VM/ESA CMS Shared File System

by R. L. Stone
T. S. Nettleship
J. Curtiss

Discussed is work toward satisfying requirements on the Conversational Monitor System (CMS) in the areas of data sharing and physical DASD space sharing. This work advances the present CMS file system design that allows only active read sharing among users on a single VM system, where each user has a reserved, private allocation of DASD space for file data. Described in this paper is the CMS Shared File System (SFS), which was designed to satisfy the data sharing and physical DASD space sharing requirements by providing a pool of DASD space that is shared among multiple users. DASD space assigned to the pool is easily extended, and read/write sharing of individual files is allowed. Also discussed is SFS security, usage of Virtual Machine/Enterprise Systems Architecture™ (VM/ESA™) data spaces for single system performance, and coordinated resource recovery to provide file data integrity in the distributed environment.

This paper discusses the Conversational Monitor System (CMS) Shared File System (SFS), which was introduced in VM/System Product (VM/SP) Release 6 and has been enhanced in Virtual Machine/Enterprise Systems Architecture™ (VM/ESA™). The CMS SFS was implemented in order to satisfy long-standing VM requirements in the areas of file sharing, space sharing, security, data addressing, transparent remote access, and data integrity. We first present the traditional CMS file system, known as the minidisk file system, which preceded SFS and today coexists with SFS. The focus is on the portions of the design of the minidisk file system that motivated the work that led to the SFS. The following sections discuss the capabilities and advantages of SFS.

Some of the design decisions that were made in order to have SFS as compatible as possible with the minidisk file system are then described. A major goal of SFS is to have applications that were written for minidisks run successfully when the data are moved to SFS. Reasons why applications may need to change are included. Finally, an example of an application that uses SFS is described.

Background

The CMS file system consists of two portions: CMS minidisks and SFS. A minidisk is a contiguous allocation of a real DASD volume and is owned by a VM user. Because a minidisk can be a subset of a real volume, there can be multiple minidisks on a real volume. VM users generally have one or more of these minidisks on which to store their data in the form of CMS format files. The minidisk portion of the CMS file system is for high-performance CMS file access; fuller function is provided by SFS. Therefore, the placement of any particular file depends on the characteristics of the usage of that file. The rest of this section describes the design of the minidisk file system that led to the requirements addressd by SFS.

The CMS minidisk file system is a flat file system in that there is only one directory per CMS

©**Copyright** 1991 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

minidisk, which contains all file names on that minidisk. There is no subdirectory support, and files on the minidisk are owned by the minidisk owner.

CMS minidisk files and space may be shared, but only with care. The normal situation is that each CMS user has a private minidisk that is not shared with other users, which means that each user has a reserved allocation of DASD space. Unused space on the minidisk is unavailable to other users, even for temporary files, which can lead to inefficiency in space utilization. If a user outgrows an allocation, a system programmer must create a new minidisk on a real volume, allocate the minidisk to that user, copy the user's files to the new minidisk, and remove the old minidisk. Such DASD space management can become time-consuming.

The owner of a minidisk may authorize other users to read or write to that minidisk. Authorization is given either by creating and giving to the other users a read or read/write password, or by authorizing the other users through a security product such as the Resource Access Control Facility (RACF). In either case, these authorized users then use the LINK command (specifying the password if that method is used) and the ACCESS command to initialize usage of the minidisk. Note that, with either passwords or RACF, sharing is on a minidisk basis. Thus, if a user has read access to the minidisk, every file on that minidisk can be read. If the user has write access, any file on the minidisk can be created, deleted, or modified. There are exceptions to these rules, such as filemode-0 files, but it is generally impossible to grant another user access to a single file or a particular subset of files on the disk or to mix authorizations such that another user can only read file A but can modify file B.

The danger in minidisk sharing is that the ACCESS command reads the minidisk directory into the user's storage. The directory contains the names of each file on the minidisk, its file attributes (e.g., record length), and its location on the minidisk. Having this information in storage is extremely important in achieving the desired performance. CMS uses this directory to find files and free space. Therefore, if the owner creates a new file, other users must ACCESS the disk again, which rereads the directory, in order to see the new file. If the owner erases a file, other users—unless they ACCESS again—are apt to think that the file still exists. Then, if the owner creates a new file that uses the same space as the erased file, those other users may see parts or all of the new file. If multiple users are sharing the minidisk in write mode, each user's CMS acts to control the allocation of space based on its in-storage directory and allocation map.

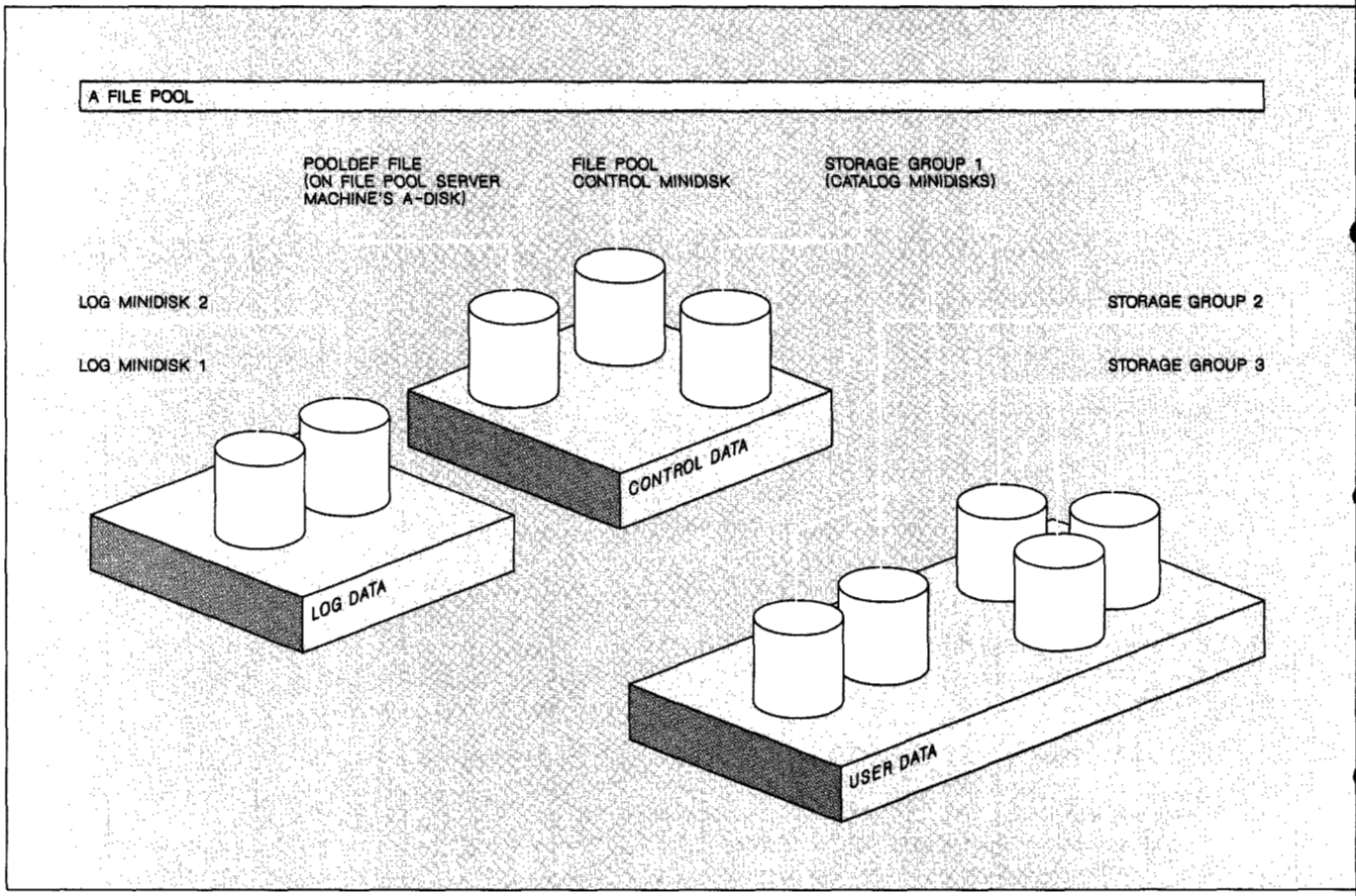
Another area in which the minidisk file system could be improved is that of data integrity. CMS commits the minidisk (or makes all changes to the minidisk files permanent) when the last file that was open for write on the minidisk is closed. It does this by writing the minidisk directory and allocation information to the minidisk. If a failure occurs before this is done, the minidisk contains the data as they were after the previous commit. This has two implications: (1) If one program, with a file open for write, calls a second program that writes data and closes its files, the data are not committed until the first program closes its file. (2) If the second program were to close all files on that minidisk in order to ensure that its data are committed, the first program's data are also committed. There is only one scope of commit on a minidisk, and that is the entire minidisk. There is no separation between programs that call one another.

