmented to handle AFS-related operations on objects accessed via open UNIX file descriptors. One of the brand-new system calls is **pioctl()**, which is much like *ioctl()* except it names targeted objects by pathname instead of file descriptor. Another is *afs_call()*, which is used to initialize the *Cache Manager* threads, as described in the section immediately following.

3.3.3 Threading

In order to execute its many roles, the *Cache Manager* is organized as a multi-threaded entity. It is implemented with (potentially multiple instantiations of) the following three thread classes:

- **CallBack Listener**: This thread implements the *Cache Manager* callback RPC interface, as described in Section 6.5.
- **Periodic Maintenance**: Certain maintenance and checkup activities need to be performed at five set intervals. Currently, the frequency of each of these operations is hard-wired. It would be a simple matter, though, to make these times configurable by adding command-line parameters to the *Cache Manager*.
  - *Thirty seconds*: Flush pending writes for NFS clients coming in through the NFS-AFS Translator facility.
  - *One minute*: Make sure local cache usage is below the assigned quota, write out dirty buffers holding directory data, and keep *flock()*s alive.

  - *Three minutes*: Check for the resuscitation of *File Server*s previously determined to be down, and check the cache of previously computed access information in light of any newly expired tickets.
  - *Ten minutes*: Check health of all *File Server*s marked as active, and garbage-collect old RPC connections.
  - *One hour*: Check the status of the root AFS volume as well as all cached information concerning read-only volumes.

- **Background Operations**: The *Cache Manager* is capable of prefetching file system objects, as well as carrying out delayed stores, occurring sometime after a *close()* operation. At least two threads are created at *Cache Manager* initialization time and held in reserve to carry out these objectives. This class of background threads implements the following three operations:
  - *Prefetch operation*: Fetches particular file system object chunks in the expectation that they will soon be needed.
  - *Path-based prefetch operation*: The prefetch daemon mentioned above operates on objects already at least partly resident in the local cache, referenced by their vnode. The path-based prefetch daemon performs the same actions, but on objects named solely by their UNIX pathname.
  - *Delayed store operation*: Flush all modified chunks from a file system object to the appropriate *File Server*'s disks.

3.4 Disposal of *Cache Manager* Records

The *Cache Manager* is free to throw away any or all of the callbacks it has received from the set of *File Server*s from which it has cached files. This housecleaning does not in any way compromise the correctness of the AFS cache consistency algorithm. The *File Server* RPC interface described in this paper provides a call to allow a *Cache Manager* to advise of such unilateral jettisoning. However, failure to use this routine still leaves the machine's cache consistent. Let us examine the case of a *Cache Manager* on machine C disposing of its callback on file X from *File Server* F. The next user access on file X on machine C will cause the *Cache Manager* to notice that it does not currently hold a callback on it (although the *File Server* will *think* it does). The *Cache Manager* on C attempts to revalidate its entry when it is entirely possible that the file is still in sync with the central store. In response, the *File Server* will extend the existing callback information it has and deliver the new promise to the *Cache Manager* on C. Now consider the case where file X is modified by a party on a machine other than C before such an access occurs on C. Under these circumstances, the *File Server* will break its callback on file X before performing the central update. The *Cache Manager* on C will receive one of these "break callback" messages. Since it no longer has a callback on file X, the *Cache Manager* on C will cheerfully acknowledge the *File Server*'s notification and move on to other matters. In either case, the callback information for both parties will eventually resynchronize. The only potential penalty paid is extra inquiries by the *Cache Manager* and thus providing for reduced performance instead of failure of operation.

Chapter 4

Common Definitions and Data Structures

This chapter discusses the definitions used in common by the *File Server* and the *Cache Manager*. They appear in the *common.xg* file, used by *Rxgen* to generate the C code instantiations of these definitions.

4.1 File-Related Definitions

4.1.1 struct AFSFid

This is the type for file system objects within AFS.

Fields

`unsigned long Volume` - This provides the identifier for the volume in which the object resides.

`unsigned long Vnode` - This specifies the index within the given volume corresponding to the object.

`unsigned long Unique` - This is a "uniquifier" or generation number for the slot identified by the `Vnode` field.

4.2 Callback-related Definitions

4.2.1 Types of Callbacks

There are three types of callbacks defined by AFS-3:

- `EXCLUSIVE`: This version of callback has not been implemented. Its intent was to allow a single *Cache Manager* to have exclusive rights on the associated file data.
- `SHARED`: This callback type indicates that the status information kept by a *Cache Manager* for the associated file is up to date. All cached chunks from this file whose version numbers match the status information are thus guaranteed to also be up to date. This type of callback is non-exclusive, allowing any number of other *Cache Managers* to have callbacks on this file and cache chunks from the file.
- `DROPPED`: This is used to indicate that the given callback promise has been cancelled by the issuing *File Server*. The *Cache Manager* is forced to mark the status of its cache entry as unknown, forcing it to stat the file the next time a user attempts to access any chunk from it.

4.2.2 struct AFSCallBack

This is the canonical callback structure passed in many *File Server* RPC interface calls.

Fields

`unsigned long CallBackVersion` - Callback version number.

`unsigned long ExpirationTime` - Time when the callback expires, measured in seconds.

`unsigned long CallBackType` - The type of callback involved, one of `EXCLUSIVE`, `SHARED`, or `DROPPED`.

4.2.3 Callback Arrays

AFS-3 sometimes does callbacks in bulk. Up to `AFSCBMAX` (50) callbacks can be handled at once. Layouts for the two related structures implementing callback arrays, `struct AFSCBFids` and `struct AFSCBs`, follow below. Note that the callback descriptor in slot

i of the array in the AFSCBs structure applies to the file identifier contained in slot i in the fid array in the matching AFSCBFids structure.

4.2.3.1 struct AFSCBFids

Fields

- `u_int AFSCBFids_len` - Number of AFS file identifiers stored in the structure, up to a maximum of AFSCBMAX.
- `AFSFid *AFSCBFids_val` - Pointer to the first element of the array of file identifiers.

4.2.3.2 struct AFSCBs

Fields

- `u_int AFSCBs_len` - Number of AFS callback descriptors stored in the structure, up to a maximum of AFSCBMAX.
- `AFSCallBack *AFSCBs_val` - Pointer to the actual array of callback descriptors

4.3 Locking Definitions

4.3.1 struct AFSDBLockDesc

This structure describes the state of an AFS lock.

Fields

- `char waitStates` - Types of lockers waiting for the lock.
- `char exclLocked` - Does anyone have a boosted, shared or write lock? (A boosted lock allows the holder to have data read-locked and then "boost" up to a write lock on the data without ever relinquishing the lock.)
- `char readersReading` - Number of readers that actually hold a read lock on the associated object.
- `char numWaiting` - Total number of parties waiting to acquire this lock in some fashion.

4.3.2 struct AFSDBCacheEntry

This structure defines the description of a *Cache Manager* local cache entry, as made accessible via the *RXAFSCB_GetCE()* callback RPC call. Note that *File Server*s do *not* make the above call. Rather, client debugging programs (such as `cmdebug`) are the agents which call *RXAFSCB_GetCE()*.

Fields

`long addr` - Memory location in the *Cache Manager* where this description is located.

`long cell` - Cell part of the fid.

`AFSFid netFid` - Network (standard) part of the fid

`long Length` - Number of bytes in the cache entry.

`long DataVersion` - Data version number for the contents of the cache entry.

`struct AFSDBLockDesc lock` - Status of the lock object controlling access to this cache entry.

`long callback` - Index in callback records for this object.

`long cbExpires` - Time when the callback expires.

`short refCount` - General reference count.

`short opens` - Number of opens performed on this object.

`short writers` - Number of writers active on this object.

`char mvstat` - The file classification, indicating one of normal file, mount point, or volume root.

`char states` - Remembers the state of the given file with a set of bits indicating, from lowest-order to highest order: stat info valid, read-only file, mount point valid, pending core file, wait-for-store, and mapped file.

4.3.3 struct AFSDBLock

This is a fuller description of an AFS lock, including a string name used to identify it.

Fields

`char name[16]` - String name of the lock.

`struct AFSDBLockDesc lock` - Contents of the lock itself.

4.4 Miscellaneous Definitions

4.4.1 Opaque structures

A maximum size for opaque structures passed via the *File Server* interface is defined as `AFSOPAQUEMAX`. Currently, this is set to 1,024 bytes. The `AFSOpaque` typedef is defined for use by those parameters that wish their contents to travel completely uninterpreted across the network.

4.4.2 String Lengths

Two common definitions used to specify basic AFS string lengths are `AFSNAMEMAX` and `AFSPATHMAX`. `AFSNAMEMAX` places an upper limit of 256 characters on such things as file and directory names passed as parameters. `AFSPATHMAX` defines the longest pathname expected by the system, composed of slash-separated instances of the individual directory and file names mentioned above. The longest acceptable pathname is currently set to 1,024 characters.

Chapter 5

File Server **Interfaces**

There are several interfaces offered by the *File Server*, allowing it to export the files stored within the set of AFS volumes resident on its disks to the AFS community in a secure fashion and to perform self-administrative tasks. This chapter will cover the three *File Server* interfaces, summarized below. There is one *File Server* interface that will not be discussed in this document, namely that used by the *Volume Server*. It will be fully described in the companion *AFS-3 Programmer's Reference: Volume Server/Volume Location Server Interface*.

- **RPC**: This is the main *File Server* interface, supporting all of the *Cache Manager*'s needs for providing its own clients with appropriate access to file system objects stored within AFS. It is closedly tied to the callback interface exported by the *Cache Manager* as described in Section 6.5, which has special implications for any application program making direct calls to this interface.
- **Signals**: Certain operations on a *File Server* must be performed by it sending UNIX signals on the machine on which it is executing. These operations include performing clean shutdowns and adjusting debugging output levels. Properly-authenticated administrative users do not have to be physically logged into a *File Server* machine to generate these signals. Rather, they may use the RPC interface exported by that machine's *BOS Server* process to generate them from any AFS-capable machine.
- **Command Line**: Many of the *File Server*'s operating parameters may be set upon startup via its command line interface. Such choices as the number of data buffers and callback records to hold in memory may be made here, along with various other decisions such as lightweight thread stack size.

5.1 RPC Interface

5.1.1 Introduction and Caveats

The documentation for the AFS-3 *File Server* RPC interface commences with some basic definitions and data structures used in conjunction with the function calls. This is followed by an examination of the set of *non-streamed* RPC functions, namely those routines whose parameters are all fixed in size. Next, the *streamed* RPC functions, those with parameters that allow an arbitrary amount of data to be delivered, are described. A code fragment and accompanying description and analysis are offered as an example of how to use the streamed RPC calls. Finally, a description of the special requirements on any application program making direct calls to this *File Server* interface appears. The *File Server* assumes that any entity making calls to its RPC functionality is a bona fide and full-fledged *Cache Manager*. Thus, it expects this caller to export the *Cache Manager*'s own RPC interface, even if the application simply uses *File Server* calls that don't transfer files and thus generate callbacks.

Within those sections describing the RPC functions themselves, the purpose of each call is detailed, and the nature and use of its parameters is documented. Each of these RPC interface routines returns an integer error code, and *a subset* of the possible values are described. A complete and systematic list of potential error returns for each function is difficult to construct and unwieldy to examine. This is due to fact that error codes from many different packages and from many different levels may arise. Instead of attempting completeness, the error return descriptions discuss error codes generated within the functions themselves (or a very small number of code levels below them) within the *File Server* code itself, and not from such associated packages as the *Rx*, volume, and protection modules. Many of these error code are defined in the companion AFS-3 documents.

By convention, a return value of zero reveals that the function call was successful and that all of its `OUT` parameters have been set by the *File Server*.

5.1.2 Definitions and Structures

5.1.2.1 Constants and Typedefs

The following constants and typedefs are required to properly use the *File Server* RPC interface, both to provide values and to interpret information returned by the calls. The constants appear first, followed by the list of typedefs, which sometimes depend on the constants above. Items are alphabetized within each group.

All of the constants appearing below whose names contain the `XSTAT` string are used in conjuction with the *extended data collection facility* supported by the *File Server*. The *File Server* defines some number of *data collections*, each of which consists of an array of longword values computed by the *File Server*.

There are currently two data collections defined for the *File Server*. The first is identified by the `AFS_XSTATSCOLL_CALL_INFO` constant. This collection of longwords relates the number of times each internal function within the *File Server* code has been executed, thus providing profiling information. The second *File Server* data collection is identified by the `AFS_XSTATSCOLL_PERF_INFO` constant. This set of longwords contains information related to the *File Server*'s performance.

5.1.2.1.1 AFS_DISKNAMESIZE *[Value = 32]* Specifies the maximum length for an AFS disk partition, used directly in the definition for the `DiskName` typedef. A `DiskName` appears as part of a `struct ViceDisk`, a group of which appear inside a `struct ViceStatistics`, used for carrying basic *File Server* statistics information.

5.1.2.1.2 AFS_MAX_XSTAT_LONGS *[Value = 1,024]* Defines the maximum size for a *File Server* data collection, as exported via the *RXAFS_GetXStats()* RPC call. It is used directly in the `AFS_CollData` typedef.

5.1.2.1.3 AFS_XSTATSCOLL_CALL_INFO *[Value = 0]* This constant identifies the *File Server*'s data collection containing profiling information on the number of times each of its internal procedures has been called.

Please note that this data collection is *not* supported by the *File Server* at this time. A request for this data collection will result the return of a zero-length array.

5.1.2.1.4 AFS_XSTATSCOLL_PERF_INFO *[Value = 1]* This constant identifies the *File Server*'s data collection containing performance-related information.

5.1.2.1.5 AFS_CollData *[typedef long AFS_CollData<AFS_MAX_XSTAT_LONGS>;]* This typedef is used by *Rxgen* to create a structure used to pass *File Server* data collections to the caller. It resolves into a C typedef statement defining a structure of the same name with the following fields:

Fields

- `u_int AFS_CollData_len` - The number of longwords contained within the data pointed to by the next field.
- `long *AFS_CollData_val` - A pointer to a sequence of `AFS_CollData_len` longwords.

5.1.2.1.6 AFSBulkStats *[typedef AFSFetchStatus AFSBulkStats<AFSCBMAX>;]* This typedef is used by *Rxgen* to create a structure used to pass a set of statistics structures, as described in the *RXAFS_BulkStatus* documentation in Section 5.1.3.21. It resolves into a C typedef statement defining a structure of the same name with the following fields:

Fields

- `u_int AFSBulkStats_len` - The number of `struct AFSFetchStatus` units contained within the data to which the next field points.
- `AFSFetchStatus *AFSBulkStats_val` - This field houses pointer to a sequence of `AFSBulkStats_len` units of type `struct AFSFetchStatus`.

5.1.2.1.7 DiskName *[typedef opaque DiskName[AFS_DISKNAMESIZE];]* The name of an AFS disk partition. This object appears as a field within a `struct ViceDisk`, a group of which appear inside a `struct ViceStatistics`, used for carrying basic *File Server* statistics information. The term *opaque* appearing above inidcates that the object being defined will be treated as an undifferentiated string of bytes.

5.1.2.1.8 ViceLockType *[typedef long ViceLockType;]* This defines the format of a lock used internally by the *Cache Manager*. The content of these locks is accessible via the *RXAFSCB_GetLock()* RPC function. An isomorphic and more refined version of the lock structure used by the *Cache Manager*, mapping directly to this definition, is `struct AFSDBLockDesc`, defined in Section 4.3.1.

5.1.2.2 struct AFSVolSync

This structure conveys volume synchronization information across many of the *File Server* RPC interface calls, allowing something akin to a 'whole-volume callback" on read-only volumes.

Fields

`unsigned long spare1 ... spare6` - The first longword, `spare1`, contains the volume's creation date. The rest are currently unused.

5.1.2.3 struct AFSFetchStatus

This structure defines the information returned when a file system object is fetched from a *File Server*.

Fields

`unsigned long InterfaceVersion` - RPC interface version, defined to be 1.

`unsigned long FileType` - Distinguishes the object as either a file, directory, symlink, or invalid.

`unsigned long LinkCount` - Number of links to this object.

`unsigned long Length` - Length in bytes.

`unsigned long DataVersion` - Object's data version number.

`unsigned long Author` - Identity of the object's author.

`unsigned long Owner` - Identity of the object's owner.

`unsigned long CallerAccess` - The set of access rights computed for the caller on this object.

`unsigned long AnonymousAccess` - The set of access rights computed for any completely unauthenticated principal.

`unsigned long UnixModeBits` - Contents of associated UNIX mode bits.

`unsigned long ParentVnode` - Vnode for the object's parent directory.

`unsigned long ParentUnique` - Uniquifier field for the parent object.

`unsigned long SegSize` - *(Not implemented).*

`unsigned long ClientModTime` - Time when the caller last modified the data within the object.

`unsigned long ServerModTime` - Time when the server last modified the data within the object.

`unsigned long Group` - *(Not implemented).*

`unsigned long SyncCounter` - *(Not implemented).*

`unsigned long spare1 ... spare4` - Spares.

5.1.2.4 struct AFSStoreStatus

This structure is used to convey which of a file system object's status fields should be set, and their new values. Several *File Server* RPC calls, including *RXAFS_StoreStatus()*, *RXAFS_CreateFile()*, *RXAFS_SymLink()*, *RXAFS_MakeDir()*, and the streamed call to store file data onto the *File Server*.

Fields

`unsigned long Mask` - Bit mask, specifying which of the following fields should be assigned into the *File Server*'s status block on the object.

`unsigned long ClientModTime` - The time of day that the object was last modified.

`unsigned long Owner` - The principal identified as the owner of the file system object.

`unsigned long Group` - *(Not implemented).*

`unsigned long UnixModeBits` - The set of associated UNIX mode bits.

`unsigned long SegSize` - *(Not implemented).*

5.1.2.5 struct ViceDisk

This structure occurs in `struct ViceStatistics`, and describes the characteristics and status of a disk partition used for AFS storage.

Fields

`long BlocksAvailable` - Number of 1 Kbyte disk blocks still available on the partition.

`long TotalBlocks` - Total number of disk blocks in the partition.

`DiskName Name` - The human-readable character string name of the disk partition (e.g., `/vicepa`).

5.1.2.6 struct ViceStatistics

This is the *File Server* statistics structure returned by the *RXAFS_GetStatistics()* RPC call.

Fields

`unsigned long CurrentMsgNumber` - *Not used*

`unsigned long OldestMsgNumber` - *Not used*

`unsigned long CurrentTime` - Time of day, as understood by the *File Server*.

`unsigned long BootTime` - Kernel's boot time.

`unsigned long StartTime` - Time when the *File Server* started up.

`long CurrentConnections` - Number of connections to *Cache Manager* instances.

`unsigned long TotalViceCalls` - Count of all calls made to the RPC interface.

`unsigned long TotalFetchs` - Total number of fetch operations, either status or data, performed.

`unsigned long FetchDatas` - Total number of data fetch operations exclusively.

`unsigned long FetchedBytes` - Total number of bytes fetched from the *File Server* since it started up.

`long FetchDataRate` - Result of dividing the `FetchedBytes` field by the number of seconds the *File Server* has been running.

`unsigned long TotalStores` - Total number of store operations, either status or data, performed.

`unsigned long StoreDatas` - Total number of data store operations exclusively.

`unsigned long StoredBytes` - Total number of bytes stored to the *File Server* since it started up.

