CMU-ITC-88-070

Design and Specification of the Cellular Andrew Environment

Edward R. Zayas
Craig F. Everhart
Information Technology Center
Carnegie Mellon University

DRAFT
2 August 1988

[/afs/andrew.cmu.edu/usr8/erz/cells/doc/spec/specification.ez]

CMU-ITC-070

Table of Contents

1. Introduction

The Andrew computing environment developed by the Information Technology Center (ITC) at Carnegie Mellon University aims at supporting a very large user base. On the CMU campus, it is envisioned that each of the roughly 7,000 students, staff, and faculty members will eventually have their own workstation, and the Andrew design reflects this projection. The distributed file system employed allows each client machine to share the same view of the single file tree, with individual workstations merely caching copies of the files in the (conceptually) centralized collection.

Several Andrew sites have begun operation, some of them within Carnegie Mellon itself. CMU's Computer Science Department and the Psychology Department service their members independent of the main campus computing organization. There is also a separate internal Andrew site devoted to testing newly-released software before it is introduced into the mainstream campus system. Externally, sites currently exist at IBM Rochester, IBM Palo Alto, and the NIH. At the time of this writing, MIT has just recently brought two cells online in support of the Athena Project. Future sites include the University of Michigan and possibly Hewlett Packard. Each Andrew installation is completely autonomous, independently managing such administrative functions as account creation, maintenance of authentication databases, backup services, server machine allocation, and configuration. As per design, each site maintains its own separate file system.

The ***Cellular Andrew*** effort was launched in April 1987 in order to allow such administratively autonomous sites, or ***cells***, to cooperatively establish a "community filespace" composed of the union of the various individual trees. While the single name space allows transparent access to any file in the cellular community from any Andrew workstation, it does not require a site to relinquish (or even share) administrative control over its own system. In particular, access to a site's files is still mediated by its own authentication system. Also, the design of the Cellular Andrew extensions had the explicit goal of minimizing the performance costs of providing such a unified community. While inter-cell file access would certainly not be uncommon, the vast majority of references were still expected to lie within a single site. Noticeable degradation of local file service due to overheads in maintaining the idiom of a common file name space was seen as greatly outweighing the advantages of such an arrangement.

The first two Andrew cells to come on line were the main CMU campus computing organization (including the ITC) and the Psychology Department. The maiden transparent cross-cell file access took place between these two sites in early August of 1987, with the cellular authentication mechanisms becoming fully functional a couple of weeks later. Utilities such as *ls* and *passwd* were converted over time, and additional functionality was put into the *fs* program so that users could take full advantage of the new cellular system's primitives. As more sites came on line and more Andrew software was upgraded to exploit the features of the cellular environment, the natural divergence took place. The ***Andrew Message System (AMS)*** [3, 5], providing a multi-media mail and bulletin board service, drove home the need

to define community standards and conventions clearly, without which inter-cell interoperation is severely hampered. This document is meant to provide a comprehensive specification of these Cellular Andrew conventions at all levels.

This paper continues by describing the design of the cellular extensions to the Andrew environment in Section 2. The reader is not assumed to have detailed knowledge of the stock Andrew mechanisms. Those wishing more explicative detail and/or specific performance figures for the ***Andrew File System (AFS)*** itself are referred to [1, 2, 4]. Section 3 begins the definition of proper cellular operation by presenting the set of required file space conventions. Some of these conventions apply to optional facilities that are *not* required to be supported at Andrew sites, namely the White Pages database and the AMS. They are nonetheless included in this document, since these same conventions are also used to inform the cellular community that certain services are *not* being provided at a particular site. Following in Section 4 are the AFS semantics that must be adhered to at the application level. The AMS requirements are presented in Section 5. Again, it is important for cell administrators to understand these requirements even if they do not plan to support AMS at their site. Administrators must still insure that the conventions are not inadvertently breached. All of the details of cellular configuration are given in Section 6. Sections 7 and 8 present the server and programming interfaces for cellular AFS respectively, and Section 9 explains the formal methods by which cell administrators may propose changes, keep their sites synchronized and just generally communicate. The plans for future AFS development by the ITC File System Group as it impacts the cellular architecture are revealed in Section 10, and the presentation is wrapped up in Section 11. Appendix 1 contains a glossary of the many terms that appear in this paper, while Appendix 2 provides detailed information on the layout, maintenance, and use of the optional White Pages database.

2. Cellular Andrew Extensions

The Andrew architecture is straightforward, and a summary is presented here. A set of *FileServer* machines house the central copies of all AFS files on their disks. Each Andrew workstation runs a special agent, the *CacheManager*, that transparently caches any files accessed by its users on the machine's local disk. This *CacheManager* is issued *callbacks* from the *FileServers* it gets these files from, which are essentially promises that notification will be issued if the central copy of the file is updated. In this way, the *CacheManager* does not have to check with the appropriate *FileServer* each time a user attempts to *open()* a file sitting in the local cache, but only when a callback promise is not held for the given file. There is usually (but not necessarily) a separate machine referred to as the *System Control Machine*, or *SCM*. This server keeps all the *FileServers* synchronized, keeps volume location information up-to-date (see below) and performs other important administrative functions. Authentication is controlled in the system by *AuthServer* processes running on a subset of the *FileServer* machines. These processes provide tokens of identity which are used in conjunction with access control lists at the directory level (not the file level).

The rest of this section analyzes in detail the portions of the Andrew File System that were affected by introduction of cellular capabilities, along with the actual changes themselves.

2.1. Authentication

2.1.1. Standard Mechanisms

Each user of the current Andrew environment identifies himself to the system by engaging in an authentication protocol. This procedure is first carried out via the *login* program at the start of the user's session. Authentication may take place several more times during the same session using the *su* or *log* programs, allowing this identity to change. After the user supplies his or her name and password, *login* (for example) attempts to contact an available *AuthServer* process to verify this information. At startup time, the *CacheManager* uses system configuration files to determine the list of machines hosting *AuthServer* processes. If none of these machines respond in a reasonable amount of time, the *SCM* is used. The *AuthServer* that is eventually reached checks to see if the given password matches the one recorded for that user. If so, it creates and returns ***clear*** and ***secret tokens*** that serve to identify the user in its dealings with the *FileServers*. These tokens contain not only the user's central ***ViceID*** but also expiration information to prevent processes from accessing the file system indefinitely. Note that *ViceIDs* do not necessarily correspond with the local *uid*, although they currently tend to match. *Login* proceeds to pass these tokens to the *CacheManager*, which stores them for when it acts as the user's agent for file manipulation. If all of the *AuthServers* are down, or if the tokens cannot be successfully handed to the *CacheManager*, a *local login* is performed. In this case, the user may still access files on his or her own workstation's local disk *iff* the password matches the one in the local copy of the password file. After a local login, the user is still treated as completely unauthenticated when attempting to access the

pool of AFS files.

An Andrew user can be logged in as several different people at once. This option is due to the fact that each user process belongs to exactly one ***process authentication group (PAG)***. PAGs are guaranteed to be unique for all process families on a machine until a reboot occurs. User tokens are associated with the corresponding PAG, so the *CacheManager* can select the right set when a process asks it to interact with the *FileServers*. Thus, different processes controlled by a user can be tied to different accounts via this PAG mechanism. Note, however, that a single process/PAG can only be authenticated as one user at any time.

2.1.2. Extensions

In the cellular Andrew environment, users have the option of specifying the site in which they wish to be authenticated when invoking *log* and its brethren. Every workstation associates itself with the administrative cell in which it physically resides, as determined by the system configuration information. By default, the local set of *AuthServers* will be contacted as before. If the user explicitly names a remote cell, a service database will be queried to determine the set of machines advertised as hosting *AuthServers* for that site. The *AuthServers* for the chosen cell are therefore contacted directly, and they return tokens which can only be properly decoded by agents in that domain. The data structure in which the *CacheManager* stores tokens is now tagged with the cell in which they were generated. Note that the token format itself has not been changed. When accessing a file in a particular cell in behalf of a client, the *CacheManager* must now select the user's tokens that correspond to that cell. The feature, where tokens obtained from each authentication transaction accumulate in the *CacheManager*, is known as ***additive authentication***.

Even with additive authentication, the system restricts users to a single identity per PAG in any one cell. The *CacheManager* enforces this restriction by only saving the last set of tokens received for a given cell within a PAG. This limitation is necessary to prevent situations where the proper identity cannot be selected automatically. To illustrate, let us assume that this restriction did not hold and that the *CacheManager* held `andrew.cmu.edu` tokens for both authors, *erz* and *cfe*, on behalf of an editor process. It is impossible to determine a file's correct owner if the editor tries to save it into a directory writable by both people.

Additive authentication is a useful feature, allowing individual command invocations to access protected files in several domains. Take the example where a protected file needs to be copied from the *erz* account in the `cs.cmu.edu` domain into `andrew.cmu.edu`'s *erz* account. If a process is authenticated as both parties, the standard *cp* command works correctly. Without additive authentication, this simple task would require two separate processes. The first process, authenticated in the `cs.cmu.edu` domain, deposits the file in some intermediate location made accessible to it (say, */tmp*) on the local workstation disk. At this point, the second process, authenticated as *erz* in `andrew.cmu.edu`, moves it to its final home. Alternatively, a pipe may be set up between the two, with the first reading the `cs.cmu.edu` file and

the second writing to the proper file in the `andrew.cmu.edu` cell. These alternatives are simply not palatable. As stated above, the *CacheManager* automatically selects the proper token to present to a *FileServer* in a particular domain. If no token exists for a cell, the *CacheManager* uses an unauthenticated connection when performing a file operation, effectively reducing the client's rights to those of the *Anonymous* user in that domain. If system security and autonomy are to be maintained, it is imperative that automatic rights reductions be performed when crossing into a protection domain in which the client has not identified himself.

While several identities per PAG are possible at any given instant, the system must be able to select at most one of them as the ***primary identity***. Many programs need to ask "*Who am I?*" in order to operate properly. The prime example of a program exhibiting this need is *Messages*, a multi-media facility that provides a common framework for performing such tasks as reading electronic bulletin boards and electronic mail. When it comes up, *Messages* must be able to determine such things as its caller's home directory and the directory holding that user's mail. If *Messages* is running in a PAG where the user is authenticated in several cells, there is no way to unambiguously discover these things in an automatic way. The *getpwuid()* family of routines are no help, since they only operate on local password files.

Primary identities are established during authentication by both convention and the explicit use of switches in the command lines of the *log* family of programs. The *CacheManager* marks the given identity as primary when it stores its tokens. At most one primary identity can be selected at any given time per PAG. A program can determine its primary identity through new routines that supplement the standard *getuid()/getpw*()* family. For example, *getvuid()* contacts the *CacheManager* and returns the ViceID corresponding to the primary identity. Also, *getvpwuid()* can be used to find the password file entry for that user's primary identity. These routines are described in more detail in Section 8.2.

This new naming and authentication scheme has several positive characteristics, and is more attractive than any attempt to maintain a single, global authentication database:

1. Most existing AFS server data structures, operations, and tools are completely unchanged by the cellular upgrade. These include the authentication database format, distribution of authentication information, and all operations for adding and deleting user accounts.

2. A cell's authentication database is completely independent of both the number of cells in the community and the corresponding sizes of their protection files. A global authentication database will certainly result in files with unworkable sizes, and will grow with the number of cells represented. Password files in the `andrew.cmu.edu` cell are already large enough to require a separate index in order to provide adequate search times.

3. Maintaining reasonably consistent copies of a global authentication

database requires a large amount of communication among the cells. The only information that must be kept reasonably current in the chosen strategy is the list of *AuthServers* for each protection domain. These lists are not expected to change very often. In addition, note that a single, global authentication database approach requires that sensitive information be transmitted across the network.

4. An error in authentication database updates by one cell could corrupt other cells if a global approach were used. To avoid this, each domain must expend energy on carrying out consistency checks on any new information received before merging it in. No such corruption is possible in this implementation, as updates never cross protection boundaries.

5. User names do not need to be unique in the global community, only within a cell. In fact, the result of this design is a global, unique naming system in which each cell has complete and exclusive control of its own user name space, totally free of any collisions with the name spaces maintained by other cells.

6. Since the remote agents are contacted directly, remote authentication activity does not raise the load on local servers.

7. Special privileges within a cell do not automatically carry over to other cells, maintaining security and exclusivity. For example, a user with System:Administrator rights in cell `A` cannot delete users in cell `B` unless he is also authenticated there as a person with those same rights. Automatic rights reduction insures that remote users who do not have accounts in another cell or have chosen not to identify themselves as that person are treated as the Anonymous user. However, an authenticated administrator for cell `B` can still carry out his full range of activities there even if they log in on a workstation that belongs to cell `A`.

2.2. Volume Location

2.2.1. Standard Mechanisms

The Andrew ***volume*** concept is a central one. Volumes are containers for a hierarchy of files, and are the basic units of data moved between *FileServers*. The current Andrew file system is composed of a collection of system and user volumes, joined together at ***mount points***. Note that Andrew mount points are AFS objects that have no real connection with mount points as defined by the Unix file system. Instead, they indicate where a particular volume is to appear in the file system tree. In this way, AFS presents the image of a single, seamless tree to its clients. *FileServers* may be instructed to *clone* read-only volumes from the read-write versions and replicate them amongst themselves for greater availability and reduced per-server demand. Mechanisms exist by which administrators may create, delete, back up, replicate, and move volumes. Complete information on the status of all Andrew volumes is kept in the ***Volume Location Data Base (VLDB)*** on the *SCM* and replicated at a subset of the *FileServer* machines. As volume operations take place, individual *FileServers* keep track of the changes to the volumes they host. Periodically, the *SCM* polls each *FileServer* machine and collects these changes, merging them back into a new VLDB and redistributing it. Volumes are identified by either name or number, both guaranteed to be unique in the Andrew environment.

At startup time, the *CacheManager* determines the volume serving as the root of the AFS tree. This is done by either calling the *RViceGetRootVolume()* file system interface routine or through explicit (and overriding) information in the bootup configuration files. It also determines the group of *FileServer* machines that provides volume location service. As the *CacheManager* encounters mount points in the course of processing path names presented by its users, it consults any *FileServer* in this group to determine the location of a volume for which that information has not been cached. Once the sites a volume resides on are determined, the *CacheManager* can access that volume's files to the full extent of the permissions held by its clients. To make certain the volume location information held by the *CacheManager* doesn't get overly stale, one of its lightweight daemon processes checks the name-to-number mappings every two hours.

