AFS-3 Programmer's Reference: Architectural Overview

Edward R. Zayas

Transarc Corporation

Version 1.0 of 2 September 1991 22:53

FS-00-D160

Contents

Chapter 1

Introduction

1.1 Goals and Background

This paper provides an architectural overview of Transarc's wide-area distributed file system, AFS. Specifically, it covers the current level of available software, the third-generation AFS-3 system. This document will explore the technological climate in which AFS was developed, the nature of problem(s) it addresses, and how its design attacks these problems in order to realize the inherent benefits in such a file system. It also examines a set of additional features for AFS, some of which are actively being considered.

This document is a member of a reference suite providing programming specifications as to the operation of and interfaces offered by the various AFS system components. It is intended to serve as a high-level treatment of distributed file systems in general and of AFS in particular. This document should ideally be read before any of the others in the suite, as it provides the organizational and philosophical framework in which they may best be interpreted.

1.2 Document Layout

Chapter 2 provides a discussion of the technological background and developments that created the environment in which AFS and related systems were inspired. Chapter 3 examines the specific set of goals that AFS was designed to meet, given the possibilities created by personal computing and advances in communication technology. Chapter 4 presents the core AFS architecture and how it addresses these goals. Finally, Chapter 5 considers how AFS functionality may be be improved by certain design changes.

1.3 Related Documents

The names of the other documents in the collection, along with brief summaries of their contents, are listed below.

- *AFS-3 Programmer's Reference: File Server/Cache Manager Interface*: This document describes the *File Server* and *Cache Manager* agents, which provide the backbone file managment services for AFS. The collection of *File Server*s for a cell supplies centralized file storage for that site, and allows clients running the *Cache Manager* component to access those files in a high-performance, secure fashion.
- *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 mapping printable user names to and from their internal AFS identifiers. The *Protection Server* also allows users to create, destroy, and manipulate "groups" of users, which are suitable for placement on Access Control Lists (ACLs).
- *AFS-3 Programmer's Reference: BOS Server Interface*: This paper covers the "nanny" service 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.

Chapter 2

Technological Background

Certain changes in technology over the past two decades greatly influenced the nature of computational resources, and the manner in which they were used. These developments created the conditions under which the notion of a **distributed file systems (DFS)** was born. This chapter describes these technological changes, and explores how a distributed file system attempts to capitalize on the new computing environment's strengths and minimize its disadvantages.

2.1 Shift in Computational Idioms

By the beginning of the 1980's, new classes of computing engines and new methods by which they may be interconnected were becoming firmly established. At this time, a shift was occurring away from the conventional mainframe-based, timeshared computing environment to one in which both workstation-class machines and the smaller personal computers (PCs) were a strong presence.

The new environment offered many benefits to its users when compared with timesharing. These smaller, self-sufficient machines moved dedicated computing power and cycles directly onto people's desks. Personal machines were powerful enough to support a wide variety of applications, and allowed for a richer, more intuitive, more graphically-based interface for them. Learning curves were greatly reduced, cutting training costs and increasing new-employee productivity. In addition, these machines provided a constant level of service throughout the day. Since a personal machine was typically only executing programs for a single human user, it did not suffer from timesharing's load-based response time degradation. Expanding the computing services for an organization was often accomplished by simply purchasing more of the relatively cheap machines. Even small organizations could now afford their own computing resources, over which they exercised full control. This provided more freedom to tailor computing services to the specific needs of particular groups.

However, many of the benefits offered by the timesharing systems were lost when the computing idiom first shifted to include personal-style machines. One of the prime casualties of this shift was the loss of the notion of a single name space for all files. Instead, workstation- and PC-based environments each had independent and completely disconnected file systems. The standardized mechanisms through which files could be transferred between machines (e.g., FTP) were largely designed at a time when there were relatively few large machines that were connected over slow links. Although the newer multi-megabit per second communication pathways allowed for faster transfers, the problem of resource location in this environment was still not addressed. There was no longer a system-wide file system, or even a file location service, so individual users were more isolated from the organization's collective data. Overall, disk requirements ballooned, since lack of a shared file system was often resolved by replicating all programs and data to each machine that needed it. This proliferation of independent copies further complicated the problem of version control and management in this distributed world. Since computers were often no longer behind locked doors at a computer center, user authentication and authorization tasks became more complex. Also, since organizational managers were now in direct control of their computing facilities, they had to also actively manage the hardware and software upon which they depended.

Overall, many of the benefits of the proliferation of independent, personal-style machines were partially offset by the communication and organizational penalties they imposed. Collaborative work and dissemination of information became more difficult now that the previously unified file system was fragmented among hundreds of autonomous machines.

2.2 Distributed File Systems

As a response to the situation outlined above, the notion of a **distributed file system (DFS)** was developed. Basically, a DFS provides a framework in which access to files is permitted regardless of their locations. Specifically, a distributed file system offers a single, common set of file system operations through which those accesses are performed.

There are two major variations on the core DFS concept, classified according to the way in which file storage is managed. These high-level models are defined below.

- **Peer-to-peer**: In this symmetrical model, each participating machine provides storage for specific set of files on its own attached disk(s), and allows others to access them remotely. Thus, each node in the DFS is capable of both *importing* files (making reference to files resident on foreign machines) and *exporting* files (allowing other machines to reference files located locally).

- **Server-client**: In this model, a set of machines designated as *servers* provide the storage for all of the files in the DFS. All other machines, known as *clients*, must direct their file references to these machines. Thus, servers are the sole exporters of files in the DFS, and clients are the sole importers.

The notion of a DFS, whether organized using the peer-to-peer or server-client discipline, may be used as a conceptual base upon which the advantages of personal computing resources can be combined with the single-system benefits of classical timeshared operation.