`long StoreDataRate` - The result of dividing the `StoredBytes` field by the number of seconds the *File Server* has been running.

`unsigned long TotalRPCBytesSent` - *Outdated*

`unsigned long TotalRPCBytesReceived` - *Outdated*

`unsigned long TotalRPCPacketsSent` - *Outdated*

`unsigned long TotalRPCPacketsReceived` - *Outdated*

`unsigned long TotalRPCPacketsLost` - *Outdated*

`unsigned long TotalRPCBogusPackets` - *Outdated*

`long SystemCPU` - Result of reading from the kernel the usage times attributed to system activities.

`long UserCPU` - Result of reading from the kernel the usage times attributed to user-level activities.

`long NiceCPU` - Result of reading from the kernel the usage times attributed to *File Server* activities that have been *nice()*d (i.e., run at a lower priority).

`long IdleCPU` - Result of reading from the kernel the usage times attributed to idling activities.

`long TotalIO` - Summary of the number of bytes read/written from the disk.

`long ActiveVM` - Amount of virtual memory used by the *File Server*.

`long TotalVM` - Total space available on disk for virtual memory activities.

`long EtherNetTotalErrors` - *Not used.*

`long EtherNetTotalWrites` - *Not used.*

`long EtherNetTotalInterupts` - *Not used.*

`long EtherNetGoodReads` - *Not used.*

`long EtherNetTotalBytesWritten` - *Not used.*

`long EtherNetTotalBytesRead` - *Not used.*

`long ProcessSize` - The size of the *File Server*'s data space in 1 Kbyte chunks.

`long WorkStations` - The total number of client *Cache Manager*s (workstations) for which information is held by the *File Server*.

`long ActiveWorkStations` - The total number of client *Cache Manager*s (workstations) that have recently interacted with the *File Server*. This number is strictly less than or equal to the `WorkStations` field.

`long Spare1 ... Spare8` - *Not used.*

`ViceDisk Disk1 ... Disk10` - Statistics concerning up to 10 disk partitions used by the *File Server*. These records keep information on *all* partitions, not just partitions reserved for AFS storage.

5.1.2.7 struct afs_PerfStats

This is the structure corresponding to the `AFS_XSTATSCOLL_PERF_INFO` data collection that is defined by the *File Server* (see Section 5.1.2.1.4). It is accessible via the *RX-AFS_GetXStats()* interface routine, as defined in Section 5.1.3.26.

The fields within this structure fall into the following classifications:

- Number of requests for the structure.
- Vnode cache information.
- Directory package numbers.
- Rx information.
- Host module fields
- Spares.

Please note that the *Rx* fields represent the contents of the `rx_stats` structure maintained by *Rx* RPC facility itself. Also, a full description of all the structure's fields is not possible here. For example, the reader is referred to the companion *Rx* document for further clarification on the *Rx*-related fields within `afs_PerfStats`.

Fields

`long numPerfCalls` - Number of performance collection calls received.

`long vcache_L_Entries` - Number of entries in large (directory) vnode cache.

`long vcache_L_Allocs` - Number of allocations for the large vnode cache.

`long vcache_L_Gets` - Number of get operations for the large vnode cache.

`long vcache_L_Reads` - Number of reads performed on the large vnode cache.

`long vcache_L_Writes` - Number of writes executed on the large vnode.cache.

long `vcache_S_Entries` - Number of entries in the small (file) vnode cache.

long `vcache_S_Allocs` - Number of allocations for the small vnode cache.

long `vcache_S_Gets` - Number of get operations for the small vnode cache.

long `vcache_S_Reads` - Number of reads performed on the small vnode cache.

long `vcache_S_Writes` - Number of writes executed on the small vnode cache.

long `vcache_H_Entries` - Number of entries in the header of the vnode cache.

long `vcache_H_Gets` - Number of get operations on the header of the vnode cache.

long `vcache_H_Replacements` - Number of replacement operations on the header of the vnode cache.

long `dir_Buffers` - Number of directory package buffers in use.

long `dir_Calls` - Number of read calls made to the directory package.

long `dir_IOs` - Number of directory I/O operations performed.

long `rx_packetRequests` - Number of *Rx* packet allocation requests.

long `rx_noPackets_RcvClass` - Number of failed packet reception requests.

long `rx_noPackets_SendClass` - Number of failed packet transmission requests.

long `rx_noPackets_SpecialClass` - Number of "special" *Rx* packet rquests.

long `rx_socketGreedy` - Did setting the *Rx* socket to `SO_GREEDY` succeed?

long `rx_bogusPacketOnRead` - Number of short packets received.

long `rx_bogusHost` - Latest host address from bogus packets.

long `rx_noPacketOnRead` - Number of attempts to read a packet when one was not physically available.

long `rx_noPacketBuffersOnRead` - Number of packets dropped due to buffer shortages.

long `rx_selects` - Number of selects performed, waiting for a packet arrival or a timeout.

long `rx_sendSelects` - Number of selects forced upon a send.

long `rx_packetsRead_RcvClass` - Number of packets read belonging to the "Rcv" class.

long `rx_packetsRead_SendClass` - Number of packets read that belong to the "Send" class.

long `rx_packetsRead_SpecialClass` - Number of packets read belonging to the "Special" class.

long `rx_dataPacketsRead` - Number of unique data packets read off the wire.

long `rx_ackPacketsRead` - Number of acknowledgement packets read.

long `rx_dupPacketsRead` - Number of duplicate data packets read.

long `rx_spuriousPacketsRead` - Number of inappropriate packets read.

long `rx_packetsSent_RcvClass` - Number of packets sent belonging to the "Rcv" class.

long `rx_packetsSent_SendClass` - Number of packets sent belonging to the "Send" class.

long `rx_packetsSent_SpecialClass` - Number of packets sent belonging to the "Special" class.

long `rx_ackPacketsSent` - Number of acknowledgement packets sent.

long `rx_pingPacketsSent` - Number of ping packets sent.

long `rx_abortPacketsSent` - Number of abort packets sent.

long `rx_busyPacketsSent` - Number of busy packets sent.

long `rx_dataPacketsSent` - Number of unique data packets sent.

long `rx_dataPacketsReSent` - Number of retransmissions sent.

long `rx_dataPacketsPushed` - Number of retransmissions pushed by a `NACK`.

long `rx_ignoreAckedPacket` - Number of packets whose `acked` flag was set at *rxi_Start()* time.

long `rx_totalRtt_Sec` - Total round trip time in seconds.

long `rx_totalRtt_Usec` - Microsecond portion of the total round trip time,

long `rx_minRtt_Sec` - Minimum round trip time in seconds.

long `rx_minRtt_Usec` - Microsecond portion of minimal round trip time.

long `rx_maxRtt_Sec` - Maximum round trip time in seconds.

long `rx_maxRtt_Usec` - Microsecond portion of maximum round trip time.

long `rx_nRttSamples` - Number of round trip samples.

long `rx_nServerConns` - Total number of server connections.

long `rx_nClientConns` - Total number of client connections.

long `rx_nPeerStructs` - Total number of peer structures.

long `rx_nCallStructs` - Total number of call structures.

long `rx_nFreeCallStructs` - Total number of call structures residing on the free list.

long `host_NumHostEntries` - Number of host entries.

long `host_HostBlocks` - Number of blocks in use for host entries.

long `host_NonDeletedHosts` - Number of non-deleted host entries.

long `host_HostsInSameNetOrSubnet` - Number of host entries in the same [sub]net as the *File Server*.

long `host_HostsInDiffSubnet` - Number of host entries in a different subnet as the *File Server*.

long `host_HostsInDiffNetwork` - Number of host entries in a different network entirely as the *File Server*.

long `host_NumClients` - Number of client entries.

long `host_ClientBlocks` - Number of blocks in use for client entries.

long `spare[32]` - Spare fields, reserved for future use.

5.1.2.8 struct AFSFetchVolumeStatus

The results of asking the *File Server* for status information concerning a particular volume it hosts.

Fields

long `Vid` - Volume ID.

long `ParentId` - Volume ID in which the given volume is "primarily" mounted. This is used to properly resolve `pwd` operations, as a volume may be mounted simultaneously at multiple locations.

char `Online` - Is the volume currently online and fully available?

char `InService` - This field records whether the volume is currently in service. It is indistinguishable from the `Blessed` field,

char `Blessed` - See the description of the `InService` field immediately above.

char `NeedsSalvage` - Should this volume be salvaged (run through a consistency-checking procedure)?

long `Type` - The classification of this volume, namely a read/write volume (`RWVOL` = 0), read-only volume (`ROVOL` = 1), or backup volume (`BACKVOL` = 2).

long `MinQuota` - Minimum number of 1 Kbyte disk blocks to be set aside for this volume. Note: this field is not currently set or accessed by any AFS agents.

`long MaxQuota` - Maximum number of 1 Kbyte disk blocks that may be occupied by this volume.

`long BlocksInUse` - Number of 1 Kbyte disk blocks currently in use by this volume.

`long PartBlocksAvail` - Number of available 1 Kbyte blocks currently unused in the volume's partition.

`long PartMaxBlocks` - Total number of blocks, in use or not, for the volume's partition.

5.1.2.9 struct AFSStoreVolumeStatus

This structure is used to convey which of a file system object's status fields should be set, and their new values. The *RXAFS_SetVolumeStatus()* RPC call is the only user of this structure.

Fields

`long Mask` - Bit mask to determine which of the following two fields should be stored in the centralized status for a given volume.

`long MinQuota` - Minimum number of 1 Kbyte disk blocks to be set aside for this volume.

`long MaxQuota` - Maximum number of 1 Kbyte disk blocks that may be occupied by this volume.

5.1.2.10 struct AFSVolumeInfo

This field conveys information regarding a particular volume through certain *File Server* RPC interface calls. For information regarding the different volume types that exist, please consult the companion document, AFS-3 Programmer's Reference: *Volume Server/Volume Location Server* Interface.

Fields

`unsigned long Vid` - Volume ID.

`long Type` - Volume type (see `struct AFSFetchVolumeStatus` in Section 5.1.2.8 above).

`unsigned long Type0 ... Type4` - The volume IDs for the possible volume types in existance for this volume.

`unsigned long ServerCount` - The number of *File Server* machines on which an instance of this volume is located.

`unsigned long Server0 ... Server7` - Up to 8 IP addresses of *File Server* machines hosting an instance on this volume. The first `ServerCount` of these fields hold valid server addresses.

`unsigned short Port0 ... Port7` - Up to 8 UDP port numbers on which operations on this volume should be directed. The first `ServerCount` of these fileds hold valid port identifiers.

5.1.3 Non-Streamed Function Calls

The following is a description of the *File Server* RPC interface routines that utilize only parameters with fixed maximum lengths. The majority of the *File Server* calls fall into this suite, with only a handful using streaming techniques to pass objects of unbounded size between a *File Server* and *Cache Manager*.

Each function is labeled with an *opcode* number. This is the low-level numerical identifier for the function, and appears in the set of network packets constructed for the RPC call.

5.1.3.1 RXAFS_FetchACL — Fetch the ACL associated with the given AFS file identifier

```
int RXAFS_FetchACL(IN struct rx_connection *a_rxConnP,
                   IN AFSFid *a_dirFidP,
                   OUT AFSOpaque *a_ACLP,
                   OUT AFSFetchStatus *a_dirNewStatP,
                   OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 131] Fetch the ACL for the directory identified by *a_dirFidP*, placing it in the space described by the opaque structure to which *a_ACLP* points. Also returned is the given directory's status, written to *a_dirNewStatP*. An ACL may thus take up at most `AFSOPAQUEMAX` (1,024) bytes, since this is the maximum size of an `AFSOpaque`.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller is not permitted to perform this operation.

`EINVAL` An internal error in looking up the client record was encountered, or an invalid fid was provided.

`VICETOKENDEAD` Caller's authentication token has expired.

5.1.3.2 RXAFS_FetchStatus — Fetch the status information regarding a given file system object

```
int RXAFS_FetchStatus(IN struct rx_connection *a_rxConnP,
                      IN AFSFid *a_fidToStatP,
                      OUT AFSFetchStatus *a_currStatP,
                      OUT AFSCallBack *a_callBackP,
                      OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 132] Fetch the current status information for the file or directory identified by *a_fidToStatP*, placing it into the area to which *a_currStatP* points. If the object resides in a read/write volume, then the related callback information is returned in *a_callBackP*.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller is not permitted to perform this operation.

`EINVAL` An internal error in looking up the client record was encountered, or an invalid fid was provided.

`VICETOKENDEAD` Caller's authentication token has expired.

5.1.3.3 **RXAFS_StoreACL** — Associate the given ACL with the named directory

```
int RXAFS_StoreACL(IN struct rx_connection *a_rxConnP,
                   IN AFSOpaque *a_ACLToStoreP,
                   IN AFSFid *a_dirFidP,
                   OUT AFSFetchStatus *a_dirNewStatP,
                   OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 134] Store the ACL information to which *a_ACLToStoreP* points to the *File Server*, associating it with the directory identified by *a_dirFidP*. The resulting status information for the *a_dirFidP* directory is returned in *a_dirNewStatP*. Note that the ACL supplied via *a_ACLToStoreP* may be at most `AFSOPAQUEMAX` (1,024) bytes long, since this is the maximum size accommodated by an `AFSOpaque`.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller is not permitted to perform this operation.

`E2BIG` The given ACL is too large.

`EINVAL` The given ACL could not translated to its on-disk format.

5.1.3.4 RXAFS_StoreStatus — Store the given status information for the specified file

```
int RXAFS_StoreStatus(IN struct rx_connection *a_rxConnP,
                      IN AFSFid *a_fidP,
                      IN AFSStoreStatus *a_currStatusP,
                      OUT AFSFetchStatus *a_srvStatusP,
                      OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 135] Store the status information to which *a_currStatusP* points, associating it with the file identified by *a_fidP*. All outstanding callbacks on this object are broken. The resulting status structure stored at the *File Server* is returned in *a_srvStatusP*.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller is not permitted to perform this operation.

`EINVAL` An internal error in looking up the client record was encountered, or an invalid fid was provided, or an attempt was made to change the mode of a symbolic link.

`VICETOKENDEAD` Caller's authentication token has expired.

5.1.3.5 RXAFS_RemoveFile — Delete the given file

```
int RXAFS_RemoveFile(IN struct rx_connection *a_rxConnP,
                     IN AFSFid *a_dirFidP,
                     IN char *a_name<AFSNAMEMAX>,
                     OUT AFSFetchStatus *a_srvStatusP,
                     OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 136] Destroy the file named `a_name` within the directory identified by `a_dirFidP`. All outstanding callbacks on this object are broken. The resulting status structure stored at the *File Server* is returned in *a_srvStatusP*.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller is not permitted to perform this operation.

`EINVAL` An internal error in looking up the client record was encountered, or an invalid fid was provided, or an attempt was made to remove "." or "..".

`EISDIR` The target of the deletion was supposed to be a file, but it is really a directory.

`ENOENT` The named file was not found.

`ENOTDIR` The `a_dirFidP` parameter references an object which is not a directory, or the deletion target is supposed to be a directory but is not.

`ENOTEMPTY` The target directory being deleted is not empty.

`VICETOKENDEAD` Caller's authentication token has expired.

5.1.3.6 RXAFS_CreateFile — Create the given file

```
int RXAFS_CreateFile(IN struct rx_connection *a_rxConnP,
                     IN AFSFid *DirFid,
                     IN char *Name,
                     IN AFSStoreStatus *InStatus,
                     OUT AFSFid *OutFid,
                     OUT AFSFetchStatus *OutFidStatus,
                     OUT AFSFetchStatus *OutDirStatus,
                     OUT AFSCallBack *CallBack,
                     OUT AFSVolSync *a_volSyncP)
```

associated with the new file.

Description

[Opcode 137] This call is used to create a file, but *not* for creating a directory or a symbolic link. If this call succeeds, it is the *Cache Manager*'s responsibility to either create an entry locally in the directory specified by *DirFid* or to invalidate this directory's cache entry.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller is not permitted to perform this operation.

`EINVAL` An internal error in looking up the client record was encountered, or an invalid fid or name was provided.

`ENOTDIR` The `DirFid` parameter references an object which is not a directory.

`VICETOKENDEAD` Caller's authentication token has expired.

5.1.3.7 RXAFS_Rename — Rename the specified file in the given directory

```
int RXAFS_Rename(IN struct rx_connection *a_rxConnP,
                 IN AFSFid *a_origDirFidP,
                 IN char *a_origNameP,
                 IN AFSFid *a_newDirFidP,
                 IN char *a_newNameP,
                 OUT AFSFetchStatus *a_origDirStatusP,
                 OUT AFSFetchStatus *a_newDirStatusP,
                 OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 138] Rename file *a_origNameP* in the directory identified by *a_origDirFidP*. Its new name is to be *a_newNameP*, and it will reside in the directory identified by *a_newDirFidP*. Each of these names must be no more than `AFSNAMEMAX` (256) characters long. The status of the original and new directories after the rename operation completes are deposited in *a_origDirStatusP* and *a_newDirStatusP* respectively. Existing callbacks are broken for all files and directories involved in the operation.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

- `EACCES` New file exists but user doesn't have `Delete` rights in the directory.
- `EINVAL` Name provided is invalid.
- `EISDIR` Original object is a file and new object is a directory.
- `ENOENT` The object to be renamed doesn't exist in the parent directory.
- `ENOTDIR` Original object is a directory and new object is a file.
- `EXDEV` Rename attempted across a volume boundary, or create a pathname loop, or hard links exist to the file.

5.1.3.8 RXAFS_Symlink — Create a symbolic link

```
int RXAFS_Symlink(IN struct rx_connection *a_rxConnP,
                  IN AFSFid *a_dirFidP,
                  IN char *a_nameP,
                  IN char *a_linkContentsP,
                  IN AFSStoreStatus *a_origDirStatP,
                  OUT AFSFid *a_newFidP,
                  OUT AFSFetchStatus *a_newFidStatP,
                  OUT AFSFetchStatus *a_newDirStatP,
                  OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 139] Create a symbolic link named *a_nameP* in the directory identified by *a_dirFidP*. The text of the symbolic link is provided in *a_linkContentsP*, and the desired status fields for the symbolic link given by *a_origDirStatP*. The name offered in *a_nameP* must be less than `AFSNAMEMAX` (256) characters long, and the text of the link to which *a_linkContentsP* points must be less than `AFSPATHMAX` (1,024) characters long. Once the symbolic link has been successfully created, its file identifier is returned in *a_newFidP*. Existing callbacks to the *a_dirFidP* directory are broken before the symbolic link creation completes. The status fields for the symbolic link itself and its parent's directory are returned in *a_newFidStatP* and *a_newDirStatP* respectively.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES`	The caller does not have the necessary access rights.
`EINVAL`	Illegal symbolic link name provided.

5.1.3.9 RXAFS_Link — Create a hard link

```
int RXAFS_Link(IN struct rx_connection *a_rxConnP,
               IN AFSFid *a_dirFidP,
               IN char *a_nameP,
               IN AFSFid *a_existingFidP,
               OUT AFSFetchStatus *a_newFidStatP,
               OUT AFSFetchStatus *a_newDirStatP,
               OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 140] Create a hard link named *a_nameP* in the directory identified by *a_dirFidP*. The file serving as the basis for the hard link is identified by *existingFidP*. The name offered in *a_nameP* must be less than `AFSNAMEMAX` (256) characters long. Existing callbacks to the *a_dirFidP* directory are broken before the hard link creation completes. The status fields for the file itself and its parent's directory are returned in *a_newFidStatP* and *a_newDirStatP* respectively.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

- `EACCES` The caller does not have the necessary access rights.
- `EISDIR` An attempt was made to create a hard link to a directory.
- `EXDEV` Hard link attempted across directories.