CMS Shared File System server structure

The desire to extend the capabilities of the CMS minidisk file system led to the design of a file server through which one user—in this case the Shared File System—owns the collection of minidisks and manages both the physical placement and user access to the data on them. Users come to the SFS server with file requests across an advanced program-to-program communication (APPC/VM) connection.

The collection of minidisks—which are owned and managed by the SFS virtual machine—is known as a *file pool*, and the SFS virtual machine is called an *SFS file pool server machine*. The file pool contains files for a number of users who are enrolled in the file pool. To manage these user files, each file pool has a certain number of files and minidisks devoted to control information. For example, the POOLDEF file contains information necessary to locate all the minidisks in the file pool. Every file pool needs a control minidisk that keeps track of all data in the file pool. The control disk contains a map of all the blocks in the file

Figure 1 File pool structure



pool. (Each minidisk is divided into sections called *blocks*, and each block is 4K bytes long.) The map tells the file pool server machine which blocks are being used and which are not.

The last item of the control data is the catalog storage group, or as it is often called, storage group 1. The catalogs contain information about files and directories that exist in the file pool, who owns them, and who is authorized to read and write them.

To help protect the integrity of the control data as well as the user data, a server maintains a log. Two copies of the log are kept on separate minidisks. In the logs, the server records changes to the file pool so that if the system fails in the middle of an operation, the file pool is not corrupted. A log is also needed so that applications have the ability to roll back and undo changes. It is recommended that the log minidisks reside on separate volumes for reasons of performance and integrity.

The remainder of the space in the file pool is for user data. To allow control over which users have files on which minidisks, the server uses a concept called *storage groups*. A storage group is a collection of minidisks within the file pool. Each storage group is identified by a number ranging from 1 to 32 767. Storage group 1 is for catalog information, and storage groups 2 through 32 767 are for user data.

For a person to use a file pool, the person must be enrolled in it. When a person is enrolled, that

person can be assigned an allocation of space in a particular storage group, and files can be created using that space. All the person's files reside on one or more of the minidisks assigned to the storage group. The user's allocation of space is referred to as the user's file space. The file spaces of many users can be assigned to a single storage group. Each user can have, at most, one file space in a given file pool. There are three kinds of data stored in a file pool, as shown in Figure 1.

Users can now group files into a directory structure for quicker access and ease of use.

A file space is a logical allocation; that is, no physical DASD space within the storage group is set aside for the user. The file pool server allocates physical disk space as needed until the user reaches the limit of the allocated file space.

The server tries to allocate blocks evenly from all physical DASD volumes in the storage group; that is, the server uses a few blocks on one volume, a few on the next, and so on. This helps to balance the input or output activity to the volumes in the group. Multiple minidisks on a single volume are not balanced, but are filled sequentially. Thus the blocks of a user's file might be scattered among several DASD volumes. There is no way for a user to determine where in the storage group the blocks of a particular file reside.

Data addressing

The CMS minidisk file system is basically a *flat file* system. A user can organize the files only at the minidisk level using user-defined file names and file types to establish any file structures. The Shared File System allows users to organize files within hierarchical directories. Users can now group files into a directory structure for quicker access and ease of use. Hierarchical directories allow the user to organize files in smaller numbers than on an entire minidisk.

SFS directory names are as follows:

- Up to eight-character file pool name followed by a colon (e.g., VMSYSU:)
- Up to eight-character user ID, corresponding to the root directory
- Up to eight-subdirectory names from 1–16 characters, each separated by a period

The file pool name and user ID may be defaulted to only a period in many cases. For example, each of the following is a valid way of referencing the same directory:

- VMSYSU:USERA.FIRSTSUBDIR.SECONDSUBDIR
- USERA.FIRSTSUBDIR.SECONDSUBDIR
- .FIRSTSUBDIR.SECONDSUBDIR

In order to use the default methods, the SET FILEPOOL command must be used. It can be added to a user or system profile; or the default file pool name can be set in the IPL statement found in the user's CP directory.

SFS directories may be accessed by using the ACCESS command. This allows SFS directories to be assigned to a CMS file mode letter. Many programs using the CMS file system can use SFS when a directory has a file mode associated with it. The SFS directory is placed in the CMS search order and operates the same as a CMS minidisk. Even though SFS has a hierarchical directory structure, no implicit searching is done within that structure. The CMS file-mode letter search order is still the method used to search for files in CMS. In addition to referencing a file via a CMS file mode letter, a user may also reference a file using the directory name along with the file name. Commands such as ACCESS, RENAME, and ERASE allow the user to reference a file directly, without having to access the directory. New CMS program functions also allow programs (and execs) to directly reference SFS files.

Another new concept introduced with SFS is the use of file aliases. A standard CMS file, that is, a file containing file data, is known within SFS as a *base file*. A *file alias* can be created that is simply another file name that references the base file data. No unique data are associated with a file alias. Aliases are a very powerful tool within SFS. They allow users to organize their own data better. Aliases also allow users to create the means by which to access other users' data. For exam-

ple, if USERB has granted USERA authority to five execs each in separate directories, USERA would ordinarily have to ACCESS all five directories in the CMS search order to have them available at a given time. Using aliases, USERA can create a directory, USERA.USERBEXECS, and create five aliases in that directory that point to USERB's five execs. USERA can then ACCESS USERA.USERBEXECS at one file mode letter, and have all the execs available.

To allow greatest concurrency, SFS acquires locks only when needed.

CMS file sharing

The Shared File System provides CMS users with the ability to share files and file data without having to know when the data were changed or having to reaccess the owner's files. Files are shared at the file level. That is, there are no conflicts when users are referencing different files. Also, multiple users can use the same file concurrently.

Multiple readers and one writer may access a file at the same time. When a user opens a file with the intent of reading it, that user obtains a consistent view of the file, even if another user is writing data to that file. In this way, the user is not given a partially updated file. This is of particular importance when considering users who are sharing a program such as an exec. Users should not be executing partially updated copies of execs.

File updates are not seen by readers of a file until the updates are complete and the writer has committed those updates to the file system. After the file system has committed these updates, future readers of the file then see the new—that is, the now-current—version. However, if any users opened the file before the updates were committed, they continue to see the previous version of the file. In order to see any new updates, users must close and then reopen the file. This differs from the use of the minidisk file system, which requires a user to reaccess an entire minidisk to see the file updates.

Once you have given many users authority to access your SFS files, you need a way to avoid concurrent updates to those files. In SFS, this is done by placing locks on a specified file or directory. There are two categories of locks, implicit and explicit. An *implicit lock* is one that SFS acquires and releases automatically. To allow greatest concurrency, SFS acquires locks only when needed and frees them as soon as possible after they are no longer needed. An implicit lock may be acquired when a file is opened. The type and duration depend on the intent of the open (read or write) and when the file is closed. This process can prevent other users from updating the file while it is being updated, but it does not prevent other users from reading the file.

CMS provides a way to lock files and directories *explicitly*, to prevent simultaneous updates. When a user is actively working with SFS files and directories, CMS automatically creates and deletes implicit locks. With the CREATE LOCK command, a user can create an explicit lock on a file, a group of files, or a directory. One can also select the type of lock and its duration. There are three types of explicit locks:

- *Share lock* allows authorized users to read the contents, but no one can make any changes.
- *Update lock* allows authorized users to read the contents, but only the lock holder can make changes.
- *Exclusive lock* allows only the lock holder to read and make changes.

There are two lock durations:

- *Session lock* lasts until the end of the CMS session, or until the connection with the server is broken, unless the user specifically deletes it.
- *Lasting lock* lasts until it is specifically deleted.

A QUERY LOCK command displays information about the locks that have been created; a DELETE LOCK command deletes explicit locks.

In CMS, the IBM System Product Editor, XEDIT, allows users to edit SFS files in the same manner as minidisk files. When several users are XEDITing a file, explicit locking is necessary because XEDIT

reads the file into memory, then closes it. Once a file is closed, the SFS implicit lock is released and another user could update the file. The new XEDIT options LOCK and NOLOCK have been added to control this process; and the default is LOCK. XEDIT gets an *update session lock* on the input file that the user is editing. Other users trying to XEDIT the same file can see the file, but receive a warning message saying that the file cannot be updated. In a sharing environment, it would be best to use the NOLOCK option when the user does not intend to update the file. Thus one user does not prevent another from updating the file.