2.2.2. Extensions

In the expanded environment, volumes from anywhere in the greater Andrew community can be mounted in the AFS tree via ***cellular mount points***. This new construct differs from the standard AFS mount point in that it also contains the name of the cell in which the associated volume resides. At startup time, the *CacheManager* now learns of the set of *FileServer* machines providing volume location service for each known cell, including its own. Internally, the *CacheManager*'s volume information is keyed on both the cell and volume numbers. This is made necessary by the fact that the 32-bit volume numbers are no longer guaranteed to be unique across cells. Path name processing is now only slightly more complex than before. When crossing a cell mount point for which no volume location

information has been cached, the *CacheManager* consults the subgroup of *FileServers* responsible for the named cell. When crossing a standard mount point, volume location requests are directed to the machines for the cell in which the parent volume resides, since the new volume also resides there.

This scheme has the same advantages as the new approach to accessing authentication services. These include the basic preservation of the existing mechanisms, independence of VLDBs, preservation of workable file sizes, avoidance of inter-cell communication needed to support a global VLDB, compartmentalization of errors, direct application of remote volume location workload to remote servers, and automatic rights reductions across protection boundaries. In addition, the proposed system has two favorable characteristics:

1. Local file servers are completely unaffected by the file traffic generated when its workstations access remote files. The only parties involved in inter-cell file transfers are the remote *FileServers* hosting the data and the individual workstations performing the accesses.

2. The cell mount point construct allows remote file systems to be rooted anywhere in the local cell's file system. This allows a cell's administrators to shape their view of the file space to suit their own purposes. By convention, though, the upper level is defined as described in Section 3.

While volumes may still be shuttled back and forth as they have always been between *FileServers* in the same cell, they are not permitted to move between cells. Each administrative domain has full control of its volumes, and is completely responsible for their housing and backup. The only difficulty with this restriction is in handling the natural movement of users between sites. Transferring a user's files between cells, i.e., from `andrew.cmu.edu` to `cs.cmu.edu`, can still be accomplished in spite of this limitation. A dump is taken of the user's `andrew.cmu.edu` volume, and the volume itself is destroyed. The dump file is copied to the `cs.cmu.edu` cell, where it is restored to a new volume belonging to that site. This operation must be carried out as a cooperative effort between the maintainers of both cells.

2.3. Mapping ViceIDs to Attributes

2.3.1. Standard Mechanisms

Such programs as *ls* (list directory) perform mappings from internal *uids*/ViceIDs to string names suitable for human consumption. Other programs map ViceIDs to such things as the user's home directory, the desired shell (command interpreter), etc. These tasks are done via the standard *getuid()* and *getpw*()*-class calls. The Andrew version of *ls* is capable of using the White Pages facility to speed these mappings, should it be available. The White Pages database is a superset of the information in */etc/passwd*, supplying such additional information as user nicknames, departmental affiliations, and mail forwarding addresses. The interface to this database allows such

sophisticated operations as fuzzy matching on user names. A full description of this optional Andrew facility may be found in [7]. Andrew programs that have been converted to take full advantage of the cellular environment use the corresponding *getvuid()* and *getvpw*()* calls instead of the standard ones. These routines know to make use of a White Pages database if one exists; otherwise, they fall back on a vanilla search of */etc/passwd.*

2.3.2. Extensions

These mappings will still be performed correctly by the above mechanism in the cellular system, but only when dealing with files and directories living in the local cell. The problem lies in the fact that this approach *always* maps *uids*/ViceIDs in relation to the local White Pages (or */etc/passwd)* regardless of the cell they really refer to. As an example of the confusion produced by this shortcoming of the standard mapping mechanism, consider the following *ls* invocation when run from a workstation in the `andrew.cmu.edu` cell:

```
% ls -ldF /afs/cs.cmu.edu/user/erz
d            2 lynn  2048 May 25 15:21 /afs/cs.cmu.edu/user/erz/
```

The ViceID corresponding to the *erz* account in the `cs.cmu.edu` cell is 67. Since the printable name string is determined from the `andrew.cmu.edu` White Pages, *ls* attributes ownership of that directory to `andrew.cmu.edu`'s ViceID 67, namely Lynn Brown.

Several solutions for this problem were considered and rejected. One proposal was to change the standard *stat()* block to carry cell information, but this would require a change to all programs depending on the *stat()* interface. Similarly unworkable was an approach which partitioned the *uid*/ViceID space so that a cell identifier appeared in the upper bits. Another rejected suggestion was to forget about performing translations on ViceIDs associated with files in remote cells at all. Rather, (in the case of *ls*) the raw number, qualified by the cell name, would be printed out. While this was the easiest to implement, it was also the least informative, so much so that it was determined to be useless.

The solution finally adopted was to establish a convention requiring that each cell locate its White Pages database (and/or its */etc/passwd* file) in a standard place within its directory structure. The *getcpwuid()* and *getcpwnam()* routines were added to the White Pages interface, allowing clients to specify the name of the cell within which ViceID mappings are to take place. Using the above convention (described in more detail in Section 5.1), that cell's translation database(s) can be easily located and used by *getcpwuid()*. Using these tools, the upgraded *ls* will always provide complete and correct owner information. Its response to the above example is shown below:

```
% ls -ldF /afs/cs.cmu.edu/user/erz
d   2 erz@cs.cmu.edu  2048 May 25 15:21 /afs/cs.cmu.edu/user/erz/
```

The printable names are sometimes qualified by the cell in which the mapping was made, as in the above example. This extra information is provided in the case where the file lives outside the cell hosting the client's primary identity (see Section 2.1.2). If no primary identity has been declared, such qualification occurs only when providing information about files outside the cell administering the workstation on which the client is running. Thus, output from the upgraded *ls* will be identical to that of the standard one for local (same-cell) files, but ownership will be fully qualified for remote files.

3. Filespace Conventions

3.1. Directory Structure

3.1.1. The /afs Root

The global Andrew file tree is rooted at */afs*. Listed below are the current contents of this top-level directory, as seen from a workstation in the `andrew.cmu.edu` cell:

```
% ls -F /afs
andrew/                  cs.cmu.edu/
andrew.cmu.edu/          psy/
athena.mit/              psy.cmu.edu/
athena.mit.edu/          tmpcmu/
beta/                    wm.mit/
beta.andrew.cmu.edu/     wm.mit.edu/
cs/
```

This illustrates the two basic classes of directory names that can appear under */afs*:

> 1. The directories with fully-specified, domain-style names represent entry points to the individual file systems of the sites participating in the Greater Andrew community. Specifically, they are mount points to each cell's root volume.
>
> 2. For each entry bearing a full cell name, there is a corresponding "short-form" name. These are simply locally-defined shortcuts for people who wish to minimize key strokes when specifying full path names. The short-form names are implemented as symbolic links to the appropriate full names in the same directory.

Full names as exhibited in class 1 are unique across all cells, and can be freely and safely used on any occasion. Shortcut names belonging to class 2 are conveniences that are not guaranteed to be unique (or even exist in another cell). Further discussion on these points and the dangers of using shortcut names is delayed until Section 3.2.1.

3.1.2. Per-Cell Directories

Cell administrators are generally free to structure their individual file systems (e.g., everything below */afs/andrew.cmu.edu*) as they see fit. However, there is one directory name whose use is "reserved" at this level of the file tree: ***service***. If a site provides such a directory, it is promising to export certain internal information in a standardized fashion. Furthermore, all directories and files within a cell's *service* subtree ***must*** reside within that cell.

Within *service* itself, the presence of any or all of the following subdirectories further obligates that cell to provide the corresponding service or services (hence

the name). Conversely, the absence of any or all of the directories listed below indicates that the given cell does *not* provide the associated service or services.

etc: This directory holds copies of the */etc/passwd* and */etc/group* files, which are also stored on each workstation's local disk. Certain programs (e.g. *ls*) expect to find this mapping information for each cell in the community at this location. In the case of */etc/passwd*, this copy is only referenced should the cell in question not support a White Pages service. Thus, if the string name associated with gid 4 has to be determined for cell `XXX`, the answer may be found in */afs/XXX/service/etc/group* (for the curious, gid 4 maps to `tty` in the `andrew.cmu.edu` cell). Administrators are free to zero the field carrying encoded password information in this copy of */etc/passwd*, since the file is never used for authentication purposes. Notice that there is only one copy of */etc/group* in this directory. The `andrew.cmu.edu` cell currently has a different copy of */etc/group* for each supported machine type. The only major difference between them is that the `rt_r3` software requires that users must belong to the `wheel` group before they can become `root`. A consolidated version will thus be placed in */afs/andrew.cmu.edu/service/etc/group* to follow through on the cellular conventions.

mailqs: This directory contains the mail delivery queues as used by the Andrew Message System (AMS). As stated above, sites *not* running AMS should not have a directory named *service/mailqs*, as its existence will be viewed by other cells as a promise to provide AMS service. More information on the structure and proper use of *service/mailqs* appears in the full discussion of AMS requirements in Section 5.1.

configuration: This directory contains AMS configuration information. As with *mailqs* above, further discussion on the contents of this directory is also carried out in Section 5.1.

printing: This directory acts as an interface to the printing services available in the given cell. Only the high-level directories expected from a cell are mentioned here. The reader is referred to the Andrew printing documentation for details of the lower-level directory organization, font representations, and accounting procedures [6]. Briefly, there are six required subdirectories of *service/printing* should a cell wish to export its printing facilities:

- *commands*: This is the collection of shell scripts used to bring up the various printer spoolers.
- *database*: This directory contains an accounting database, divided into a set of files for ease of locking. User entries contain such information as the uid, the maximum printing quota, how much of this quota has been used, and how many printing jobs have been issued in the user's name.
- *device*: All device-specific information is kept in this subtree. In `andrew.cmu.edu`, there are three devices available, and each gets its own subdirectory in *service/device*: *ibm3812*, *ibm3820*, and *postscript*.

For each device `ddd`, there are two required files in *service/device/ddd*: *translation.list*, which holds the font-character translation table, and *fonts.list*, which holds font-specific information.

fonts: The actual font libraries live here, partitioned into two subdirectories: *fdb* and *ibm3820*.

printcaps: Machine-specific printing configuration is archived in this directory.

spool: All print spool queues are kept in this directory. For each printer `ppp`, there is a corresponding *service/printing/spool/ppp* directory.

servers: Although details have not yet been worked out as to how a cell's pool of free machines might be made available to foreign users in a reasonable way, presence of this directory implies that the given cell is willing to make its excess computing power available to others. This directory contains some configuration files for the *Butler*, the manager of the free workstation pool, along with a set of subdirectories it needs for its operation. Although the ability to run programs on external workstations is not currently exportable to other sites, the *servers* directory still appears on this list, as this ability is expected to be implemented in the future. Further documentation is expected to be made available.

systypes: This plain ASCII file lists the legal set of values for the *@sys* special name in the given cell. There is no particular format for the contents, as it is only intended to be read by humans.

wp: This directory is the home for the given cell's White Pages database files, should they decide to support this facility. The database maintained at the `andrew.cmu.edu` cell is very large, with the White Pages b-tree partitioned into 94 separate files.

3.2. Volume Conventions

3.2.1. Naming

Each cell is required to maintain a volume named `root.afs`, containing that cell's view of the global, top-level */afs* directory. There are several reasons why each cell maintains its own cellular root volume instead of having only one volume shared between all sites. First, this scheme avoids dependence on the availability and integrity of any one site in the community. It also avoids the bottleneck created by having to go off-site for *each and every* pathname translation performed by the user workstations. Furthermore, the partition of top-level entries into regular mount points and cell mount points differs from cell to cell. Finally, each cell is free to assign its own particular shortcut names (see Section 3.1.1) for the unique, fully-specified cell entry points defined at this level. These short names represent a tradeoff between typing ease and uniqueness. Although it turns out that uniqueness is still preserved in the current cellular community, consider the following scenario. Suppose Newcell University's Computer Science Department comes on-line as the latest Cellular Andrew site. Their full name, `cs.newcell.edu`, is added to the `root.afs` volume

(and hence the */afs* directory) in each cell. However, a shortcut version of this full name, *cs*, is placed in the `root.afs` maintained at Newcell University for the convenience of its users. Notice that the meaning of pathname */afs/cs* now varies depending on the workstation interpreting it. Since workstations maintained by Newcell University mount the local version of the cellular super-root volume, someone running on one of these machines will be directed to */afs/cs.newcell.edu*, whereas someone sitting at a workstation in the CMU Computer Science Department will have that name translated to */afs/cs.cmu.edu*! For this reason, **it is important that hard-wired pathnames, either in programs or in files, use the fully-qualified versions of cell names.** This will insure that the use of that program or file will be location-transparent.

Along with the above naming requirement for each system's "super-root" volume, each cell must use a standardized name for the root volume of its own individually-maintained file system. Specifically, each cell must name its own root volume `root.cell`. This allows a given cell's file system to be mounted into an image of the global tree by creating a cell mount point consisting of the cell's name, a colon, and the string "`root.cell`".

3.2.2. Replication

For reliability, availability, and load-sharing, cell administrators will typically create read-only replicas of important system and binary volumes, and distribute them across several *FileServer* machines. It is ***strongly suggested*** that each cell create clones of its `root.afs` volume and force its workstations to mount one of these read-only copies at */afs*, *even if it is only running a single FileServer.* The reason has to do with the inheritance of volume characteristics across mount points. When crossing a mount point, the *CacheManager* will locate and use a read-only replica of the target volume if the following three conditions are met:

1. The "parent" volume, i.e., that which houses the mount point being crossed, is a clone.
2. The read/write version of the target volume has not been explicitly selected (via the *-rw* switch in the *fs* utility, which is used to create mount points).
3. A read-only clone of the target volume exists.

By mounting a read-only copy of its own `root.afs` volume for the shared file system, a cell's workstations exploit read-only clones wherever they are available in the global tree. This makes remote access more reliable, since any replication performed by external sites on their own behalf is also taken advantage of. Mounting a read-only version of `root.afs` is also an act of courtesy towards the other cells in the community. As mentioned above, cloning is also used for load sharing in Andrew. Having a cell pile up its requests on the read-write versions of remote volumes when replicas are available is considered anti-social.

It is also desirable that the root volume for each cell's individual file system

(`root.cell`) be cloned, for the reasons listed above.

3.2.3. Directory Structure, Revisited

While accessing cloned volumes both locally and remotely is the best recourse for normal computing activities, there are times when the read-write versions must be manipulated. This action is required to carry out such system administration activities as updating binaries and adding entry points for new cells. The upgraded read-write volumes are cloned once again, and the new read-only copies replace the old ones at the *FileServers* acting as replication sites within the cell. Although the necessary read-write volumes can be explicitly mounted at a temporary location when such operations are needed, it is more convenient to have the complete read-write tree mounted and available at all times.