5.1.3.10 RXAFS_MakeDir — Create a directory

```
int RXAFS_MakeDir(IN struct rx_connection *a_rxConnP,
                  IN AFSFid *a_parentDirFid,P
                  IN char *a_newDirNameP,
                  IN AFSStoreStatus *a_currStatP,
                  OUT AFSFid *a_newDirFidP,
                  OUT AFSFetchStatus *a_dirFidStatP,
                  OUT AFSFetchStatus *a_parentDirStatP,
                  OUT AFSCallBack *a_newDirCallBackP,
                  OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 141] Create a directory named *a_newDirNameP* within the directory identified by *a_parentDirFidP*. The initial status fields for the new directory are provided in *a_currStatP*. The new directory's name must be less than `AFSNAMEMAX` (256) characters long. The new directory's ACL is inherited from its parent. Existing callbacks on the parent directory are broken before the creation completes. Upon successful directory creation, the new directory's file identifier is returned in *a_newDirFidP*, and the resulting status information for the new and parent directories are stored in *a_dirFidStatP* and *a_parentDirStatP* respectively. In addition, a callback for the new directory is returned in *a_newDirCallBackP*.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller does not have the necessary access rights.

`EINVAL` The directory name provided is unacceptable.

5.1.3.11 RXAFS_RemoveDir — Remove a directory

```
int RXAFS_RemoveDir(IN struct rx_connection *a_rxConnP,
                    IN AFSFid *a_parentDirFidP,
                    IN char *a_dirNameP,
                    OUT AFSFetchStatus *a_newParentDirStatP,
                    OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 142] Remove the directory named *a_dirNameP* from within its parent directory, identified by *a_parentDirFid*. The directory being removed must be empty, and its name must be less than AFSNAMEMAX (256) characters long. Existing callbacks to the directory being removed and its parent directory are broken before the deletion completes. Upon successful deletion, the status fields for the parent directory are returned in *a_newParentDirStatP*.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

EACCES The caller does not have the necessary access rights.

5.1.3.12 **RXAFS_GetStatistics** — Get common *File Server* statistics

```
int RXAFS_GetStatistics(IN struct rx_connection *a_rxConnP,
                        OUT ViceStatistics *a_FSInfoP)
```

Description

[Opcode 146] Fetch the structure containing a set of common *File Server* statistics. These numbers represent accumulated readings since the time the *File Server* last restarted. For a full description of the individual fields contained in this structure, please see Section 5.1.2.6.

Rx connection information for the related *File Server* is contained in *a_rxConnP*.

Error Codes

`---` No error codes generated.

5.1.3.13 **RXAFS_GiveUpCallBacks** — Ask the *File Server* to break the given set of callbacks on the corresponding set of file identifiers

```
int RXAFS_GiveUpCallBacks(IN struct rx_connection *a_rxConnP,
                          IN AFSCBFids *a_fidArrayP,
                          IN AFSCBs *a_callBackArrayP)
```

Description

[Opcode 147] Given an array of up to `AFSCBMAX` file identifiers in *a_fidArrayP* and a corresponding number of callback structures in *a_callBackArrayP*, ask the *File Server* to remove these callbacks from its register. Note that this routine only affects callbacks outstanding on the given set of files for the host issuing the *RXAFS_GiveUpCallBacks* call. Callback promises made to other machines on any or all of these files are not affected.

Rx connection information for the related *File Server* is contained in *a_rxConnP*.

Error Codes

`EINVAL` More file identifiers were provided in the *a_fidArrayP* than callbacks in the *a_callBackArray*.

5.1.3.14 RXAFS_GetVolumeInfo — Get information about a volume given its name

```
int RXAFS_GetVolumeInfo(IN struct rx_connection *a_rxConnP,
                        IN char *a_volNameP,
                        OUT VolumeInfo *a_volInfoP)
```

Description

[Opcode 148] Ask the given *File Server* for information regarding a volume whose name is *a_volNameP*. The volume name must be less than `AFSNAMEMAX` characters long, and the volume itself must reside on the *File Server* being probed.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Please note that definitions for the error codes with `VL_` prefixes may be found in the *vlserver.h* include file

Error Codes

`1` Could not contact any of the corresponding *Volume Location Servers*.

`VL_BADNAME` An improperly-formatted volume name provided.

`VL_ENTDELETED` An entry was found for the volume, reporting that the volume has been deleted.

`VL_NOENT` The given volume was not found.

5.1.3.15 RXAFS_GetVolumeStatus — Get basic status information for the named volume

```
int RXAFS_GetVolumeStatus(IN struct rx_connection *a_rxConnP,
                          IN long a_volIDP,
                          OUT AFSFetchVolumeStatus *a_volFetchStatP,
                          OUT char *a_volNameP,
                          OUT char *a_offLineMsgP,
                          OUT char *a_motdP)
```

Description

[Opcode 149] Given the numeric volume identifier contained in *a_volIDP*, fetch the basic status information corresponding to that volume. This status information is stored into *a_volFetchStatP*. A full description of this status structure is found in Section 5.1.2.8. In addition, three other facts about the volume are returned. The volume's character string name is placed into *a_volNameP*. This name is guaranteed to be less than `AFSNAMEMAX` characters long. The volume's offline message, namely the string recording why the volume is off-line (if it is), is stored in *a_offLineMsgP* . Finally, the volume's "Message of the Day" is placed in *a_motdP*. Each of the character strings deposited into *a_offLineMsgP* and *a_motdP* is guaranteed to be less than `AFSOPAQUEMAX` (1,024) characters long.

Rx connection information for the related *File Server* is contained in *a_rxConnP*.

Error Codes

`EACCES` The caller does not have the necessary access rights.

`EINVAL` A volume identifier of zero was specified.

5.1.3.16 RXAFS_SetVolumeStatus — Set the basic status information for the named volume

```
int RXAFS_SetVolumeStatus(IN struct rx_connection *a_rxConnP,
                          IN long a_volIDP,
                          IN AFSStoreVolumeStatus *a_volStoreStatP,
                          IN char *a_volNameP,
                          IN char *a_offLineMsgP,
                          IN char *a_motdP)
```

Description

[Opcode 150] Given the numeric volume identifier contained in *a_volIDP*, set that volume's basic status information to the values contained in *a_volStoreStatP*. A full description of the fields settable by this call, including the necessary masking, is found in Section 5.1.2.9. In addition, three other items relating to the volume may be set. Non-null character strings found in *a_volNameP*, *a_offLineMsgP*, and *a_motdP* will be stored in the volume's printable name, off-line message, and "Message of the Day" fields respectively. The volume name provided must be less than `AFSNAMEMAX` (256) characters long, and the other two strings must be less than `AFSOPAQUEMAX` (1,024) characters long each.

Rx connection information for the related *File Server* is contained in *a_rxConnP*.

Error Codes

`EACCES` The caller does not have the necessary access rights.

`EINVAL` A volume identifier of zero was specified.

5.1.3.17 **RXAFS_GetRootVolume** — Return the name of the root volume for the file system

```
int RXAFS_GetRootVolume(IN struct rx_connection *a_rxConnP,
                        OUT char *a_rootVolNameP)
```

Description

[Opcode 151] Fetch the name of the volume which serves as the root of the AFS file system and place it into *a_rootVolNameP*. This name will always be less than `AFSNAMEMAX` characters long. Any *File Server* will respond to this call, not just the one hosting the root volume. The queried *File Server* first tries to discover the name of the root volume by reading from the `/usr/afs/etc/RootVolume` file on its local disks. If that file doesn't exist, then it will return the default value, namely "`root.afs`".

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`---` No error codes generated.

5.1.3.18 RXAFS_CheckToken — (Obsolete) Check that the given user identifier matches the one in the supplied authentication token

```
int RXAFS_CheckToken(IN struct rx_connection *a_rxConnP,
                     IN long ViceId,
                     IN AFSOpaque *token)
```

Description

[Opcode 152] This function only works for the now-obsolete RPC facility used by AFS, *R*. For modern systems using the *Rx* RPC mechanism, we always get an error return from this routine.

Rx connection information for the related *File Server* is contained in *a_rxConnP*.

Error Codes

`ECONNREFUSED` Always returned on *Rx* connections.

5.1.3.19 RXAFS_GetTime — Get the *File Server*'s time of day

```
int RXAFS_GetTime(IN struct rx_connection *a_rxConnP,
                  OUT unsigned long *a_secondsP,
                  OUT unsigned long *a_uSecondsP)
```

Description

[Opcode 153] Get the current time of day from the *File Server* specified in the *Rx* connection information contained in *a_rxConnP*. The time is returned in elapsed seconds (*a_secondsP*) and microseconds (*a_uSecondsP*) since that standard UNIX "start of the world".

Error Codes

--- No error codes generated.

5.1.3.20 RXAFS_NGetVolumeInfo — Get information about a volume given its name

```
int RXAFS_NGetVolumeInfo(IN struct rx_connection *a_rxConnP,
                         IN char *a_volNameP,
                         OUT AFSVolumeInfo *a_volInfoP)
```

Description

[Opcode 154] This function is identical to *RXAFS_GetVolumeInfo()* (see Section 5.1.3.14), except that it returns a `struct AFSVolumeInfo` instead of a `struct VolumeInfo`. The basic difference is that `struct AFSVolumeInfo` also carries an accompanying UDP port value for each *File Server* listed in the record.

5.1.3.21 RXAFS_BulkStatus — Fetch the status information regarding a set of given file system objects

```
int RXAFS_BulkStatus(IN struct rx_connection *a_rxConnP,
                     IN AFSCBFids *a_fidToStatArrayP,
                     OUT AFSBulkStats *a_currStatArrayP,
                     OUT AFSCBs *a_callBackArrayP,
                     OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 155] This routine is identical to *RXAFS_FetchStatus()* as described in Section 5.1.3.2, except for the fact that it allows the caller to ask for the current status fields for a set of up to `AFSCBMAX` (50) file identifiers at once.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller does not have the necessary access rights.

`EINVAL` The number of file descriptors for which status information was requested is illegal.

5.1.3.22 RXAFS_SetLock — Set an advisory lock on the given file identifier

```
int RXAFS_SetLock(IN struct rx_connection *a_rxConnP,
                  IN AFSFid *a_fidToLockP,
                  IN ViceLockType a_lockType,
                  OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 156] Set an advisory lock on the file identified by a_fidToLockP. There are two types of locks that may be specified via *a_lockType*: `LockRead` and `LockWrite`. An advisory lock times out after `AFS_LOCKWAIT` (5) minutes, and must be extended in order to stay in force (see *RXAFS_ExtendLock()*, Section 5.1.3.23).

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller does not have the necessary access rights.

`EINVAL` An illegal lock type was specified.

`EWOULDBLOCK` The lock was already incompatibly granted to another party.

5.1.3.23 **RXAFS_ExtendLock** — Extend an advisory lock on a file

```
int RXAFS_ExtendLock(IN struct rx_connection *a_rxConnP,
                     IN AFSFid *a_fidToBeExtendedP,
                     OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 157] Extend the advisory lock that has already been granted to the caller on the file identified by *a_fidToBeExtendedP*.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EINVAL` The caller does not already have the given file locked.

5.1.3.24 RXAFS_ReleaseLock — Release the advisory lock on a file

```
int RXAFS_ReleaseLock(IN struct rx_connection *a_rxConnP,
                      IN AFSFid *a_fidToUnlockP,
                      OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 158] Release the advisory lock held on the file identified by *a_fidToUnlockP*. If this was the last lock on this file, the *File Server* will break all existing callbacks to this file.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller does not have the necessary access rights.

5.1.3.25 RXAFS_XStatsVersion — Get the version number associated with the *File Server*'s extended statistics structure

```
int RXAFS_XStatsVersion(IN struct rx_connection *a_rxConnP,
                        OUT long *a_versionNumberP)
```

Description

[Opcode 159] This call asks the *File Server* for the current version number of the extended statistics structures it exports (see *RXAFS_GetXStats()*, Section 5.1.3.26). The version number is placed into *a_versionNumberP*.

Rx connection information for the related *File Server* is contained in *a_rxConnP*.

Error Codes

--- No error codes generated.

5.1.3.26 RXAFS_GetXStats — Get the current contents of the specified extended statistics structure

```
int RXAFS_GetXStats(IN struct rx_connection *a_rxConnP,
                    IN long a_clientVersionNumber,
                    IN long a_collectionNumber,
                    OUT long *a_srvVersionNumberP,
                    OUT long *a_timeP,
                    OUT AFS_CollData *a_dataP)
```

Description

[Opcode 160] This function fetches the contents of the specified *File Server* extended statistics structure. The caller provides the version number of the data it expects to receive in *a_clientVersionNumber*. Also provided in *a_collectionNumber* is the numerical identifier for the desired **data collection**. There are currently two of these data collections defined: `AFS_XSTATSCOLL_CALL_INFO`, which is the list of tallies of the number of invocations of internal *File Server* procedure calls, and `AFS_XSTATSCOLL_PERF_INFO`, which is a list of performance-related numbers. The precise contents of these collections are described in Sections 5.1.2.7. The current version number of the *File Server* collections is returned in *a_srvVersionNumberP*, and is always set upon return, even if the caller has asked for a different version. If the correct version number has been specified, and a supported collection number given, then the collection data is returned in *a_dataP*. The time of collection is also returned, being placed in *a_timeP*.

Rx connection information for the related *File Server* is contained in *a_rxConnP*.

Error Codes

`---` No error codes are generated.

5.1.4 Streamed Function Calls

There are two *streamed* functions in the *File Server* RPC interface, used to fetch and store arbitrary amounts of data from a file. While some non-streamed calls pass such variable-length objects as `struct AFSCBFids`, these objects have a pre-determined maximum size.

The two streamed RPC functions are also distinctive in that their single *Rxgen* declarations generate not one but *two* client-side stub routines. The first is used to ship the IN parameters off to the designated *File Server*, and the second to gather the OUT parameters and the error code. If a streamed definition declares a routine named *X_YZ()*, the two resulting stubs will be named *StartX_YZ()* and *EndX_YZ()*. It is the application programmer's job to first invoke *StartX_YZ()*, then manage the unbounded data transfer, then finish up by calling *EndX_YZ()*. The first longword in the unbounded data stream being fetched from a *File Server* contains the number of data bytes to follow. The application then reads the specified number of bytes from the stream.

The following sections describe the four client-side functions resulting from the *FetchData()* and *StoreData()* declarations in the *Rxgen* interface definition file. These are the actual routines the application programmer will include in the client code. For reference, here are the interface definitions that generate these functions. Note that the split keyword is what causes *Rxgen* to generate the separate start and end routines. In each case, the number after the equal sign specifies the function's identifying opcode number. The opcode is passed to the *File Server* by the *StartRXAFS_FetchData()* and *StartRXAFS_StoreData()* stub routines.

```
FetchData(IN  AFSFid *a_fidToFetchP,
          IN  long a_offset,
          IN  long a_lenInBytes,
          OUT AFSFetchStatus *a_fidStatP,
          OUT AFSCallBack *a_callBackP,
          OUT AFSVolSync *a_volSyncP) split = 130;
```

```
StoreData(IN  AFSFid *Fid,
          IN  AFSStoreStatus *InStatus,
          IN  long Pos,
          IN  long Length,
          IN  long FileLength,
          OUT AFSFetchStatus *OutStatus,
          OUT AFSVolSync *a_volSyncP) split = 133;
```

5.1.4.1 StartRXAFS_FetchData — Begin a request to fetch file data

```
int StartRXAFS_FetchData(IN struct rx_call *a_rxCallP,
                         IN AFSFid *a_fidToFetchP,
                         IN long a_offset,
                         IN long a_lenInBytes)
```

Description

Begin a request for *a_lenInBytes* bytes of data starting at byte offset *a_offset* from the file identified by *a_fidToFetchP*. After successful completion of this call, the data stream will make the desired bytes accessible. The first longword in the stream contains the number of bytes to actually follow.

Rx call information to the related *File Server* is contained in *a_rxCallP*.

Error Codes

--- No error codes generated.

5.1.4.2 EndRXAFS_FetchData — Conclude a request to fetch file data

```
int EndRXAFS_FetchData(IN struct rx_call *a_rxCallP,
                       OUT AFSFetchStatus *a_fidStatP,
                       OUT AFSCallBack *a_callBackP,
                       OUT AFSVolSync *a_volSyncP)
```

Description

Conclude a request to fetch file data, as commenced by an *StartRXAFS_FetchData()* invocation. By the time this routine has been called, all of the desired data has been read off the data stream. The status fields for the file from which the data was read are stored in *a_fidStatP*. If the file was from a read/write volume, its callback information is placed in *a_callBackP*.

Rx call information to the related *File Server* is contained in *a_rxCallP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller does not have the necessary access rights.

`EIO` Given file could not be opened or statted on the *File Server*, or there was an error reading the given data off the *File Server*'s disk.

`-31` An *Rx* write into the stream ended prematurely.

5.1.4.3 StartRXAFS_StoreData — Begin a request to store file data

```
int StartRXAFS_StoreData(IN struct rx_call *a_rxCallP,
                         IN AFSFid *a_fidToStoreP,
                         IN AFSStoreStatus *a_fidStatusP,
                         IN long a_offset,
                         IN long a_lenInBytes,
                         IN long a_fileLenInBytes)
```

Description

Begin a request to write *a_lenInBytes* of data starting at byte offset *a_offset* to the file identified by *a_fidToStoreP*, causing that file's length to become *a_fileLenInBytes* bytes. After successful completion of this call, the data stream will be ready to begin accepting the actual data being written.

Rx call information to the related *File Server* is contained in *a_rxCallP*.

Error Codes

--- No error codes generated.

5.1.4.4 EndRXAFS_StoreData — Conclude a request to store file data

```
int EndRXAFS_StoreData(IN struct rx_call *a_rxCallP,
                       OUT AFSFetchStatus *a_fidStatP,
                       OUT AFSCallBack *a_callBackP,
                       OUT AFSVolSync *a_volSyncP)
```

Description

Conclude a request to store file data, as commenced by a *StartRXAFS_StoreData()* invocation. By the time this routine has been called, all of the file data has been inserted into the data stream. The status fields for the file to which the data was written are stored in *a_fidStatP*. All existing callbacks to the given file are broken before the store concludes.

Rx call information to the related *File Server* is contained in *a_rxCallP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller does not have the necessary access rights.

`EISDIR` The file being written to is a symbolic link.

`ENOSPEC` A write to the *File Server*'s file on local disk failed.

`-32` A short read was encountered by the *File Server* on the data stream.

5.1.5 Example of Streamed Function Call Usage

5.1.5.1 Preface

The following code fragment is offered as an example of how to use the streamed *File Server* RPC calls. In this case, a client fetches some amount of data from the given *File Server* and writes it to a local file it uses to cache the information. For simplicity, many issues faced by a true application programmer are *not* addressed here. These issues include locking, managing file chunking, data version number mismatches, volume location, *Rx* connection management, defensive programming (e.g., checking parameters before using them), client-side cache management algorithms, callback management, and full error detection and recovery. Pseudocode is incorporated when appropriate to keep the level of detail reasonable. For further descriptions of some of these details and issues, the reader is referred to such companion documents as *AFS-3 Programmer's Reference: Specification for the Rx Remote Procedure Call Facility*, *AFS-3 Programmer's Reference:Volume Server/Volume Location Server Interface*, and *AFS-3 Programmer's Reference: Architectural Overview.*

A discussion of the methods used within the example code fragment follows immediately afterwards in Section 5.1.5.3.