File security

The Shared File System gives users the ability to assign authorities to individual files. One user thus gives another user authority to read or write a file by using the GRANT AUTHORITY command. The authority is controlled by the owner of the file, and the owner can grant or revoke another user's authority to any file or group of files. The following are authorities for SFS files:

- *Read.* A user with read authority on a file may only read that file.
- *Write.* A user with write authority may read and write the file.

When granting authority, one user can give authority to another user, a group of users (using nicknames), or to all users that can connect to the file pool. For the last case, the original user grants authority to PUBLIC, thus giving all users the ability to read or write the file. Like files, the owner of the directory can grant read or write authority to other users. Directory authorities are separate from file authorities. That is, a user who has read authority on the directory, does not necessarily have read authority on all the files in that directory. The directory-level authorities are:

- *Read.* A user with read authority can see the list of files in that directory. Read does not imply that the user can read any of these files. This authority is required to use the ACCESS command to reference an SFS directory.
- *Write.* A user with write authority can add files to that directory. The files, after they have been created in the directory, then belong to the owner of the directory, and the creator of the file is automatically given write authority to the file.
- *New read.* A user with new read authority has read authority to future files in the directory; this, however, does not imply that the user has read authority on the directory or on any existing files in that directory.
- *New write.* A user with new write authority has read and write authority to future files in the directory; this, however, does not imply that the user has write authority on the directory or on any existing files in that directory.

New read and new write give the directory owner the ability to grant authority to directories that are dynamic, without individually granting authorities when each new file is added.

With these file and directory authorities, a user can give access to a file or group of files to a single user or a group of users. The GRANT command allows the owner to give other users authority to files and directories. If USERA wants to give USERB the ability to use the ACCESS command to get to the .DATAFILES directory, USERA would issue the following command:

```
GRANT AUTH USERA.DATAFILES TO USERB (READ
```

This gives USERB read authority to the directory USERA.DATAFILES. If USERA wants to give USERB read authority to the file USERA DATA in directory USERA.DATAFILES, USERA issues the following command:

```
GRANT AUTH USERA DATA USERA.DATAFILES TO
USERB (READ
```

Now suppose USERA wants all the users in a nickname file TESTDEPT to have write authority on all future files in USERA.DATAFILES. USERA issues the following command:

```
GRANT AUTH USERA.DATAFILES TO TESTDEPT
(NEWWRITE
```

The difference between granting file-level authority and directory-level authority depends on the scope of the data one user wants other users to be able to see. If the granting user wants to give another user the ability to reference a file but not the directory, only file-level authority is appropriate. An example of this is granting a user read authority to execute an exec in a granting user's tools directory. What the receiving user commonly does is to create an alias in the receiving directory to this exec. In this way, the receiver does not need the authority on the directory to

reference the file. If, on the other hand, one user wants another user to be able to execute any exec or program in that directory, the granting user gives read authority on all files in the directory and read authority on the directory. This is an easier way to handle this situation than creating a large number of aliases.

Along with the authorities that are provided with SFS, an external security manager can be used to determine whether a user has access to an SFS file. RACF, for example, can be installed so that when a user tries to access an SFS file, SFS passes the request to RACF to determine whether the user has the correct authority. If the user has the required authority, SFS services the request; if not, an error message is returned to that user.

Types of SFS directories

There are two types of SFS directories. There is a *filecontrol* directory that allows users to grant authorities on individual files. All SFS directories in VM/SP Release 6 are file control directories. The other SFS directory type is a directory-control directory, termed *dircontrol*. This type of directory resembles a CMS minidisk, and it is new with VM/ESA Release 1. There are no individual file-level authorities. If one has read authority on the directory, that user also has read authority on all files and all future files in that directory. Also, only one user can access a *dircontrol* directory read-write at a time. All other users receive read-only access until the user with read-write access releases the directory.

Another attribute of a *dircontrol* directory is that it can be placed in a VM/ESA data space. This function is available in VM/ESA Release 1.1. The SFS administrator can assign this attribute to a *dircontrol* directory. When the directory is accessed, it is put into a data space by the SFS server, unless the requester is on a different system than the SFS server. When a user references a file in the data space, CMS obtains the file data directly from the data space and does not make a request for data from the SFS server. This can result in a sizable performance savings. An SFS directory in a data space can be accessed only in a read-only mode, similar to a shared segment. If a user accesses it as a read-write entity, CMS communicates with the server in the normal way. Because a data space can be shared among multiple virtual machines, a single copy of an SFS directory and its files can be shared among many users. Placing a tools library in a data-space-eligible *dircontrol* directory allows for sharing among users. Such a placement of tools also yields performance benefits because of direct access to the files, because there is no need to communicate with the server machine. It is also possible to take advantage of other SFS functions such as space management and SFS authorizations.

Because multiple users are drawing from the same DASD space, free space is available.

The following authorities have implications on both the directory and the files contained in that directory:

- *Directory read.* A user with directory read authority has the same authority on the directory as if it were a CMS minidisk. This implies that the user has read authority on the directory, all existing files within the directory, and all future files within the directory. This authority may be used only with directory control (*dircontrol*) directories.
- *Directory write.* A user with directory write authority has the same authority on the directory as if it were a CMS minidisk. This implies that the user has read and write authority on the directory, on all existing files within the directory, and on all future files within the directory. This authority may be used only with directory control (*dircontrol*) directories.

These authorities are mutually exclusive with the read and write authorities on filecontrol directories. That is, one cannot grant read authority to a *dircontrol* directory, or directory read authority to a filecontrol directory.

Storage management

In the Shared File System, DASD space is allocated as a collection of CMS minidisks that are owned by the SFS server machine and are not

accessible directly by users. This collection of minidisks is known as the *file pool*. Minidisks allocated to the file pool can be grouped into disjoint groups known as *storage groups*. A storage group is the pool of shared storage defined and set up by an installation to support a specific user group. Because multiple users are drawing from the same pool of DASD space, free (i.e., unused) space is available for use by multiple users, rather than the fixed allocation of space on an individual minidisk. All DASD space in SFS is allocated on demand in units of 4K-byte blocks. Users do not hold unused 4K-byte blocks. Unused blocks are owned by the storage group and thus are available for other users in that storage group.

The SFS system administrator creates these storage groups. The administrator then gives each user in that storage group a logical space allocation known as the user's *file space*. File spaces represent the number of 4K-byte blocks that the user is allowed within the storage group. Actual DASD space is allocated only when the space is needed. The way a file space works can be best illustrated by an example:

- Suppose storage group 2 has a DASD allocation of 10 000 blocks and 10 users.
- Assume that each user has a file space of 2000 blocks. Notice, we can make a logical allocation that is greater than the number of physical blocks in the storage group, because as of yet, no one has used any physical DASD space.
- If USERA creates a file that is 500 blocks long, USERA's available file space is 1500 blocks, and the available storage group space is 9500 blocks.

A user (USERA) can continue until the 2000-block file space limit is reached or until all users together have taken the 10 000 blocks of the storage group. The SFS system administrator can dynamically increase (or decrease) a user's file space and can also add more minidisks to the storage group to increase the physical storage. Either way, this allows an installation much greater flexibility to allocate its system's DASD resources to users.

Files in SFS are not necessarily contiguous. A ten-block-long file may be spread over more than one minidisk within the storage group. SFS keeps track of where the file blocks are and their proper order within the file. This management of 4K-byte blocks allows CMS users to have files that are larger than one minidisk, even larger than one physical volume. The management of DASD storage by the Shared File System may easily be one of the greatest benefits to many installations.

Remote access

Files stored in a file pool are shareable across multiple VM systems. This includes VM systems connected in a Systems Network Architecture (SNA) network and VM systems in a Transparent Services Access Facility (TSAF) collection.