Thus, by convention, each cell will also explicitly mount the read/write version of the other cells' individual root volumes in */afs*, as maintained by their local `root.afs`. Since these entries are not needed during the vast majority of normal activities, they are "hidden" from the casual observer by prepending a period to the fully-qualified names. To illustrate, let us once again generate a listing of the top-level */afs* directory, but this time we'll include all the "hidden" subdirectories representing these read-write mount points:

```
% ls -aF /afs
./                          .wm.mit.edu/            cs/
../                         andrew/                 cs.cmu.edu/
.andrew.cmu.edu/            andrew.cmu.edu/         psy/
.athena.mit.edu/            athena.mit/             psy.cmu.edu/
.beta.andrew.cmu.edu/       athena.mit.edu/         wm.mit/
.cs.cmu.edu/                beta/                   wm.mit.edu/
.psy.cmu.edu/               beta.andrew.cmu.edu/
```

As an example of the direct manipulation of a cell's read/write root volume via this mechanism, suppose the administrators for the `andrew.cmu.edu` cell wish to create a directory, *usr26*, to hold additional user accounts (which are currently stored in directories *usr0* through *usr25*). The following command will start the ball rolling:

```
% mkdir /afs/.andrew.cmu.edu/usr26
```

To make this new user directory visible along the normal paths, the adminstrators then clone and distribute the `root.cell` volume in the `andrew.cmu.edu` cell. When the clones arrive at the replication sites and the VLDB is updated, workstations begin to use these new clones. At this point, */afs/andrew.cmu.edu/usr26* appears to the general populace.

3.3. Handling Special Names

While processing pathname components, the *CacheManager* handles the string *@sys*

in a special way. Any occurrance of *@sys* is replaced with a string describing the type of CPU and operating system the *CacheManager*'s workstation is running. For example, *@sys* is replaced with `rt_r3` for an IBM PC RT running Release 3 (basically Berkeley 4.3 Unix) in the `andrew.cmu.edu` cell, `sun3_34` for a Sun3 workstation running Sun Unix 3.4 there, and so on.

This feature allows a program to use a given pathname without worrying about which machine type and operating system type it is running on. For example, if a user were to invoke */afs/andrew.cmu.edu/@sys/usr/andy/bin/ez* from the command interpreter, they would actually run the */afs/andrew.cmu.edu/**rt_r3**/usr/andy/bin/ez* editor on the above RT configuration. Similarly, this same command would result in starting up the */afs/andrew.cmu.edu/**sun3_34**/usr/andy/bin/ez* version of the same editor on a Sun3.

Extending this facility to operate "correctly" in the cellular environment has been found to be intractable. The difficulty stems from the fact that each cell is free to introduce new CPU types into the Andrew stable, and to perform its own (possibly incompatible) operating system releases. Hence, it will also produce its own particular *@sys* strings to reflect these new systems. This complicates the transparent operation of the *CacheManager*. When encountering an *@sys* reference within the context of an external cell, the *CacheManager* must discover the "compatible" translation string, assuming one exists at all.

The nearest thing to a viable solution is to establish a standard, generic set of *@sys* strings. Each cell provides a table which maps these global names to the compatible local string. For example, let us assume `rt_r3` has been agreed upon as a standard *@sys* string in the Andrew community. The `andrew.cmu.edu` cell provides the identity mapping for it, while the `cs.cmu.edu` cell may decide to map it to something like `rt_mach`. When handling an *@sys* reference in a pathname component lying in cell `X`, the *CacheManager* uses the local string that `X` designates in its table. The biggest drawback to this scheme is the amount of communication required to keep the mapping information up to date. New standard names will have to be constantly negotiated. Also, each cell administration will have to decide exactly what local system(s) the new configuration is compatible with in order to add reasonable entries in their exported mapping tables. This task rapidly becomes impossible in a large community.

It has thus been determined that the *CacheManager* will map occurrances of *@sys* according to the conventions established in its own cell, regardless of where in the pathname this special name is seen. To continue with the above example, if a given cell does not have such an *rt_r3* directory in the specified location, the pathname parse will simply fail, even if a compatible version is available under a different name, say *rt_mach*.

4. AFS Semantics

The */afs/andrew.cmu.edu/common/usr/andy/doc/vdoc/venus.vdoc* file contains a complete specification document for the programming interface to the Andrew *CacheManager*. Every cell in the Greater Andrew community must adhere to this interface, regardless of whether they decide to write their own *CacheManager* or to simply make changes to the one available from the ITC. This section presents a set of points regarding the AFS semantics which are of interest to Andrew *applications* programmers, and the importance of fully publicizing proposed semantic changes to the system as early as possible. It is strongly recommended that the reader review the above document before examining this section.

4.1. Applications Programming

1. The *CacheManager* provides special codes in `errno` for conditions peculiar to the AFS. It is expected that any such AFS-specific error conditions generate the appropriate code in `errno`. In particular, the loss-of-connection condition produces `ETIMEDOUT`, the loss-of-*CacheManager* condition produces `ENXIO`, restarting the *CacheManager* produces `ENOTTY` for operations on old file descriptors, and the absence of a mount point's destination produces `ENODEV`. It is also assumed that `VOFFLINE` and `VBUSY`, if they occur, reflect temporary failures on a *FileServer*. In particular, the `errno` code `ENOENT` must be returned only as an authoritative observation that a file or directory is not present, and never simply that the *CacheManager* was unable to determine whether or not a file or directory existed.

2. It is assumed that `EFBIG` is the `errno` value returned if a program attempts to create a file in a directory that is at its maximum size limit.

3. If and when the AFS begins use of the domain system to determine the legitimate subdirectories under */afs* (instead of relying on a relatively static `root.afs` volume), temporary failures from the domain system itself are reflected via `errno` values other than `ENOENT`. A new and distinguished value for `errno` must be chosen when scanning */afs* in these cases, examples of which are `SERVFAIL`s, non-authoritative `NXDOMAIN`s, and no-such-record responses.

4. Even after migrating the handling of */afs* subdirectories to the domain system, the cell name returned by the `VIOC_FILE_CELL_NAME` *pioctl()* and the `VIOCIGETCELL` *ioctl()* will be the fully-qualified domain name and *not* an abbreviated one. We expect that these two operations will return `EINVAL` when presented with files that do not reside on any server in any AFS cell. These calls are intended as a reliable mechanism by which applications can determine whether or not a given file lives in the AFS. This replaces the previous approach, wherein the `st_gid` field in the *stat()* block was set to 32,767 iff the file resided in the AFS.

6. The *fsync()* system call, when directed at an AFS file, causes that file to be stored on its *FileServer*, and furthermore results in that file remaining open with all *flock()*s at the time of the *fsync()* still in force.

7. [***Specifically for the AMS***] When a file is created, its owner (the value in the `st_uid` field after a *stat()*) is the ViceID held in the *FileServer* connection that was responsible for the creation. It is expected that whenever a rename is possible on a file such that the `st_uid` field is *not* updated, the person responsible for the destination directory will be able to trust the person responsible for the source directory. An example of this situation arises when renames are done within the same volume. The AMS relies on this convention for files inserted into *~/Mailbox* directories. Thus, when a file tagged with a certain ViceID is inserted into a mailbox directory, it is expected that it was inserted by a process authenticated as that same ViceID. When a file is renamed into that directory without updating its `st_uid` (as in the above example), we expect that the rename was performed by a trusted entity. The `andrew.cmu.edu` cell's administration carries out this part of the expectation by assigning each user an individual AFS volume. Thus, the only files that can be renamed into the user's *~/Mailbox* directory without changing the `st_uid` field are those that were stored in the user's volume to begin with. Presumably, the user will be able to trust such files.

4.2. Handling Future Changes

Suggested changes to the AFS semantics must be aired publically at the earliest reasonable time to allow both system and applications programmers from all cells in the community to assess their consequences. Sometimes, seemingly insignificant changes have profound and devastating effects on certain Andrew programs. For example, consider the following proposal made by the CMU Computer Science Department. They suggested that the AFS support the "File system full -- pausing" feature, wherein a program that attempts to allocate storage beyond what is available is suspended until the allocation request can be satisfied. If this behavior had been universally applied without recourse, it would have disrupted sites using the AMS facility to provide mail delivery service. PostOffice server machines would be susceptible to hanging for extended periods, since they don't normally have users logged into them to see the problem. This would effectively halt mail and bulletin board processing on that server machine. Specifically, an attempt to deliver mail to a user that is over quota results in a denial of service to the rest of the community until the situation is corrected. In this case, the issue was resolved allowing pausing to be disabled on ENOSPC errors. Thus, machines that cannot tolerate such behavior (such as those serving as PostOffices) merely change it to suit their needs.

5. The Andrew Message System

The Andrew Message System (AMS) is a highly desirable facility, but not a prerequisite for membership in the cellular community. This section provides a set of cellular conventions to be used by those sites that wish to run their own version of this advanced mail delivery system. Even sites using other delivery systems must be aware of these conventions to make certain they are not inadvertently violated. Such violations may give the appearance to the rest of the Andrew world that their cell is providing a particular service or services when in fact it is not.

The AMS is not only integrated into the cellular environment, but also takes advantage of the unique opportunities it offers. It is possible for sites to participate in AMS delivery both at a full-compliance level and at intermediate levels of participation. This section also explores the details of such varying degrees of conformance (including the possible non-participation of some sites) and the tradeoffs involved.

As stated earlier in this document, each cell has a fully-qualified Internet-style domain name. This full domain name is referred to as the "cell name" or "`CELLNAME`". These two terms will be used interchangeably throughout this discussion. As a reminder, the (CMU campus) cells currently in existence are `andrew.cmu.edu`, `cs.cmu.edu`, `psy.cmu.edu`, and `beta.andrew.cmu.edu`.

5.1. Requirements for Full Cooperation

This section defines full AMS cooperation, meaning AMS delivery system installation. Intermediate levels of support are discussed in Section 5.3.

Under full compliance, each cell maintains and exports a White Pages database describing its users, and this b-tree database is expected to reside in */afs/CELLNAME/service/wp*, with a root file named */afs/CELLNAME/service/wp/wp*. A cell's White Pages contains an entry for each account entity known to its administrators. These entries are derived from a file in */etc/passwd*-format, whose fields are interpreted as follows:

`pw_name`: The login name for the AFS user (e.g., `cfe`).
`pw_passwd`: May be any one of the following: 1) The encoded local-disk password, 2) A "*" prohibiting logins, 3) A "" (null string) allowing unauthenticated, password-free logins or 4) A "V" or "X" signalling that AFS authentication should be trusted for purposes of workstation login.
`pw_uid`: The ViceID for that AFS user.
`pw_gid`: The group ID for that AFS user.
`pw_gecos`: The full name for that AFS user (e.g. "Craig Everhart").
`pw_dir`: The home directory for that AFS user.
`pw_shell`: The initial shell program to be used when that user logs in.

Thus, the authoritative translations between ViceID values and the login names

corresponding to those values are kept here. The cell's White Pages facility is used by such programs as *ls* to determine the print-names for the owners of files in their own or external cells.

By agreement, the cell name is the prevailing mail domain name. For example, the AMS constructs "From:" headers in which the "@domain" text is "@CELLNAME". In particular, the cellular-capable AMS treats the "@domain" part of any mail address as a possible reference to a cell name. Thus, since the CMU Computer Science Department's cell is named `cs.cmu.edu`, the AMS looks to that cell to resolve and deliver mail addressed to the "cs.cmu.edu" domain.

Also by convention, standard network services such as Finger, SMTP mail delivery, Telnet, and FTP are provided through this cell name. For any cell name, there must be a machine with an identical domain name. Finger, Telnet, and FTP services must therefore be provided via processes on this machine. It is possible to redirect a sizable portion of the SMTP mail traffic away from this specific machine by publishing the proper set of MX records. However, there are many mail-sending machines in the internet that do not use MX mail delivery records. Thus, we are effectively forced to provide SMTP service at this location anyway.

The following algorithm is used to generate the "local-part" for mail addresses by the AMS authenticated in a given cell:

1. Take the ViceID returned by the *CacheManager* for the authentication tokens in that cell.
2. Use that cell's White Pages to translate that ViceID to a `pw_name` value.
3. Append a plus sign ("+") to the `pw_name` value.

(This behavior may be modified in some higher-level cases; see the discussion in the third paragraph following.)

The "From:" address in mail is derived from the message sender's primary AFS authentication. Other authentication traces, however (such as Return-path information, or the authentication information preserved by the AMS and presented in message captions), will be generated from any authentication, primary or not, possessed by the message sender.

The AMS places in its "From:" fields the `pw_gecos` field of the authenticated user, as an RFC822 "phrase" before an RFC822 "route". Thus, it will quote the `pw_gecos` field contents if necessary to conform to RFC822.

Should a cell administration be able to guarantee the uniqueness of its `pw_gecos` field contents, the algorithm for generating "From:" fields and for the validated form of "To:" and "CC:" recipients can be modified as follows. A file named */afs/CELLNAME/service/configuration/name-separator* is created by the administrators (and recall that this file must be stored within the CELLNAME cell). If this file exists and its first character is not alphanumeric, canonical mail addresses in CELLNAME are

constructed by replacing all spaces in the pw_gecos field with that first character (optionally quoting them to serve as a legal RFC822 "local-part") and then appending "@CELLNAME". Thus, a cell that desires its validated mail addresses to usually be of the form "Firstname.Lastname@CELLNAME" should create a name-separator file whose first character is a period. Similarly, the name-separator file should begin with an underscore or a space if the addresses are to be of the form "Firstname_Lastname@CELLNAME" or "Firstname Lastname@CELLNAME" respectively. Other values for this character are not recommended for a cell that uses the AMS delivery system. When an address is rewritten using a name-separator character, no RFC822 "phrase"/"route" pair is generated. Instead, the pw_gecos field is simply used as the RFC822 "local-part" for the address, and it is left as "local-part@CELLNAME". Choosing a space or a period for the name-separator character may require AMS programs to quote the resultant "local-part". Furthermore, using a space implies that any pw_gecos field with a space will be quoted. Finally, using a period implies that any pw_gecos field with a period next to a space must be quoted. For example,

Craig Everhart@CELLNAME
Craig.F..Everhart@CELLNAME

are not legal addresses; they must be quoted as

"Craig Everhart"@CELLNAME
"Craig.F..Everhart"@CELLNAME

Another caveat is that even if a cell alters the form of its validated mail addresses, it must still recognize the "userid+@CELLNAME" form. This is not only because some addresses generated at low levels of the system will always generate this older form, but also because it may not always be possible for an AMS program to read the name-separator file (e.g., if communications are disrupted). In such cases, an AMS program will revert to the "userid+@CELLNAME" form of validated address if it cannot be certain about the existence or contents of the published name-separator file.