5.1.5.2 Code Fragment Illustrating Fetch Operation

```
int code;                   /*Return code*/
long bytesRead;             /*Num bytes read from Rx*/
struct myConnInfo *connP; /*Includes Rx conn info*/
struct rx_call *rxCallP;  /*Rx call ptr*/
struct AFSFid *afsFidP;   /*Fid for file to fetch*/
int lclFid;                 /*Fid for local cache file*/
long offsetBytes;           /*Starting fetch offset*/
long bytesToFetch;          /*Num bytes to fetch*/
long bytesFromFS;           /*Num bytes FileServer returns*/
char *fetchBuffP;           /*Buffer to hold stream data*/
int currReadBytes;          /*Num bytes for current read*/

    /*
     * Assume that connP, afsFidP, offsetBytes, lclFid,and
     * bytesToFetch have all been given their desired values.
     */
        .
        .
        .
    rxCallP = rx_NewCall(connP->rxConnP);
    code = StartRXAFS_FetchData(
                rxCallP,        /*Rx call to use*/
```

```
                afsFidP,        /*Fid being fetched from*/
                offsetBytes,    /*Offset in bytes*/
                bytesToFetch);  /*Num bytes wanted*/
    if (code == 0) {
        bytesRead = rx_Read(rxCallP, &bytesFromFS, sizeof(long));
        if (bytesRead != sizeof(long))
            ExitWithError(SHORT_RX_READ);
        bytesFromFS = ntohl(bytesFromFS);
        xmitBuffer = malloc(FETCH_BUFF_BYTES);
        lclFid = open(CacheFileName, O_RDWR, mode);
        pos = lseek(lclFid, offsetBytes, L_SET);
        while (bytesToFetch > 0) {
            currReadBytes =
                (bytesToFetch > FETCH_BUFF_BYTES) ?
                    FETCH_BUFF_BYTES : bytesToFetch;
            bytesRead = rx_Read(rxCallP, fetchBuffP, currReadBytes);
            if (bytesRead != currReadBytes)
                ExitWithError(SHORT_RX_READ);
            code = write(lclFid, fetchBuffP, currReadBytes);
            if (code)
                ExitWithError(LCL_WRITE_FAILED);
            bytesToFetch -= bytesRead;
        } /*Read from the Rx stream*/
        close(lclFid);
    }
    else
        ExitWithError(code);
    code = EndRXAFS_FetchData(
                rxCallP,        /*Rx call to use*/
                fidStatP,       /*Resulting stat fields*/
                fidCallBackP,   /*Resulting callback info*/
                volSynchP);     /*Resulting volume sync info*/
    code = rx_EndCall(rxCallP, code);
    return(code);
        .
        .
        .
```

5.1.5.3 Discussion and Analysis

The opening assumption in this discussion is that all the information required to do the fetch has already been set up. These mandatory variables are the client-side connection information for the *File Server* hosting the desired file, the corresponding AFS file identifier, the byte offset into the file, the number of bytes to fetch, and the identifier for the local file serving as a cached copy.

Given the *Rx* connection information stored in the client's connP record, *rx_NewCall()* is used to create a new *Rx* call to handle this fetch operation. The structure containing this call handle is placed into `rxCallP`. This call handle is used immediately in the invocation of *StartRXAFS_FetchData()*. If this setup call fails, the fragment exits. Upon success, though, the *File Server* will commence writing the desired data into the *Rx* data stream. The *File Server* first writes a single longword onto the stream announcing to the client how many bytes of data will actually follow. The fragment reads this number with its first *rx_Read()* call. Since all *Rx* stream data is written in network byte order, the fragment translates the byte count to its own host byte order first to properly interpret it. Once the number of bytes to appear on the stream is known, the client code proceeds to open the appropriate cache file on its own local disk and seeks to the appropriate spot within it. A buffer into which the stream data will be placed is also created at this time.

The example code then falls into a loop where it reads all of the data from the *File Server* and stores it in the corresponding place in the local cache file. For each iteration, the code decides whether to read a full buffer's worth or the remaining number of bytes, whichever is smaller. After all the data is pulled off the *Rx* stream, the local cache file is closed. At this point, the example finishes off the RPC by calling *EndRXAFS_FetchData()*. This gathers in the required set of `OUT` parameters, namely the status fields for the file just fetched, callback and volume synchronization information, and the overall error code for the streamed routine. The *Rx* call created to perform the fetch is then terminated and cleaned up by invoking *rx_EndCall()*.

5.1.6 Required Caller Functionality

The AFS *File Server* RPC interface was originally designed to interact only with *Cache Manager* agents, and thus made some assumptions about its callers. In particular, the *File Server* expected that the agents calling it would potentially have stored callback state on file system objects, and would have to be periodically pinged in order to garbage-collect its records, removing information on dead client machines. Thus, any entity making direct calls to this interface **must** mimic certain *Cache Manager* actions, and respond to certain *Cache Manager* RPC interface calls.

To be safe, any application calling the *File Server* RPC interface directly should export the entire *Cache Manager* RPC interface. Realistically, though, it will only need to provide stubs for the three calls from this interface that *File Servers* know how to make: *RXAFSCB_InitCallBackState()*, *RXAFSCB_Probe()* and *RXAFSCB_CallBack()*. The very first *File Server* call made by this application will prompt the given *File Server* to call *RXAFSCB_InitCallBackState()*. This informs the application that the *File Server* has no record of its existence and hence this "*Cache Manager*" should clear all callback information for that server. Once the application responds positively to the inital *RX-AFSCB_InitCallBackState()*, the *File Server* will treat it as a bona fide, fully-fledged

Cache Manager, and probe it every so often with *RXAFSCB_Probe()* calls to make sure it is still alive.

5.2 Signal Interface

While the majority of communication with AFS *File Server*s occurs over the RPC interface, some important operations are invoked by sending UNIX signals to the process. This section describes the set of signals recognized by the *File Server* and the actions they trigger upon receipt, as summarized below:

- `SIGQUIT`: Shut down a *File Server*.
- `SIGTSTP`: Upgrade debugging output level.
- `SIGHUP`: Reset debugging output level.
- `SIGTERM`: Generate debugging output specifically concerning open files within the *File Server* process.

5.2.1 `SIGQUIT`: Server Shutdown

Upon receipt of this signal, the *File Server* shuts itself down in an orderly fashion. It first writes a message to the console and to its log file (*/usr/afs/logs/FileLog*) stating that a shutdown has commenced. The *File Server* then flushes all modified buffers and prints out a set of internal statistics, including cache and disk numbers. Finally, each attached volume is taken offline, which means the volume header is written to disk with the appropriate bits set.

In typical usage, human operators do not send the `SIGQUIT` signal directly to the *File Server* in order to affect an orderly shutdown. Rather, the *BOS Server* managing the server processes on that machine issues the signal upon receipt of a properly-authorized shutdown RPC request.

5.2.2 `SIGTSTP`: Upgrade Debugging Level

Arrival of a `SIGTSTP` signal results in an increase of the debugging level used by the *File Server*. The routines used for writing to log files are sensitive to this debugging level, as recorded in the global LogLevel variable. Specifically, these routines will only generate output if the value of LogLevel is greater than or equal to the value of its threshold parameter. By default, the *File Server* sets LogLevel to zero upon startup. If a SIGTSTP signal is received when the debugging level is zero, it will be bumped to 1. If the signal arrives when the debugging level is positive, its value will be multiplied by 5. Thus, as more SIGTSTPs are received, the set of debugging messages eligible to be delivered to log files grows.

Since the SIGTSTP signal is not supported under IBM's AIX 2.2.1 operating system, this form of debugging output manipulation is not possible on those platforms.

5.2.3 SIGHUP: Reset Debugging Level

Receiving a SIGHUP signal causes a *File Server* to reset its debugging level to zero. This effectively reduces the set of debugging messages eligible for delivery to log files to a bare minimum. This signal is used in conjunction with SIGTSTP to manage the verbosity of log information.

Since the SIGHUP signal is not supported under IBM's AIX 2.2.1 operating system, this form of debugging output manipulation is not possible on those platforms.

5.2.4 SIGTERM: File Descriptor Check

Receipt of a SIGTERM signal triggers a routine which sweeps through the given *File Server*'s UNIX file descriptors. For each possible UNIX fid slot, an *fstat()* is performed on that descriptor, and the particulars of each open file are printed out. This action is designed solely for debugging purposes.

5.3 Command Line Interface

Another interface exported by the *File Server* is the set of command line switches it accepts. Using these switches, many server parameters and actions can be set. Under normal conditions, the *File Server* process is started up by the *BOS Server* on that machine, as described in *AFS-3 Programmer's Reference: BOS Server Interface*. So, in order to utilize any combination of these command-line options, the system administrator must define the *File Server* bnode in such a way that these parameters are properly included. Note that the switch names must be typed exactly as listed, and that abbreviations are not allowed. Thus, specifying **-b 300** on the command line is unambiguous, directing that 300 buffers are to be allocated. It is *not* an abbreviation for the **-banner** switch, asking that a message is to be printed to the console periodically.

A description of the set of currently-supported command line switches follows.

- **-b** <*# buffers*> Choose the number of 2,048-byte data buffers to allocate at system startup. If this switch is not provided, the *File Server* will operate with 70 such buffers by default.
- **-banner** This switch instructs the *File Server* to print messages to the console every 10 minutes to demonstrate it is still running correctly. The text of the printed message is: `File Server is running at` <*time*>.
- **-cb** <*# callbacks stored*> Specify the maximum number of callback records stored simultaneously by the *File Server*. The default pool size is 20,000 records.
- **-d** <*debug level*> Set the debugging output level at which *File Server* runs to the value provided. Specifically, the `LogLevel` global variable is set to the given value (See Section 5.2.2). If this switch is not provided, the default initial *File Server* debugging level is set to zero, producing the minimal debugging output to the log files.
- **-k** <*stack size*> Set the stack size to provide server LWPs upon creation, measured in 1,024-byte blocks. The default LWP stack size is 24 blocks, or 24,576 bytes.
- **-l** <*large (directory) vnodes*> Select the number of "large" vnodes the *File Server* will cache. These vnodes are suitable for recording information about AFS directories. The extra space in the vnode allows ACL information to be stored along with the directory. The default allocation value is 200 directory vnodes.
- **-pctspare** <*percent overrun blocks past quota*> Similar to the **-spare** switch, except that the number of allowable overrun blocks is expressed as a percentage of the given volume's quota. Note: this switch cannot be used in combination with the **-spare** switch.
- **-rxdbg** Instruct the *File Server* to open a file named `rx_dbg` in the current directory, into which the *Rx* package will write general debugging information. If the file is already open (due to the appearance of the **-rxdbge** switch earlier in the command line), this results in a no-op.
- **-rxdbge** Instruct the *File Server* to open a file named `rx_dbg` in the current directory, into which the *Rx* package will write debugging information related to its event-scheduling activities. If the file is already open (due to the appearance of the **-rxdbg** switch earlier in the command line), this results in a no-op.

- **-rxpck** <*# packets*> Set the number of extra *Rx* packet buffers to hold in reserve. These pre-allocated buffers assist in responding to spikes in network traffic demands. By default, 100 such packet buffers are maintained.

- **-s** <*small (file) vnodes*> Select the number of "small" vnodes the *File Server* will cache. These vnodes are suitable for recording information about non-directory files. As with directory vnodes, the *File Server* will allocate 200 small vnodes by default.

- **-spare** <*# overrun blocks to allow*> Tell the *File Server* to allow users performing a store operation to overrun the host volume's disk quota by a certain number of (1,024-byte) blocks. In other words, the first store resulting in a quota overrun will be allowed to succeed if and only if it uses no more than these many blocks beyond the quota. Further store operations will be rejected until the volume's storage is once again reduced below quota. By default, overruns of 1,024 blocks of 1,024 bytes each (1 megabyte total) are tolerated. Note: this switch cannot be used in combination with the **-pctspare** switch.

- **-w** <*callback wait interval in seconds*> This switch determines how often the *File Server* periodic daemon lightweight processes run. Among other things, these daemon LWPs check on the validity of callback records, keep disk usage statistics up to date, and check the health of the various client machines that have previously interacted with the *File Server*. For a full description of these daemon LWPs, consult Section 2.3. The associated argument specifies the number of seconds to sleep between daemon invocations. By default, these periodic daemons run every 300 seconds (5 minutes).

Chapter 6

Cache Manager **Interfaces**

6.1 Overview

There are several interfaces offered by the *Cache Manager*, allowing clients to access the files stored by the community of AFS *File Servers*, to configure the *Cache Manager*'s behavior and resources, to store and retrieve authentication information, to specify the location of community *Authentication Server* and *Volume Location Server* services, and to observe and debug the *Cache Manager*'s state and actions. This chapter will cover the following five interfaces to the *Cache Manager*:

- **ioctl()**: The standard UNIX **ioctl()** system call has been extended to include more operations, namely waiting until data stores to a *File Server* complete before returning to the caller (`VIOCCLOSEWAIT`) and getting the name of the cell in which an open file resides (`VIOCIGETCELL`).
- **pioctl()**: An additional system call is provided through which applications can access operations specific to AFS, which are often tied to a particular pathname. These operations include Access Control List (ACL) and mount point management, Kerberos ticket management, cache configuration, cell configuration, and status of *File Servers*.
- **RPC**: Interface by which outside servers and investigators can manipulate the *Cache Manager*. There are two main categories of routines: callback management, typically called by the *File Server*, and debugging/statistics, called by programs such as *cmdebug* and via the *xstat* user-level library for collection of extended statistics.

- **Files**: Much of the *Cache Manager*'s configuration information, as well as its view of the AFS services available from the outside world, is obtained from parsing various files. One set of these files is typically located in */usr/vice/etc*, and includes *CellServDB*, *ThisCell*, and *cacheinfo*. Another set is usually found in */usr/vice/cache*, namely *CacheItems*, *VolumeItems*, and *AFSLog*.
- **Mariner**: This is the interface by which file transfer activity between the *Cache Manager* and *File Server*s may be monitored. Specifically, it is used to monitor the names of the files and directories being fetched and/or stored over the network.

Another important component not described in this document is the *afsd* program. It is *afsd*'s job to initialize the *Cache Manager* on a given machine and to start up its related daemon threads. It accepts a host of configuration decisions via its command-line interface. In addition, it parses some of the information kept in the configuration files mentioned above and passes that information to the *Cache Manager*. The reader may find a full description of *afsd* in the *AFS 3.0 Command Reference Manual*[2].

6.2 Definitions

This section defines data structures that are used by the **pioctl()** calls.

6.2.1 struct VenusFid

The *Cache Manager* is the sole active AFS agent aware of the cellular architecture of the system. Since AFS file identifiers are not guaranteed to be unique across cell boundaries, it must further qualify them for its own internal bookkeeping. The `struct VenusFid` provides just such additional qualification, attaching the *Cache Manager*'s internal cell identifier to the standard AFS fid.

Fields

`long Cell` - The internal identifier for the cell in which the file resides.

`struct ViceFid Fid` - The AFS file identifier within the above cell.

6.2.2 struct ClearToken

This is the clear-text version of an AFS token of identity. Its fields are encrypted into the secret token format, and are made easily available to the *Cache Manager* in this structure.

Fields

`long AuthHandle` - Key version number.

`char HandShakeKey[8]` - Session key.

`long ViceId` - Identifier for the AFS principal represented by this token.

`long BeginTimestamp` - Timestamp of when this token was minted, and hence came into effect.

`long EndTimestamp` - Timestamp of when this token is considered to be expired, and thus disregarded.

6.3 ioctl() Interface

The standard UNIX **ioctl()** system call performs operations on file system objects referenced with an open file descriptor. AFS has augmented this system call with two additional operations, one to perform "safe stores", and one to get the name of the cell in which a file resides. A third **ioctl()** extension is now obsolete, namely aborting a store operation currently in progress.

6.3.1 VIOCCLOSEWAIT

[Opcode 1] Normally, a client performing a UNIX *close()* call on an AFS file resumes once the store operation on the given file data to the host *File Server* has commenced but before it has completed. Thus, it is possible that the store could actually fail (say, because of network partition or server crashes) without the client's knowledge. This new ioctl opcode specifies to the *Cache Manager* that all future *close()* operations will wait until the associated store operation to the *File Server* has completed fully before returning.

6.3.2 VIOCABORT

[Opcode 2] This **ioctl()** extension is now obsolete. This call results in a noop.

The original intention of this call was to allow a store operation currently in progress to a *File Server* on the named fid to be aborted.

6.3.3 VIOIGETCELL

[Opcode 3] Get the name of the cell in which the given fid resides. If the file is not an AFS file, then ENOTTY is returned. The output buffer specified in the data area must be large enough to hold the null-terminated string representing the file's cell, otherwise EFAULT is returned. However, an out_size value of zero specifies that the cell name is *not* to be copied into the output buffer. In this case, the caller is simply interested in whether the file is in AFS, and not its exact cell of residence.

6.4 pioctl() Interface

6.4.1 Introduction

There is a new UNIX system call, **pioctl()**, which has been defined especially to support the AFS *Cache Manager*. Its functional definition is as follows:

```
int afs_syscall_pioctl(IN char *a_pathP,
                       IN int a_opcode,
                       IN struct ViceIoctl *a_paramsP,
                       IN int a_followSymLinks)
```

This new call is much like the standard **ioctl()** call, but differs in that the affected file (when applicable) is specified by its path, not by a file descriptor. Another difference is the fourth parameter, a_followSymLinks, determines which file should be used should *a_pathP* be a symbolic link. If *a_followSymLinks* be set to 1, then the symbolic link is followed to its target, and the **pioctl()** is applied to that resulting file. If *a_followSymLinks* is set to 0, then the **pioctl()** applies to the symbolic link itself.

Not all **pioctl()** calls affect files. In those cases, the *a_pathP* parameter should be set to a null pointer. The second parameter to **pioctl()**, *a_opcode*, specifies which operation is to be performed. The opcode for each of these operations is included in the text of the description. Note that not all **pioctl()** opcodes are in use. These unused values correspond to obsolete operations.

The descriptions that follow identify some of the possible error codes for each **pioctl()** opcode, but do not offer a comprehensive lists. All **pioctl()** calls return 0 upon success.

The rest of this section proceeds to describe the individual opcodes available. First, though, one asymmetry in this opcode set is pointed out, namely that while various operations are defined on AFS mount points, there is no direct way to *create* a mount point.

This documentation partitions the **pioctl()** into several groups:

- **Volume operations**
- *File Server* **operations**
- **Cell Operations**
- **Authentication Operations**
- **ACL Operations**
- **Cache operations**
- **Miscellaneous operations**

For all **pioctl()**s, the fields within the *a_paramsP* parameter will be referred to directly. Thus, the values of *in*, *in_size*, *out*, and *out_size* are discussed, rather than the settings for *a_paramsP->in*, *a_paramsP->in_size*, *a_paramsP->out*, and *a_paramsP->out_size*.