Many distributed file systems have been designed and deployed, operating on the fast local area networks available to connect machines within a single site. These systems include DOMAIN [9], DS [15], RFS [16], and Sprite [10]. Perhaps the most widespread of distributed file systems to date is a product from Sun Microsystems, NFS [13] [14], extending the popular UNIX file system so that it operates over local networks.

2.3 Wide-Area Distributed File Systems

Improvements in long-haul network technology are allowing for faster interconnection bandwidths and smaller latencies between distant sites. Backbone services have been set up across the country, and T1 (1.5 megabit/second) links are increasingly available to a larger number of locations. Long-distance channels are still at best approximately an order of magnitude slower than the typical local area network, and often two orders of magnitude slower. The narrowed *difference* between local-area and wide-area data paths opens the window for the notion of a **wide-area distributed file system (WADFS)**. In a WADFS, the transparency of file access offered by a local-area DFS is extended to cover machines across much larger distances. Wide-area file system functionality facilitates collaborative work and dissemination of information in this larger theater of operation.

Chapter 3

AFS-3 Design Goals

3.1 Introduction

This chapter describes the goals for the AFS-3 system, the first commercial WADFS in existence.

The original AFS goals have been extended over the history of the project. The initial AFS concept was intended to provide a single distributed file system facility capable of supporting the computing needs of Carnegie Mellon University, a community of roughly 10,000 people. It was expected that most CMU users either had their own workstation-class machine on which to work, or had access to such machines located in public *clusters*. After being successfully implemented, deployed, and tuned in this capacity, it was recognized that the basic design could be augmented to link autonomous AFS installations located within the greater CMU campus. As described in Section 2.3, the long-haul networking environment developed to a point where it was feasible to further extend AFS so that it provided wide-area file service. The underlying AFS communication component was adapted to better handle the widely-varying channel characteristics encountered by intra-site and inter-site operations.

A more detailed history of AFS evolution may be found in [3] and [18].

3.2 System Goals

At a high level, the AFS designers chose to extend the single-machine UNIX computing environment into a WADFS service. The UNIX system, in all of its numerous incarnations, is an important computing standard, and is in very wide use. Since AFS was originally intended to service the heavily UNIX-oriented CMU campus, this decision served an important tactical purpose along with its strategic ramifications.

In addition, the server-client discipline described in Section 2.2 was chosen as the organizational base for AFS. This provides the notion of a central file store serving as the primary residence for files within a given organization. These centrally-stored files are maintained by server machines and are made accessible to computers running the AFS client software.

Listed in the following sections are the primary goals for the AFS system. Chapter 4 examines how the AFS design decisions, concepts, and implementation meet this list of goals.

3.2.1 Scale

AFS differs from other existing DFSs in that it has the specific goal of supporting a very large user community with a small number of server machines. Unlike the rule-of-thumb ratio of approximately 20 client machines for every server machine (20:1) used by Sun Microsystem's widespread NFS distributed file system, the AFS architecture aims at smoothly supporting client/server ratios more along the lines of 200:1 within a single installation. In addition to providing a DFS covering a single organization with tens of thousands of users, AFS also aims at allowing thousands of independent, autonomous organizations to join in the single, shared name space (see Section 3.2.2 below) without a centralized control or coordination point. Thus, AFS envisions supporting the file system needs of tens of millions of users at interconnected yet autonomous sites.

3.2.2 Name Space

One of the primary strengths of the timesharing computing environment is the fact that it implements a single name space for all files in the system. Users can walk up to any terminal connected to a timesharing service and refer to its files by the identical name. This greatly encourages collaborative work and dissemination of information, as everyone has a common frame of reference. One of the major AFS goals is the extension of this concept to a WADFS. Users should be able to walk up to any machine acting as an AFS client, anywhere in the world, and use the identical file name to refer to a given object.

In addition to the common name space, it was also an explicit goal for AFS to provide complete **access transparency** and **location transparency** for its files. *Access* *transparency* is defined as the system's ability to use a single mechanism to operate on a file, regardless of its location, local or remote. *Location transparency* is defined as the inability to determine a file's location from its name. A system offering location transparency may also provide transparent file mobility, relocating files between server machines without visible effect to the naming system.

3.2.3 Performance

Good system performance is a critical AFS goal, especially given the scale, client-server ratio, and connectivity specifications described above. The AFS architecture aims at providing file access characteristics which, on average, are similar to those of local disk performance.

3.2.4 Security

A production WADFS, especially one which allows and encourages transparent file access between different administrative domains, must be extremely conscious of security issues. AFS assumes that server machines are "trusted" within their own administrative domain, being kept behind locked doors and only directly manipulated by reliable administrative personnel. On the other hand, AFS client machines are assumed to exist in inherently insecure environments, such as offices and dorm rooms. These client machines are recognized to be unsupervisable, and fully accessible to their users. This situation makes AFS servers open to attacks mounted by possibly modified client hardware, firmware, operating systems, and application software. In addition, while an organization may actively enforce the physical security of its own file servers to its satisfaction, other organizations may be lax in comparison. It is important to partition the system's security mechanism so that a security breach in one administrative domain does not allow unauthorized access to the facilities of other autonomous domains.

The AFS system is targeted to provide confidence in the ability to protect system data from unauthorized access in the above environment, where untrusted client hardware and software may attempt to perform direct remote file operations from anywhere in the world, and where levels of physical security at remote sites may not meet the standards of other sites.

3.2.5 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. This mechanism was considered too coarse-grained by the AFS designers. It was seen as insufficient for specifying the exact set of individuals and groups which may properly access any given file, as well as the operations these principals may perform. The UNIX group mechanism was also considered too coarse and inflexible. AFS was designed to provide more flexible and finer-grained control of file access, improving the ability to define the set of parties which may operate on files, and what their specific access rights are.