Information concerning the authenticated source of every piece of mail is presented to AMS users. This information is ultimately derived from the contents of the st_uid field given by *stat()* for files living in AFS, which is translated via that cell's White Pages to a pw_gecos value. Each cell provides a distinguished account that is trusted by that cell's users to preserve authentication information for messages transferred within that same cell. The name of this account is defined to be the value of PostmasterName in the configuration files, which is typically postman.

A cell's AMS mail queues are those subdirectories in the */afs/CELLNAME/service/mailqs* directory whose names start with the prefix "q". Again, it is stipulated that these subdirectories must be located in CELLNAME itself. Thus, */afs/andrew.cmu.edu/service/mailqs/q003* names a mail queue for the andrew.cmu.edu cell. A cell's background mail queues are those subdirectories whose names start with the prefix "sq". So, */afs/andrew.cmu.edu/service/mailqs/sq2*

is an example of a background mail queue, again in the `andrew.cmu.edu` cell.

If the *mailqs* directory exists for a cell, it must contain at least one regular and one background mail queue subdirectory. The presence of */afs/CELLNAME/service/mailqs* signals to the AMS running in any cell that `CELLNAME` supports the AMS delivery system. Furthermore, it promises that delivery into the `CELLNAME` mail domain may be accomplished by enqueueing a mail request into one of the AMS mail queues found there. Conventional access control permissions on these AMS mail queues include "System:AnyUser li". A mail request, roughly speaking, is a triple of files in a mail queue directory. The three files are grouped together by virtue of the fact that they have the same file name except for the first letter of that name. The remainder of the file names are arbitrary. The first two letters of the three file names are "SF", "QF" and "GF". The QF file contains the text to be sent, the SF file contains the out-of-band information (the sender, the recipient and some additional tracking information) and the GF file contains nothing -- its presence alone signifies that its companion QF and SF files have been completely written.

The presence of AMS mail queues in a cell indicates that they will be serviced frequently by daemons that read the queued messages there and deliver them to their intended recipients. The conventional way in which this takes place is by running (at least) one AMS Post Office machine, which is a workstation solely dedicated to accepting and delivering mail. Each AMS mail queue is serviced by (at least) one process running on a Post Office machine. These processes must be authenticated in their cell as the trusted `postman` user. Recall that the name of that trusted user may be configured at system-build time as the value of the `PostmasterName` configuration variable. AMS Post Office machines are, by convention, run in a physically secure environment. They may also serve as the ports by which ordinary network mail (e.g., SMTP, UUCP etc.) enters and leaves the cell.

The bodies of mail messages in the distributed, inter-cell AMS may be formatted in various ways, including ATK datastreams. The format of such mail bodies is given by an RFC822 header associated with that mail body ("Content-type:"). Processes that read messages from the AMS must be aware of the possibility that the body might be formatted. Thus, they must also be prepared to un-format messages when the recipients cannot understand them (e.g. SMTP, NNTP, or UUCP recipients). There are library functions to perform this de-formatting.

If cells `A` and `B` are running the AMS delivery system, then message delivery between them may be accomplished either via direct insertion into the proper mailbox directories or by use of the mail queues. These messages will retain their source formatting and "Content-type:" headers.

Some configuration decisions about how a cell is run are represented by the contents of the directory */afs/CELLNAME/service/configuration.* Yet again, it must be stressed that the subtree rooted at this directory must reside completely within `CELLNAME`. Should it exist, the */afs/CELLNAME/service/configuration/Postmaster* file contains the mailing address of those people who are responsible for electronic mail at that site.

Notice, though, that not *all* configuration decisions are represented in this directory. For example, a cell administration announces that it runs the AMS delivery system by simply providing an */afs/CELLNAME/service/mailqs* directory. Also, the file named */afs/CELLNAME/service/configuration/WP_Update*, if it exists, contains the mailing address of a process that accepts mailed requests to update fields of the White Pages database. In the `andrew.cmu.edu` cell, the file */afs/andrew.cmu.edu/service/configuration/Postmaster* contains the string "Postmaster@andrew.cmu.edu" (with or without a trailing newline). If a file named */afs/CELLNAME/service/configuration/name-separator* exists, it means that canonical mail addresses in CELLNAME are derived from the user's full name rather than from the userid. Other characteristics of the *name-separator* file have already been discussed above.

The */afs/CELLNAME/service/configuration/AMS-Server* file contains information used by the AMS's *MessageServer* program to configure itself when the AMS home cell (generally the primary authentication) for a user is not the same cell responsible for that workstation's administration. All cells operating an AMS delivery system should install such a file to allow the *MessageServer* to operate as though it had been able to get its *AndrewSetup* information from a workstation in its home cell. This file is ordinarily generated to contain all cell-configurable *MessageServer* options in the conventional manner when installing the AMS delivery system.

There must be at most one distinguished home directory in the AFS for any user listed in its White Pages. This allows mail to that user to be delivered directly to their *~/Mailbox* directory. (As an aside, *~/Mailbox* directories deny "w" (write) access to everyone and grant "k" (lock) access to accounts that may insert files there. The access list is generally "System:AnyUser lik; <owner> rlidka".) Mail is delivered to this *~/Mailbox* directory by creating a new file there. In more detail, a file is created, that file is *flock()*ed, the contents of the file are written, and the file is closed. Readers must *flock()* files they discover in *~/Mailbox* before assuming that they are reading a completely-delivered file. This *flock()* on reading prevents confusion due to multiple simultaneous readers, in addition to ensuring that simultaneous attempts at writing and reading do not collide. It is wise to choose file names such that multiple simultaneous attempts at writing do not share a file name. The absence of "w" (write) permission on the *~/Mailbox* directory helps assure that writers do not collide.

The *queuemail* program must be copied to the workstation's */etc* directory, and it must be setuid-root to allow it to change its local identity and to manipulate the */usr/spool/ViceMsgQueue* queue. The file */usr/spool/ViceMsgQueue* (or whatever is configured as "`LocalQueue`" in the *mailconfig* process) must exist on the workstation, such that only root may write the files contained there. The *startmailsystem* script must also exist on the workstation's */etc*. This script should be responsible for setting the permissions on that directory, for starting the *queuemail* and *guardian* daemons, and for reporting error conditions on the local disk to the local postmaster.

5.2. Implications

If the AMS generates a mail address "foo@bar", the expectation is that when the site "bar" receives mail destined for "foo", it will be delivered to the intended user.

Let us work through an example where the sender has primary authentication in the `andrew.cmu.edu` cell as login name "cfe", with a `pw_gecos` value of "Craig F. Everhart". He also holds tokens for login name "everhart" in the `cs.cmu.edu` cell, where his `pw_gecos` reads "Craig Everhart". The "From:" address in messages sent will appear as ""Craig F. Everhart" <cfe+@andrew.cmu.edu>". If mail is sent to recipients within the `andrew.cmu.edu` cell, the AMS presents the originator of that mail as "Craig F. Everhart". If mail is sent to recipients in the `cs.cmu.edu` cell, the AMS presents the originator of that mail as "Craig Everhart". If mail is sent to any other cell, the AMS may present the originator of that mail as "Craig F. Everhart *" (authenticated format) or as "Craig F. Everhart@andrew.cmu.edu" (unauthenticated format). The "Return-path:" of the mail received in the `andrew.cmu.edu` cell will be "<cfe+@andrew.cmu.edu>"; the "Return-path:" of the mail received in the `cs.cmu.edu` cell may be the same, or it may be "<everhart+@cs.cmu.edu>".

In this example, the mail receiver for mail domain "andrew.cmu.edu" is expected to route mail addressed to "cfe+@andrew.cmu.edu" via the appropriate Andrew mailbox or any mail forwarding that has been established. Likewise, the mail receiver for mail domain "cs.cmu.edu" is expected to route mail addressed to "everhart+@cs.cmu.edu" to the primary Computer Science mailbox or to wherever that mail address forwards its mail.

5.3. Fallbacks from Full Cooperation

Cell administrators may choose not to provide the capacity for complete interoperation with other cells. In this section, we outline the possible fallback positions that are supported by the distributed inter-cell Andrew Message System. Unsupported options must remain forbidden, so that the choices made by one cell's administrators do not corrupt AMS operation in other cells. That is, any site that chooses not to maintain full compliance must do so in a way that will be recognized by the AMS facilities running elsewhere.

A cell, `A`, may choose to provide neither a White Pages nor any mail queues at all. The AMS delivery system will be unable to run with primary authentication in such a cell. AMS user interface programs may be specially configured to run with primary authentication in such a cell, but they will not attempt AMS delivery system or local White Pages functions. If an AMS program runs in another cell, `B`, that supports the AMS delivery system, then `B`'s program will be unable to resolve mail names or find final mail destinations for `A`. Additional restrictions are given in this section's closing paragraph.

Cell `C` may choose to provide a White Pages but not mail queues. Its White Pages remains suitable for use by such programs as *ls*, but not for mail delivery. As with `A`, the AMS delivery system will be unable to run with primary authentication in cell `C`, and AMS user interface programs there will not attempt AMS delivery system

functions. Similarly, AMS programs running in other cells will be unable to validate mail names in C. This situation thus corresponds to a cell that chooses not to run the AMS mail delivery system. The AMS, run with its primary authentication in such a cell, would not attempt inter-cell mail delivery of any kind. The AMS will ignore any secondary authentications in such cells as C, because the only path by which mail may be delivered into such a cell will be through such mechanisms as SMTP. Again, additional restrictions appear below.

If a cell wishes to advertise that it runs the AMS delivery system, it will provide both a set of mail queues and a White Pages database in the standard fashion. It is not absolutely required that a cell's mail queues be open to anonymous insertion from another cell, but it is a good idea. If user Anonymous cannot insert into the mail queues, then mail must be delivered into that cell either in accordance with the White Pages or through some facility outside of AMS such as SMTP.

Sites that support both White Pages and publicly-insertable mail queues are the fullest participants in this cellular scheme. Such cells may exchange messages that are fully authenticated by a combination of the standard AFS authentication mechanisms and the recipient's trusted mail delivery agent.

As mentioned above, cells that choose not to run the AMS delivery system impose some additional restrictions on people using their workstations. In order to send mail with the AMS delivery system from a given workstation, the cell in which that workstation ordinarily resides must itself run the AMS delivery system. Consider the user whose primary authentication is in a cell running a full AMS delivery system, yet whose workstation is sitting in a different cell that does *not* fully support AMS. In this situation, the user will be able to compose and send mail, but it may not be delivered for several hours since the workstation does not support the AMS dropoff function. (In general, AMS delivery requires that a good deal of the delivery-specific work occur on the workstation itself.) Alternatively, if a user's primary authentication is in a cell not running the full AMS delivery system, but the user's workstation is in another cell, that user's home for the purposes of reading and sending mail and bulletin board notices will ***not*** be considered to be where the user has primary authentication, but rather in the workstation's cell. If the user has no authentication in the workstation's cell, AMS programs like *Messages* will be unable to run. If the primary authentication is neither in the workstation-native cell nor in a cell with AMS delivery, AMS programs will not know how to configure themselves. Configuration information for cells running AMS delivery is available by convention and via the */afs/CELLNAME/service* directory, whereas most configuration options for cells not running AMS delivery are not made public this way.

6. Configuration Information

The introduction of cooperating Andrew installations has created the need to maintain configuration information for this community. Each Andrew cell must know the names of the other cells in the confederation as well as which machines are acting as their volume location and authentication servers. This section supplies the information needed by system operators to understand the contents of these files, make changes when necessary and diagnose problems caused by omission or corruption of these files on a workstation's local disk. All examples are written from the point of view of the `andrew.cmu.edu` cell; other sites should easily be able to extrapolate.

6.1. The */usr/vice/etc* Files

All of the Andrew cellular configuration files are stored in both the AFS and each workstation's local disk. The */afs/andrew.cmu.edu/common/etc* directory acts as the central AFS repository. At boot time, the *package* program insures that the configuration files are present and up to date in */usr/vice/etc*, located on the workstation itself. There are currently four cellular configuration files, only two of which are necessary for proper operation. One of the four, *domain.auth-backup*, is scheduled for removal. Let's look at each file individually:

ThisCell: This file contains the name of the Andrew cell to which the given workstation belongs. It is permissible to have leading and trailing white space in this file, along with any number of trailing linefeeds/carriage returns. This file is the only source of this information, so it is critical to the operation of the workstation's *CacheManager*. On startup, one of the first things the *CacheManager* does is to read this file to find out which cell it's working for. If it can't read it, the *CacheManager* will decide it is running in a mythical cell called `localcell`. If the version of */usr/vice/etc/ThisCell* read has garbage in it, the *CacheManager* will adopt the first string it can read from the file as its cell name. Errors of this type will manifest themselves in bizarre ways later, as described below.

CellServDB: This file contains the full list of known Andrew cells (including your own), along with the names and IP addresses of the machines providing their volume location and authentication services. The information is in simple ASCII text format, so it is possible to change with any editor. The current contents of *CellServDB* in the `andrew.cmu.edu` cell is reproduced below for easy reference:

```
>andrew.cmu.edu          #ITC/Campus cell
128.2.10.2                       #vice2.fs.andrew.cmu.edu
128.2.10.7                       #vice7.fs.andrew.cmu.edu
128.2.249.123                    #scm.fs.andrew.cmu.edu
>beta.andrew.cmu.edu     #ASA's beta cell
128.2.72.1                       #beta1.bfs.andrew.cmu.edu
128.2.72.2                       #beta2.bfs.andrew.cmu.edu
```

```
128.2.72.10                     #scm.bfs.andrew.cmu.edu
>cs.cmu.edu                  #Computer Science cell
128.2.222.180                   #mango.srv.cs.cmu.edu
128.2.242.81                    #peach.srv.cs.cmu.edu
128.2.242.86                    #lemon.srv.cs.cmu.edu
128.2.250.187                   #guava.srv.cs.cmu.edu
128.2.217.45                    #apple.srv.cs.cmu.edu
128.2.222.199                   #papaya.srv.cs.cmu.edu
>psy.cmu.edu                 #Psychology cell
128.2.248.147                   #thistle.psy.cmu.edu
```

Every line starting with a ">" marks the beginning of the information for a particular cell. The opening ">" is followed immediately by the domain name of the cell being described, some amount of whitespace, a "#" and finally a wordier description of the cell. Each line between this marker and the next one (or end-of-file) describes a machine providing volume location/authentication services for that cell. The description starts with the machine's IP address in dot notation and is followed by some whitespace, a "#" and finally its full domain name. The software manipulating this file parses and stores this information, and in fact tends to use the explicit IP addresses only as a last resort. The last server description line for each cell must identify its System Control Machine (SCM). While the SCM normally provides neither volume location nor authentication service, it is still necessary to include it here so that passwords may be changed in other cells. The SCM is the only place where this type of update can take place in a cell, forcing its appearance on this list.

domain.auth-backup: This file is *CellServDB*'s predecessor, and is no longer being used by anyone or anything. It is scheduled for removal in the near future.

cacheinfo: This file contains the following three necessary pieces of configuration information, separated by colons, for the in-kernel *CacheManager*:

1. The Unix directory on the workstation's local disk on which the AFS will be mounted. This is expected to be */afs* for a workstation participating in the cellular community, but may take on other values for such reasons as debugging.
2. The Unix directory on the workstation's local disk to use for the *CacheManager*'s disk cache.
3. The maximum number of 1Kbyte blocks on the workstation's local disk which may be devoted to the *CacheManager*'s disk cache.