For convenience of reference, a list of the actively-supported **pioctl()**s, their opcodes, and brief description appears (in opcode order) below.

- *[1]* `VIOCSETAL` : Set the ACL on a directory
- *[2]* `VIOCGETAL` : Get the ACL for a directory
- *[3]* `VIOCSETTOK` : Set the caller's token for a cell
- *[4]* `VIOCGETVOLSTAT` : Get volume status

- *[5]* VIOCSETVOLSTAT : Set volume status
- *[6]* VIOCFLUSH : Flush an object from the cache
- *[8]* VIOCGETTOK : Get the caller's token for a cell
- *[9]* VIOCUNLOG : Discard authentication information
- *[10]* VIOCCKSERV : Check the status of one or more *File Server*s
- *[11]* VIOCCKBACK : Mark cached volume info as stale
- *[12]* VIOCCKCONN : Check caller's tokens/connections
- *[14]* VIOCWHEREIS : Find host(s) for a volume
- *[20]* VIOCACCESS : Check caller's access on object
- *[21]* VIOCUNPAG : See *[9]* VIOCUNLOG
- *[22]* VIOCGETFID : Get fid for named object
- *[24]* VIOCSETCACHESIZE : Set maximum cache size in blocks
- *[25]* VIOCFLUSHCB : Unilaterally drop a callback
- *[26]* VIOCNEWCELL : Set cell service information
- *[27]* VIOCGETCELL : Get cell configuration entry
- *[28]* VIOCAFS_DELETE_MT_PT : Delete a mount point
- *[29]* VIOC_AFS_STAT_MT_PT : Get the contents of a mount point
- *[30]* VIOC_FILE_CELL_NAME : Get cell hosting a given object
- *[31]* VIOC_GET_WS_CELL : Get caller's home cell name
- *[32]* VIOC_AFS_MARINER_HOST : Get/set file transfer monitoring output
- *[33]* VIOC_GET_PRIMARY_CELL : Get the caller's primary cell
- *[34]* VIOC_VENUSLOG : Enable/disable *Cache Manager* logging
- *[35]* VIOC_GETCELLSTATUS : Get status info for a cell entry
- *[36]* VIOC_SETCELLSTATUS : Set status info for a cell entry
- *[37]* VIOC_FLUSHVOLUME : Flush cached data from a volume

- *[38]* `VIOC_AFS_SYSNAME` : Get/set the @sys mapping
- *[39]* `VIOC_EXPORTAFS` : Enable/disable NFS/AFS translation
- *[40]* `VIOCGETCACHEPARAMS` : Get current cache parameter values

6.4.2 Mount Point Asymmetry

There is an irregularity which deserves to be mentioned regarding the **pioctl()** interface. There are **pioctl()** operations for getting information about a mount point (`VIOC_AFS_STAT_MT_PT`) and for deleting a mount point (`VIOC_AFS_DELETE_MT_PT`), but no operation for creating mount points. To create a mount point, a symbolic link obeying a particular format must be created. The first character must be either a "%" or a "#", depending on the type of mount point being created (see the discussion in Section 6.4.4.4). If the mount point carries the name of the cell explicitly, the full cell name will appear next, followed by a colon. In all cases, the next portion of the mount point is the volume name. By convention, the last character of a mount point must always be a period ("."). This trailing period is not visible in the output from `fs lsmount`.

6.4.3 Volume Operations

There are several **pioctl()** opcodes dealing with AFS volumes. It is possible to get and set volume information (`VIOCGETVOLSTAT, VIOCSETVOLSTAT`), discover which volume hosts a particular file system object (`VIOCWHEREIS`), remove all objects cached from a given volume (`VIOC_FLUSHVOLUME`), and revalidate cached volume information (`VIOCCKBACK`).

6.4.3.1 `VIOCGETVOLSTAT`: Get volume status for pathname

[Opcode 4] Fetch information concerning the volume that contains the file system object named by *a_pathP*. There is no other input for this call, so *in_size* should be set to zero. The status information is placed into the buffer named by *out*, if *out_size* is set to a value of `sizeof(struct VolumeStatus)` or larger. Included in the volume information are the volume's ID, quota, and number of blocks used in the volume as well as the disk partition on which it resides. Internally, the *Cache Manager* calls the *RXAFS_GetVolumeInfo()* RPC (See Section 5.1.3.14) to fetch the volume status.

Among the possible error returns, `EINVAL` indicates that the object named by *a_pathP* could not be found.

6.4.3.2 `VIOCSETVOLSTAT`: Set volume status for pathname

[Opcode 5] Set the status fields for the volume hosting the file system object named by *a_pathP*. The first object placed into the input buffer *in* is the new status image. Only those fields that may change, namely *MinQuota* and *MaxQuota* fields, are interpreted upon receipt by the *File Server*, and are set to the desired values. Immediately after the `struct VolumeStatus` image, the caller must place the null-terminated string name of the volume involved in the input buffer. New settings for the offline message and MOTD (Message of the Day) strings may appear after the volume name. If there are no changes in the offline and/or MOTD messages, a null string must appear for that item. The *in_size* parameter must be set to the total number of bytes so inserted, including the nulls after each string. Internally, the *Cache Manager* calls the *RXAFS_SetVolumeStatus()* RPC (See Section 5.1.3.16) to store the new volume status.

Among the possible error returns, `EINVAL` indicates that the object named by *a_pathP* could not be found.

6.4.3.3 `VIOCWHEREIS`: Find the server(s) hosting the pathname's volume

[Opcode 14] Find the set of machines that host the volume in which the file system object named by *a_pathP* resides. The input buffer *in* is not used by this call, so *in_size* should be set to zero. The output buffer indicated by *out* is filled with up to 8 IP addresses, one for each *File Server* hosting the indicated volume. Thus, *out_size* should be set to at least `(8*sizeof(long))`. This group of hosts is terminated by the first zeroed IP address that appears in the list, but under no circumstances are more than 8 host IP addresses returned.

Among the possible error returns is `EINVAL`, indicating that the pathname is not in AFS, hence is not contained within a volume. If `ENODEV` is returned, the associated volume information could not be obtained.

6.4.3.4 `VIOC_FLUSHVOLUME`: Flush all data cached from the pathname's volume

[Opcode 37] Determine the volume in which the file system object named by *a_pathP* resides, and then throw away all currently cached copies of files that the *Cache Manager* has obtained from that volume. This call is typically used should a user suspect there is some cache corruption associated with the files from a given volume.

6.4.3.5 `VIOCCKBACK`: Check validity of all cached volume information

[Opcode 11] Ask the *Cache Manager* to check the validity of all cached volume information. None of the call's parameters are referenced in this call, so *a_pathP* and *in* should be set to the null pointer, and *in_size* and *out_size* should be set to zero.

This operation is performed in two steps:

1. The *Cache Manager* first refreshes its knowledge of the root volume, usually named `root.afs`. On success, it wakes up any of its own threads waiting on the arrival of this information, should it have been previously unreachable. This typically happens should the *Cache Manager* discover in its startup sequence that information on the root volume is unavailable. Lacking this knowledge at startup time, the *Cache Manager* settles into a semi-quiescent state, checking every so often to see if volume service is available and thus may complete its own initialization.

2. Each cached volume record is flagged as being stale. Any future attempt to access information from these volumes will result in the volume record's data first being refreshed from the *Volume Location Server*.

6.4.4 *File Server* Operations

One group of **pioctl()** opcodes is aimed at performing operations against one or more *File Servers* directly. Specifically, a caller may translate a pathname into the corresponding AFS fid (`VIOCGETFID`), unilaterally discard a set of callback promises (`VIOCFLUSHCB`), get status on mount points (`VIOC_AFS_STAT_MT_PT`), delete unwanted mount points (`VIOC_AFS_DELETE_MT_PT`), and check the health of a group of *File Servers* (`VIOCCKSERV`).

6.4.4.1 `VIOCGETFID`: Get augmented fid for named file system object

[Opcode 22] Return the augmented file identifier for the file system object named by *a_pathP*. The desired `struct VenusFid` is placed in the output buffer specified by *out*. The output buffer size, as indicated by the *out_size* parameter, must be set to the value of `sizeof(struct VenusFid)` or greater. The input buffer is not referenced in this call, so *in* should be set to the null pointer and *in_size* set to zero.

Among the possible error returns, `EINVAL` indicates that the object named by *a_pathP* was not found.

6.4.4.2 `VIOCFLUSHCB`: Unilaterally drop a callback

[Opcode 25] Remove any callback information kept by the *Cache Manager* on the file system object named by *a_pathP*. Internally, the *Cache Manager* executes a call to the *RXAFS_GiveUpCallBacks()* RPC (See Section 5.1.3.13) to inform the appropriate *File Server* that it is being released from its particular callback promise. Note that if the named file resides on a read-only volume, then the above call is not made, and success is returned immediately. This optimization is possible because AFS *File Server*s do not grant callbacks on files from read-only volumes.

Among the possible error returns is `EINVAL`, which indicates that the object named by *a_pathP* was not found.

6.4.4.3 `VIOC_AFS_DELETE_MT_PT`: Delete a mount point

[Opcode 28] Remove an AFS mount point. The name of the directory in which the mount point exists is specified by *a_pathP*, and the string name of the mount point within this directory is provided through the *in* parameter. The input buffer length, *in_size*, is set to the length of the mount point name itself, including the trailing null. The output buffer is not accessed by this call, so *out* should be set to the null pointer and *out_size* to zero.

One important note is that the *a_followSymLinks* argument **must be set to zero for correct operation**. This is counter-intuitive, since at first glance it seems that a symbolic link that resolves to a directory should be a valid pathname parameter. However, recall that mount points are implemented as symbolic links that do not actually point to another file system object, but rather simply contain cell and volume information (see the description in Section 6.4.2). This "special" symbolic link must not be resolved by the **pioctl()**, but rather presented as-is to the *Cache Manager*, which then properly interprets it and generates a reference to the given volume's root directory. As an unfortunate side-effect, a perfectly valid symbolic link referring to a directory will be rejected out of hand by this operation as a value for the *a_pathP* parameter.

Among the possible error returns, `EINVAL` reports that the named directory was not found, and `ENOTDIR` indicates that the pathname contained within *a_pathP* is not a directory.

6.4.4.4 `VIOC_AFS_STAT_MT_PT`: Get the contents of a mount point

[Opcode 29] Return the contents of the given mount point. The directory in which the mount point in question resides is provided via the *a_pathP* argument, and the *in* buffer

contains the name of the mount point object within this directory. As usual, *in_size* is set to the length of the input buffer, including the trailing null. If the given object is truly a mount point and the *out* buffer is large enough (its length appears in *out_size*), the mount point's contents are stored into *out*.

The mount point string returned obeys a stylized format, as fully described in Section 5.6.2 of the *AFS 3.0 System Administrator's Guide*[1]. Briefly, a leading pound sign ("#") indicates a standard mount point, inheriting the read-only or read-write preferences of the mount point's containing volume. On the other hand, a leading percent sign ("%") advises the *Cache Manager* to cross into the read-write version of the volume, regardless of the existence of read-only clones. If a colon (":") separator occurs, the portion up to the colon itself denotes the fully-qualified cell name hosting the volume. The rest of the string is the volume name itself.

Among the possible error codes is `EINVAL`, indicating that the named object is not an AFS mount point. Should the name passed in *a_pathP* be something other than a directory, then `ENOTDIR` is returned.

6.4.4.5 `VIOCCKSERV`: Check the status of one or more *File Servers*

[Opcode 10] Check the status of the *File Servers* that have been contacted over the lifetime of the *Cache Manager*. The *a_pathP* parameter is ignored by this call, so it should be set to the null pointer. The input parameters as specified by *in* are completely optional. If something is placed in the input buffer, namely *in_size* is not zero, then the first item stored there is a longword used as a bit array of flags. These flags carry instructions as to the domain and the "thoroughness" of this check.

Only the settings of the least-significant two bits are recognized. Enabling the lowest bit tells the *Cache Manager* not to ping its list of servers, but simply report their status as contained in the internal server records. Enabling the next-higher bit limits the search to only those *File Servers* in a given cell. If *in_size* is greater than `sizeof(long)`, a null-terminated cell name string follows the initial flag array, specifying the cell to check. If this search bit is set but no cell name string follows the longword of flags, then the search is restricted to those servers contacted from the same cell as the caller.

This call returns at least one longword into the output buffer *out*, specifying the number of hosts it discovered to be down. If this number is not zero, then the longword IP address for each dead (or unreachable) host follows in the output buffer. At most 16 server addresses will be returned, as this is the maximum number of servers for which the *Cache Manager* keeps information.

Among the possible error returns is `ENOENT`, indicating that the optional cell name string input value is not known to the *Cache Manager*.

6.4.5 Cell Operations

The *Cache Manager* is the only active AFS agent that understands the system's cellular architecture. Thus, it keeps important information concerning the identities of the cells in the community, which cell is in direct administrative control of the machine upon which it is running, status and configuration of its own cell, and what cell-specific operations may be legally executed. The following **pioctl()**s allow client processes to access and update this cellular information. Supported operations include adding or updating knowledge of a cell, including the cell overseeing the caller's machine (VIOCNEWCELL), fetching the contents of a cell configuration entry (VIOCGETCELL), finding out which cell hosts a given file system object (VIOC_FILE_CELL_NAME), discovering the cell to which the machine belongs (VIOC_GET_WS_CELL), finding out the caller's "primary" cell (VIOC_GET_PRIMARY_CELL), and getting/setting certain other per-cell system parameters (VIOC_GETCELLSTATUS, VIOC_SETCELLSTATUS).

6.4.5.1 VIOCNEWCELL: Set cell service information

[Opcode 26] Give the *Cache Manager* all the information it needs to access an AFS cell. Exactly eight longwords are placed at the beginning of the *in* input buffer. These specify the IP addresses for the machine providing AFS authentication and volume location authentication services. The first such longword set to zero will signal the end of the list of server IP addresses. After these addresses, the input buffer hosts the null-terminated name of the cell to which the above servers belong. The *a_pathP* parameter is not used, and so should be set to the null pointer.

Among the possible error returns is EACCES, indicating that the caller does not have the necessary rights to perform the operation. Only root is allowed to set cell server information. If either the IP address array or the server name is unacceptable, EINVAL will be returned.

6.4.5.2 VIOCGETCELL: Get cell configuration entry

[Opcode 27] Get the *i*'th cell configuration entry known to the *Cache Manager*. The index of the desired entry is placed into the *in* input buffer as a longword, with the first legal value being zero. If there is a cell associated with the given index, the output buffer will be filled with an array of 8 longwords, followed by a null-terminated string.

The longwords correspond to the list of IP addresses of the machines providing AFS authentication and volume location services. The string reflects the name of the cell for which the given machines are operating. There is no explicit count returned of the number of valid IP addresses in the longword array. Rather, the list is terminated by the first zero value encountered, or when the eighth slot is filled.

This routine is intended to be called repeatedly, with the index starting at zero and increasing each time. The array of cell information records is kept compactly, without holes. A return value of `EDOM` indicates that the given index does not map to a valid entry, and thus may be used as the terminating condition for the iteration.

6.4.5.3 `VIOC_FILE_CELL_NAME`: Get cell hosting a given object

[Opcode 30] Ask the *Cache Manager* to return the name of the cell in which the file system object named by *a_pathP* resides. The input arguments are not used, so *in* should be set to the null pointer and *in_size* should be set to zero. The null-terminated cell name string is returned in the *out* output buffer.

Among the possible error values, `EINVAL` indicates that the pathname provided in *a_pathP* is illegal. If there is no cell information associated with the given object, `ESRCH` is returned.

6.4.5.4 `VIOC_GET_WS_CELL`: Get caller's home cell name

[Opcode 31] Return the name of the cell to which the caller's machine belongs. This cell name is returned as a null-terminated string in the output buffer. The input arguments are not used, so *in* should be set to the null pointer and *in_size* should be set to zero.

Among the possible error returns is `ESRCH`, stating that the caller's home cell information was not available.

6.4.5.5 `VIOC_GET_PRIMARY_CELL`: Get the caller's primary cell

[Opcode 33] Ask the *Cache Manager* to return the name of the caller's primary cell. Internally, the *Cache Manager* scans its user records, and the cell information referenced by that record is used to extract the cell's string name. The input arguments are not used, so *in* should be set to the null pointer and *in_size* should be set to zero. The *a_pathP* pathname argument is not used either, and should similarly be set to the null

pointer. The null-terminated cell name string is placed into the output buffer pointed to by *out* if it has sufficient room.

Among the possible error returns is `ESRCH`, stating that the caller's primary cell information was not available.

6.4.5.6 `VIOC_GETCELLSTATUS`: Get status info for a cell entry

[Opcode 35] Given a cell name, return a single longword of status flags from the *Cache Manager*'s entry for that cell. The null-terminated cell name string is expected to be in the *in* parameter, with *in_size* set to its length plus one for the trailing null. The status flags are returned in the *out* buffer, which must have *out_size* set to `sizeof(long)` or larger.

The *Cache Manager* defines the following output flag values for this operation:

- `0x1` This entry is considered the caller's primary cell.
- `0x2` The UNIX *setuid()* operation is not honored.
- `0x4` An obsolete version of the *Volume Location Server*'s database is being used. While defined, this flag should no longer be set in modern systems.

Among the possible error returns is `ENOENT`, informing the caller that the *Cache Manager* has no knowledge of the given cell name.

6.4.5.7 `VIOC_SETCELLSTATUS`: Set status info for a cell entry

[Opcode 36] Given a cell name and an image of the cell status bits that should be set, record the association in the *Cache Manager*. The input buffer *in* must be set up as follows. The first entry is the longword containing the cell status bits to be set (see the `VIOC_GETCELLSTATUS` description above for valid flag definitions). The next entry is another longword, ignored by the *Cache Manager*. The third and final entry in the input buffer is a null-terminated string containing the name of the cell for which the status flags are to be applied.

Among the possible error returns is `ENOENT`, reflecting the *Cache Manager*'s inability to locate its record for the given cell. Only `root` is allowed to execute this operation, and an `EACCES` return indicates the caller was not effectively `root` when the call took place.

6.4.6 Authentication Operations

The *Cache Manager* serves as the repository for authentication information for AFS clients. Each client process belongs to a single **Process Authentication Group (PAG)**. Each process in a given PAG shares authentication information with the other members, and thus has the identical rights with respect to AFS Access Control Lists (ACLs) as all other processes in the PAG. As the *Cache Manager* interacts with *File Servers* as a client process' agent, it automatically and transparently presents the appropriate authentication information as required in order to gain the access to which the caller is entitled. Each PAG can host exactly one *token* per cell. These tokens are objects that unequivocally codify the principal's identity, and are encrypted for security. Token operations between a *Cache Manager* and *File Server* are also encrypted, as are the interchanges between clients and the *Authentication Servers* that generate these tokens.

There are actually two different flavors of tokens, namely **clear** and **secret**. The data structure representing clear tokens is described in Section 6.2.2, and the secret token appears as an undifferentiated byte stream.

This section describes the operations involving these tokens, namely getting and setting the caller's token for a particular cell (`VIOCGETTOK`, `VIOCSETTOK`), checking a caller's access on a specified file system object (`VIOCACCESS`), checking the status of caller's tokens associated with the set of *File Server* connections maintained on its behalf (`VIOCCKCONN`), and discarding tokens entirely (`VIOCUNLOG`, `VIOCUNPAG`). These abilities are used by such programs as *login*, *klog*, *unlog*, and *tokens*, which must generate, manipulate, and/or destroy AFS tokens.