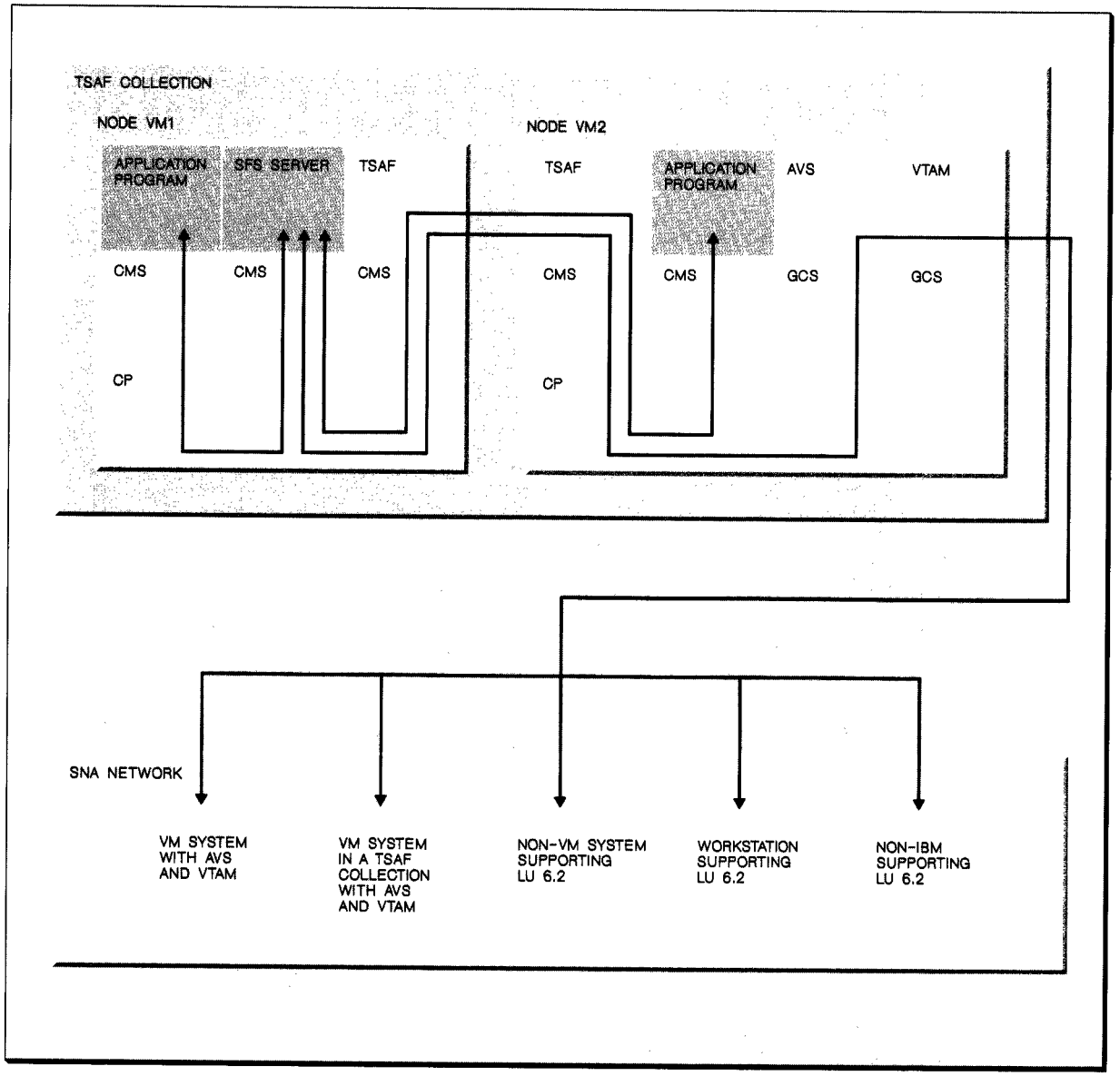
The remote user or application would use the same interfaces and CMS commands that are available for local CMS files. The location of the server being accessed is transparent to the end user, with the exception of performance delays that can be encountered with remote access.

Figure 2 illustrates how an SFS server can be accessed in relation to a TSAF collection and SNA network. The diagram shows two nodes within a TSAF collection. Node VM1 has three virtual machines, a CMS application, an SFS server, and a TSAF virtual machine. Node VM2 also contains a TSAF virtual machine, another CMS application, and the components necessary to communicate to an SNA network. VM SNA communication is discussed in Reference 1. Figure 2 shows that the SFS server located in node VM1 can be accessed by the CMS application within VM1. The SFS server also can be accessed from VM2 through the TSAF virtual machines and from somewhere in the SNA network.

Some administrative activity is required to make the user known on the remote system. For local users and users in the TSAF collection, no special action is required. For users outside the TSAF collection, the remote user must first be defined to the local system using APPC/VM directory services. Once this is done, a user issues the GRANT command as though the remote user were local.

The Shared File System allows users to share and access files through any connected VM system, thereby giving users the ability to operate (e.g., XEDIT) on files on remote VM systems. For example, if a CMS user on node VM2 wants to access files contained on an SFS server on node VM1, the user simply issues the CMS ACCESS command

Figure 2 VM connectivity



specifying the filepool and directory name. TSAF resolves this resource name and routes the request to VM1. This is totally transparent to the user that issued the ACCESS command on VM2.

Data integrity

One intent of the SFS design was to remove the minidisk file system limitation of the minidisk-level commit. In order to do this, SFS initially provided the following application functions:

- Ability to coordinate the commit of changes to a user-defined set of files. This set of files is also known as a commit scope or a CMS work unit within a single-file pool. The term *coordinate* means that either all changes are committed, or if a failure occurs, all changes are discarded.

- Ability to discard (roll back) all uncommitted changes in a CMS work unit
- Ability to have multiple, concurrent, independent work units

  This allows an application the flexibility to control the point at which changes are committed and to separate its changes from those made by other programs it may invoke or which invoke the application.

  CMS provides functions for getting and returning work units and for making a particular work unit the default work unit. The default work unit is used if the application does not specify a work unit at the time of the function call. This default is also used for the FSREAD/FSWRITE interface, which does not allow a work unit specification.

  A work unit is not limited to a single set of files or a single commit. An application may make changes to one set of files and commit them, and then make changes to another set and commit those changes. A work unit may be used for the life of the application. VM/SP Release 6 included the restriction that a commit could not occur until all files in the work unit were closed.

The following improvements[2] have been made in VM/ESA:

- Files need not be closed in order to commit changes. Changes that have been made to files in the work unit are committed, and open files remain open.
- Coordinated Resource Recovery allows the work unit to include multiple file pools.

Shared File System command interfaces

SFS functions are provided through new and enhanced CMS commands. Most existing CMS commands work with SFS files, and some also allow one to reference SFS directories, without accessing them as file modes. Commands that can reference SFS directories now accept a directory ID. This can be in one of the following formats:

- File mode letter of an accessed SFS directory
- An SFS directory name
- A plus (+) modification to an accessed directory name

The latter format uses a file mode letter of a currently accessed directory as a shortcut. For example, if the directory, VMSYSU:USERA.TOOLS were to be accessed at file mode letter B, the directory VMSYSU:USERA.TOOLS.EXECS could be accessed by issuing the command ACCESS + B.EXECS C. The directory accessed at file mode B (USERA.TOOLS) is substituted in the command to form the full directory name. This allows users with extensive directory structures to reduce the amount of typing needed to reference SFS directories. There is also a minus (−) syntax that works similarly to the plus format.

All current CMS commands that use file mode letters will work with accessed SFS directories unless they are specifically related to minidisks (e.g., the FORMAT command). Several CMS commands have been enhanced to also accept a directory ID, as previously described. These are ERASE, RENAME, ACCESS, RELEASE, SET, and QUERY. Both RENAME and ERASE have been modified to work on SFS directories as well as files.

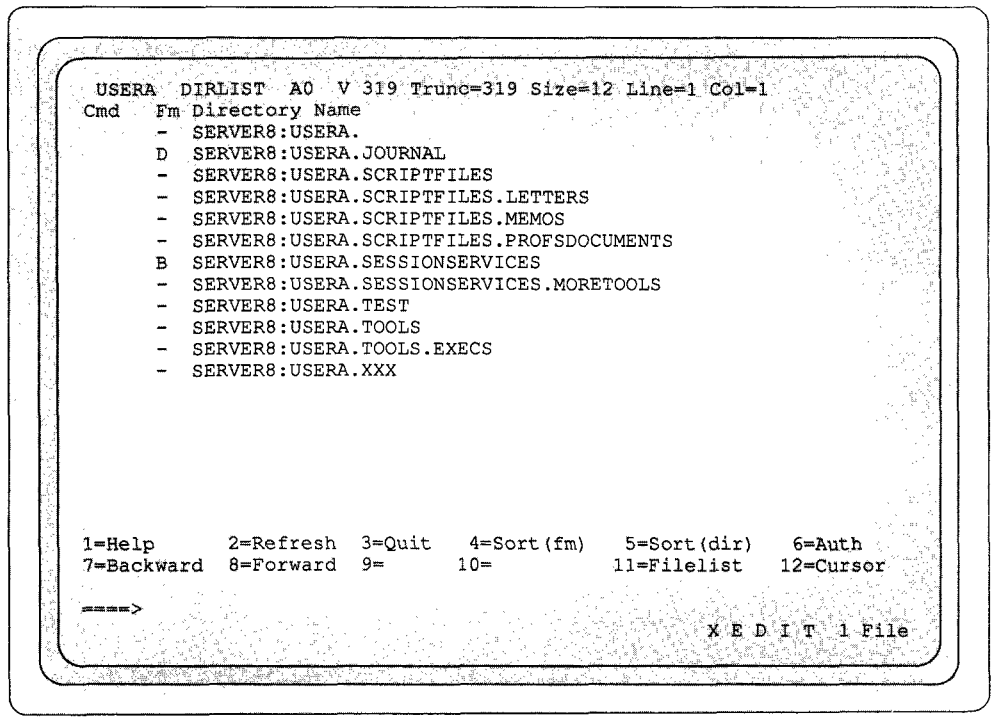
New CMS commands that have been added to allow users to operate on SFS files and directories include the following:

- CREATE—Creates aliases, directories, locks, and empty files
- DELETE—Deletes locks
- GRANT—Grants authorities to files and directories
- LISTDIR—Lists SFS directories
- RELOCATE—Moves SFS files and directories
- REVOKE—Removes granted authorities

The Shared File System may also be accessed by using many of the productivity aids that are commonly used today with CMS minidisks. One of these is DIRLIST, which is a new productivity aid that has been added to CMS specifically to allow users to navigate through their SFS directory structures. This command will display SFS directories and their subdirectories in a full-screen format. An example of USERA issuing a DIRLIST command is shown in Figure 3.