3.2.6 Reliability

The crash of a server machine in any distributed file system causes the information it hosts to become unavailable to the user community. The same effect is observed when server and client machines are isolated across a network partition. Given the potential size of the AFS user community, a single server crash could potentially deny service to a very large number of people. The AFS design reflects a desire to minimize the visibility and impact of these inevitable server crashes.

3.2.7 Administrability

Driven once again by the projected scale of AFS operation, one of the system's goals is to offer easy administrability. With the large projected user population, the amount of file data expected to be resident in the shared file store, and the number of machines in the environment, a WADFS could easily become impossible to administer unless its design allowed for easy monitoring and manipulation of system resources. It is also imperative to be able to apply security and access control mechanisms to the administrative interface.

3.2.8 Interoperability/Coexistence

Many organizations currently employ other distributed file systems, most notably Sun Microsystem's NFS, which is also an extension of the basic single-machine UNIX system. It is unlikely that AFS will receive significant use if it cannot operate concurrently with other DFSs without mutual interference. Thus, coexistence with other DFSs is an explicit AFS goal.

A related goal is to provide a way for other DFSs to interoperate with AFS to various degrees, allowing AFS file operations to be executed from these competing systems. This is advantageous, since it may extend the set of machines which are capable of interacting with the AFS community. Hardware platforms and/or operating systems to which AFS is not ported may thus be able to use their native DFS system to perform AFS file references.

These two goals serve to extend AFS coverage, and to provide a migration path by which potential clients may sample AFS capabilities, and gain experience with AFS. This may result in data migration into native AFS systems, or the impetus to acquire a native AFS implementation.

3.2.9 Heterogeneity/Portability

It is important for AFS to operate on a large number of hardware platforms and operating systems, since a large community of unrelated organizations will most likely utilize a wide variety of computing environments. The size of the potential AFS user community will be unduly restricted if AFS executes on a small number of platforms. Not only must AFS support a largely heterogeneous computing base, it must also be designed to be easily portable to new hardware and software releases in order to maintain this coverage over time.

Chapter 4

AFS High-Level Design

4.1 Introduction

This chapter presents an overview of the system architecture for the AFS-3 WADFS. Different treatments of the AFS system may be found in several documents, including [3], [4], [5], and [2]. Certain system features discussed here are examined in more detail in the set of accompanying AFS programmer specification documents.

After the archtectural overview, the system goals enumerated in Chapter 3 are revisited, and the contribution of the various AFS design decisions and resulting features is noted.

4.2 The AFS System Architecture

4.2.1 Basic Organization

As stated in Section 3.2, a server-client organization was chosen for the AFS system. A group of trusted server machines provides the primary disk space for the central store managed by the organization controlling the servers. File system operation requests for specific files and directories arrive at server machines from machines running the AFS client software. If the client is authorized to perform the operation, then the server proceeds to execute it.

In addition to this basic file access functionality, AFS server machines also provide related system services. These include authentication service, mapping between printable and numerical user identifiers, file location service, time service, and such administrative operations as disk management, system reconfiguration, and tape backup.

4.2.2 Volumes

4.2.2.1 Definition

Disk partitions used for AFS storage do not directly host individual user files and directories. Rather, connected subtrees of the system's directory structure are placed into containers called **volumes**. Volumes vary in size dynamically as the objects it houses are inserted, overwritten, and deleted. Each volume has an associated **quota**, or maximum permissible storage. A single UNIX disk partition may thus host one or more volumes, and in fact may host as many volumes as physically fit in the storage space. However, the practical maximum is currently 3,500 volumes per disk partition. This limitation is imposed by the *salvager* program, which examines and repairs file system metadata structures.

There are two ways to identify an AFS volume. The first option is a 32-bit numerical value called the **volume ID**. The second is a human-readable character string called the **volume name**.

Internally, a volume is organized as an array of mutable objects, representing individual files and directories. The file system object associated with each index in this internal array is assigned a **uniquifier** and a **data version number**. A subset of these values are used to compose an AFS file identifier, or **FID**. FIDs are not normally visible to user applications, but rather are used internally by AFS. They consist of ordered triplets, whose components are the volume ID, the index within the volume, and the uniquifier for the index.

To understand AFS FIDs, let us consider the case where index i in volume v refers to a file named *example.txt*. This file's uniquifier is currently set to one (1), and its data version number is currently set to zero (0). The AFS client software may then refer to this file with the following FID: (v, i, 1). The next time a client overwrites the object identified with the (v, i, 1) FID, the data version number for *example.txt* will be promoted to one (1). Thus, the data version number serves to distinguish between different versions of the same file. A higher data version number indicates a newer version of the file.

Consider the result of deleting file (v, i, 1). This causes the body of *example.txt* to be discarded, and marks index i in volume v as unused. Should another program create a file, say *a.out*, within this volume, index i may be reused. If it is, the creation operation will bump the index's uniquifier to two (2), and the data version number is reset to zero (0). Any client caching a FID for the deleted *example.txt* file thus cannot affect the completely unrelated *a.out* file, since the uniquifiers differ.

4.2.2.2 Attachment

The connected subtrees contained within individual volumes are attached to their proper places in the file space defined by a site, forming a single, apparently seamless UNIX tree. These attachment points are called mount points. These mount points are persistent file system 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.

A single volume is chosen as the root of the AFS file space for a given organization. By convention, this volume is named `root.afs`. Each client machine belonging to this organization peforms a UNIX *mount()* of this root volume (not to be confused with an AFS mount point) on its empty */afs* directory, thus attaching the entire AFS name space at this point.

4.2.2.3 Administrative Uses

Volumes serve as the administrative unit for AFS file system data, providing as the basis for **replication**, **relocation**, and **backup** operations.

4.2.2.4 Replication

Read-only snapshots of AFS volumes may be created by administrative personnel. These *clones* may be deployed on up to eight disk partitions, on the same server machine or across different servers. Each clone has the identical volume ID, which must differ from its read-write parent. Thus, at most one clone of any given volume v may reside on a given disk partition. File references to this read-only clone volume may be serviced by *any* of the servers which host a copy.