6.4.6.1 `VIOCSETTOK`: Set the caller's token for a cell

[Opcode 3] Store the caller's secret and clear tokens within the *Cache Manager*. The input buffer is used to hold the following quantities, laid out end to end. The first item placed in the buffer is a longword, specifying the length in bytes of the secret token, followed by the body of the secret token itself. The next field is another longword, this time describing the length in bytes of the `struct ClearToken`, followed by the structure. These are all required fields. The caller may optionally include two additional fields, following directly after the required ones. The first optional field is a longword which is set to a non-zero value if the cell in which these tokens were generated is to be marked as the caller's primary cell. The second optional argument is a null-terminated string specifying the cell in which these tokens apply. If these two optional arguments do not appear, the *Cache Manager* will default to using its home cell and marking the entry as non-primary. The *a_pathP* pathname parameter is not used, and thus should be

set to the null pointer.

If the caller does not have any tokens registered for the cell, the *Cache Manager* will store them. If the caller already has tokens for the cell, the new values will overwrite their old values. Because these are stored per PAG, the new tokens will thus determine the access rights of all other processes belonging to the PAG.

Among the possible error returns is `ESRCH`, indicating the named cell is not recognized, and `EIO`, if information on the local cell is not available.

6.4.6.2 `VIOCGETTOK`: Get the caller's token for a cell

[Opcode 8] Get the specified authentication tokens associated with the caller. The *a_pathP* parameter is not used, so it should be set to the null pointer. Should the input parameter *in* be set to a null pointer, then this call will place the user's tokens for the machine's home cell in the *out* output buffer, if such tokens exist. In this case, the following objects are placed in the output buffer. First, a longword specifying the number of bytes in the body of the secret token is delivered, followed immediately by the secret token itself. Next is a longword indicating the length in bytes of the clear token, followed by the clear token. The input parameter may also consist of a single longword, indicating the index of the token desired. Since the *Cache Manager* is capable of storing multiple tokens per principal, this allows the caller to iteratively extract the full set of tokens stored for the PAG. The first valid index value is zero. The list of tokens is kept compactly, without holes. A return value of `EDOM` indicates that the given index does not map to a valid token entry, and thus may be used as the terminating condition for the iteration.

Other than `EDOM`, another possible error return is `ENOTCONN`, specifying that the caller does not have any AFS tokens whatsoever.

6.4.6.3 `VIOCACCESS`: Check caller's access on object

[Opcode 20] This operation is used to determine whether the caller has specific access rights on a particular file system object. A single longword is placed into the input buffer, *in*, representing the set of rights in question. The acceptable values for these access rights are listen in Section 6.4.5. The object to check is named by the *a_pathP* parameter. The output parameters are not accessed, so *out* should be set to the null pointer, and *out_size* set to zero.

If the call returns successfully, the caller has at least the set of rights denoted by the bits set in the input buffer. Otherwise, EACCESS is returned.

6.4.6.4 `VIOCCKCONN`: Check status of caller's tokens/connections

[Opcode 12] Check whether the suite of *File Server* connections maintained on behalf of the caller by the *Cache Manager* has valid authentication tokens. This function always returns successfully, communicating the health of said connections by writing a single longword value to the specified output buffer in *out*. If zero is returned to the output buffer, then two things are true. First, the caller has tokens for at least one cell. Second, all tokens encountered upon a review of the caller's connections have been properly minted (i.e., have not been generated fraudulently), and, in addition, have not yet expired. If these conditions do not currently hold for the caller, then the output buffer value will be set to EACCES. Neither the *a_pathP* nor input parameters are used by this call.

6.4.6.5 `VIOCUNLOG`: Discard authentication information

[Opcode 9] Discard all authentication information held in trust for the caller. The *Cache Manager* sweeps through its user records, destroying all of the caller's associated token information. This results in reducing the rights of all processes within the caller's PAG to the level of file system access granted to the special `system:anyuser` group.

This operation always returns successfully. None of the parameters are referenced, so they should all be set to null pointers and zeroes as appropriate.

6.4.6.6 `VIOCUNPAG`: Discard authentication information

[Opcode 21] This call is essentially identical to the VIOCUNLOG operation, and is in fact implemented internally by the same code for VIOCUNLOG.

6.4.7 ACL Operations

This set of opcodes allows manipulation of AFS Access Control Lists (ACLs). Callers are allowed to fetch the ACL on a given directory, or to set the ACL on a directory. In AFS-3, ACLs are only maintained on directories, not on individual files. Thus, a directory ACL determines the allowable accesses on all objects within that directory in conjunction with their normal UNIX mode (owner) bits. Should the *a_pathP* parameter specify a file instead of a directory, the ACL operation will be performed on the directory in which the given file resides.

These **pioctl()** opcodes deal only in *external formats* for ACLs, namely the actual text stored in an AFS ACL container. This external format is a character string, composed of a descriptive header followed by some number of individual principal-rights pairs. AFS ACLs actually specify *two* sublists, namely the *positive* and *negative rights* lists. The positive list catalogues the set of rights that certain principals (individual users or groups of users) have, while the negative list contains the set of rights specifically denied to the named parties.

These external ACL representations differ from the *internal format* generated by the *Cache Manager* after a parsing pass. The external format may be easily generated from the internal format as follows. The header format is expressed with the following *printf()* statement:

```
printf("%d\n%d\n", NumPositiveEntries, NumNegativeEntries);
```

The header first specifies the number of entries on the positive rights list, which appear first in the ACL body. The number of entries on the negative list is the second item in the header. The negative entries appear after the last positive entry.

Each entry in the ACL proper obeys the format imposed by the following *printf()* statement:

```
printf("%s\t%d\n", UserOrGroupName, RightsMask);
```

Note that the string name for the user or group is stored in an externalized ACL entry. The *Protection Server* stores the mappings between the numerical identifiers for AFS principals and their character string representations. There are cases where there is no mapping from the numerical identifier to a string name. For example, a user or group may have been deleted sometime after they were added to the ACL and before the *Cache Manager* externalized the ACL for storage. In this case, the *Cache Manager* sets *UserOrGroupName* to the string version of the principal's integer identifier. Should the `erz` principal be deleted from the *Protection Server*'s database in the above scenario, then the string "`1019`" will be stored, since it corresponded to `erz`'s former numerical identifier.

The *RightsMask* parameter to the above call represents the set of rights the named principal may exercise on the objects covered by the ACL. The following flags may be OR'ed together to construct the desired access rights placed in *RightsMask*:

```
#define PRSFS_READ         1 /*Read files*/
#define PRSFS_WRITE        2 /*Write & write-lock existing files*/
#define PRSFS_INSERT       4 /*Insert & write-lock new files*/
#define PRSFS_LOOKUP       8 /*Enumerate files and examine ACL*/
#define PRSFS_DELETE      16 /*Remove files*/
#define PRSFS_LOCK        32 /*Read-lock files*/
#define PRSFS_ADMINISTER  64 /*Set access list of directory*/
```

6.4.7.1 VIOCSETAL: Set the ACL on a directory

[Opcode 1] Set the contents of the ACL associated with the file system object named by *a_pathP*. Should this pathname indicate a file and not a directory, the *Cache Manager* will apply this operation to the file's parent directory. The new ACL contents, expressed in their externalized form, are made available in *in*, with *in_size* set to its length in characters, including the trailing null. There is no output from this call, so *out_size* should be set to zero. Internally, the *Cache Manager* will call the *RXAFS_StoreACL()* RPC (see Section 5.1.3.3 to store the new ACL on the proper *File Server*.

Possible error codes include EINVAL, indicating that one of three things may be true: the named path is not in AFS, there are too many entries in the specified ACL, or a non-existent user or group appears on the ACL.

6.4.7.2 VIOCGETAL: Get the ACL for a directory

[Opcode 2] Get the contents of the ACL associated with the file system object named by *a_pathP*. Should this pathname indicate a file and not a directory, the *Cache Manager* will apply this operation to the file's parent directory. The ACL contents, expressed in their externalized form, are delivered into the *out* buffer if *out_size* has been set to a value which indicates that there is enough room for the specified ACL. This ACL string will be null-terminated. There is no input to this call, so *in_size* should be set to zero. Internally, the *Cache Manager* will call the *RXAFS_FetchACL()* RPC (see Section 5.1.3.1) to fetch the ACL from the proper *File Server*.

Possible error codes include EINVAL, indicating that the named path is not in AFS.

6.4.8 Cache Operations

It is possible to inquire about and affect various aspects of the cache maintained locally by the *Cache Manager* through the group of **pioctl()**s described below. Specifically, one may force certain file system objects to be removed from the cache (`VIOCFLUSH`), set the maximum number of blocks usable by the cache (`VIOCSETCACHESIZE`), and ask for information about the cache's current state (`VIOCGETCACHEPARAMS`).

6.4.8.1 `VIOCFLUSH`: Flush an object from the cache

[Opcode 6] Flush the file system object specified by *a_pathP* out of the local cache. The other parameters are not referenced, so they should be set to the proper combination of null pointers and zeroes.

Among the possible error returns is `EINVAL`, indicating that the value supplied in the *a_pathP* parameter is not acceptable.

6.4.8.2 `VIOCSETCACHESIZE`: Set maximum cache size in blocks

[Opcode 24] Instructs the *Cache Manager* to set a new maximum size (in 1 Kbyte blocks) for its local cache. The input buffer located at *in* contains the new maximum block count. If zero is supplied for this value, the *Cache Manager* will revert its cache limit to its value at startup time. Neither the *a_pathP* nor output buffer parameters is referenced by this operation. The *Cache Manager* recomputes its other cache parameters based on this new value, including the number of cache files allowed to be dirty at once and the total amount of space filled with dirty chunks. Should the new setting be smaller than the number of blocks currently being used, the *Cache Manager* will throw things out of the cache until it obeys the new limit.

The caller is required to be effectively running as `root`, or this call will fail, returning `EACCES`. If the *Cache Manager* is configured to run with a memory cache instead of a disk cache, this operation will also fail, returning `EROF`.

6.4.8.3 `VIOCGETCACHEPARAMS`: Get current cache parameter values

[Opcode 40] Fetch the current values being used for the cache parameters. The output buffer is filled with `MAXGCSTATS` (16) longwords, describing these parameters. Only the first two longwords in this array are currently set. The first contains the value of `afs_cacheBlocks`, or the maximum number of 1 Kbyte blocks which may be used in the cache (see Section 6.4.8.2 for how this value may be set). The second longword contains the value of the *Cache Manager*'s internal `afs_blocksUsed` variable, or the number of these cache blocks currently in use. All other longwords in the array are set to zero. Neither the *a_pathP* nor input buffer arguments are referenced by this call.

This routine always returns successfully.

6.4.9 Miscellaneous Operations

There are several other AFS-specific operations accessible via the **pioctl()** interface that don't fit cleanly into the above categories. They are described in this section, and include manipulation of the socket-based Mariner file trace interface (`VIOC_AFS_MARINER_HOST`), enabling and disabling of the file-based *AFSLog* output interface for debugging (`VIOC_VENUSLOG`), getting and setting the value of the special `@sys` pathname component mapping (`VIOC_AFS_SYSNAME`), and turning the NFS-AFS translator service on and off (`VIOC_EXPORTAFS`).

6.4.9.1 `VIOC_AFS_MARINER_HOST`: Get/set file transfer monitoring output

[Opcode 32] This operation is used to get or set the IP address of the host destined to receive Mariner output. A detailed description of the *Cache Manager* Mariner interface may be found in Section 6.7.

The input buffer located at *in* is used to pass a single longword containing the IP address of the machine to receive output regarding file transfers between the *Cache Manager* and any *File Server*. If the chosen host IP address is `0xffffffff`, the *Cache Manager* is prompted to turn off generation of Mariner output entirely. If the chosen host IP address is zero, then the *Cache Manager* will not set the Mariner host, but rather return the current Mariner host as a single longword written to the *out* output buffer. Any other value chosen for the host IP address enables Mariner output (if it was not already enabled) and causes all further traffic to be directed to the given machine.

This function always returns successfully.

6.4.9.2 `VIOC_VENUSLOG`: Enable/disable *Cache Manager* logging

[Opcode 34] Tell the *Cache Manager* whether to generate debugging information, and what kind of debugging output to enable. The input buffer located at *in* is used to transmit a single longword to the *Cache Manager*, expressing the caller's wishes. Of the four bytes making up the longword, the highest byte indicates the desired value for the internal `afsDebug` variable, enabling or disabling general trace output. The next highest byte indicates the desired value for the internal `netDebug` variable, enabling or disabling network-level debugging traces. The third byte is unused, and the low-order byte represents an overall on/off value for the functionality. There is a special value

for the low-order byte, 99, which instructs the *Cache Manager* to return the current debugging setting as a single longword placed into the output buffer pointed to by *out*. The *a_pathP* parameter is not referenced by this routine.

Trace output is delivered to the *AFSLog* file, typically located in the */usr/vice/etc* directory. When this form of debugging output is enabled, the existing *AFSLog* file is truncated, and its file descriptor is stored for future use. When this debugging is disabled, a *close()* is done on the file, forcing all its data to disk. For additional information on the *AFSLog* file for collecting *Cache Manager* traces, please see the description in Section 6.6.2.1.

This call will only succeed if the caller is effectively running as `root`. If this is not the case, an error code of `EACCES` is returned.

6.4.9.3 `VIOC_AFS_SYSNAME`: Get/set the `@sys` mapping

[Opcode 38] Get or set the value of the special *@sys* pathname component understood by the *Cache Manager*. The input buffer pointed to by *in* is used to house a longword whose value determines whether the `@sys` value is being set (1) or whether the current value is being fetched (0). If it is being set, then a null-terminated string is expected to follow in the input buffer, specifying the new value of `@sys`. Otherwise, if we are asking the *Cache Manager* for the current `@sys` setting, a null-terminated string bearing that value will be placed in the `out` output buffer. The *a_pathP* parameter is not used by this call, and thus should be set to a null pointer.

There are no special privileges required of the caller to fetch the value of the current `@sys` mapping. However, a native caller must be running effectively as `root` in order to successfully alter the mapping. An unauthorized attempt to change the `@sys` setting will be ignored, and cause this routine to return `EACCES`. This requirement is relaxed for `VIOC_AFS_SYSNAME` **pioctl()** calls emanating from foreign file systems such as NFS and accessing AFS files through the NFS-AFS translator. Each such remote caller may set its own notion of what the `@sys` mapping is without affecting native AFS clients. Since the uid values received in calls from NFS machines are inherently insecure, it is impossible to enforce the fact that the caller is truly `root` on the NFS machine. This, while any principal running on an NFS machine may change that foreign machine's perception of `@sys`, it does not impact native AFS users in any way.

6.4.9.4 `VIOC_EXPORTAFS`: Enable/disable NFS/AFS translation

[Opcode 39] Enable or disable the ability of an AFS-capable machine to export AFS

access to NFS clients. Actually, this is a general facility allowing exportation of AFS service to any number of other file systems, but the only support currently in place is for NFS client machines. A single longword is expected in the input buffer *in*. This input longword is partitioned into individual bytes, organized as follows. The high-order byte communicates the type of foreign client to receive AFS file services. There are currently two legal values for this field, namely 0 for the null foreign file system and 1 for NFS. The next byte determines whether the *Cache Manager* is being asked to get or set this information. A non-zero value here is interpreted as a command to set the export information according to what's in the input longword, and a zero-valued byte in this position instructs the *Cache Manager* to place a longword in the output buffer *out*, which contains the current export settings for the foreign system type specified in the high-order byte. The third input byte is not used, and the lowest-order input buffer byte determines whether export services for the specified system are being enabled or disabled. A non-zero value will turn on the services, and a zero value will shut them down. The *a_pathP* pathname parameter is not used by this call, and the routine generates output only if the export information is being requested instead of being set.

The caller must be effectively running as `root` in order for this operation to succeed. The call returns `EACCES` if the caller is not so authorized. If the caller specifies an illegal foreign system type in the high-order byte of the input longword, then `ENODEV` is returned. Again, NFS is the only foreign file system currently supported.

Practically speaking, the machine providing NFS-AFS translation services must enable this service with this **pioctl()** before any NFS client machines may begin accessing AFS files. Conversely, if an administrator turns off this export facility, the export code on the translator machine will immediately stop responding to traffic from its active NFS clients.

6.5 RPC Interface

6.5.1 Introduction

This section covers the structure and workings of the *Cache Manager*'s RPC interface. Typically, these calls are made by *File Server* processes. However, some of the calls are designed specifically for debugging programs (e.g., the `cmdebug` facility) and for collection of statistical and performance information from the *Cache Manager*. Any client application that makes direct calls on the *File Server* RPC interface must be prepared to export a subset of the *Cache Manager* RPC interface, as discussed in Section 5.1.6.

This section will first examine the *Cache Manager*'s use of locks, whose settings may be observed via one of the RPC interface calls. Next, it will present some definitions and data structures used in the RPC interface, and finally document the individual calls available through this interface.

6.5.2 Locks

The *Cache Manager* makes use of locking to insure its internal integrity in the face of its multi-threaded design. A total of 11 locks are maintained for this purpose, one of which is now obsolete and no longer used (see below). These locks are strictly internal, and the *Cache Manager* itself is the only one able to manipulate them. The current settings for these system locks are externally accessible for debugging purposes via the *AFSRXCB_GetLock()* RPC interface call, as described in Section 6.5.5.4. For each lock, its index in the locking table is given in the following text.

- **afs_xvcache** *[Index 0]*: This lock controls access to the status cache entries maintained by the *Cache Manager*. This stat cache keeps *stat()*-related information for AFS files it has dealt with. The stat information is kept separate from actual data contents of the related file, since this information may change independently (say, as a result of a UNIX *chown() call.*
- **afs_xdcache** *[Index 1]*: This lock moderates access to the *Cache Manager*'s data cache, namely the contents of the file system objects it has cached locally. As stated above, this data cache is separate from the associated *stat()* information.
- **afs_xserver** *[Index 2]*: This lock controls access to the *File Server* machine description table, which keeps tabs on all *File Server*s contacted in recent history. This lock thus indirectly controls access to the set of per-server RPC connection descriptors the *File Server* table makes visible.
- **afs_xvcb** *[Index 3]*: This lock supervises access to the volume callback information kept by the *Cache Manager*. This table is referenced, for example, when a client decides to remove one or more callbacks on files from a given volume (see the *RXAFS_GiveUpCallBacks()* description on Section 5.1.3.13).
- **afs_xbrs** *[Index 4]*: This lock serializes the actions of the *Cache Manager*'s background daemons, which perform prefetching and background file storage duties.
- **afs_xcell** *[Index 5]*: This lock controls the addition, deletion, and update of items on the linked list housing information on cells known to the *Cache Manager*.
- **afs_xconn** *[Index 6]*: This lock supervises operations concerning the set of RPC connection structures kept by the system. This lock is used in combination with the

afs_xserver lock described above. In some internal *Cache Manager* code paths, the *File Server* description records are first locked, and then the afs_xconn lock is used to access the associated *Rx* connection records.