In the example of DIRLIST, all the subdirectories are under the top directory (USERA, in this case). If a subdirectory was accessed, the file mode letter is displayed in the column labeled Fm. The command area (Cmd column) allows users to issue commands to be used on these directories, as in the case of a directory of files. Also, a com-

Figure 3 DIRLIST example screen



mand or function key allows users to navigate through the directory structure. Using DIRLIST, the user can go directly to an SFS directory and operate on the files within that directory. If the SFS directory is unaccessed, DIRLIST picks an available file mode and accesses that directory. When the user exits, the file mode returns to CMS.

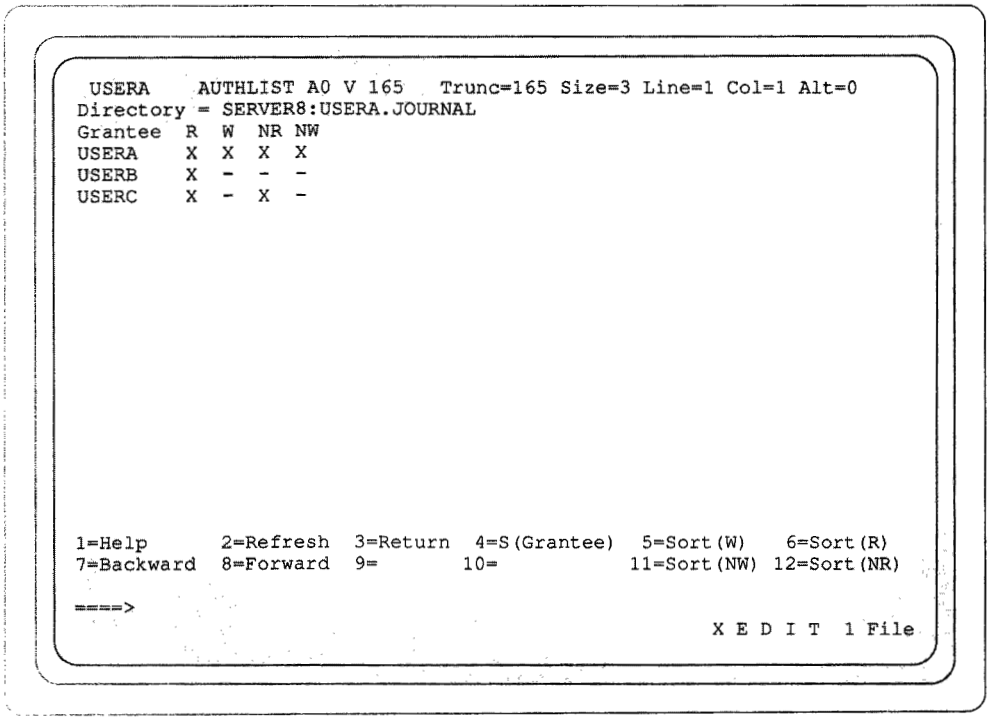
Another function that DIRLIST provides is the ability to see who has authority to a given directory. If the cursor is placed on the directory, USERA.JOURNAL, the authorization listing screen can be shown as in Figure 4.

In this example, USERA owns the directory, all read, write, new read, and new write authority. USERB has read authority to the directory, and USERC has read authority to the directory and also to future files in that directory.

The command to access the directory of files (FILELIST in Figure 5) has also been updated to display files in SFS directories. Here, the directory SERVER8:USERA.SESSIONSERVICES (displayed on the second line) is accessed as file mode letter B. Except for the directory name on the second line, this screen looks like the normal file directory (FILELIST) screen to the user. A new function key definition was also added for SFS directories; PF10 (SHARE) displays information concerning SFS sharing attributes. The new SHARE screen shown in Figure 6 can be displayed by using PF10 from the main FILELIST screen, by specifying the SHARE option on the FILELIST command or by setting SHARE as a FILELIST default using the DEFAULTS command.

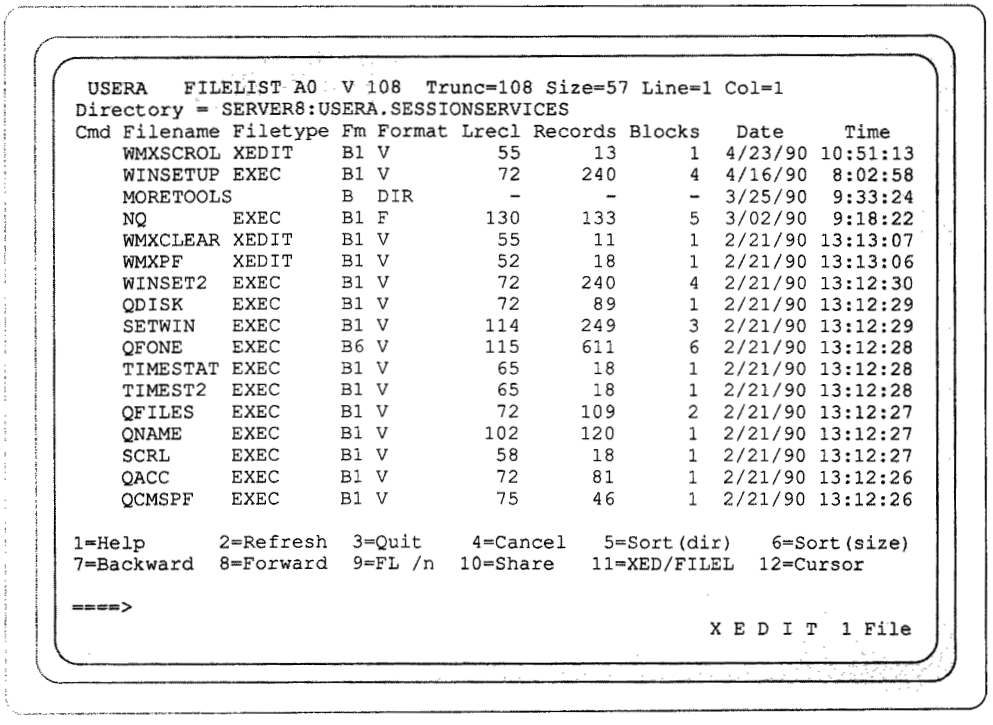
In this example, the file names are displayed along with the type of SFS file (base or alias) and the authorities the user has on these files. There are also new function key definitions to check more information, such as authority information (PF6) and file aliases (PF9). Note also that USERA has three aliases to execs that are owned by USERB. One of these execs is named TIMESTAT. To learn more about this, place the cursor on that line and use PF9 (alias), which displays the ALIALIST screen shown in Figure 7. Here, we can see that the alias TIMESTAT EXEC points to the file TESTIT EXEC, in USERB's directory MYTOOLS.