4.2.2.5 Backup

Volumes serve as the unit of tape backup and restore operations. Backups are accomplished by first creating an on-line *backup volume* for each volume to be archived. This backup volume is organized as a *copy-on-write shadow* of the original volume, capturing the volume's state at the instant that the backup took place. Thus, the backup volume may be envisioned as being composed of a set of object pointers back to the original image. The first update operation on the file located in index i of the original volume triggers the copy-on-write association. This causes the file's contents at the time of the snapshot to be physically written to the backup volume before the newer version of the file is stored in the parent volume.

Thus, AFS on-line backup volumes typically consume little disk space. On average, they are composed mostly of links and to a lesser extent the bodies of those few files which have been modified since the last backup took place. Also, the system does not have to be shut down to insure the integrity of the backup images. Dumps are generated from the unchanging backup volumes, and are transferred to tape at any convenient time before the next backup snapshot is performed.

4.2.2.6 Relocation

Volumes may be moved transparently between disk partitions on a given file server, or between different file server machines. The transparency of volume motion comes from the fact that neither the user-visible names for the files nor the internal AFS FIDs contain server-specific location information.

Interruption to file service while a volume move is being executed is typically on the order of a few seconds, regardless of the amount of data contained within the volume. This derives from the staged algorithm used to move a volume to a new server. First, a dump is taken of the volume's contents, and this image is installed at the new site. The second stage involves actually locking the original volume, taking an incremental dump to capture file updates since the first stage. The third stage installs the changes at the new site, and the fourth stage deletes the original volume. Further references to this volume will resolve to its new location.

4.2.3 Authentication

AFS uses the Kerberos [22] [23] authentication system developed at MIT's Project Athena to provide reliable identification of the principals attempting to operate on the files in its central store. Kerberos provides for **mutual authentication**, not only assuring AFS servers that they are interacting with the stated user, but also assuring AFS clients that they are dealing with the proper server entities and not imposters. Authentication information is mediated through the use of *tickets*. Clients register passwords with the authentication system, and use those passwords during authentication sessions to secure these tickets. A ticket is an object which contains an encrypted version of the user's name and other information. The file server machines may request a caller to present their ticket in the course of a file system operation. If the file server can successfully decrypt the ticket, then it knows that it was created and delivered by the authentication system, and may trust that the caller is the party identified within the ticket.

Such subjects as mutual authentication, encryption and decryption, and the use of session keys are complex ones. Readers are directed to the above references for a complete treatment of Kerberos-based authentication.

4.2.4 Authorization

4.2.4.1 Access Control Lists

AFS implements per-directory **Access Control Lists (ACLs)** to improve the ability to specify which sets of users have access to the files within the directory, and which operations they may perform. ACLs are used *in addition* to the standard UNIX mode bits. ACLs are organized as lists of one or more *(principal, rights)* pairs. A *principal* may be either the name of an individual user or a group of individual users. There are seven expressible rights, as listed below.

- **Read (r)**: The ability to read the contents of the files in a directory.
- **Lookup (l)**: The ability to look up names in a directory.
- **Write (w)**: The ability to create new files and overwrite the contents of existing files in a directory.
- **Insert (i)**: The ability to insert new files in a directory, but *not* to overwrite existing files.
- **Delete (d)**: The ability to delete files in a directory.
- **Lock (k)**: The ability to acquire and release advisory locks on a given directory.
- **Administer (a)**: The ability to change a directory's ACL.

4.2.4.2 AFS Groups

AFS users may create a certain number of *groups*, differing from the standard UNIX notion of group. These AFS groups are objects that may be placed on ACLs, and simply contain a list of AFS user names that are to be treated identically for authorization purposes. For example, user `erz` may create a group called `erz:friends` consisting of the `kazar`, `vasilis`, and `mason` users. Should `erz` wish to grant read, lookup, and insert rights to this group in directory *d*, he should create an entry reading *(erz:friends, rli)* in *d*'s ACL.

AFS offers three special, built-in groups, as described below.

1. **system:anyuser**: Any individual who accesses AFS files is considered by the system to be a member of this group, whether or not they hold an authentication ticket. This group is unusual in that it doesn't have a stable membership. In fact, it doesn't have an explicit list of members. Instead, the `system:anyuser` "membership" grows and shrinks as file accesses occur, with users being (conceptually) added and deleted automatically as they interact with the system.

   The `system:anyuser` group is typically put on the ACL of those directories for which some specific level of completely public access is desired, covering any user at any AFS site.

2. **system:authuser**: Any individual in possession of a valid Kerberos ticket minted by the organization's authentication service is treated as a member of this group. Just as with `system:anyuser`, this special group does not have a stable membership. If a user acquires a ticket from the authentication service, they are automatically "added" to the group. If the ticket expires or is discarded by the user, then the given individual will automatically be "removed" from the group.

   The `system:authuser` group is usually put on the ACL of those directories for which some specific level of intra-site access is desired. Anyone holding a valid ticket within the organization will be allowed to perform the set of accesses specified by the ACL entry, regardless of their precise individual ID.

3. **system:administrators**: This built-in group defines the set of users capable of performing certain important administrative operations within the cell. Members of this group have explicit "a" (ACL administration) rights on every directory's ACL in the organization. Members of this group are the only ones which may legally issue administrative commands to the file server machines within the organization. This group is *not* like the other two described above in that it *does* have a stable membership, where individuals are added and deleted from the group explicitly.

The `system:administrators` group is typically put on the ACL of those directories which contain sensitive administrative information, or on those places where only administrators are allowed to make changes. All members of this group have implicit rights to change the ACL on any AFS directory within their organization. Thus, they don't have to actually appear on an ACL, or have "a" rights enabled in their ACL entry if they do appear, to be able to modify the ACL.