Here are the contents of this file in the `andrew.cmu.edu` cell:

```
/afs:/usr/vice/cache:30000
```

In this case, the *CacheManager* is being told that the AFS should be mounted locally at */afs*, that the */usr/vice/cache* directory will serve as its cache directory and that 30 Mbytes (30,000 1Kbyte blocks) are available for cache storage on disk.

The user-level *CacheManager* does not use this file. Instead, it depends on */etc/vstab* to supply this information.

6.2. Use of Configuration Files

The *librauth.a* library contains a module (*cellconfig.c*) which serves to interpret the cell configuration files and make their information available to various application programs. Let us consider these applications individually:

log, passwd: These two programs use the configuration file information to contact the proper authentication servers when someone invokes an operation that must be handled by another cell. In the case of *log*, a user may wish to generate tokens so they can manipulate protected files in a remote cell. Take the example where someone with user name *frank* wishes to establish authentication with the Psychology cell, which is named `psy.cmu.edu`. The user types in "*log frank -c psy.cmu.edu*". The *log* program recognizes that an explicit cell is mentioned on the command line and deals directly with the authentication servers in that cell. In the case of *passwd*, reference to another cell will cause it to contact the last server in that cell's list, which is guaranteed to be the SCM (see the description of */usr/vice/etc/CellServDB* in Section 6.1 above).

vcellconfig: This program passes all of the information in the *CellServDB* file to the workstation's *CacheManager* The different types of *CacheManager* begin life with different levels of knowledge concerning the members of the cellular community. As discussed earlier, the user-level *CacheManager* only reads the *ThisCell* file upon invocation. Along with the information it picks up from */etc/vstab*, it starts off knowing everything it needs to about the cell in which it's running, but no others. Thus, before a machine running a user-level *CacheManager* can access files in other cells, *vcellconfig* must be run to round out its knowledge. In the `andrew.cmu.edu` cell, *vcellconfig* is run automatically as part of the bootup script. So, by the time the login prompt is issued at a workstation running a user-level *CacheManager* process, information on all advertised cells is available. The in-kernel *CacheManager*, on the other hand, reads the information in *CellServDB* on its own upon initialization. Thus, it comes up with a full awareness of the other sites in the community.

Users may run *vcellconfig* as many times as they want; multiple invocations won't hurt anything. At worst, the *CacheManager* will receive the same picture of the state of the cellular world as it had before. However, should a cell have been added or deleted, running *vcellconfig* will update the

CacheManager's knowledge. The *-v* switch tells *vcellconfig* to be verbose about its activities. Anyone who's more curious about this program can feel free to use this switch.

AMS, guardian: These programs have been modified to extract cellular information from the *CacheManager*. A version of the *CacheManager* capable of providing this information to such applications has been released on both the */usr/andy* and */usr/andrew* sides. If *Messages* asks about a remote cell and the (user-level) *CacheManager* hasn't been prepared by *vcellconfig*, it won't know how to answer.

6.3. Updating Configuration Files

Operators should not normally be changing the cellular configuration files in the AFS directories themselves. Their responsibility lies in discovering that they are either not present or damaged on the workstation's local disk and repairing the problem. When a change has to be made, it is carried out on the AFS copy (or copies) and propagated to individual workstations using whatever configuration tools are available. In the `andrew.cmu.edu` cell, the *package* facility performs these updates upon workstation reboot.

In those sites using *package*, operators should avoid editing the versions of these files that are on local disk. Unless they protect them (as `root`) against writing, the repair will disappear the next time the machine boots. ***These files are never to be left with the owner's write protection disabled!*** Doing this will prevent the *package* facility from overwriting stale files with updated ones. Incidentally, *package* has been instructed to boot the machine *again* if any of these files have changed since the last time it ran. Otherwise, the *CacheManager* that is already running will use old and often crippling information. After the second boot, the *CacheManager* is guaranteed to have an accurate view of the world.

6.4. Typical Errors and How to Handle Them

There are typically a small number of errors that indicate that something is wrong with the cellular configuration files.

If the *CacheManager* or an application complains about not being able to read a certain configuration file, the presence, integrity and access rights of the file must be checked. As stated above, if one of these files has been corrupted on the local disk, the operator should resist simply editing it in place. The correct response is to ensure the file is included in the *package* list, update the affected file(s) and reboot the workstation.

If someone is having trouble getting to certain AFS files and the normal checks reveal nothing, see what the *CacheManager* knows about cells. The "*fs listcells /afs*" command will print out a complete picture of the cellular world as the *CacheManager* currently sees it. If this information includes "`localcell`" or some unusual string anywhere, then there was trouble reading the */usr/vice/etc/ThisCell* file. Otherwise,

check the printout against the contents of the AFS copy of *CellServDB*. Differences here could cause access problems by telling the *CacheManager* to contact the wrong servers when it is trying to fetch and/or store files. By the way, a direct check on how well *ThisCell* was read is to issue the "*fs wscell /afs*" command, which echos the contents of this file at the time the *CacheManager* looked at it. If the printout looks fine (it'll read " *This workstation belongs to cell 'andrew.cmu.edu'* "), find out which file in particular is causing the problem. Type in "*fs whichcell* ***filename***", and it will tell you in which cell the file resides. If information for that cell doesn't appear in the output produced by "*fs listcells /afs*" or if the command returns an error, then the user needs to run the *vcellconfig* utility, which will refresh the *CacheManager*'s image of the cellular configuration.

In general, unusual problems having to do with authentication and users not being able to access certain files should prompt operators to check on the status of the cell configuration files.

7. Server Interfaces

One of the most important interfaces in the Cellular Andrew environment is that between a *CacheManager* and *FileServer*. Not only must the interface routines be well-defined, but so must the characteristics and architecture of the underlying RPC.

These topics are too complex to deal with in this paper. The reader is referred to [8] and [9] for a full treatment.

8. Programming Interfaces

This section explores the Andrew programming interfaces that are affected by the upgrade to a cellular environment. The interface to the *CacheManager* has seen both the addition of new routines and the modification of existing ones, and these will be covered in detail. The optional White Pages facility has also experienced changes in its interface, and is examined next. Programmers are well advised to use the White Pages interface even if their cell does not provide this service. At such sites, these routines fall back on the local password file. Furthermore, should the decision be made at some later point to add White Pages services, all such software will automatically access the database without recoding. Finally, the changes to the authentication library are revealed. In each of these three areas, sample code will illustrate the proper usage of the new or modified operations.

8.1. The *CacheManager* Interface

The full *CacheManager* interface is described in a separate document, which may be found in */afs/andrew.cmu.edu/common/usr/andy/doc/vdoc/venus.vdoc*. The portions relating to the cellular *pioctl()*s and the single cellular *ioctl()*, *VIOCIGETCELL*, have been extracted and reproduced in Section 8.1.1 for convenience. The reader is referred to the above document for a full explanation of the *CacheManager* interface. After the excerpts describing the new or altered interface operations, some examples of their use are given. The *fs* utility provides access to much of the *CacheManager* interface for the user sitting at his workstation, and the programming examples offered in Section 8.1.2 are taken from this code.

8.1.1. Affected Operations

The VIOCIGETCELL ioctl()
Get the cell name associated with the given open file descriptor. If the file is not in the AFS, this call returns an error (ENOTTY or EINVAL). Otherwise, it returns one output parameter: the null-terminated name of the cell containing the file open on this descriptor. This is the recommended way of telling whether or not a file is stored in the AFS (when a file descriptor for it is available).

VIOCSETTOK
Set authentication tokens. This call has one input parameter, the encoded authentication tokens. These tokens are associated with the Unix uid of the process performing the call. They are obtained (usually by *log* or *login*) by contacting an *AuthServer*, using the user's password as an encryption key. Thus, authentication with any *FileServer* is possible without actually having the user's password stored in memory.

Tokens come in **clear token** and **secret token** pairs. The secret token is encrypted (by the *AuthServer*) with the *FileServer*'s "password", and can be safely transmitted over the network. The clear token is the unencrypted version of the same and should not be transmitted over the network, as it contains a session key in the clear that

would allow others to observe one's communication.

The input parameter encodes these tokens as follows. The first 4 bytes are the size of the secret token. The next `sizeof(SecretToken)` bytes contain the secret token itself. Similarly, the next 4 bytes give the size of the clear token and the next `sizeof(ClearToken)` bytes hold the clear token. Each field must be byte-aligned. Fields are copied around using *bcopy()*.

If the tokens are not valid, the code EPERM is returned. EINVAL is returned if the token sizes do not match what the file system expects.

Like *VIOCGETTOK* (described below), this call has also been extended to support multiple cells. In particular, one may provide extra information after the tokens. If present, this extra data is interpreted as a long integer specifying whether this is the primary cell ID for the user, followed by a character string giving the cell name for the tokens. This call will fail if the cell has not yet been configured on the *CacheManager*.

VIOCGETTOK

Get authentication tokens. This call gets the authentication tokens associated with the uid making the call. There are no input parameters and one output parameter, namely the authentication tokens. They are encoded as in the *VIOCSETTOK* call. The pathname parameter must be a file in the AFS.

This call has been extended to support multiple cells. It now has one optional input parameter, which, if present, tells us we're using the new calling convention. This parameter is a sequencer, between 0 and infinity, telling the *CacheManager* which set of tokens to return. There is no connection between this number and any internal cell number used by the *CacheManager*. Rather, it is just an iteration mechanism for performing multiple calls to the *CacheManager* to get all tokens associated with the caller. When using the new calling convention, we return a long integer after the token information, indicating whether this is the primary cell ID, and a character string indicating the cell name. If the *CacheManager* is holding *n* tokens for the caller, invoking *VIOCGETTOK* with an iterator greater than *n* will cause an error return (EDOM).

VIOCNEWCELL

Tell the *CacheManager* to configure a new cell. The input parameter contains up to MAXHOSTS (8) hosts in network byte order that provide the volume location and authentication servers for this cell. The parameter must be null-terminated if there are fewer than the maximum number of hosts. This 32-byte array is followed by the null-terminated, fully-qualified cell name.

VIOCGETCELL

Get a cell name and its associated hosts. Takes one input parameter, a zero-based index. Returns the same structure as in *VIOCNEWCELL*. When the index gets too large (i.e., exceeds the number of cells currently configured into the system), this call

returns a standard Unix error (EDOM).

VIOC_FILE_CELL_NAME
Given a file name, determine the name of the cell in which that file resides. This call has no input parameters, and returns a character string identifying the file's corresponding cell. If the file doesn't exist, or for some other reason cannot be accessed, the call will fail and `errno` will indicate the reason for the error.

VIOC_GET_WS_CELL
Get the name of the cell to which the workstation belongs. This call has no input parameters, and returns a character string identifying the "home cell" for the workstation.

VIOC_GET_PRIMARY_CELL
Return the name of the cell in which the caller has established his "primary" identity (i.e., where he has done his *login*). Note: the *log* program can be used to establish both primary and non-primary tokens of identity in any number of cells; any non-primary identities are not reported here. If the user does not currently have a primary identity (i.e., they have done an *unlog*), this call will return the null string.

8.1.2. *fs* Coding Examples

Two examples are provided below to illustrate the proper use of these *CacheManager* operations. They have been taken from the *fs* program, which allows users at a workstation to communicate their desires to their *CacheManager* and to extract information from it. The first example demonstrates a non-iterative call, simply getting the cell in which the caller has his primary authentication (if he has one at all):

```
...
#define MAXSIZE 2000
char space[MAXSIZE];
struct ViceIoctl blob;
     ...
    else if (!strcmp(argv[1], "whichcell")) {
        /*
         * Find out which cell a given file/directory lives in.
         */
        if (argc != 3) {
            printf("%s: Syntax error in whichcell command.\n", pn);
            exit(1);
        }

        blob.in_size = sizeof(argv[2]);
        blob.in = argv[2];
        blob.out_size = MAXSIZE;
        blob.out = space;
```

```
        code = pioctl(argv[2], VIOC_FILE_CELL_NAME, &blob, 1);
        if (code) {
            Die(code, argv[2]);
        }
        else
            printf("File lives in cell '%s'\n", space);
    }
    else...
```

As described in the full *CacheManager* interface document, the `blob` structure is used as a communication buffer. The first true argument to *fs whichcell* is in `argv[2]`, and is the name of the file whose place of residence we wish to discover. We load this filename as the input parameter to the *pioctl()*, and set up the `space` array as the output parameter into which the associated cell name will be placed. All that has to be done at this point is to call *pioctl(VIOC_FILE_CELL_NAME)*, passing the address of the `blob` and using the third parameter to instruct it to follow any symbolic links encountered in the input argument. If the return code is zero, then the *CacheManager* has successfully returned the associated cell name in the user's `space` array. Otherwise, the *Die()* routine, listed below, is used to print out what went wrong:

```
Die(code, filename)
    int   code;
    char *filename;

{ /*Die*/

    char FullError[256];

    if (errno == EINVAL) {
        printf("%s: Invalid argument.\n", pn);
        printf("%s: Possible that file is not in AFS.\n", pn);
    }
    else if (errno == ENOENT)
        printf("%s: File '%s' doesn't exist\n", pn, filename);
    else if (errno == EROFS)
        printf("%s: You may not change a backup volume\n", pn);
    else if (errno == EACCES)
        printf("%s:  You  don't  have  the  required  access  rights  on
'%s'\n",
                pn, filename);
    else {
        sprintf(FullError, "%s:'%s'", pn, filename);
        perror(FullError);
    }
    commandError = 1;    /*So we exit(1)*/

} /*Die*/
```

The following excerpt from the *fs* program demonstrates the proper use of a *pioctl()* meant to be used iteratively. In this case, the caller has invoked *fs listcells*, which must generate a list of all cells the *CacheManager* currently knows about, along with their associated servers:

```
...
else if (!strcmp(argv[1], "listcells")) {
        /*
         * List all cells.
         */
        if (argc != 3) {
            printf("%s: Syntax error in listcells command.\n", pn);
            exit(1);
        }
        for(i=0; i<1000; i++) {
            blob.out_size = MAXSIZE;
            blob.in_size = sizeof(long);
            blob.in = space;
            blob.out = space;
            bcopy(&i, space, sizeof(long));

            code = pioctl(argv[2], VIOCGETCELL, &blob, 1);

            if (code < 0) {
                if (errno == EDOM)
                    break;      /*Done with the list*/
                else {
                    Die(code, argv[2]);
                    exit(1);
                }
            }
            printf("Cell %s on hosts", space+8*sizeof(long));
            for(j=0; j < 8; j++) {
                bcopy(space + j*sizeof(long), &clear, sizeof(long));
                if (clear == 0) break;
                thp = gethostbyaddr(&clear, sizeof(long), AF_INET);
                if (thp) {
                    printf(" %s", thp->h_name);
                }
                else {
                    printf(" %08x", clear);
                }
            }
            printf(".\n");
        }
    }
    else...
```

This time we loop, not expecting to find more than 1,000 cells configured into the *CacheManager*. The iterator is stuffed into the input parameter area for each *pioctl(VIOCGETCELL)* call. For the *i*th call, we print out the *i*th configured cell name and its servers (getting the servers' names from their IP addresses with the *gethostbyaddr()* utility). On some call *j*, we'll get back a failing (non-zero) error code. The special case that is watched for here is EDOM, signifying that there are no more configured cells, with the last valid one numbered *j*-1. Any other error value is handled as a true error with the *Die()* routine described above. The choice of 1,000 for the upper limit will have to be increased should more than this number of Andrew cells ever be in existence.