Figure 4 AUTHLIST (authorization listing) example screen



```
 USERA    AUTHLIST A0 V 165   Trunc=165 Size=3 Line=1 Col=1 Alt=0
Directory = SERVER8:USERA.JOURNAL
Grantee  R  W  NR NW
USERA    X  X  X  X
USERB    X  -  -  -
USERC    X  -  X  -









1=Help       2=Refresh  3=Return  4=S(Grantee)  5=Sort(W)   6=Sort(R)
7=Backward   8=Forward  9=        10=           11=Sort(NW) 12=Sort(NR)

====>
                                                    X E D I T  1 File
```

Figure 5 FILELIST example screen



```
 USERA   FILELIST A0  V 108  Trunc=108 Size=57 Line=1 Col=1
Directory = SERVER8:USERA.SESSIONSERVICES
Cmd Filename Filetype Fm Format Lrecl Records Blocks   Date      Time
    WMXSCROL XEDIT    B1 V         55      13      1  4/23/90 10:51:13
    WINSETUP EXEC     B1 V         72     240      4  4/16/90  8:02:58
    MORETOOLS         B  DIR        -       -      -  3/25/90  9:33:24
    NQ       EXEC     B1 F       130     133      5  3/02/90  9:18:22
    WMXCLEAR XEDIT    B1 V         55      11      1  2/21/90 13:13:07
    WMXPF    XEDIT    B1 V         52      18      1  2/21/90 13:13:06
    WINSET2  EXEC     B1 V         72     240      4  2/21/90 13:12:30
    QDISK    EXEC     B1 V         72      89      1  2/21/90 13:12:29
    SETWIN   EXEC     B1 V        114     249      3  2/21/90 13:12:29
    QFONE    EXEC     B6 V        115     611      6  2/21/90 13:12:28
    TIMESTAT EXEC     B1 V         65      18      1  2/21/90 13:12:28
    TIMEST2  EXEC     B1 V         65      18      1  2/21/90 13:12:28
    QFILES   EXEC     B1 V         72     109      2  2/21/90 13:12:27
    QNAME    EXEC     B1 V        102     120      1  2/21/90 13:12:27
    SCRL     EXEC     B1 V         58      18      1  2/21/90 13:12:27
    QACC     EXEC     B1 V         72      81      1  2/21/90 13:12:26
    QCMSPF   EXEC     B1 V         75      46      1  2/21/90 13:12:26

1=Help       2=Refresh  3=Quit    4=Cancel   5=Sort(dir)   6=Sort(size)
7=Backward   8=Forward  9=FL /n  10=Share   11=XED/FILEL  12=Cursor

====>
                                                    X E D I T  1 File
```

Figure 6 FILELIST SHARE example screen

```
 USERA   FILELIST A0  V 149  Trunc=149 Size=57 Line=1 Col=1
Directory = SERVER8:USERA.SESSIONSERVICES
Cmd   Filename Filetype Fm Owner   Type   R  W
      WMXSCROL XEDIT    B1 USERA   BASE   X  X
      WINSETUP EXEC     B1 USERA   BASE   X  X
      MORETOOLS         B  USERA   DIR    X  X
      NQ       EXEC     B1 USERB   ALIAS  X  -
      WMXCLEAR XEDIT    B1 USERB   ALIAS  X  -
      WMXPF    XEDIT    B1 USERA   BASE   X  X
      WINSET2  EXEC     B1 USERA   BASE   X  X
      QDISK    EXEC     B1 USERA   ALIAS  X  X
      SETWIN   EXEC     B1 USERA   BASE   X  X
      QFONE    EXEC     B6 USERA   BASE   X  X
      TIMESTAT EXEC     B1 USERB   ALIAS  X  -
      TIMEST2  EXEC     B1 USERA   BASE   X  X
      QFILES   EXEC     B1 USERA   BASE   X  X
      QNAME    EXEC     B1 USERA   BASE   X  X
      SCRL     EXEC     B1 USERA   BASE   X  X
      QACC     EXEC     B1 USERA   ALIAS  X  X
      QCMSPF   EXEC     B1 USERA   BASE   X  X

1=Help      2=Refresh  3=Quit     4=Cancel   5=Sort(dir)   6=Auth
7=Backward  8=Forward  9=Alias  10=Stats   11=XED/FILEL  12=Cursor

====>
                                                  X E D I T  1 File
```

Figure 7 ALIALIST example screen

```
 USERA    ALIALIST A0  V 190  Trunc=190 Size=1 Line=1 Col=1 Alt=0
Alias = TIMESTAT EXEC SERVER8:USERA.SESSIONSERVICES
Userid   Num Filename Filetype Directory
USERB      1 TESTIT   EXEC     .MYTOOLS









1=Help     2=Refresh  3=Return   4=Sort(type) 5=Sort(name) 6=Sort(dir)
7=Backward 8=Forward  9=S(user) 10=          11=          12=

====>
                                                  X E D I T  1 File
```

FILELIST also displays subdirectories and files within an SFS directory. The subdirectory MORETOOLS is displayed in the FILELIST example shown in Figure 5. By placing the cursor on the MORETOOLS line and using PF11 (XED/FILEL), the MORETOOLS directory is displayed in another FILELIST screen. Here, PF11 has a dual function. If the object is a file, XEDIT is called. If the object is a subdirectory, a FILELIST is done. FILELIST is sensitive to the type of object the user is looking at and acts accordingly.

Compatibility

CMS support for file sharing includes support designed to allow existing CMS applications to work on files managed by the Shared File System with little or no change to the application. Application conversion required depends on interfaces used by the application and whether files are actually shared.

Existing CMS applications that use any of the following interfaces are supported for files managed by the Shared File System:

- CMS File System macros
- OS and DOS macro simulation
- REXX file system functions (EXECIO)

Applications that do not share data or that use read-only sharing and use only published CMS interfaces should run without change. Changes may be required for applications that use CMS internal control blocks or CMS internal functions. If an installation wants to use existing applications to operate on files that are to be shared among multiple users with write authority to the files, then some application changes may be required.

There are fundamental differences between the way basic file functions work under minidisk level sharing and the way they work under file level sharing. Many existing CMS file system applications are written under the assumption that locking and authorization checking are done at the minidisk level and that only a single user is authorized to write to the minidisk. As a result, such applications are not prepared to deal with exception conditions that can occur in a file-level sharing environment (e.g., when encountering file locks).

The existing CMS interfaces have been modified to run on both minidisk files and shared files. Use of these existing interfaces on shared files protects the application from conditions that might be true with shared files that are not true in the minidisk environment. As a result, an application written to the old interfaces cannot take full advantage of the CMS Shared File System functions. It may be necessary to change the application to use the new Shared File System program function interface in order to take full advantage of file level sharing.

There are several instances where CMS tries to protect applications from compatibility problems. First, the ACCESS command accesses another user's directory by default in a read-only mode, even though the user may have read-write authority to that directory. The reason for this is to protect applications that use static file names. An application running simultaneously on two virtual machines may be accessing the same directory and unknowingly at the same time using the same file. Since the directory was accessed as read-only by the nonowner, the application receives an error rather than overlaying the file data. This may be bypassed by using the FORCERW option on the ACCESS command to give applications the ability to access another user's directory in a read-write mode.

Another example of trying to protect applications from compatibility problems is that of using the STATE function. If the user issuing the STATE function to check for the existence of a file does not have read authority on the file, STATE will not return a found condition, even though the user has read authority on the directory and can see the file name. The same is true with STATEW, which checks write authority to a file. It will not find a file in a read-write file mode if the user has only read authority to that file. New functions in CMS are provided for applications that need to get file and directory information in a file-sharing environment.

Migration/Coexistence

Existing applications that previously operated on files on minidisks can now operate on files on minidisks or in SFS directories. In some cases, this involves changes to existing application programs. The number and type of changes depend on the level of sharing of an application's data. If the application is using files on a read-write minidisk and the user wants to transfer these files

to a private read-write SFS directory, the number and type of changes (if any) to the application are small.

Private read-write files and read-only sharing. This environment involves files that are not being shared (private) or shared but not updated (read-only sharing). This SFS environment most closely resembles the use of minidisks. This environment requires the least number of possible changes for a CMS application. An application can work unchanged under any one of the following conditions:

- A file has no authorities granted to other users.
- No aliases have been created for the file.
- There are no dependencies on the use of minidisk addresses.

If either of the first two points is false, then the files are considered shared. (This is discussed in the following section.) Existing applications may or may not require changes. An application must change so as not to use CMS minidisk addresses and instead to use SFS directory names. This involves deleting uses of the LINK commands and changing the minidisk address on the ACCESS command to the name of the appropriate SFS directory. If the applications do not depend on CMS minidisk addresses (i.e., use predefined file modes) or do not use file system internals, no changes are necessary.

Read-write sharing. This will be the most common file-sharing environment. Typically, the owner of a file grants other users read authority on the file. When an application opens a file, it receives a copy of the last committed version of the file. The data in the file remain consistent until the file is closed and reopened, during which time the file is updated. The directory containing this file does not have to be reaccessed in order to see the changes.