- **afs_xuser** *[Index 7]*: This lock serializes access to the per-user structures maintained by the *Cache Manager*.
- **afs_xvolume** *[Index 8]*: This lock is used to control access to the *Cache Manager*'s volume information cache, namely the set of entries currently in memory, a subset of those stably housed in the *VolumeItems* disk file (see Section 6.6.2.3).
- **afs_puttofileLock** *[Index 9]*: This lock is obsolete, and while still defined by the system is no longer used. It formerly serialized writes to a debugging output interface buffer, but the internal mechanism has since been updated and improved.
- **afs_ftf** *[Index 10]*: This lock is used when flushing cache text pages from the machine's virtual memory tables. For each specific machine architecture on which the *Cache Manager* runs, there is a set of virtual memory operations which must be invoked to perform this operation. The result of such activities is to make sure that the latest contents of new incarnations of binaries are used, instead of outdated copies of previous versions still resident in the virtual memory system.

6.5.3 Definitions and Typedefs

This section documents some macro definitions and typedefs referenced by the *Cache Manager*'s RPC interface. Specifically, these definitions and typedefs are used in the *RXAFSCB_GetXStats()* and *RXAFSCB_XStatsVersion* calls as described in Sections 6.5.5.6 and 6.5.5.7.

```
/*
 * Define the version of CacheManager and FileServer extended
 * statistics being implemented.
 */
const AFSCB_XSTAT_VERSION = 1;

/*
 * Define the maximum arrays for passing extended statistics
 * info for the CacheManager and FileServer back to our caller.
 */
const AFSCB_MAX_XSTAT_LONGS = 2048;

typedef long AFSCB_CollData<AFSCB_MAX_XSTAT_LONGS>;

/*
```

```
 * Define the identifiers for the accessible extended stats data
 * collections.
 */
const AFSCB_XSTATSCOLL_CALL_INFO = 0;   /*CM call counting & info*/
const AFSCB_XSTATSCOLL_PERF_INFO = 1;   /*CM performance info*/
```

6.5.4 Structures

This section documents some structures used in the *Cache Manager* RPC interface. As with the constants and typedefs in the previous section, these items are used in the *RXAFSCB_GetXStats()* and *RXAFSCB_XStatsVersion* calls as described in Sections 6.5.5.6 and 6.5.5.7.

6.5.4.1 struct afs_MeanStats

This structure may be used to collect a running average figure. It is included in some of the statistics structures described below.

Fields

`long average` - The computed average.

`long elements` - The number of elements sampled for the above aveage.

6.5.4.2 struct afs_CMCallStats

This structure maintains profiling information, communicating the number of times internal *Cache Manager* functions are invoked. Each field name has a "`C_`" prefix, followed by the name of the function being watched. As this structure has entries for over 500 functions, it will not be described further here. Those readers who wish to see the full layout of this structure are referred to Appendix A.

The `AFSCB_XSTATSCOLL_CALL_INFO` data collection includes the information in this structure.

6.5.4.3 struct afs_CMMeanStats

This is the other part of the information (along with the `struct afs_CMCallStats` construct described above) returned by the `AFSCB_XSTATSCOLL_CALL_INFO` data collection defined by the *Cache Manager* (see Section 6.5.3). It is accessible via the *RXAFSCB_GetXStats()* interface routine, as defined in Section 6.5.5.7.

This structure represents the beginning of work to compute average values for some of the extended statistics collected by the *Cache Manager*.

Fields

`struct afs_MeanStats something` - Intended to collect averages for some of the *Cache Manager* extended statistics; not yet implemented.

6.5.4.4 struct afs_CMStats

This structure defines the information returned by the `AFSCB_XSTATSCOLL_CALL_INFO` data collection defined by the *Cache Manager* (see Section 6.5.3). It is accessible via the *RXAFSCB_GetXStats()* interface routine, as defined in Section 6.5.5.7.

Fields

`struct afs_CallStats callInfo` - Contains the counts on the number of times each internal *Cache Manager* function has been called.

`struct afs_MeanStats something` - Intended to collect averages for some of the *Cache Manager* extended statistics; not yet implemented.

6.5.4.5 struct afs_CMPerfStats

This is the information returned by the `AFSCB_XSTATSCOLL_PERF_INFO` data collection defined by the *Cache Manager* (see Section 6.5.3). It is accessible via the *RXAFSCB_GetXStats()* interface routine, as defined in Section 6.5.5.7.

Fields

`long numPerfCalls` - Number of performance calls received.

`long epoch` - *Cache Manager* epoch time.

`long numCellsContacted` - Number of cells contacted.

`long dlocalAccesses` - Number of data accesses to files within the local cell.

`long vlocalAccesses` - Number of stat accesses to files within the local cell.

`long dremoteAccesses` - Number of data accesses to files outside of the local cell.

`long vremoteAccesses` - Number of stat accesses to files outside of the local cell.

`long cacheNumEntries` - Number of cache entries.

`long cacheBlocksTotal` - Number of (1K) blocks configured for the AFS cache.

`long cacheBlocksInUse` - Number of cache blocks actively in use.

`long cacheBlocksOrig` - Number of cache blocks configured at bootup.

`long cacheMaxDirtyChunks` - Maximum number of dirty cache chunks tolerated.

`long cacheCurrDirtyChunks` - Current count of dirty cache chunks.

`long dcacheHits` - Number of data file requests satisfied by the local cache.

`long vcacheHits` - Number of stat entry requests satified by the local cache.

`long dcacheMisses` - Number of data file requests *not* satisfied by the local cache.

`long vcacheMisses` - Number of stat entry requests *not* satisfied by the local cache.

`long cacheFlushes` - Number of files flushed from the cache.

`long cacheFilesReused` - Number of cache files reused.

`long numServerRecords` - Number of records used for storing information concerning *File Servers*.

`long ProtServerAddr` - IP addres of the *Protection Server* used *(not implemented)*.

`long spare[32]` - A set of longword spares reserved for future use.

6.5.5 Function Calls

This section discusses the *Cache Manager* interface calls. No special permissions are required of the caller for any of these operations. A summary of the calls making up the interface appears below:

- *RXAFSCB_Probe()* "Are-you-alive" call.
- *RXAFSCB_CallBack()* Report callbacks dropped by a *File Server*.
- *RXAFSCB_InitCallBackState()* Purge callback state from a *File Server*.
- *RXAFSCB_GetLock()* Get contents of *Cache Manager* lock table.
- *RXAFSCB_GetCE()* Get cache file description.
- *RXAFSCB_XStatsVersion()* Get version of extended statistics package.
- *RXAFSCB_GetXStats()* Get contents of extended statistics data collection.

6.5.5.1 RXAFSCB_Probe — Acknowledge that the underlying callback service is still operational

`int` **RXAFSCB_Probe**`(IN struct rx_call *a_rxCallP)`

Description

[Opcode 206] This call simply implements an "are-you-alive" operation, used to determine if the given *Cache Manager* is still running. Any *File Server* will probe each of the *Cache Managers* with which it has interacted on a regular basis, keeping track of their health. This information serves an important purpose for a *File Server*. In particular, it is used to trigger purging of deceased *Cache Managers* from the *File Server*'s callback records, and also to instruct a new or "resurrected" *Cache Manager* to purge its own callback state for the invoking *File Server*.

Rx call information for the related *Cache Manager* is contained in *a_rxCallP*.

Error Codes

`---` No error codes are generated.

6.5.5.2 RXAFSCB_CallBack — Report callbacks dropped by a *File Server*

```
int RXAFSCB_CallBack(IN struct rx_call *a_rxCallP,
                     IN AFSCBFids *a_fidArrayP,
                     IN AFSCBs *a_callBackArrayP)
```

Description

[Opcode 204] Provide information on dropped callbacks to the *Cache Manager* for the calling *File Server*. The number of fids involved appears in *a_fidArrayP->AFSCBFids_len*, with the fids themselves located at *a_fidArrayP->AFSCBFids_val*. Similarly, the number of associated callbacks is placed in *a_callBackArrayP->AFSCBs_len*, with the callbacks themselves located at *a_callBackArrayP->AFSCBs_val*.

Rx call information for the related *Cache Manager* is contained in *a_rxCallP*.

Error Codes

---	No error codes are generated.

6.5.5.3 RXAFSCB_InitCallBackState — Purge callback state from a *File Server*

```
int RXAFSCB_InitCallBackState(IN struct rx_call *a_rxCallP)
```

Description

[Opcode 205] This routine instructs the *Cache Manager* to purge its callback state for all files and directories that live on the calling host. This function is typically called by a *File Server* when it gets a request from a *Cache Manager* that does not appear in its internal records. This handles situations where *Cache Manager*s survive a *File Server*, or get separated from it via a temporary network partition. This also happens upon bootup, or whenever the *File Server* must throw away its record of a *Cache Manager* because its tables have been filled.

Rx call information for the related *Cache Manager* is contained in *a_rxCallP*.

Error Codes

`---` No error codes are generated.

6.5.5.4 RXAFSCB_GetLock — Get contents of *Cache Manager* lock table

```
int RXAFSCB_GetLock(IN struct rx_call *a_rxCall,
                    IN long a_index,
                    OUT AFSDBLock *a_lockP)
```

Description

[Opcode 207] Fetch the contents of entry *a_index* in the *Cache Manager* lock table. There are 11 locks in the table, as described in Section 6.5.2. The contents of the desired lock, including a string name representing the lock, are returned in *a_lockP*.

This call is not used by *File Server*s, but rather by debugging tools such as `cmdebug`.

Rx call information for the related *Cache Manager* is contained in *a_rxCallP*.

Error Codes

1 The index value supplied in *a_index* is out of range; it must be between 0 and 10.

6.5.5.5 RXAFSCB_GetCE — Get cache file description

```
int RXAFSCB_GetCE(IN struct rx_call *a_rxCall,
                  IN long a_index,
                  OUT AFSDBCacheEntry *a_ceP)
```

Description

[Opcode 208] Fetch the description for entry *a_index* in the *Cache Manager* file cache, storing it into the buffer to which *a_ceP* points. The structure returned into this pointer variable is described in Section 4.3.2.

This call is not used by *File Server*s, but rather by debugging tools such as `cmdebug`.

Rx call information for the related *Cache Manager* is contained in *a_rxCallP*.

Error Codes

`1` The index value supplied in *a_index* is out of range.

6.5.5.6 RXAFSCB_XStatsVersion — Get version of extended statistics package

```
int RXAFSCB_XStatsVersion(IN struct rx_call *a_rxCall,
                          OUT long *a_versionNumberP)
```

Description

[Opcode 209] This call asks the *Cache Manager* for the current version number of the extended statistics structures it exports (see *RXAFSCB_GetXStats()*, Section 6.5.5.7). The version number is placed in *a_versionNumberP*.

Rx call information for the related *Cache Manager* is contained in *a_rxCallP*.

Error Codes

--- No error codes are generated.

6.5.5.7 RXAFSCB_GetXStats — Get contents of extended statistics data collection

```
int RXAFSCB_GetXStats(IN struct rx_call *a_rxCall,
                      IN long a_clientVersionNumber,
                      IN long a_collectionNumber,
                      OUT long *a_srvVersionNumberP,
                      OUT long *a_timeP,
                      OUT AFSCB_CollData *a_dataP)
```

Description

[Opcode 210] This function fetches the contents of the specified *Cache Manager* extended statistics structure. The caller provides the version number of the data it expects to receive in *a_clientVersionNumber*. Also provided in *a_collectionNumber* is the numerical identifier for the desired **data collection**. There are currently two of these data collections defined: `AFSCB_XSTATSCOLL_CALL_INFO`, which is the list of tallies of the number of invocations of internal *Cache Manager* procedure calls, and `AFSCB_XSTATSCOLL_PERF_INFO`, which is a list of performance-related numbers. The precise contents of these collections are described in Section 6.5.4. The current version number of the *Cache Manager* collections is returned in *a_srvVersionNumberP*, and is always set upon return, even if the caller has asked for a different version. If the correct version number has been specified, and a supported collection number given, then the collection data is returned in *a_dataP*. The time of collection is also returned, being placed in *a_timeP*.

Rx call information for the related *Cache Manager* is contained in *a_rxCallP*.

Error Codes

1 The collection number supplied in *a_collectionNumber* is out of range.

6.6 Files

The *Cache Manager* gets some of its start-up configuration information from files located on the client machine's hard disk. Each client is required to supply a */usr/vice/etc* directory in which this configuration data is kept. Section 6.6.1 describes the format and purpose of the three files contributing this setup information: *ThisCell*, *CellServDB*, and *cacheinfo*.

6.6.1 Configuration Files

6.6.1.1 *ThisCell*

The *Cache Manager*, along with various applications, needs to be able to determine the cell to which its client machine belongs. This information is provided by the *ThisCell* file. It contains a single line stating the machine's fully-qualified cell name.

As with the *CellServDB* configuration file, the *Cache Manager* reads the contents of *ThisCell* exactly once, at start-up time. Thus, an incarnation of the *Cache Manager* will maintain precisely one notion of its home cell for its entire lifetime. Thus, changes to the text of the *ThisCell* file will be invisible to the running *Cache Manager*. However, these changes will affect such application programs as `klog`, which allows a user to generate new authentication tickets. In this example, `klog` reads *ThisCell* every time it is invoked, and then interacts with the set of *Authentication Server*s running in the given home cell, unless the caller specifies the desired cell on the command line.

The *ThisCell* file is not expected to be changed on a regular basis. Client machines are not imagined to be frequently traded between different administrative organizations. The Unix mode bits are set to specify that while everyone is allowed to read the file, only `root` is allowed to modify it.

6.6.1.2 *CellServDB*

To conduct business with a given AFS cell, a *Cache Manager* must be informed of the cell's name and the set of machines running AFS database servers within that cell. Such servers include the *Volume Location Server*, *Authentication Server*, and *Protection Server*. This particular cell information is obtained upon startup by reading the *CellServDB* file. Thus, when the *Cache Manager* initialization is complete, it will be able to communicate with the cells covered by *CellServDB*.

The following is an excerpt from a valid *CellServDB* file, demonstrating the format used.

```
    .
    .
    .
>transarc.com           #Transarc Corporation
192.55.207.7                    #henson.transarc.com
192.55.207.13                   #bigbird.transarc.com
192.55.207.22                   #ernie.transarc.com
>andrew.cmu.edu         #Carnegie Mellon University
128.2.10.2                      #vice2.fs.andrew.cmu.edu
128.2.10.7                      #vice7.fs.andrew.cmu.edu
128.2.10.10                     #vice10.fs.andrew.cmu.edu
    .
    .
    .
```

There are four rules describing the legal *CellServDB* file format:

1. Each cell has a separate entry. The entries may appear in any order. It may be convenient, however, to have the workstation's local cell be the first to appear.
2. No blank lines should appear in the file, even at the end of the last entry.
3. The first line of each cell's entry begins with the ">" character, and specifies the cell's human-readable, Internet Domain-style name. Optionally, some white space and a comment (preceded by a "#") may follow, briefly describing the specified cell.
4. Each subsequent line in a cell's entry names one of the cell's database server machines. The following must appear on the line, in the order given:
   - The Internet address of the server, in the standard 4-component dot notation.
   - Some amount of whitespace.
   - A "#", followed by the machine's complete Internet host name. **In this instance, the "#" sign and the text beyond it specifying the machine name are NOT treated as a comment. This is required information.** The *Cache Manager* will use the given host name to determine its current address via an Internet Domain lookup. If and only if this lookup fails does the *Cache Manager* fall back to using the dotted Internet address on the first part of the line. This dotted address thus appears simply as a hint in case of Domain database downtime.

The *CellServDB* file is only parsed once, when the *Cache Manager* first starts. It is possible, however, to amend existing cell information records or add completely new ones at any time after *Cache Manager* initialization completes. This is accomplished via the `VIOCNEWCELL` **pioctl()** (see Section 6.4.5.1.

6.6.1.3 *cacheinfo*

This one-line file contains three fields separated by colons:

- **AFS Root Directory**: This is the directory where the *Cache Manager* mounts the AFS root volume. Typically, this is specified to be */afs*.
- **Cache Directory**: This field names the directory where the *Cache Manager* is to create its local cache files. This is typically set to */usr/vice/cache*.
- **Cache Blocks**: The final field states the upper limit on the number of 1,024-byte blocks that the *Cache Manager* is allowed to use in the partition hosting the named cache directory.

Thus, the following `cacheinfo` file would instruct the *Cache Manager* to mount the AFS filespace at */afs*, and inform it that it may expect to be able to use up to 25,000 blocks for the files in its cache directory, */usr/vice/cache*.

```
/afs:/usr/vice/cache:25000
```

6.6.2 Cache Information Files

6.6.2.1 *AFSLog*

This is the AFS log file used to hold *Cache Manager* debugging output. The file is set up when the *Cache Manager* first starts. If it already exists, it is truncated. If it doesn't, it is created. Output to this file is enabled and disabled via the the `VIOC_VENUSLOG` **pioctl()** (see Section 6.4.9.2). Normal text messages are written to this file by the *Cache Manager* when output is enabled. Each time logging to this file is enabled, the *AFSLog* file is truncated. Only `root` can read and write this file.

6.6.2.2 *CacheItems*

The *Cache Manager* only keeps a subset of its data cache entry descriptors in memory at once. The number of these in-memory descriptors is determined by *afsd*. All of the data cache entry descriptors are kept on disk, in the *CacheItems* file. The file begins with a header region, taking up four longwords:

```
struct fheader {
long magic AFS_FHMAGIC   0x7635fab8
long firstCSize: First chunk size
long otherCSize: Next chunk sizes
long spare
}
```

The header is followed by one entry for each cache file. Each is:

```
struct fcache {
    short hvNextp;              /* Next in vnode hash table, or freeDCList */
    short hcNextp;              /* Next index in [fid, chunk] hash table */
    short chunkNextp;           /* File queue of all chunks for a single vnode */
    struct VenusFid fid;        /* Fid for this file */
    long modTime;               /* last time this entry was modified */
    long versionNo;             /* Associated data version number */
    long chunk;                 /* Relative chunk number */
    long inode;                 /* Unix inode for this chunk */
    long chunkBytes;            /* Num bytes in this chunk */
    char states;                /* Has this chunk been modified? */
};
```

6.6.2.3 *VolumeItems*

The *Cache Manager* only keeps at most `MAXVOLS` (50) in-memory volume descriptions. However, it records all volume information it has obtained in the *VolumeItems* file in the chosen AFS cache directory. This file is truncated when the *Cache Manager* starts. Each volume record placed into this file has the following `struct fvolume` layout:

```
struct fvolume {
    long cell;                  /*Cell for this entry*/
    long volume;                /*Numerical volume ID */
    long next;                  /*Hash index*/
    struct VenusFid dotdot;     /*Full fid for .. dir */
    struct VenusFid mtpoint;    /*Full fid for mount point*/
};
```

6.7 Mariner Interface

The *Cache Manager* Mariner interface allows interested parties to be advised in real time as to which files and/or directories are being actively transferred between the client machine and one or more *File Server*s. If enabled, this service delivers messages of two different types, as exemplified below:

```
Fetching myDataDirectory
Fetching myDataFile.c
Storing myDataObj.o
```

In the first message, the *myDataDirectory* directory is shown to have just been fetched from a *File Server*. Similarly, the second message indicates that the C program *myDataFile.c* had just been fetched from its *File Server* of residence. Finally, the third message reveals that the *myDataObj.o* object file has just been written out over the network to its respective server.