4.2.5 Cells

A cell is the set of server and client machines managed and operated by an administratively independent organization, as fully described in the original proposal [17] and specification [18] documents. The cell's 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 where to obtain default system resources and services.

The cell concept allows autonomous sites to retain full administrative control over their facilities while allowing them to collaborate in the establishment of a single, common name space composed of the union of their individual name spaces. By convention, any file name beginning with */afs* is part of this shared global name space and can be used at any AFS-capable machine. The original mount point concept was modified to contain cell information, allowing volumes housed in foreign cells to be mounted in the file space. Again by convention, the top-level */afs* directory contains a mount point to the `root.cell` volume for each cell in the AFS community, attaching their individual file spaces. Thus, the top of the data tree managed by cell `xyz` is represented by the */afs/xyz* directory.

Creating a new AFS cell is straightforward, with the operation taking three basic steps:

1. **Name selection**: A prospective site has to first select a unique name for itself. Cell name selection is inspired by the hierarchical Domain naming system. Domain-style names are designed to be assignable in a completely decentralized fashion. Example cell names are `transarc.com`, `ssc.gov`, and `umich.edu`. These names correspond to the AFS installations at Transarc Corporation in Pittsburgh, PA, the Superconducting Supercollider Lab in Dallas, TX, and the University of Michigan at Ann Arbor, MI. respectively.
2. **Server installation**: Once a cell name has been chosen, the site must bring up one or more AFS file server machines, creating a local file space and a suite of local

services, including authentication (Section 4.2.6.4) and volume location (Section 4.2.6.2).

3. **Advertise services**: In order for other cells to discover the presence of the new site, it must advertise its name and which of its machines provide basic AFS services such as authentication and volume location. An established site may then record the machines providing AFS system services for the new cell, and then set up its mount point under */afs*. By convention, each cell places the top of its file tree in a volume named `root.cell`.

4.2.6 Implementation of Server Functionality

AFS server functionality is implemented by a set of user-level processes which execute on server machines. This section examines the role of each of these processes.

4.2.6.1 *File Server*

This AFS entity is responsible for providing a central disk repository for a particular set of files within volumes, and for making these files accessible to properly-authorized users running on client machines.

4.2.6.2 *Volume Location Server*

The *Volume Location Server* maintains and exports the **Volume Location Database (VLDB)**. This database tracks the server or set of servers on which volume instances reside. Among the operations it supports are queries returning volume location and status information, volume ID management, and creation, deletion, and modification of VLDB entries.

The VLDB may be replicated to two or more server machines for availability and load-sharing reasons. A *Volume Location Server* process executes on each server machine on which a copy of the VLDB resides, managing that copy.

4.2.6.3 *Volume Server*

The *Volume Server* allows administrative tasks and probes to be performed on the set of AFS volumes residing on the machine on which it is running. These operations include volume creation and deletion, renaming volumes, dumping and restoring volumes, altering the list of replication sites for a read-only volume, creating and propagating a new read-only volume image, creation and update of backup volumes, listing all volumes on a partition, and examining volume status.

4.2.6.4 *Authentication Server*

The AFS *Authentication Server* maintains and exports the **Authentication Database (ADB)**. This database tracks the encrypted passwords of the cell's users. The *Authentication Server* interface allows operations that manipulate ADB entries. It also implements the Kerberos mutual authentication protocol, supplying the appropriate identification tickets to successful callers.

The ADB may be replicated to two or more server machines for availability and load-sharing reasons. An *Authentication Server* process executes on each server machine on which a copy of the ADB resides, managing that copy.

4.2.6.5 *Protection Server*

The *Protection Server* maintains and exports the **Protection Database (PDB)**, which maps between printable user and group names and their internal numerical AFS identifiers. The *Protection Server* also allows callers to create, destroy, query ownership and membership, and generally manipulate AFS user and group records.

The PDB may be replicated to two or more server machines for availability and load-sharing reasons. A *Protection Server* process executes on each server machine on which a copy of the PDB resides, managing that copy.

4.2.6.6 *BOS Server*

The *BOS Server* is an administrative tool which runs on each file server machine in a cell. This server is responsible for monitoring the health of the AFS agent processess on that machine. The *BOS Server* brings up the chosen set of AFS agents in the proper order after a system reboot, answers requests as to their status, and restarts them when they fail. It also accepts commands to start, suspend, or resume these processes, and install new server binaries.

4.2.6.7 *Update Server/Client*

The *Update Server* and *Update Client* programs are used to distribute important system files and server binaries. For example, consider the case of distributing a new *File Server* binary to the set of Sparcstation server machines in a cell. One of the Sparcstation servers is declared to be the distribution point for its machine class, and is configured to run an *Update Server*. The new binary is installed in the appropriate local directory on that Sparcstation distribution point. Each of the other Sparcstation servers runs an *Update Client* instance, which periodically polls the proper *Update Server*. The new *File Server* binary will be detected and copied over to the client. Thus, new server binaries need only be installed manually once per machine type, and the distribution to like server machines will occur automatically.

4.2.7 Implementation of Client Functionality

4.2.7.1 Introduction

The portion of the AFS WADFS which runs on each client machine is called the *Cache Manager*. This code, running within the client's kernel, is a user's representative in communicating and interacting with the *File Servers*. The *Cache Manager*'s primary responsibility is to create the illusion that the remote AFS file store resides on the client machine's local disk(s).

As implied by its name, the *Cache Manager* supports this illusion by maintaining a *cache* of files referenced from the central AFS store on the machine's local disk. All file operations executed by client application programs on files within the AFS name space are handled by the *Cache Manager* and are realized on these cached images. Client-side AFS references are directed to the *Cache Manager* via the standard **VFS** and **vnode** file system interfaces pioneered and advanced by Sun Microsystems [21]. The *Cache Manager* stores and fetches files to and from the shared AFS repository as necessary to satisfy these operations. It is responsible for parsing UNIX pathnames on *open()* operations and mapping each component of the name to the *File Server* or group of *File Servers* that house the matching directory or file.