Files for this application can reside on either a minidisk or in an SFS directory. Existing applications using File System (FS) macros, as well as execs using EXECIO, can reference files in SFS directories. In this case, an application used by either the reader or writer may encounter situations that are new in this file sharing environment. Applications may have to change to account for any of the following conditions:

- New CMS return codes. There are some new CMS return codes to indicate conditions such as a file being locked. An application may have to be aware of this to work properly in an SFS environment.
- Updated strategy for handling temporary files. A common way to update minidisk files is to create a temporary file, erase the old file and rename the temporary file to the old file name. Using SFS, when the old file is erased, the authorities and aliases associated with the file are also deleted. Thus, the new file is created, but the authorities and aliases are lost. When the temporary file is created, all users who have been granted new read and new write authority on the directory also have authority to the file.
- Minidisk application assumptions. (1) If one file on an accessed file mode can be written to, all files on that file mode can be written to. It is not necessarily true that a user can write to all files in a read-write SFS directory. An SFS directory accessed as read-write may have an alias that points to a file to which the issuer has read-only authority. In this case, an error could result when the user attempts to write to the file through the alias name. (2) A particular user can always read or write an existing file. A file may exist but be locked. Applications that check for the existence of a file, then assume that they have authorization to read it, may receive CMS return codes indicating the file is locked. (3) If the file exists, the user knows it has at least one record of data. Starting with VM/ESA Release 1.1, SFS supports empty CMS files. This file will look like a normal CMS file, except the number of records and number of blocks used will be zero.
- Applications referencing file system internals, such as file system control blocks, may have to change.

Many of the existing CMS applications that reference minidisk files work with little or no changes to them. The number of changes that must be made for existing applications depends on the levels of file sharing as described previously. If a user wants an application to take fuller advantage of the capabilities of the Shared File System, the application should be updated using the new application interfaces that are described later in this paper.

Programming interfaces

The enhanced capabilities of the Shared File System can be exploited by both assembler level and high-level language (HLL) applications. A new

feature of CMS provided in VM/SP Release 6 was the Callable Services Library (CSL), which contains most application interfaces to the Shared File System. Thus the CSL interface allows both assembler and high-level languages, such as FORTRAN, PL/1, COBOL, C, and REXX, to call these services directly from the application. In the past, when an application required a native VM service instead of a language-provided service, a high-level language would have to call an assembler subroutine to execute that service (such as FSOPEN). All of the SFS programming interfaces are contained in a CSL library called VMLIB. This library is loaded by the VM system profile exec at initialization time so that the services are present when needed.

The Shared File System is essentially a server machine that manages many files for many VM users.

In addition to the standard open, close, read, write functions, SFS provides many new functions that are not available on the minidisk file system. Some of these functions include the following:

- Creating directories and aliases
- Creating and deleting locks
- Granting and revoking of authorities
- Creating empty files
- Relocating files and directories
- Asynchronous read and write
- Committing file data without closing the file

These functions are all available through the CSL programming interface. Another function of the CSL interface is that an application can front-end the IBM-provided programs. A user can create a CSL routine with the same name as the IBM-supplied routine. The IBM-supplied routine can be loaded under a different name. The user routine then gains control when called, takes the indicated actions, and then passes control to the IBM routine. For example, the DMSOPEN CSL routine allows programs to open an SFS file. This routine can be loaded by the user under the name DMSOPENX, and the user can supply a DMSOPEN routine. The user can now trap a DMSOPEN call and take some action before passing control to the IBM DMSOPENX routine to actually open the file.[3,4]

SFS administration

The Shared File System is essentially a server machine that manages many files for many VM users. Given this, it is necessary to be able to administer the resources within the SFS file pool. Administration authority gives access to the resources of a file pool. With this authority, one can add minidisks to the file pool, enroll and delete users, and so on. One need not be an enrolled user to be an administrator. Administrator authority also gives quasi-ownership of every object stored in the file pool. The administrator automatically has write authority on all the base files, aliases, and directories in the file pool and can do anything with them that the owners can do.

An SFS administrator's primary responsibilities include the following:

- Enrolling and deleting users in a file pool
- Controlling the size of file spaces for individual users. If a user wants to increase a file space (e.g., add 500 blocks), an administrator can do this on line, and the space becomes immediately available to the user.
- Adding minidisks to the file pool. This becomes necessary as the physical storage fills up. Threshold warnings alert an administrator of this condition.
- Backing up the file pool
- Resolving user situations. Having administrator authority allows someone to resolve such conditions as a user's placing an explicit lasting lock on a file and then going away on vacation.

For more information on administrating SFS file pools, see Reference 5.

An example of SFS usage

Growth of the VM development organization in recent years has created a need for better and more efficient methods of managing code changes. With the large number of people working on CP and CMS, an automated system of code control is critical to the on-time delivery of new

VM releases. The need for a code library system led to the development of the IBM Endicott Development Library (EDL), one of the first major applications to be written using the CMS Shared File System. In the following, EDL refers to the code library system itself.

Other code libraries are available within IBM, several of which were considered for VM code control. However, during system test, VM development is in the practice of installing prerelease VM on the IBM Endicott Programming Laboratory (EPL) production systems as soon as the code is relatively stable. That means any library system used by VM developers must be able to run on code that is one to two releases ahead of the version available to customers (and other IBM sites). None of the other library systems could guarantee support for that type of environment. Because VM/SP Release 6 was in system test at the time that EDL was developed, a code control library was the best test of a new file system.

EDL provides a wide range of functions used in code development. One of these is named code bases. The code base for a release or project is kept on a set of directories. Each code base is assigned a name describing what is in it. (For example, CMS6+SRV contains code for CMS Release 6 plus all service updates.) This eliminates the need for most developers to know where the code is kept. They can simply tell EDL to access the code base for them. Another function is that of editing and updating facilities. After a code base is accessed, EDL functions allow developers to change the code using XEDIT. The UPDATE facility of XEDIT is used to make incremental updates involving distinct pieces of code to each of the modules. Compiling facilities is a function by which code updates are applied using the UPDATE command, followed by code compilation. The compile can be done in the developer's virtual machine or by a batch machine. System build facilities are provided so that after the code is updated and compiled, developers can build a private copy of the system and test the code before making the updates available to others. Code integration facilities make it possible to combine new source code updates with the existing base code and previous updates. When code is integrated, it becomes accessible to all of the developers.

The Endicott Development Library (EDL) operating environment consists of several parts, as shown in Figure 8. The EDL file pool contains the directories used for each named code base that is defined in EDL. The directories that are used for module ownership control are also located there. EDL programs and tables are kept on an SFS directory that is accessible to all developers. This directory is also in the EDL file pool. The EDLADMIN service machine is enrolled as an administrator for the EDL file pool. It processes code integration requests and also performs a limited number of privileged file pool requests on behalf of developers.

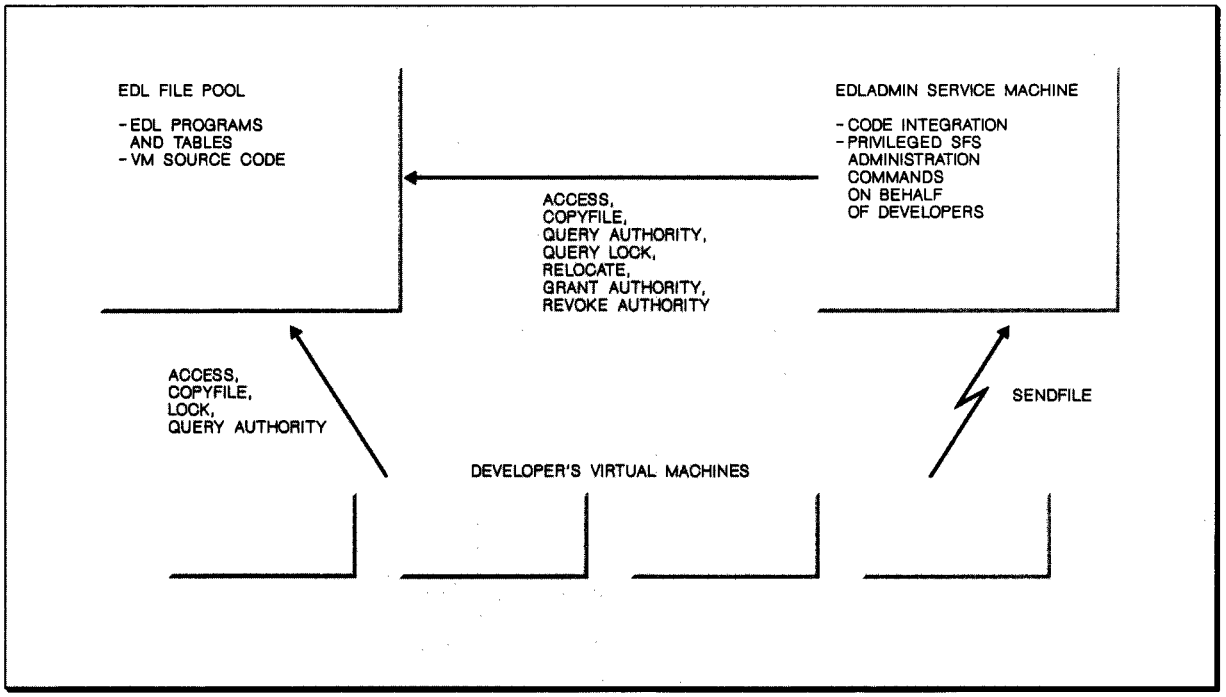
The Shared File System function is utilized by EDL in a number of ways, several of which are presented in the sections that follow.