In actuality, the text of the messages carries a string prefix to indicate whether a Fetch or Store operation had been performed. So, the full contents of the above messages are as follows:

```
fetch$Fetching myDataDirectory
fetch$Fetching myDataFile.c
store$Storing myDataObj.o
```

The Mariner service may be enabled or disabled for a particular machine by using the `VIOC_AFS_MARINER_HOST` **pioctl()** (see Section 6.4.9.1). This operation allows any host to be specified as the recipient of these messages. A potential recipient must have its host be declared the target of such messages, then listen to a socket on port 2106.

Internally, the *Cache Manager* maintains a cache of `NMAR` (10) vnode structure pointers and the string name (up to 19 characters) of the associated file or directory. This cache is implemented as an array serving as a circular buffer. Each time a file is involved in a create or lookup operation on a *File Server*, the current slot in this circular buffer is filled with the relevant vnode and string name information, and the current position is advanced. If Mariner output is enabled, then an actual network fetch or store operation will trigger messages of the kind shown above. Since a fetch or store operation normally occurs shortly after the create or lookup, the mapping of vnode to name is likely to still be in the Mariner cache when it comes time to generate the appropriate message. However, since an unbounded number of other lookups or creates could have been performed in the interim, there is no guarantee that the mapping entry will not have been overrun. In these instances, the Mariner message will be a bit vaguer. Going back to our original example,

```
Fetching myDataDirectory
Fetching a file
Storing myDataObj.o
```

In this case, the cached association between the vnode containing *myDataFile.c* and its string name was thrown out of the Mariner cache before the network fetch operation could be performed. Unable to find the mapping, the generic phrase "a file" was used to identify the object involved.

Mariner messages only get generated when RPC traffic for fetching or storing a file system object occurs between the *Cache Manager* and a *File Server*. Thus, file accesses that are handled by the *Cache Manager*'s on-board data cache do not trigger such announcements.

Appendix A

struct afs_CMCallStats

This appendix contains the full specification of the `afs_CMCallStats` structure, as originally described in Section 6.5.4.2. This construct is returned by the *Cache Manager* when a client requests data collection `AFSCB_XSTATSCOLL_CALL_INFO`, as defined in Section 6.5.3.

In the following tabular presentation, each field name listed is preceded by its (longword) offset in the `afs_CMCallStats` structure.

0. C_afs_init	1. C_gop_rdwr	2. C_aix_gnode_rele
3. C_gettimeofday	4. C_m_cpytoc	5. C_aix_vattr_null
6. C_afs_gn_ftrunc	7. C_afs_gn_rdwr	8. C_afs_gn_ioctl
9. C_afs_gn_lockctl	10. C_afs_gn_readlink	11. C_afs_gn_readdir
12. C_afs_gn_select	13. C_afs_gn_strategy	14. C_afs_gn_symlink
15. C_afs_gn_revoke	16. C_afs_gn_link	17. C_afs_gn_mkdir
18. C_afs_gn_mknod	19. C_afs_gn_remove	20. C_afs_gn_rename
21. C_afs_gn_rmdir	22. C_afs_gn_fid	23. C_afs_gn_lookup
24. C_afs_gn_open	25. C_afs_gn_create	26. C_afs_gn_hold
27. C_afs_gn_close	28. C_afs_gn_map	29. C_afs_gn_rele
30. C_afs_gn_unmap	31. C_afs_gn_access	32. C_afs_gn_getattr
33. C_afs_gn_setattr	34. C_afs_gn_fclear	35. C_afs_gn_fsync
36. C_pHash	37. C_DInit	38. C_DRead
39. C_FixupBucket	40. C_afs_newslot	41. C_DRelease
42. C_DFlush	43. C_DFlushEntry	44. C_DVOffset
45. C_DZap	46. C_DNew	47. C_shutdown_bufferpackage
48. C_afs_CheckKnownBad	49. C_afs_RemoveVCB	50. C_afs_NewVCache
51. C_afs_FlushActiveVcaches	52. C_afs_VerifyVCache	53. C_afs_WriteVCache
54. C_afs_GetVCache	55. C_afs_StuffVcache	56. C_afs_FindVCache
57. C_afs_PutDCache	58. C_afs_PutVCache	59. C_CacheStoreProc
60. C_afs_FindDCache	61. C_afs_TryToSmush	62. C_afs_AdjustSize
63. C_afs_CheckSize	64. C_afs_StoreWarn	65. C_CacheFetchProc

66. C_UFS_CacheStoreProc
67. C_UFS_CacheFetchProc
68. C_afs_GetDCache
69. C_afs_SimpleVStat
70. C_afs_ProcessFS
71. C_afs_InitCacheInfo
72. C_afs_InitVolumeInfo
73. C_afs_InitCacheFile
74. C_afs_CacheInit
75. C_afs_GetDSlot
76. C_afs_WriteThroughDSlots
77. C_afs_MemGetDSlot
78. C_afs_UFSGetDSlot
79. C_afs_StoreDCache
80. C_afs_StoreMini
81. C_shutdown_cache
82. C_afs_StoreAllSegments
83. C_afs_InvalidateAllSegments
84. C_afs_TruncateAllSegments
85. C_afs_CheckVolSync
86. C_afs_wakeup
87. C_afs_CFileOpen
88. C_afs_CFileTruncate
89. C_afs_GetDownD
90. C_afs_WriteDCache
91. C_afs_FlushDCache
92. C_afs_GetDownDSlot
93. C_afs_FlushVCache
94. C_afs_GetDownV
95. C_afs_QueueVCB
96. C_afs_call
97. C_afs_syscall_call
98. C_syscall
99. C_lpioctl
100. C_lsetpag
101. C_afs_syscall
102. C_afs_CheckInit
103. C_afs_shutdown
104. C_shutdown_BKG
105. C_shutdown_afstest
106. C_SRXAFSCB_GetCE
107. C_ClearCallBack
108. C_SRXAFSCB_GetLock
109. C_SRXAFSCB_CallBack
110. C_SRXAFSCB_InitCallBackState
111. C_SRXAFSCB_Probe
112. C_afs_RXCallBackServer
113. C_shutdown_CB
114. C_afs_Chunk
115. C_afs_ChunkBase
116. C_afs_ChunkOffset
117. C_afs_ChunkSize
118. C_afs_ChunkToBase
119. C_afs_ChunkToSize
120. C_afs_SetChunkSize
121. C_afs_config
122. C_mem_freebytes
123. C_mem_getbytes
124. C_fpalloc
125. C_kluge_init
126. C_ufdalloc
127. C_ufdfree
128. C_commit
129. C_dev_ialloc
130. C_ffree
131. C_iget
132. C_iptovp
133. C_ilock
134. C_irele
135. C_iput
136. C_afs_Daemon
137. C_afs_CheckRootVolume
138. C_BPath
139. C_BPrefetch
140. C_BStore
141. C_afs_BBusy
142. C_afs_BQueue
143. C_afs_BRelease
144. C_afs_BackgroundDaemon
145. C_shutdown_daemons
146. C_exporter_add
147. C_exporter_find
148. C_afs_gfs_kalloc
149. C_IsAfsVnode
150. C_SetAfsVnode
151. C_afs_gfs_kfree
152. C_gop_lookupname
153. C_gfsvop_getattr
154. C_gfsvop_rdwr
155. C_afs_uniqtime
156. C_gfs_vattr_null
157. C_afs_lock
158. C_afs_unlock
159. C_afs_update
160. C_afs_gclose
161. C_afs_gopen
162. C_afs_greadlink
163. C_afs_select
164. C_afs_gbmap
165. C_afs_getfsdata
166. C_afs_gsymlink
167. C_afs_namei
168. C_printgnode
169. C_HaveGFSLock
170. C_afs_gmount
171. C_AddGFSLock
172. C_RemoveGFSLock
173. C_afs_grlock
174. C_afs_gumount
175. C_afs_gget
176. C_afs_glink
177. C_afs_gmkdir
178. C_afs_sbupdate
179. C_afs_unlink
180. C_afs_grmdir
181. C_afs_makenode
182. C_afs_grename
183. C_afs_rele
184. C_afs_syncgp
185. C_afs_getval
186. C_afs_gfshack
187. C_afs_trunc
188. C_afs_rwgp
189. C_afs_stat
190. C_afsc_link
191. C_hpsobind
192. C_hpsoclose
193. C_hpsocreate
194. C_hpsoreserve
195. C_afs_vfs_mount
196. C_devtovfs
197. C_igetinode
198. C_afs_syscall_iopen
199. C_iopen
200. C_afs_syscall_iincdec
201. C_afs_syscall_ireadwrite
202. C_iincdec
203. C_ireadwrite
204. C_oiread
205. C_AHash
206. C_QTOA
207. C_afs_FindPartByDev
208. C_aux_init
209. C_afs_GetNewPart
210. C_afs_InitAuxVolFile
211. C_afs_CreateAuxEntry
212. C_afs_GetAuxSlot
213. C_afs_GetDownAux
214. C_afs_FlushAuxCache
215. C_afs_GetAuxInode

216. C_afs_PutAuxInode
217. C_afs_ReadAuxInode
218. C_afs_WriteAuxInode
219. C_afs_auxcall
220. C_tmpdbg_auxtbl
221. C_tmpdbg_parttbl
222. C_idec
223. C_iinc
224. C_iread
225. C_iwrite
226. C_getinode
227. C_trygetfs
228. C_iforget
229. C_afs_syscall_icreate
230. C_icreate
231. C_Lock_Init
232. C_Lock_Obtain
233. C_Lock_ReleaseR
234. C_Lock_ReleaseW
235. C_afs_BozonLock
236. C_afs_BozonUnlock
237. C_osi_SleepR
238. C_osi_SleepS
239. C_osi_SleepW
240. C_osi_Sleep
241. C_afs_BozonInit
242. C_afs_CheckBozonLock
243. C_xxxinit
244. C_KernelEntry
245. C_afs_InitMemCache
246. C_afs_LookupMCE
247. C_afs_MemReadBlk
248. C_afs_MemReadUIO
249. C_afs_MemWriteBlk
250. C_afs_MemCacheStoreProc
251. C_afs_MemCacheTruncate
252. C_afs_MemWriteUIO
253. C_afs_MemCacheFetchProc
254. C_afs_vnode_pager_create
255. C_next_KernelEntry
256. C_afs_GetNfsClientPag
257. C_afs_FindNfsClientPag
258. C_afs_PutNfsClientPag
259. C_afs_nfsclient_reqhandler
260. C_afs_nfsclient_GC
261. C_afs_nfsclient_hold
262. C_afs_nfsclient_stats
263. C_afs_nfsclient_sysname
264. C_afs_nfsclient_shutdown
265. C_afs_rfs_readdir_fixup
266. C_afs_rfs_dispatch
267. C_afs_xnfs_svc
268. C_afs_xdr_putrddirres
269. C_afs_rfs_readdir
270. C_afs_rfs_rddirfree
271. C_rfs_dupcreate
272. C_rfs_dupsetattr
273. C_Nfs2AfsCall
274. C_afs_sun_xuntext
275. C_osi_Active
276. C_osi_FlushPages
277. C_osi_FlushText
278. C_osi_CallProc
279. C_osi_CancelProc
280. C_osi_Invisible
281. C_osi_Time
282. C_osi_Alloc
283. C_osi_SetTime
284. C_osi_Dump
285. C_osi_Free
286. C_shutdown_osi
287. C_osi_UFSOpen
288. C_osi_Close
289. C_osi_Stat
290. C_osi_Truncate
291. C_osi_Read
292. C_osi_Write
293. C_osi_MapStrategy
294. C_shutdown_osifile
295. C_osi_FreeSendSpace
296. C_osi_FreeSmallSpace
297. C_pkt_iodone
298. C_shutdown_osinet
299. C_afs_cs
300. C_osi_AllocSendSpace
301. C_osi_AllocSmallSpace
302. C_osi_CloseToTheEdge
303. C_osi_xgreedy
304. C_osi_FreeSocket
305. C_osi_NewSocket
306. C_trysblock
307. C_osi_NetSend
308. C_WaitHack
309. C_osi_CancelWait
310. C_osi_InitWaitHandle
311. C_osi_Wakeup
312. C_osi_Wait
313. C_dirp_Read
314. C_dirp_SetCacheDev
315. C_Die
316. C_dirp_Cpy
317. C_dirp_Eq
318. C_dirp_Write
319. C_dirp_Zap
320. C_PSetVolumeStatus
321. C_PFlush
322. C_PNewStatMount
323. C_PGetTokens
324. C_PUnlog
325. C_PCheckServers
326. C_PMariner
327. C_PCheckAuth
328. C_PCheckVolNames
329. C_PFindVolume
330. C_Prefetch
331. C_PGetCacheSize
332. C_PRemoveCallBack
333. C_PSetCacheSize
334. C_PViceAccess
335. C_PListCells
336. C_PNewCell
337. C_PRemoveMount
338. C_HandleIoctl
339. C__AFSIOCTL
340. C__VALIDAFSIOCTL
341. C_PGetCellStatus
342. C_PSetCellStatus
343. C_PVenusLogging
344. C_PFlushVolumeData
345. C_PSetSysName
346. C_PExportAfs
347. C_HandleClientContext
348. C_afs_ioctl
349. C_afs_xioctl
350. C_afs_pioctl
351. C_afs_syscall_pioctl
352. C_HandlePioctl
353. C_PGetAcl
354. C_PGetFID
355. C_PSetAcl
356. C_PBogus
357. C_PGetFileCell
358. C_PGetWSCell
359. C_PNoop
360. C_PGetUserCell
361. C_PSetTokens
362. C_PGetVolumeStatus
363. C_afs_ResetAccessCache
364. C_afs_FindUser
365. C_afs_ResetUserConns

366. C_afs_ResourceInit
367. C_afs_GetCell
368. C_afs_GetCellByIndex
369. C_afs_GetCellByName
370. C_afs_NewCell
371. C_afs_GetUser
372. C_afs_PutUser
373. C_afs_SetPrimary
374. C_CheckVLDB
375. C_afs_GetVolume
376. C_afs_GetVolumeByName
377. C_InstallVolumeEntry
378. C_InstallVolumeInfo
379. C_afs_FindServer
380. C_afs_PutVolume
381. C_afs_random
382. C_ranstage
383. C_RemoveUserConns
384. C_afs_MarinerLog
385. C_afs_vtoi
386. C_afs_GetServer
387. C_afs_SortServers
388. C_afs_Conn
389. C_afs_ConnByHost
390. C_afs_ConnByMHosts
391. C_afs_Analyze
392. C_afs_PutConn
393. C_afs_ResetVolumeInfo
394. C_StartLogFile
395. C_afs_SetLogFile
396. C_EndLogFile
397. C_afs_dp
398. C_fprf
399. C_fprint
400. C_fprintn
401. C_afs_CheckLocks
402. C_puttofile
403. C_shutdown_AFS
404. C_afs_CheckCacheResets
405. C_afs_GCUserData
406. C_VSleep
407. C_afs_CheckCode
408. C_afs_CopyError
409. C_afs_FinalizeReq
410. C_afs_cv2string
411. C_afs_FindVolCache
412. C_afs_GetVolCache
413. C_afs_GetVolSlot
414. C_afs_WriteVolCache
415. C_afs_UFSGetVolSlot
416. C_afs_CheckVolumeNames
417. C_afs_MemGetVolSlot
418. C_print_internet_address
419. C_CheckVLServer
420. C_HaveCallBacksFrom
421. C_ServerDown
422. C_afs_CheckServers
423. C_afs_AddToMean
424. C_afs_GetCMStat
425. C_afs_getpage
426. C_afs_putpage
427. C_afs_nfsrdwr
428. C_afs_map
429. C_afs_cmp
430. C_afs_cntl
431. C_afs_dump
432. C_afs_realvp
433. C_afs_PageLeft
434. C_afsinit
435. C_afs_mount
436. C_afs_unmount
437. C_afs_root
438. C_afs_statfs
439. C_afs_sync
440. C_afs_vget
441. C_afs_mountroot
442. C_afs_swapvp
443. C_afs_AddMarinerName
444. C_afs_setpag
445. C_genpag
446. C_getpag
447. C_afs_GetMariner
448. C_afs_badop
449. C_afs_index
450. C_afs_noop
451. C_afs_open
452. C_afs_closex
453. C_afs_close
454. C_afs_MemWrite
455. C_afs_write
456. C_afs_UFSWrite
457. C_afs_rdwr
458. C_afs_MemRead
459. C_afs_read
460. C_FIXUPSTUPIDINODE
461. C_afs_UFSRead
462. C_afs_CopyOutAttrs
463. C_afs_getattr
464. C_afs_VAttrToAS
465. C_afs_setattr
466. C_EvalMountPoint
467. C_afs_access
468. C_ENameOK
469. C_HandleAtName
470. C_getsysname
471. C_strcat
472. C_afs_lookup
473. C_afs_create
474. C_afs_LocalHero
475. C_FetchWholeEnchilada
476. C_afs_remove
477. C_afs_link
478. C_afs_rename
479. C_afs_InitReq
480. C_afs_mkdir
481. C_BlobScan
482. C_afs_rmdir
483. C_RecLen
484. C_RoundToInt
485. C_afs_readdir_with_offlist
486. C_DIRSIZ_LEN
487. C_afs_readdir_move
488. C_afs_readdir_iter
489. C_HandleFlock
490. C_afs_readdir
491. C_afs_symlink
492. C_afs_HandleLink
493. C_afs_MemHandleLink
494. C_afs_UFSHandleLink
495. C_afs_readlink
496. C_afs_fsync
497. C_afs_inactive
498. C_afs_ustrategy
499. C_afs_bread
500. C_afs_brelse
501. C_afs_bmap
502. C_afs_fid
503. C_afs_strategy
504. C_afs_FakeClose
505. C_afs_FakeOpen
506. C_afs_StoreOnLastReference
507. C_afs_GetAccessBits
508. C_afs_AccessOK
509. C_shutdown_vnodeops
510. C_afsio_copy
511. C_afsio_trim
512. C_afs_page_read
513. C_afs_page_write
524. C_afsio_skip
515. C_afs_read1dir

516. C_afs_get_groups_from_pag
517. C_afs_get_pag_from_groups
518. C_PagInCred
519. C_afs_getgroups
520. C_setpag
521. C_afs_setgroups
522. C_afs_page_in
523. C_afs_page_out
524. C_AddPag
525. C_afs_AdvanceFD
526. C_afs_lockf
527. C_afs_xsetgroups
528. C_afs_nlinks
529. C_DoLockWarning
530. C_afs_lockctl
531. C_afs_xflock

Bibliography

[1] Transarc Corporation. *AFS 3.0 System Administrator's Guide*, F-30-0-D102, Pittsburgh, PA, April 1990.

[2] Transarc Corporation. *AFS 3.0 Command Reference Manual*, F-30-0-D103, Pittsburgh, PA, April 1990.

[3] CMU Information Technology Center. *Synchronization and Caching Issues in the Andrew File System*, USENIX Proceedings, Dallas, TX, Winter 1988.

[4] Sun Microsystems, Inc. *NFS: Network File System Protocol Specification*, RFC 1094, March 1989.

[5] Sun Microsystems, Inc. *Design and Implementation of the Sun Network File System*, USENIX Summer Conference Proceedings, June 1985.

[6] S.P. Miller, B.C. Neuman, J.I. Schiller, J.H. Saltzer. *Kerberos Authentication and Authorization System*, Project Athena Technical Plan, Section E.2.1, M.I.T., December 1987.

[7] Bill Bryant. *Designing an Authentication System: a Dialogue in Four Scenes*, Project Athena internal document, M.I.T, draft of 8 February 1988.

Index