8.2. White Pages

As explained in various parts of this document, the White Pages is an optional service available to cell administrators. It provides a superset of the local password file information, with special features useful in running the AMS. It also provides some interface routines that have been upgraded to work in the Cellular Andrew environment. Programmers now have extensions to the standard *getpw*()*-class calls that allow them to specify the cell in which the information is gathered, and also whether information for the primary authentication is desired. Again, these interface routines may still be used whether or not the cell decides to provide a White Pages service. They are available as part of the normal system build. They default to operations on the local password file should a White Pages database not be available. It is wise, in fact, to use these routines even when White Pages service is not being currently supported at the programmer's site. Should the cell step up and introduce a White Pages for its users at a later date, its full power will be immediately available to all the locally-produced software without the need for recoding.

The complete White Pages interface may be found in */usr/andy/include/wp.h*, but is not of immediate interest to us here. The *wp.h* file specifies those routines which will only work should a White Pages actually be installed. The subject of these next two sections will be the White Pages-related routines defined in */usr/andy/include/util.h* and implemented in the system's standard *libutil.a* library, which is always available to any AFS installation. These routines augment the power of the vanilla Unix *getpw*()* calls. In this description, they are split into two "classes", *V* and *C*.

8.2.1. *V*-Class Routines

The standard *getpwuid()*, *getpwnam()*, *setpwent()*, *endpwent()* and *getpwent()* routines in Unix allow the programmer to manipulate the local password file, typically stored in */etc/passwd*. Documentation on their arguments, return values and uses is found in the conventional Unix man pages. However, these functions need to be augmented in order to access the full power of the Cellular Andrew world. First, processes hold ViceIDs, used for AFS authentication purposes, which may differ from their actual Unix process IDs. Furthermore, any process may be associated with several ViceIDs, as authentication in several cells at once is possible. The so-called *V*-class routines are identical to the above Unix routines, with the sole exception that the information

they gather is for the primary identity associated with the calling process. Should the caller not have a primary identity (see Section 2.1.2 for a full description and the mechanism by which processes can forego primary identities), these routines will fail, and the standard Unix routines may be used instead.

For convenience, here is the full specification for the *V*-class routines, as taken from the various source modules that implement them:

```
int getvuid()

struct passwd *getvpwuid(vuid)
    int vuid;

struct passwd *getvpwnam(vnam)
    char *vnam;

int setvpwent()

int endpwent()

struct passwd *getvpwent()
```

8.2.2. *C*-Class Routines

The *V*-class routines described above must determine the user's primary identity, which could be in any cell at the time they are called. There is a different class of routines provided via this special White Pages interface which allows the user to specify the exact cell in which the information will be gathered. This *C*-class is made up of the following two routines:

```
struct passwd *getcpwuid(vuid, vcell)
    int vuid; char *vcell;

struct passwd *getcpwnam(vnam, vcell)
    char *vnam, *vcell;
```

Both of the above functions are called just as the standard Unix routines from which their names are derived, except that they take a second argument. The additional parameter is the string name of the cell in which the password-style lookup is to take place. These are useful for those occasions when the particular cell that must be searched is known. They are also particularly useful in that they do not depend on any particular authentication to be designated as "primary".

8.2.3. Example From *ls*

The use of these routines should be fairly straightforward to an experienced Unix programmer. Regardless, this section illustrates the use of a *C*-class routine,

getcpwuid(), by the *ls* utility.

```
    ...
 /*
  * Element in a queue of cell names seen so far.
  */
   struct HTCellName {
        struct HTCellName *next;    /*Ptr to next cell name on list*/
        char full[64];               /*Full cell name*/
        };
 ...
struct afile {
                ...
                struct HTCellName *cellname;
                ...
                };
register struct passwd  *pw;    /*Passwd entry*/
 short uid;
struct afile *fileinfop;
...
 /*
  * Our cache didn't have the uid -> name mapping.  Look it up in
  * the passwd database.
  */
   HT_uidCacheMisses++;
   if (DB_ls)
       printf("[%s:%s] Cache miss, looking up uid %d in cell '%s'\n",
              mn, rn, uid, fileinfop->cellname->full);

   pw = getcpwuid((int) uid,
                  fileinfop->cellname->full /*Full cell name*/);

   if (pw == 0) {
       if (DB_ls)
           printf("[%s:%s] Can't map uid %d in cell '%s'\n",
                  mn, rn, uid, fileinfop->cellname->full);
       sprintf(MappedName, "%d", uid);
   }
   else
       sprintf(MappedName, "%s", pw->pw_name);
```

The standard *ls* program, when asked to provide printable names for the uid corresponding to an owner of a file, uses the local password database to derive these mappings. The cellular-capable version of *ls* used at Andrew sites must also worry about performing these mappings for files in different cells from its own. Once *ls* determines which cell a particular file lives in (with the *VIOC_FILE_CELL_NAME pioctl()*, not shown above; see Section 8.1.1) and the owner's uid (from a *stat()*), it

uses *getcpwuid()* to grab the appropriate password record. It extracts the printable name associated with the given uid from this record. Should that given uid not have a name mapping in the chosen cell, *ls* chooses to report the raw uid in its place.

The *getcpwuid()* operation is used freqently by *ls*, making the derived information particularly amenable to caching. It is expected that most files in a directory are owned by a small number of people, often only one. The above *ls* excerpt alludes to a hash table in which *([uid, cell] --> name)* mappings are kept. In practice, very high hit ratios are achieved on these caches. Performing such caching of information from remote databases not only makes software much faster, but also reduces network traffic and the load on the *FileServers*. When writing important utilities, programmers should keep this in mind.

8.3. Authentication

The *CacheManager* may now store several token pairs for each of its clients, as described in Section 2.1.2. Thus, the Andrew authentication interface was modified to reflect this new situation. This section describes the routines added to the interface in support of additive authentication. The *libauth.a* library implements all of these operations. Two component source files were affected: *auser.c* and *avenus.c*, both living in directory */afs/andrew.cmu.edu/itc/src/rel/vice/auth*.

Most of these routines have non-cellular counterparts so that programs written before the cellular changes would run unchanged. However, the internals of these counterparts have been altered to call upon the new interface routines directly, so they are now little more than stubs. These new functions, in turn, communicate with the *CacheManager* by calling the appropriate *pioctl()*s (see Section 8.1.1).

In addition to the changes in the existing *auser.c* and *avenus.c* modules, a new pair of source files was added to the authentication library: *cellconfig.c* and its interface, *cellconfig.h*. The header file is installed in the */usr/andy/server/include* directory. The functions provided in this module allow an in-memory version of the file-based cell configuration information to be built, as well as discovering the name of the cell to which the local workstation is attached. This frees the programmer from having to know the exact names and locations of the configuration files, and even if they are files at all (the information may be kept in the domain system in future implementations).

Readers interested in the actual code for the routines described below are encouraged to look at the sources. After presenting the new authentication interface routines, some examples of their use are provided. Keep in mind when reading these routine descriptions that *Venus* is an older term for the Andrew *CacheManager*.

8.3.1. Routines in *auser.c*

```
/*-----------------------------------------------------------------------------
 * U_CellAuthenticate
 *
 * Description:
 *      Talks to an Authentication Server for the given cell and obtains tokens
 *      on behalf of user uName.  Gets back clear & secret tokens for this user.
 *
 * Arguments:
 *      uName      : Ptr to the user's login name.
 *      uPassword  : Ptr to the user's password.
 *      cellID     : Ptr to the name of the cell to contact.
 *      cToken     : Ptr to the location in which to deposit the clear token.
 *      sToken     : Ditto for the secret token.
 *
 * Returns:
 *      >0 if the operation succeeded and the tokens were generated,
 *       0 otherwise.
 *
 * Environment:
 *      Nothing interesting.
 *
 * Side Effects:
 *      None.
 *---------------------------------------------------------------------------*/

int U_CellAuthenticate(uName, uPasswd, cellID, cToken, sToken)
        char                    *uName;
        char                    *uPasswd;
        char                    *cellID;
        ClearToken              *cToken;
        EncryptedSecretToken    *sToken;
```

```
/*-----------------------------------------------------------------------------
 * U_CellChangePassword
 *
 * Description:
 *      Binds to the AuthServer running on the SCM in cell cellName and
 *      changes user uName's password to newPasswd if myName is the
 *      same as uName or a system administrator (at the target cell).
 *      MyPasswd is used to validate myName.
 *
 * Arguments:
 *      uName       : The user id being changed.
 *      newPasswd   : The new password desired.
 *      myName      : The user performing the change.
 *      myPasswd    : The password of the user making the change.
 *      cellName    : The Andrew cell where uName lives.
 *
 * Returns:
 *      Whatever is returned by U_CellBindToServer(), AuthNameToID() or
 *      AuthChangePasswd.
 *
 * Environment:
 *      Nothing special.
 *
 * Side Effects:
 *      As advertised.
 *-----------------------------------------------------------------------------*/

int U_CellChangePassword(uName, newPasswd, myName, myPasswd, cellName)
    char *uName;
    char *newPasswd;
    char *myName;
    char *myPasswd;
    char *cellName;
```

```
/*-----------------------------------------------------------------------------
 * U_CellBindToServer
 *
 * Description:
 *      Binds to an AuthServer in the given cell on behalf of user uName using
 *      uPassword as the password.  Sets RPCid to the value of the connection
 *      id established, if any.
 *
 * Arguments:
 *      write     : Whether the SCM is to be used as the chosen AuthServer.
 *      uName     : The user name to use.
 *      uPasswd   : The user's password.
 *      cellID    : The name of the cell to contact.
 *      RPCid     : Set to the RPC id established with the AuthServer we
 *                  contacted, if any.
 *
 * Returns:
 *       0        if everything went well,
 *      >0        otherwise.
 *
 * Environment:
 *      Variable SetGlobalVars determines whether the global settings of
 *      GlobalUserName and GlobalPassword are set here.  This routine
 *      should also make sure we aren't doing replays by calling a couple
 *      of procs in the AuthServer.
 *
 * Side Effects:
 *      Loads up the set of AuthServers, SCM to talk to from the Domain
 *      system, or from the local fallback file in case of failure.  Sets the
 *      name to be used in pioctl()s at this time, too.
 *-----------------------------------------------------------------------------*/

int U_CellBindToServer(write, uName, uPasswd, cellID, RPCid)
        int                      write;
        char                    *uName;
        char                    *uPasswd;
        char                    *cellID;
        struct r_connection     **RPCid;
```

```
/*-----------------------------------------------------------------------------
 * DetermineAuthServers
 *
 * Description:
 *      Given a cell name, determine the set of AuthServers advertised by that
 *      cell.  If the primary Domain system search fails, we fall back on the
 *      locally-maintained database.
 *
 * Arguments:
 *      cellID : The string name of the cell we want to talk to.
 *
 * Returns:
 *      AUTH_SUCCESS            if everything goes well.
 *      AUTH_NOSUCHCELL         if the primary Domain service has never
 *                              heard of the given cell.
 *      AUTH_NOTINBACKUP        if the primary Domain service is not available
 *                              and the cell doesn't appear in the local
 *                              backup file.
 *      AUTH_FAILED             if any other failure condition is encountered.
 *
 * Environment:
 *      We use the cellconfig package to do basically all the work for us.
 *      The global CDBp and cellInfop pointers are set here.
 *
 * Side Effects:
 *      Sets numHosts, lHosts array.
 *-----------------------------------------------------------------------------*/

int DetermineAuthServers(cellID)
        char *cellID;
```