The *Cache Manager* has additional responsibilities. It also serves as a reliable repository for the user's authentication information, holding on to their tickets and wielding them as necessary when challenged during *File Server* interactions. It caches volume location information gathered from probes to the VLDB, and keeps the client machine's local clock synchronized with a reliable time source.

4.2.7.2 Chunked Access

In previous AFS incarnations, whole-file caching was performed. Whenever an AFS file was referenced, the entire contents of the file were stored on the client's local disk. This approach had several disadvantages. One problem was that no file larger than the amount of disk space allocated to the client's local cache could be accessed.

AFS-3 supports *chunked* file access, allowing individual 64 kilobyte pieces to be fetched and stored. Chunking allows AFS files of any size to be accessed from a client. The chunk size is settable at each client machine, but the default chunk size of 64K was chosen so that most UNIX files would fit within a single chunk.

4.2.7.3 Cache Management

The use of a file cache by the AFS client-side code, 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 at the server. 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.

4.2.8 Communication Substrate: *Rx*

All AFS system agents employ **remote procedure call (RPC)** interfaces. Thus, servers may be queried and operated upon regardless of their location.

The *Rx* RPC package is used by all AFS agents to provide a high-performance, multi-threaded, and secure communication mechanism. The *Rx* protocol is adaptive, conforming itself to widely varying network communication media encountered by a WADFS. It allows user applications to define and insert their own security modules, allowing them to execute the precise end-to-end authentication algorithms required to suit their specific needs and goals. *Rx* offers two built-in security modules. The first is the *null* module, which does not perform any encryption or authentication checks. The second built-in security module is *rxkad*, which utilizes Kerberos authentication.

Although pervasive throughout the AFS distributed file system, all of its agents, and many of its standard application programs, *Rx* is entirely separable from AFS and does not depend on any of its features. In fact, *Rx* can be used to build applications engaging in RPC-style communication under a variety of UNIX-style file systems. There are in-kernel and user-space implementations of the *Rx* facility, with both sharing the same interface.

4.2.9 Database Replication: *ubik*

The three AFS system databases (VLDB, ADB, and PDB) may be replicated to multiple server machines to improve their availability and share access loads among the replication sites. The *ubik* replication package is used to implement this functionality. A full description of *ubik* and of the *quorum completion* algorithm it implements may be found in [19] and [20].

The basic abstraction provided by *ubik* is that of a disk file replicated to multiple server locations. One machine is considered to be the *synchronization site*, handling all write operations on the database file. Read operations may be directed to any of the active members of the *quorum*, namely a subset of the replication sites large enough to insure integrity across such failures as individual server crashes and network partitions. All of the quorum members participate in regular *elections* to determine the current synchronization site. The *ubik* algorithms allow server machines to enter and exit the quorum in an orderly and consistent fashion.

All operations to one of these replicated "abstract files" are performed as part of a *transaction*. If all the related operations performed under a transaction are successful, then the transaction is *committed*, and the changes are made permanent. Otherwise, the transaction is *aborted*, and all of the operations for that transaction are undone.

Like *Rx*, the *ubik* facility may be used by client applications directly. Thus, user applicatons may easily implement the notion of a replicated disk file in this fashion.

4.2.10 System Management

There are several AFS features aimed at facilitating system management. Some of these features have already been mentioned, such as volumes, the *BOS Server*, and the pervasive use of secure RPCs throughout the system to perform administrative operations from any AFS client machinein the worldwide community. This section covers additional AFS features and tools that assist in making the system easier to manage.

4.2.10.1 Intelligent Access Programs

A set of intelligent user-level applications were written so that the AFS system agents could be more easily queried and controlled. These programs accept user input, then translate the caller's instructions into the proper RPCs to the responsible AFS system agents, in the proper order.

An example of this class of AFS application programs is `vos`, which mediates access to the *Volume Server* and the *Volume Location Server* agents. Consider the `vos move` operation, which results in a given volume being moved from one site to another. The *Volume Server* does not support a complex operation like a volume move directly. In fact, this move operation involves the *Volume Servers* at the current and new machines, as well as the *Volume Location Server*, which tracks volume locations. Volume moves are accomplished by a combination of full and incremental volume dump and restore operations, and a VLDB update. The `vos move` command issues the necessary RPCs in the proper order, and attempts to recovers from errors at each of the steps.

The end result is that the AFS interface presented to system administrators is much simpler and more powerful than that offered by the raw RPC interfaces themselves. The learning curve for administrative personnel is thus flattened. Also, automatic execution of complex system operations are more likely to be successful, free from human error.

4.2.10.2 Monitoring Interfaces

The various AFS agent RPC interfaces provide calls which allow for the collection of system status and performance data. This data may be displayed by such programs as `scout`, which graphically depicts *File Server* performance numbers and disk utilizations. Such monitoring capabilites allow for quick detection of system problems. They also support detailed performance analyses, which may indicate the need to reconfigure system resources.

4.2.10.3 Backup System

A special backup system has been designed and implemented for AFS, as described in [6]. It is not sufficient to simply dump the contents of all *File Server* partitions onto tape, since volumes are mobile, and need to be tracked individually. The AFS backup system allows hierarchical **dump schedules** to be built based on volume names. It generates the appropriate RPCs to create the required backup volumes and to dump these snapshots to tape. A database is used to track the backup status of system volumes, along with the set of tapes on which backups reside.

4.2.11 Interoperability

Since the client portion of the AFS software is implemented as a standard VFS/vnode file system object, AFS can be installed into client kernels and utilized without interference with other VFS-style file systems, such as vanilla UNIX and the NFS distributed file system.