SFS is a distributed file system. It is very important in the EPL computing environment to have distributed file handling capability, because development resources are spread over six computer systems: Four systems run development-level, new-release code; one system runs VM/SP HPO Release 5 CP with VM/SP Release 6 CMS; and one system runs VM/XA SP 2. With the exception of the VM/XA system, developers can obtain data in the EDL file pool from any of the EPL computer systems via transparent services access facility (TSAF). In fact, using the facilities provided by APPC/VM VTAM support (AVS), authorized developers at other IBM sites can also access VM source code.

SFS provides better security. In addition to allowing flexibility in accessing source code, SFS also keeps that code secure, in that no one can gain access to the VM source code without first being enrolled in the EDL file pool. Once enrolled, developers can look at any of the source code for VM and VM-related products. If there were a need for subdivisions of security, SFS authorization functions could be used to do that too. (For example, one might want to limit access only to CMS.)

SFS allows enforcement of module ownership. With a large number of developers and a large amount of new function being added to VM, it may happen that two or more developers need to be updating a module concurrently. A system of module ownership was introduced in EDL that helps in this situation. Primary and secondary owners are defined for each module in the system. By default, owners have authority to integrate changes to the modules they own. If a developer needs to update a module that that person does

Figure 8 EDL structure



not own, the developer can ask a module owner to grant the authority to integrate the change. The restricting of unauthorized changes, however, is not the main intent of module ownership. Such restriction is merely intended as a formalized communications channel that allows an owner to control module updates in an orderly fashion, without the problems of sequence errors or overlaying of existing code.

The locking functions in SFS also help during concurrent development. If a developer checks out a file for updating, others who try to make changes to the file receive a warning that someone is updating the file. Also, SFS locking functions are used to keep two developers from integrating completed code changes for the same module simultaneously.

SFS simplifies code integration. The actual code integration process has been greatly improved using SFS. Before EDL was implemented, developers would send their completed code changes to a person in the Product Control Group (PCG) using the SENDFILE command, then the developer was required to deliver to the PCG a sheet of paper listing what had been sent and the code base to which it belonged. At major code checkpoints, when a large amount of code was integrated at one time, this resulted in a mountain of paper.

EDL uses a different approach. The developer creates a file that lists all the updates to be integrated into the system. When the code is presented for integration, EDL takes the following actions:

1. Verifies that all the files listed are in the developer's CMS search order
2. Verifies that there are no update locks on any of the modules being updated
3. Verifies that the developer has authority to integrate the changes
4. Copies the updates to a staging directory. (All developers enrolled in the EDL file pool have write authority to the staging directory.)

When the files have all been copied successfully, the developer simply presses a key that tells EDL

to process the integration. EDL copies the update list to the staging directory and sends a message to the EDLADMIN service machine. EDLADMIN reverifies integration authority, moves the files into a collection directory, and grants PUBLIC read authority on the files. Once the files have been moved to the collection directory, everyone who is using that code base can see the updates automatically, because the file information for accessed SFS directories is always kept up to date in the user's machine. From the collection directory, PCG processes the files, rebuilds the system, and moves the files to the directory where valid updates are kept. Each code base uses a different set of directories, so there is never any confusion about which updates belong on which base.

EDL continues to grow and evolve. Used daily by nearly every programmer in the Endicott Programming Laboratory, it helps developers get their jobs done more efficiently. EDL has provided quantitative benefits by decreasing from hours to minutes, the amount of time needed to integrate a change. This allows the PCG to provide daily a rebuilt system with all new updates applied. (The process can be done twice a day if necessary.) Testers find fewer duplicate problems, because the code-fixes quickly appear in the next system build. EDL has provided qualitative benefits by giving developers a consistent set of easy-to-use interfaces for developing code. Also, by using EDL and exercising its SFS interfaces during Release 6 system test, VM development found and fixed several errors in the new file system. The development and use of EDL has contributed to a greater understanding within VM development of how customers might use SFS and how it can be improved in the future.

Summary

The intent of this paper is to provide the reader with an understanding and appreciation of the capabilities of the CMS Shared File System (SFS). The design adds to the functions available with the CMS minidisk file system and is as compatible as possible with it.

Specifically, the following areas were addressed with SFS:

- File sharing is allowed at the file level. Individual users may be granted separate read or read-write authorities for individual files and directories.
- Space sharing allows a group of users to share the same physical space and yet have individual limits on the usage of that space.
- Security permits the owner of data to control the usage of those data as regulated by user ID, not by the use of a password.
- Data addressing allows a user to create directories and to organize files in those directories.
- Remote access allows users on separate VM systems to share the data.
- Data integrity gives the user the assurance that if a failure occurs the data will be at a predefined, consistent level when operations resume.

In applications such as the Endicott Development Library (EDL), the Endicott Programming Laboratory has realized benefits from these capabilities of SFS and is using this experience as one source of input for future enhancements. SFS has also provided the laboratory with the ability to share files and documents among design, development, planning, and other organizations.

Virtual Machine/Enterprise Systems Architecture and VM/ESA are trademarks of International Business Machines Corporation.

Cited references

1. *VM/ESA Connectivity Planning, Administration, and Operation Manual*, SC24-5448, IBM Corporation; available through IBM branch offices or authorized dealers.
2. C. C. Barnes, A. Coleman, J. M. Showalter, and M. L. Walker, "VM/ESA Support for Coordinated Recovery of Files," *IBM Systems Journal* **30**, No. 1, 107–125 (1991, this issue).
3. *VM/ESA CMS Application Development Guide*, SC24-5450-00, IBM Corporation; available through IBM branch offices or authorized dealers.
4. *VM/ESA CMS Application Development Reference*, SC24-5451-00, IBM Corporation; available through IBM branch offices or authorized dealers.
5. *VM/ESA CMS Planning and Administration Guide*, SC24-5445, IBM Corporation; available through IBM branch offices or authorized dealers.

Richard L. Stone *IBM Data Systems Division, P.O. Box 6, Endicott, New York 13760.* Mr. Stone is currently an advisory programmer in VM Product Planning at the Endicott Programming Laboratory. He attended Purdue University, West Lafayette, Indiana, where he received his B.S. degree in mathematics. He joined IBM in Endicott in 1968 as a junior programmer in the programming development area. Mr. Stone spent several years in the VS1 supervisor development and project office groups. He was one of the lead designers on the MVS/Operator Communication Control Facility, VM/Group

Control System, and CMS Shared File System projects. His current responsibilities include product planning for VM file and data management.

T. Scott Nettleship *IBM Data Systems Division, P.O. Box 6, Endicott, New York 13760.* Mr. Nettleship is a senior associate programmer in the CMS File System Development department in the Endicott Programming Laboratory. Mr. Nettleship joined IBM in 1983 in Endicott, New York. He has worked in the development of the Group Control System (GCS) component of VM. He has worked on the development of the Shared File System since its inception and continues an active roll in its current development. Mr. Nettleship is a coauthor of a Technical Disclosure Bulletin, "VM GCS Dispatching Program," IBM (1985). He is a coauthor of the Technical Report *GCS Overview and Application Example*, IBM (1988). Mr. Nettleship was the CMS GUIDE user group representative from 1988 to 1990 and is now an IBM representative to the SHARE user group. He received his B.S. degree from Rutgers University, New Brunswick, New Jersey, in 1982 and his M.S. degree from the State University of New York at Binghamton in 1987.

Jay Curtiss *IBM US Marketing & Services, P.O. Box 81868, Lincoln, Nebraska 68508.* Mr. Curtiss received his B.S. degree in computer science from the University of Nebraska at Lincoln in 1983. The same year, he joined the Endicott Programming Laboratory as a programmer working on the development of VM/GCS (Group Control System). He then worked on performance enhancements in CMS Release 5. In 1985, Mr. Curtiss became an original member of the team that designed the CMS Shared File System that was announced in Release 6 of CMS. Since then, he has been involved in many enhancements to CMS for VM/ESA. At the same time, he was a member of the team that developed the initial versions of the Endicott Development Library (EDL). He also served on a committee assigned to review software invention disclosures submitted from the IBM Endicott laboratory. Most recently, Mr. Curtiss transferred to the IBM branch office in Lincoln, Nebraska, where he is a systems engineer involved in large systems support.

Reprint Order No. G321-5423.