8.3.2. Routines in *avenus.c*

```
/*-----------------------------------------------------------------------------
 * U_CellSetLocalTokens
 *
 * Description:
 *      Tells Venus about the clear and secret tokens obtained from the
 *      AuthServer, as well as whether this identity is the Primary one and
 *      which cell these tokens are valid in. If setPag is true, a setpag
 *      system call is made. Returns 0 on success, -1 on failure.
 *
 * Arguments:
 *      setPag      : If true, do a setpag system call.
 *      cToken      : Ptr to the clear token.
 *      sToken      : Ptr to the secret token.
 *      cellID      : String name for the cell these tokens are valid in.
 *      primaryFlag : Does this token set represent the primary identity?
 *
 * Returns:
 *      0 on success,
 *      -1 otherwise.
 *
 * Environment:
 *      Nothing interesting.
 *
 * Side Effects:
 *      None.
 *-----------------------------------------------------------------------------*/

int U_CellSetLocalTokens(IN setPag, IN cToken, IN sToken,
                         IN cellID, IN primaryFlag)
    int                  setPag;
    ClearToken           *cToken;
    EncryptedSecretToken *sToken;
    char                 *cellID;
    int                  primaryFlag;
```

```
/*----------------------------------------------------------------------------
 * U_CellGetLocalTokens
 *
 * Description:
 *      Gets the specified tokens from Venus, filling in cToken and sToken
 *       (and possibly cellID and pIsPrimary).
 *
 * Arguments:
 *    useCellEntry : If true, supply the optional cellEntry argument to Venus,
 *                   prompting Venus to return information for a specific cell.
 *    cellEntry    : Ask Venus to return info about the tokens for the
 *                   cellEntry'th cell associated with the caller.
 *       cToken    : Ptr to the clear token buffer to fill.
 *       sToken    : Ptr to the secret token buffer to fill.
 *       cellID    : String name for the cell these tokens are valid in. Filled
 *                   iff useCellEntry is true.
 *       pIsPrimary : Does this token set represent the primary identity?
 *                    Filled iff useCellEntry is true.
 *
 * Returns:
 *       0       on success,
 *    EDOM if cellEntry is used and is out of range,
 *       -1      otherwise.
 *
 * Environment:
 *       Nothing interesting.
 *
 * Side Effects:
 *       None.
 *----------------------------------------------------------------------------*/

int U_CellGetLocalTokens(IN useCellEntry, IN cellEntry,
                         OUT cToken, OUT sToken, OUT cellID,
                         OUT pIsPrimary)
    int                  useCellEntry;
    int                  cellEntry;
    ClearToken           *cToken;
    EncryptedSecretToken *sToken;
    char                 *cellID;
    int                  *pIsPrimary;
```

8.3.3. Routines Exported by *cellconfig.h*

```
/*
 * Complete server info for one cell.
 */
struct CellServers {
    struct CellServers *pNext;     /*Ptr to next entry*/
    char cellName[MAXCELLCHARS];   /*Cell name*/
    short numServers;              /*Num active servers for the cell*/
    struct in_addr cellHostAddr[MAXHOSTSPERCELL]; /*IP addrs for cell's
servers*/
    char cellHostName[MAXHOSTSPERCELL][MAXHOSTCHARS]; /*Their names*/
};

extern char LclCellName[MAXCELLCHARS];

extern int GetLocalCellName();
     /*
      * Args:
      *   None.
      *
      * Returns:
      *   Indication of whether the local cell name was correctly determined.
      */

extern struct CellServers *ReadCellDatabase();
     /*
      * Args:
      *   None.
      *
      * Returns:
      *   Ptr to the first entry of the in-memory copy of the cell/servers
      *     database, or NULL if an error was encountered.
     */

extern int CellLookup();
     /*
      * Args:
      *   char                *cellToFind;
      *   struct CellServers  *pCellDatabase;
      *   struct CellServers **ppCellRec;
      *
      * Returns:
      *   CCONF_SUCCESS if the search succeeded,
      *   CCONF_NOTFOUND if the search failed.
      *   CCONF_FAILURE if the params were screwed up.
     */
```

```
extern void ReclaimCellDatabase();
      /*
       * Args:
       *      struct CellServers *pDBToDitch;
       *
       * Returns:
       *      Nothing.
       */

extern void PrintCellDatabase();
      /*
       * Args:
       *      struct CellServers *pDBToPrint;
       *
       * Returns:
       *      Nothing.
       */
```

8.3.4. Examples from *log*

The *log* program allows a user to generate authentication tokens and pass them onto the *CacheManager*, which acts on his behalf during file access operations. The user enters his login name, password and optionally the specific cell that is to be contacted. By default, this is the same as the workstation's cell. One of the first things that *log* has to do is find out exactly which cell it's running in:

```
  ...
static char lclCellID[100] = { '\0'    };
int rc;
...
 /*
  * Get our local cell's name and copy it into the local cellID buffer.
  */
   rc = GetLocalCellName();
   if (rc != CCONF_SUCCESS)
       fprintf(stderr, "%s: Can't get local cell name!  Using '%s'\n",
               rn, LclCellName);
   strcpy(lclCellID, LclCellName);
```

After all the user's information is gathered, it is passed to the chosen cell's *AuthServers*, which generate and return tokens representing that user:

```
   ...
struct passwd pwent;
struct passwd *pw = &pwent;
static char passwd[100] = { '\0' };
SecretToken sToken;
ClearToken cToken;
static char cellID[100] = { '\0' };
...
 /*
  * Get the corresponding set of tokens from an AuthServer.
  */
   if ((rc = U_CellAuthenticate(pw->pw_name, passwd, cellID, &cToken,
&sToken))
               != AUTH_SUCCESS) {
       fprintf(stderr, "Invalid login.\n");
       exit(rc);
 }
```

Once the tokens are successfully in hand, *log* must pass them on to the *CacheManager*:

```
   ...
int setPrimary = 0;
```

```
...
/*
 * Give the tokens to the Cache Manager, along with the cell they're good for
 * and how we want them treated as far as primary identity goes.
 */
  if (U_CellSetLocalTokens(0, &cToken, &sToken, cellID, setPrimary))
      fprintf(stderr,
              "Local login only; couldn't contact CacheManager.\n");
```

Should the user's primary identity have changed as a result of this call, *log* is careful to warn the user of this fact:

```
  ...
static char origPrimaryCell[100] = { '\0' };
...
if (setPrimary != 0)
  if (strcmp(origPrimaryCell, cellID) != 0) {
    fprintf(stderr, "\n*** WARNING ***\n");
    fprintf(stderr, "Your primary identity is now in the '%s' cell\n",
            cellID);
    fprintf(stderr, "instead of the '%s' cell!  AMS programs\n",
            origPrimaryCell);
    fprintf(stderr,  "such  as  Messages  may  fail  in  unpredictable
ways!!\n\n");
}
```

9. Communication Between Cell Administrators

It is expected that the set of cell adminstrators will maintain open communication channels on which they will discuss matters of importance to the Cellular Andrew community. This section outlines some topics of common interest. It also underlines the importance of careful and timely discussions regarding changes visible to the community at large and lists the actual mechansims in place for carrying out such exchanges of information.

9.1. The Importance of Synchrony

In the diverse community promised by a cooperative of autonomous sites, it is very easy for incompatibilites to creep in. One cell may change the semantics provided by its *CacheManager* without advertising that fact, another might add a feature to its version of the *log* program which changes the way primary identities are handled without advising the cell administrators at large, and so on. Because of these changes, software that had been working quite well across cells might suddenly begin misbehaving in unusual ways, with the symptoms appearing in places far away from the actual culprit. At best, this would cause an inconvenience to a few users and at worst it might scramble or sever an important community service.

It is important for site administrators to recognize the danger of such unilateral, unannounced acts. Furthermore, they must resist the implementation of such changes before their impact can be assessed by the rest of the community and a transition plan can be worked out by all. It is important to establish recognized and appropriate forums for discussion of such changes, and Section 9.3 reveals the mechanisms currently in place.

9.2. Topics of Community Interest

There are two basic categories of externally visible changes. The first category comprises all upgrades to the services provided by a particular cell. Such upgrades include the introduction of new volume location/authentication servers, the installation of additional printers along with their associated directories and the implementation of a White Pages facility for the cell. The common thread to these items is that they are non-disruptive. Since only new services are being provided, other cells may safely be informed of the changes after the fact. Existing services are in no way affected by these changes.

This leads to the second category of externally-visible changes, namely those that reduce the existing services provided by a cell, or those that propose to alter the cellular conventions established through this document. In the reduction-of-service case, it is important to announce such events well in advance, as other sites may have to make alternate arrangements. In the specific case of reassigning a *FileServer* machine to other duty, it is also in that cell's best interest to publicize the change ahead of time. Otherwise, workstations in other cells will continue to direct their volume location and authentication requests to it. This unwanted network load could possibly confuse whatever new software is running, and will definitely slow that

machine down. Proposals to change the cellular conventions must be duly considered by all the members of the community, and a consensus must be reached on both the change itself and a viable transition plan.

The following is a (necessarily) incomplete list of events or plans that should be made known to the Cellular Andrew community, either before or after the fact, following the above guidelines. Actual mechanisms for communication are discussed in Section 9.3.

- Addition of a cellular service: One or more *FileServers* handling volume location and authentication requests, new printers, White Pages support, and AMS support are examples.

- Removal of one or more of the above services.

- Addition of machine types and new versions of operating systems, along with technical information as to their operation, relation to existing systems, and associated *@sys* string (see Section 3.3).

- News concerning the site's network availability, including problems with or upgrades to routers, the state of communication lines, and so on.

- Announcements of new cells, along with pertinent configuration information.

- Changes, either proposed or actual, to system support programs and/or important applications (e.g., *Messages*).

- Scheduled downtimes for servers.

- Announcements of new software that, although not directly connected with the cellular architecture, may be of general interest.

9.3. Available Mechanisms

There are several communication channels in place for the smooth flow of information between cell administrators. First, the *AFS-Administrators* mailing list simplifies the sending of mail concerning matters of importance to all sites. To be added to this list, a prospective member need only direct his request to *AFS-Administrators-Request@andrew.cmu.edu*. For those people with direct access to the electronic bulletin boards managed by the `andrew.cmu.edu` cell, two relevant forums are `org.itc.cells` and `org.itc.afs`. The former is used not only to report developments concerning the cellular system, but also as a vehicle for discussion on these matters. The latter bulletin board is used for news and discussion of events dealing with the AFS itself. Finally, administrators from those cells on or near the CMU campus gather at the ITC the first Thursday of each month. These face-to-face encounters have proven useful for more detailed discussions concerning cellular issues.

Making proposals and documents available to the cell administrators or the cell community as a whole is greatly simplified because of the great connectivity offered by the AFS. The existence and location of such a document may be announced through the mailing list or bulletin boards described above. Interested parties may simply pull these files into their editors directly and read them at their leisure. Thus, even highly-formatted documents are easily accessible through the shared file space.

10. Future Plans

There are several fronts along which further improvements can be made to the Cellular Andrew system as it stands today. This section looks at the ITC File System Group's future plans regarding four of these areas.

10.1. Delegation of Responsibility

Some cells may wish to trade a portion of their autonomy in return for certain services which it cannot or chooses not to provide for itself. There has already been a good deal of work done by the AMS group to allow each site to select the level of Andrew delivery system support it wishes to buy into, as described in Section 5. In spite of this internal flexibility, though, it is still impossible to delegate this task to another cell. Another specific case brought to our attention is volume backup. Smaller sites may not have the proper staging and tape backup equipment necessary to save and restore its own volumes. The current implementation makes it somewhat awkward to delegate this (or any other) responsibility to a foreign cell. While remote sites may access files in other cells, they do not have the right to perform volume operations there directly. The general question of whether such piecemeal control over cellular autonomy is feasible in the proposed system must be studied further. If methods are found to accomplish this goal within the established framework, they may then be applied to the stated backup and mail delivery problems.

10.2. Global Authentication

It would be extremely useful to be able to support access lists that include user names from other cells. For example, one author could have the following access list for his `andrew.cmu.edu` home directory, */afs/andrew.cmu.edu/usr8/erz*:

```
% fs la ~
Normal rights:
System:AnyUser rl
erz rlidwka
vanryzin@cs.cmu.edu rl
```

Notice that this access list would allow David VanRyzin's account (`vanryzin`) in the `cs.cmu.edu` cell to have read and lookup rights in this directory, which lives in `andrew.cmu.edu`. This implies that there is some form of shared secret between the *AuthServers* running in the `andrew.cmu.edu` and `cs.cmu.edu` cells, allowing the tokens generated by these two sites to be understood (or at least verified) in both places. There is no proposal currently on the table for implementing this desirable feature, but further investigation is envisioned.

10.3. AFS Upgrades

10.3.1. Planned Improvements

Work is currently underway to replace the existing second-generation AFS with a much more powerful system that is also easier to operate from an administrative point of view. One of the improvements being developed that will have direct and crucial impact on the Cellular Andrew community is a new RPC facility. This new base communication layer will be able to adapt itself to the speed of the underlying transport medium. The current RPC's timeouts are tuned to the performance characteristics of a local area network. Thus, it does not allow geographically distant sites to join in the existing Andrew community, since connections typically time out before a remote procedure call can complete. With a fully adaptive protocol, new Andrew cells are free to spring up anywhere and easily participate in the global file system regardless of the speed of their interconnections.

Although connectivity regardless of location is guaranteed by the above upgrade, it is still not pleasant to wait for file transfers conducted by the *CacheManager* to complete across relatively slow links. There are two features scheduled to appear in the next-generation AFS that will reduce these transfer times, both actual and perceived. First, the kernelized *CacheManager* will perform read-ahead for its clients. Basically, this means that a file *open()* in Andrew will complete upon the arrival of the first block in the file, with the rest of the fetch occurring (still at high priority) in the background. A process performing a Unix *read()* on a region of the file will block iff that portion is still outstanding. Thus, a user program can begin using its data as soon as it arrives at the workstation, and doesn't have to wait until the entire file transfer completes before it can read its first byte after an *open()*. This feature will lower the perceived data transfer times (it still takes just as long to transfer the file, but early data is available almost immediately) and is expected to increase throughput. The second relevant feature is the projected ability of the kernelized *CacheManager* to perform *partial file transfers* when necessary or desirable. Instead of insisting that whole-file transfers always be used, only chunks of files will be shipped over to the workstation in certain cases, driven by the access pattern there. This strategy is typically used for very large files (i.e. those that couldn't fit on the local disk, even if the cache were empty) or when there isn't enough free space to cache a full copy of a smaller file and performing the regular cache replacement strategy is considered non-optimal. In many cases, partial file transfers will result in a net decrease in the amount of data that must travel across the slow link. Thus, this mechanism may decrease both perceived and actual file transfer times. Read-ahead and partial file transfers should actually work together to improve performance past what either alone could accomplish.

There is also another project being developed by the ITC File System Group to allow access times more typical of local networks even across relatively narrow communication channels. A site may install one or more *CachingServers*, which accumulate copies of distant files referenced by its own workstations. The first access for a file not held by the *CachingServer* will incur the full cost of transfer over the slow communication channel. However, once located in the cell's *CachingServer*,

further access by the same or different workstations in the same cell will result in local-network transfer speeds as long as that cached copy remains current. This also has the added benefit in that it produces a net reduction of callback state across cells. With accesses performed through *CachingServers*, a *FileServer* in the source cell will typically only have to keep callback state for a single "workstation" in the destination cell, namely the *CachingServer* itself, instead of each individual workstation in that remote cell holding a copy of a given file. Thus, a cascading system of callbacks will be in effect for these cases.