Certain machines either cannot or choose not to run the AFS client software natively. If these machines run NFS, it is still possible to access AFS files through a *protocol translator*. The **NFS-AFS Translator** may be run on any machine at the given site that runs both NFS and the AFS *Cache Manager*. All of the NFS machines that wish to access the AFS shared store proceed to NFS-mount the translator's */afs* directory. File references generated at the NFS-based machines are received at the translator machine, which is acting in its capacity as an NFS server. The file data is actually obtained when the translator machine issues the corresponding AFS references in its role as an AFS client.

4.3 Meeting AFS Goals

The AFS WADFS design, as described in this chapter, serves to meet the system goals stated in Chapter 3. This section revisits each of these AFS goals, and identifies the specific architectural constructs that bear on them.

4.3.1 Scale

To date, AFS has been deployed to over 140 sites world-wide, with approximately 60 of these cells visible on the public Internet. AFS sites are currently operating in several European countries, in Japan, and in Australia. While many sites are modest in size, certain cells contain more than 30,000 accounts. AFS sites have realized client/server ratios in excess of the targeted 200:1.

4.3.2 Name Space

A single uniform name space has been constructed across all cells in the greater AFS user community. Any pathname beginning with */afs* may indeed be used at any AFS client. A set of common conventions regarding the organization of the top-level */afs* directory and several directories below it have been established. These conventions also assist in the location of certain per-cell resources, such as AFS configuration files.

Both access transparency and location transparency are supported by AFS, as evidenced by the common access mechanisms and by the ability to transparently relocate volumes.

4.3.3 Performance

AFS employs caching extensively at all levels to reduce the cost of "remote" references. Measured data cache hit ratios are very high, often over 95%. This indicates that the file images kept on local disk are very effective in satisfying the set of remote file references generated by clients. The introduction of file system callbacks has also been demonstrated to be very effective in the efficient implementation of cache synchronization. Replicating files and system databases across multiple server machines distributes load among the given servers. The *Rx* RPC subsystem has operated successfully at network speeds ranging from 19.2 kilobytes/second to experimental gigabit/second FDDI networks.

Even at the intra-site level, AFS has been shown to deliver good performance, especially in high-load situations. One often-quoted study [1] compared the performance of an older version of AFS with that of NFS on a large file system task named the **Andrew Benchmark**. While NFS sometimes outperformed AFS at low load levels, its performance fell off rapidly at higher loads while AFS performance degradation was not significantly affected.

4.3.4 Security

The use of Kerberos as the AFS authentication system fits the security goal nicely. Access to AFS files from untrusted client machines is predicated on the caller's possession of the appropriate Kerberos ticket(s). Setting up per-site, Kerveros-based authentication services compartmentalizes any security breach to the cell which was compromised. Since the *Cache Manager* will store multiple tickets for its users, they may take on different identities depending on the set of file servers being accessed.

4.3.5 Access Control

AFS extends the standard UNIX authorization mechanism with per-directory Access Control Lists. These ACLs allow specific AFS principals and groups of these principals to be granted a wide variety of rights on the associated files. Users may create and manipulate AFS group entities without administrative assistance, and place these tailored groups on ACLs.

4.3.6 Reliability

A subset of file server crashes are masked by the use of read-only replication on volumes containing slowly-changing files. Availability of important, frequently-used programs such as editors and compilers may thus been greatly improved. Since the level of replication may be chosen per volume, and easily changed, each site may decide the proper replication levels for certain programs and/or data.

Similarly, replicated system databases help to maintain service in the face of server crashes and network partitions.

4.3.7 Administrability

Such features as pervasive, secure RPC interfaces to all AFS system components, volumes, overseer processes for monitoring and management of file system agents, intelligent user-level access tools, interface routines providing performance and statistics information, and an automated backup service tailored to a volume-based environment all contribute to the administrability of the AFS system.

4.3.8 Interoperability/Coexistence

Due to its VFS-style implementation, the AFS client code may be easily installed in the machine's kernel, and may service file requests without interfering in the operation of any other installed file system. Machines either not capable of running AFS natively or choosing not to do so may still access AFS files via NFS with the help of a protocol translator agent.

4.3.9 Heterogeneity/Portability

As most modern kernels use a VFS-style interface to support their native file systems, AFS may usually be ported to a new hardware and/or software environment in a relatively straightforward fashion. Such ease of porting allows AFS to run on a wide variety of platforms.

Chapter 5

Future AFS Design Refinements

5.1 Overview

The current AFS WADFS design and implementation provides a high-performance, scalable, secure, and flexible computing environment. However, there is room for improvement on a variety of fronts. This chapter considers a set of topics, examining the shortcomings of the current AFS system and considering how additional functionality may be fruitfully constructed.

Many of these areas are already being addressed in the next-generation AFS system which is being built as part of Open Software Foundation's (OSF) Distributed Computing Environment [7] [8].

5.2 UNIX Semantics

Any distributed file system which extends the UNIX file system model to include remote file accesses presents its application programs with failure modes which do not exist in a single-machine UNIX implementation. This semantic difference is difficult to mask.

The current AFS design varies from pure UNIX semantics in other ways. In a single-machine UNIX environment, modifications made to an open file are immediately visible to other processes with open file descriptors to the same file. AFS does not reproduce this behavior when programs on different machines access the same file. Changes made to one cached copy of the file are not made immediately visible to other cached copies. The changes are only made visible to other access sites when a modified version of a file is stored back to the server providing its primary disk storage. Thus, one client's changes may be entirely overwritten by another client's modifications. The situation is further complicated by the possibility that dirty file chunks may be flushed out to the *File Server* before the file is closed.

The version of AFS created for the OSF offering extends the current, untyped callback notion to a set of multiple, independent synchronization guarantees. These synchronization *tokens* allow functionality not offered by AFS-3, including byte-range mandatory locking, exclusive file opens, and read and write privileges over portions of a file.

5.3 Improved Name Space Management

Discovery of new AFS cells and their integration into each existing cell's name space is a completely manual operation in the current system. As the rate of new cell creations increases, the load imposed on system administrators also increases. Also, representing each cell's file space entry as a mount point object in the */afs* directory leads to a potential problem. As the number of entries in the */afs* directory increase, search time through the directory also grows.

One improvement to this situation is to implement the top-level */afs* directory through a Domain-style database. The database would map cell names to the set of server machines providing authentication and volume location services for that cell. The *Cache Manager* would query the cell database in the course of pathname resolution, and cache its lookup results.

In this database-style environment, adding a new cell entry under */afs* is accomplished by creating the appropriate database entry. The new cell information is then immediately accessible to all AFS clients.

5.4 Read/Write Replication

The AFS-3 servers and databases are currently equipped to handle read/only replication exclusively. However, other distributed file systems have demonstrated the feasibility of providing full read/write replication of data in environments very similar to AFS [11]. Such systems can serve as models for the set of required changes.

5.5 Disconnected Operation

Several facilities are provided by AFS so that server failures and network partitions may be completely or partially masked. However, AFS does *not* provide for completely **disconnected operation** of file system clients. Disconnected operation is a mode in which a client continues to access critical data during accidental or intentional inability to access the shared file repository. After some period of autonomous operation on the set of cached files, the client reconnects with the repository and resynchronizes the contents of its cache with the shared store.

Studies of related systems provide evidence that such disconnected operation is feasible [11] [12]. Such a capability may be explored for AFS.

5.6 Multiprocessor Support

The LWP lightweight thread package used by all AFS system processes assumes that individual threads may execute non-preemeptively, and that all other threads are quiescent until control is explicitly relinquished from within the currently active thread. These assumptions conspire to prevent AFS from operating correctly on a multiprocessor platform.

A solution to this restriction is to restructure the AFS code organization so that the proper locking is performed. Thus, critical sections which were previously only implicitly defined are explicitly specified.

Bibliography

[1] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, Michael J. West, *Scale and Performance in a Distributed File System*, ACM Transactions on Computer Systems, Vol. 6, No. 1, February 1988, pp. 51-81.

[2] Michael L. Kazar, *Synchronization and Caching Issues in the Andrew File System*, USENIX Proceedings, Dallas, TX, Winter 1988.

[3] Alfred Z. Spector, Michael L. Kazar, *Uniting File Systems*, Unix Review, March 1989,

[4] Johna Till Johnson, *Distributed File System Brings LAN Technology to WANs*, Data Communications, November 1990, pp. 66-67.

[5] Michael Padovano, PADCOM Associates, *AFS widens your horizons in distributed computing*, Systems Integration, March 1991.

[6] Steve Lammert, *The AFS 3.0 Backup System*, LISA IV Conference Proceedings, Colorado Springs, Colorado, October 1990.

[7] Michael L. Kazar, Bruce W. Leverett, Owen T. Anderson, Vasilis Apostolides, Beth A. Bottos, Sailesh Chutani, Craig F. Everhart, W. Anthony Mason, Shu-Tsui Tu, Edward R. Zayas, *DEcorum File System Architectural Overview*, USENIX Conference Proceedings, Anaheim, Texas, Summer 1990.

[8] *AFS Drives DCE Selection*, Digital Desktop, Vol. 1, No. 6, September 1990.

[9] Levine, P.H., *The Apollo DOMAIN Distributed File System*, in *NATO ASI Series: Theory and Practice of Distributed Operating Systems*, Y. Paker, J-P. Banatre, M. Bozyigit, editors, Springer-Verlag, 1987.

[10] M.N. Nelson, B.B. Welch, J.K. Ousterhout, *Caching in the Sprite Network File System*, ACM Transactions on Computer Systems, Vol. 6, No. 1, February 1988.

[11] James J. Kistler, M. Satyanarayanan, *Disconnected Operaton in the Coda File System*, CMU School of Computer Science technical report, CMU-CS-91-166, 26 July 1991.

[12] Puneet Kumar, M. Satyanarayanan, *Log-Based Directory Resolution in the Coda File System*, CMU School of Computer Science internal document, 2 July 1991.

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

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

[15] C.H. Sauer, D.W Johnson, L.K. Loucks, A.A. Shaheen-Gouda, and T.A. Smith, *RT PC Distributed Services Overview*, Operating Systems Review, Vol. 21, No. 3, July 1987.

[16] A.P. Rifkin, M.P. Forbes, R.L. Hamilton, M. Sabrio, S. Shah, and K. Yueh, *RFS Architectural Overview*, Usenix Conference Proceedings, Atlanta, Summer 1986.

[17] Edward R. Zayas, *Administrative Cells: Proposal for Cooperative Andrew File Systems*, Information Technology Center internal document, Carnegie Mellon University, 25 June 1987.

[18] Ed. Zayas, Craig Everhart, *Design and Specification of the Cellular Andrew Environment*, Information Technology Center, Carnegie Mellon University, CMU-ITC-070, 2 August 1988.

[19] Kazar, Michael L., Information Technology Center, Carnegie Mellon University. *Ubik - A Library For Managing Ubiquitous Data*, ITCID, Pittsburgh, PA, Month, 1988.

[20] Kazar, Michael L., Information Technology Center, Carnegie Mellon University. *Quorum Completion*, ITCID, Pittsburgh, PA, Month, 1988.

[21] S. R. Kleinman. *Vnodes: An Architecture for Multiple file System Types in Sun UNIX*, Conference Proceedings, 1986 Summer Usenix Technical Conference, pp. 238-247, El Toro, CA, 1986.

[22] 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.

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

Index