10.3.2. Desired Yet Unplanned Features

There are some AFS features that appear useful, but are currently not being seriously considered. The first of these is a mechanism by which an access list check can be performed in a uniform way between cells. That is, we wish to allow user X can ask its *CacheManager* what permissions user Y has on directory D. This could be done directly, or by simply allowing user X to ask whether user Y is in group G, or by simply allowing user X to enumerate the contents of group G. In a related fashion, it might be possible to allow some user X (or an unauthenticated connection) can ask server A to identify the machines with which it has connections authenticated as user Y, possibly only if user X has, say, "read" permissions on a particular directory stored on server A (e.g. "*~Y/.FingeringOK*").

10.4. Use of the Internet Domain System

As mentioned earlier, it is feasible to replace the use of the (relatively static) `root.afs` volume in Cellular Andrew with a lookup into the domain system in order to access the individual cells' file systems. The advantage to this mechanism is that addition or deletion of a cell can be done very simply, without requiring each site to make changes to its root volume and then release it. As soon as a cell name (and information about its associated volume location and authentication servers, of course) is placed into the domain database, references to this cell's files will cause this subtree to immediately appear in that cell's */afs* "directory" as maintained by that workstation's *CacheManager*. There is also a significant disadvantage to this approach. Using the domain system for this purpose would make it impossible to enumerate fully the contents of */afs*. Also, it is more than a little hard to explain to naive (even experienced?) Unix users that there are (potentially) many more things in */afs* than are shown by *ls*. Consider the following example, which assumes a reduced cell membership and the use of a domain-based system. The user has previously used the path names */afs/andrew.cmu.edu/usr8/erz* and */afs/cs.cmu.edu/user/erz* successfully. After consulting with the domain system, that person's *CacheManager* has learned all it needs to about these two cells - but *only* about these two cells. Observe the following exchange:

```
% ls /afs
andrew.cmu.edu/
cs.cmu.edu/
% ls /afs/beta.andrew.cmu.edu
```

```
bin/     lib/                    vmunix.itcmin*
boot*    usr/                    vmunix.itcserv*
dev/     vmunix* vmunix.org*
etc/     vmunix.afs*
kadb*    vmunix.itcall*
% ls /afs
andrew.cmu.edu/
beta.andrew.cmu.edu/
cs.cmu.edu/
```

No explicit *mkdir /afs/beta.andrew.cmu.edu* has been performed, and common sense (and/or accumulated Unix experience) tells you that the second command above will fail. Yet not only doesn't it fail, but a new directory has appeared behind the user's back. Thus, the plan to explore the use of the domain system may result in increased flexibility at the expense of some understandability.

11. Conclusions

The existing Cellular Andrew environment allows any number of sites to cooperate in providing a unified file name space without sacrificing administrative autonomy. The fact that each cell manages its own protection and volume databases in an exclusive fashion minimizes the network communication between cells, compartmentalizes errors and allows an arbitrary number of cells to enter the community without causing undue local disturbance. Each process in the cellular system can still be properly identified, and automatic rights reductions assure the community's integrity. The allure and convenience of transparent access to files across protection boundaries has already been felt by the users in existing sites. With the planned improvements to the AFS and hence the prospect of geographically distant sites joining the community, Cellular Andrew offers to integrate a very large collection of users into a single, cohesive file system and computing environment.

Appendix 1: Glossary

[All terms appearing in bold are themselves key words or phrases described in this glossary.]

additive authentication
The ability of the Andrew *CacheManager* to hold several sets of authentication tokens for any given user. Thus, the user may take on different identities at the same time, but only one per cell.

AFS
The *Andrew File System*, sometimes referred to by its old name, *Vice*. This is the distributed file system developed by the Information Technology Center (*ITC*), closely resembling Unix, to serve a large (~7,000) community of workstations belonging to students, faculty and staff at Carnegie Mellon University. Files are managed by a set of *FileServer* machines, with the workstation local disk serving strictly as a cache. Currently, only whole-file transfer is used between *FileServers* and client workstations. Each machine in the community sees exactly the same file space. A distinguishing AFS performance optimization is its use of ***callbacks***, or promises from the *FileServers* to inform individual workstations when their cached files are no longer current. Thus, workstations can operate on their cached file copies freely and without contacting the *FileServers* as long as the callback promises are intact.

AMS
The *Andrew Message System* is a portable, distributed mechanism for creating, viewing and manipulating multi-media mail and bulletin board posts in an integrated fashion. The AMS architecture permits access to the message database exclusively through a server process, so a wide variety of interfaces are possible. Thus, low-end "glass-tty" machines as well as powerful workstations with bit-mapped screens can interact with the message database.

AuthServer
A server program running on a secure machine which provides authentication services for the associated cell. Given a user's account name and encrypted password, the *AuthServer* generates and returns clear and secret tokens representing that user's identity. These tokens are held by the ***CacheManager*** and used to verify his identity when performing file operations. Exactly one *AuthServer* is allowed to accept and process requests to change passwords. This particular *AuthServer* process typically runs on the **SCM**, if one exists.

CacheManager
The Andrew *CacheManager* is either a user-level process or a kernelized system acting as a user's representative and intermediary when he performs file accesses in the **AFS**. File images are copied to the workstation local disk by the *CacheManager* when they are accessed for the first time or when the already cached copy is stale. All AFS file operations actually occur on these cached copies. The **callback** mechanism is used to keep workstation caches synchronized with the state of the file images in the central collection as maintained by the ***FileServer*** processes.

CachingServer
A server process which intercepts **AFS** requests destined for external cells and services these requests when possible from its own local cache. This facility is part of the future plans for Cellular Andrew development. See Section 10.3.1 for a full discussion.

callback
Mechanism to maintain cache consistency on Andrew workstations. See the entry for ***AFS***.

cell
An Andrew installation, running its own distinct *FileServer* and *AuthServer* machines, that cooperates with other such sites in the generation and maintenance of a common, shared file space. A cell has full control over such administrative functions as ***volume*** assignment and backup, creation and deletion of user accounts, and management of printing facilities. Details of cellular operation are legislated by the contents of this document.

cell mount point
This is an extension of the standard Andrew ***mount point***, which "glues" together ***volumes*** to form the file space visible from a workstation. A cell mount point contains not only the name of the volume to be attached, but also the name of the cell in which that volume resides. This mechanism allows externally-managed subtrees to be mounted transparently in a workstation's file space.

clear token
The unencoded version of a data structure generated by a **cell**'s *AuthServers*, used to reliably determine a user's true identity. Since they are not encoded, clear tokens are never exchanged on the network. See **secret token** for a description of its counterpart.

FileServer
This server process runs on a restricted machine and is responsible for maintaining and exporting a set of **volumes** containing files in the **AFS**. A *FileServer* receives requests to store, read, and get status on its files, determines if the caller has the authority to perform such operations, and ships out and receives file images and/or status information to and from qualified parties. It also maintains **callback** information on all such copies cached on workstations. It revokes callbacks to those file images that become outdated. Certain miscellaneous services are also provided, such as time of

day. *FileServers* regularly report the set of volumes they manage to the **SCM**, if one exists. They also respond to validated requests to move, destroy, or create read-only replicas of their volumes.

mount point
See ***cell mount point*** for a description of the cellular extension of this facility.

PAG
A Process Authentication Group, used to differentiate between different "families" of users for authentication purposes. Each user process belongs to exactly one PAG, with which its set of tokens is associated.

pioctl()
The "path *ioctl()*" interface operations exported by the ***CacheManager***. See Section 8.1.1 for a full description of those *pioctl()*s related to cellular activities.

primary identity
The identity (set of tokens) marked as "special" in the ***CacheManager***. The primary identity is only used by such facilities as **AMS** to select among the identities created by a given user, for such purposes as the proper selection of directory from which to read mail.

SCM
The **System Control Machine**, responsible for keeping all ***FileServers*** coordinated. It regularly collects the list of volumes each *FileServer* manages, creating a single unified database with volume location information, the **VLDB**. This VLDB is then distributed to the subset of the *FileServers* that has advertised itself as providing volume location service. The SCM also performs various other housekeeping functions, such as redistribution of *FileServer* and ***AuthServer*** binaries.

secret token
The encoded version of a data structure generated by a **cell**'s ***AuthServers***, used to reliably determine a user's true identity. Since they are encrypted, secret tokens are occasionally passed on the network as required by the underlying RPC protocol. See **clear token** for a description of its counterpart.

ViceID
An integer value, similar to the Unix uid, used to identify an Andrew user for authentication purposes.

VLDB
Volume Location Database. See the entry for **SCM**.

volume
A container for a heirarchy of files, conforming to a Unix subtree. These are the units of storage, relocation, and backup implemented by the ***FileServers***. Volumes are "glued" together into a single coherent Unix rooted tree via **mount points** and **cell**

mount points.

White Pages

A superset of the information kept in the password file. The White Pages is intimately involved with the **AMS** facility. See Section 5 for details on the use of the White Pages. Also, see Appendix 2 for a description of the format for this optional yet important database.

Appendix 2: White Pages Format

The optional White Pages facility requires that its database be structured as a B-tree in the conventional format (generated by the *makeboth* program, located in */usr/andrew/etc* in the `andrew.cmu.edu` cell, for instance). Entries in the White Pages are drawn largely from the */etc/passwd*-format file given as source to *makeboth*. The possible fields in each entry of the White Pages are as follows:

N Full name; initialized from `pw_gecos`.
Tk Parts of that full name; generated from the `N` and `WN` fields.
WN A sequence of alternate full names.
ID The Unix login id; initialized from `pw_name`.
EK A mask indicating the source of the information; the "1" bit means the */etc/passwd*-format file.
NI The ViceID, if any; initialized from `pw_uid`.
GI The group ID; initialized from `pw_gid`.
PW A password; initialized from `pw_passwd`.
HD The distinguished home directory; initialized from `pw_dir`.
Sh The login shell; initialized from `pw_shell`.
Af Any recognizable "affiliation"; initialized from components of the `pw_dir` field.
Fwd The account's forwarding address, or the string "`**unknown**`"; initialized as the contents of the *HD/.forward* file at *makeboth* time..
DK "Delivery Kind:" At this juncture, one of "`DIST`" or "`NNTP`".
DP "Delivery Parameter:" Extra information needed by the given `DK` field. At present, only "`DIST`" needs a parameter, providing the name of the distribution list file.
D Phonetically-canonicalized surnames; generated automatically.
X Phonetically-canonicalized parts of names; generated automatically.
SI A sequence number for entries that have no `NI` field.

The value of any field in any entry may be overridden in the *passwd.chg* file, which is a sequence of lines containing five colon-separated fields: the `ID` field to which it applies, the name of the field to change, the old value of that field, the new value of that field, and an integer by which multiple modifications may be sequenced. Thus, to change the spelling of the name *Craig Everhart*, an entry could be:

```
everhart:N:Craig Everhardt:Craig Everhart:0
```

To add Craig's name as a `WN` field, add the *White Pages Mania* alias for him, and forward his mail to the Postmaster, the White Pages entries could be:

```
everhart:WN:Craig Everhart:Craig Fulmer Everhart;White Pages
Mania:0
everhart:Fwd:+ :postman++:0
```

(There are three things to note here. First, the "old value" for the WN *field is just the value of the* N *field. Second, a sequence of full names is separated by semicolons (but not additional spaces). Third, "+" is the quoting character for the field separator; a null entry is specified as plus-space ("+ "), a single plus character by plus-plus ("++"), and an embedded colon by plus-equals ("+=").)*

The *wp.add* file specifies complete new entries to be added to the White Pages. Generally, these additional entries are the only ones that use the DK, DP, and SI fields. Each such entry must contain a unique SI field.

There are other entries in the White Pages, as well, that assist the phonetic name matcher. These entries are generated from the names/nickmap and names/override files in the *makeboth* procedure.

Conventionally, the AMS assumes that it can deliver mail using the following algorithm. If the White Pages entry for a user can be found, the AMS looks at several fields of that entry. If the DK field exists and is the text "NNTP", the message is sent as a netnews submission in the conventional fashion. If the DK field exists and is the text "DIST", the DP field is used as the name of a distribution list. Other values for DK cause error messages to be generated. If no DK field exists but a Fwd field exists, that field value is used as an address list to which mail for that user is to be sent *if* that field value is not the text string "**unknown**". When "**unknown**" is encountered, the AMS looks for a HD (home directory) field, appends the string "/.forward" to it, and looks for a file with that name. If the AMS can read that file, its contents are used as an address list to which mail for that user is to be sent. If the AMS can determine that the file does not exist, or if the Fwd field itself doesn't exist, the AMS looks for a HD field, appends the string "/Mailbox" to that field value, and delivers mail for that user by inserting a file into the directory named by the result after assuring itself that the result is a directory in the AFS.

Bibliography

[1]
An Overview of the Andrew File System
John H. Howard
Information Technology Center
Carnegie Mellon University
In 1988 Dallas Usenix (Also available as a CMU technical report: CMU-ITC-62)

[2]
Synchronization and Caching Issues in the Andrew File System
Michael Leon Kazar
Information Technology Center, Carnegie Mellon University
In 1988 Dallas Usenix (Also available as a CMU technical report: CMU-ITC-063)

[3]
A Multi-Media Message System for Andrew
Nathaniel Borenstein, Craig Everhart, Jonathan Rosenberg and Adam Stoller
Information Technology Center, Carnegie Mellon University
In 1988 Dallas Usenix (Also available as a CMU technical report: CMU-ITC-64)

[4]
Scale and Performance in a Distributed File System
John Howard, David Nichols, M. Satyanarayanan, Bob Sidebotham, Mike West
Information Technology Center, Carnegie Mellon University
In ACM Transactions on Computer Systems, 6(1), February 1988, pp. 51-81.

[5]
An Overview of the Andrew Message System
Jonathan Rosenberg, Craig F. Everhart and Nathaniel S. Borenstein
Information Technology Center, Carnegie Mellon University
In SIGCOMM '87 Workshop: Frontiers in Computer Communications Technology
Stowe, VT, 11-13 August 1987.

[6]
Printing in the Andrew Environment
Information Technology Center, Carnegie Mellon University
[Internal document, in preparation]

[7]
Building and Maintaining the Andrew White Pages Facility
Craig F. Everhart
Information Technology Center, Carnegie Mellon University
[Internal document, in preparation]

[8]
The Andrew File System Interface
Information Technology Center, Carnegie Mellon University
[Internal document, in preparation]

[9]
Rx: The Andrew RPC Protocol
Bob Sidebotham
Information Technology Center, Carnegie Mellon University
[Internal document, in preparation]