2006

Ad hoc file system

Swarup Datta

Rochester Institute of Technology

Ad-hoc File System

MS Project Report

By:
Swarup Datta
sxd2009@cs.rit.edu

Date: May 22, 2006

Chairman: Prof. Hans-Peter Bischof
Reader: Prof. James E. Heliotis
Observer: Prof. Alan Kaminsky
Observer: Prof. Roxanne Canosa

ABSTRACT

Ad hoc File System (AFS) is middleware for peers wishing to share and access data in ad hoc fashion. AFS permits processes to export local files and import files shared by other processes. Communication between hosts is facilitated by the M2MI Framework, a novel paradigm for building collaborative systems. The API provided by AFS allows for data to be exported to the middleware, which become accessible to hosts running AFS. Interested peers can import exported data to their local system, allowing uniform access to local and remote files. Peers can also unexport and unimport data undoing effects of export and import operations. To illustrate functionalities of Ad hoc File System, two user applications have been developed. The first is a character based console application similar to shells, such as the K shell, and the second is a visual application that demonstrates how the file system in AFS changes due to different operations.

TABLE OF CONTENTS

1 Introduction

AFS, a stateless distributed file system, provides an API for sharing and accessing data on hosts participating in an ad hoc network. AFS was developed using Many-to-Many Invocation (M2MI) Framework, which is middleware designed to facilitate communication between processes in a mobile ad hoc network. This project attempts to address two important issues faced by distributed file systems in an ad-hoc network: accessing data in a network without central authority, and gracefully handling imported data that become unavailable due to topological changes in a network. The project report details functionalities implemented in AFS with a broad overview into the design of the file system.

The rest of the document is divided in the following fashion. Section 2 takes a general look at ad hoc networks and discusses the need for a distributed file system targeting hosts in these types of networks. Section 3 details the functionality of AFS. Architecture of M2MI and Ad hoc File System is provided in section 4. The design of the main classes constituting the AFS middleware and the file system API is discussed in section 5. Section 6 describes the two applications developed to demonstrate usability of AFS. The report concludes with a comparison of AFS and MFS, another distributed file system, and future work.

2 Ad-hoc Networks

Ad hoc Networks are a new networking paradigm, where hosts can communicate with other hosts without the presence of any fixed infrastructure. In traditional networks, wired or wireless, hosts rely on dedicated machines and access points for routing information, access to network resources and host discovery. In an ad hoc network, hosts rely on each other, working collectively to provide network connectivity and access to resources. Peers participating in ad hoc networks tend to be small battery powered mobile wireless devices, which contribute to an unpredictable and very dynamic network topology.

In any type of networks, traditional or ad hoc, the ability to share and access data among hosts is desirable. Distributed file systems allow hosts to share and accesses network data in stable and reliable manner. File systems are made up of underlying data storage and organization system, designed to provide easy access to files and directories. Examples of distributed file systems are Network File System (NFS), Google File System (GFS), and Coda File System, with NFS being the most widely used.

The NFS protocol allows a host to function both as a server and a client. However, in the usual configuration, one or more dedicated machines are used as file servers, responsible for storing network data. To export a file from a NFS server, the *etc/exports* file must be configured, where clients allowed to access exported files are be specified followed by optional flags. Clients

of NFS servers use the mount command to access exported files, and the name of the host server is required a priori along with name of the exported paths.

In an ad hoc network, requiring prior knowledge of host names and exported paths pose a problem, as hosts can join and leave the network at will. Thus a mechanism is required where a process can learn dynamically which hosts are exporting data. Also, existence of small battery powered devices with limited computation capabilities and storage makes the concept of dedicated servers unattractive. Due to lack of dedicated servers, each device is expected to act both as a server, exporting local directories, and as a client, accessing data exported by other devices.

Meanwhile, the dynamic nature of ad hoc networks generates a need for a mechanism that can gracefully handle events when hosts expectedly or unexpectedly leave a network. For example, in an ad hoc network with multiple devices, let there be a node whose removal can lead to partitioning of the network. If the critical node suddenly leaves the network, data currently being accessed can become unavailable. Hence, a new distributed file system is required to address issues posed by ad hoc networks. Ad-hoc File System, developed under this project, focuses on the two aforementioned issues.

3 Functionality

The concept of accessing remote data is fundamental to a distributed file system. The bind functions in AFS allow a process to gain access to networked files and directories by importing the remote objects into a process's file system. However, in order for a process to access data, other processes have to be willing to share data. In AFS the Export mechanism permits processes to share locally stored data.

Files visible to AFS are internally maintained in a tree like structure, referred to as the virtual file system. The term native file system is used in this document to refer to the file system provided by a device's operating system. In AFS, the root directory of the virtual file system is a directory on the native file system, which can be specified by the user when AFS is launched. The virtual file system in AFS is generated by copying the structure of the root directory on the native file system.

If a device is running two applications that are using AFS, where the applications are running in separate Java™ Virtual Machines, each application runs a separate instance of AFS, with each instance maintaining its own copy of a virtual file system. These two applications can share files among each other using AFS. In addition to exporting and importing data, the distributed file system allows users to perform basic file operations, such as read, write, open and close. Operations executed on imported files or directories are reflected in the virtual and native file system exporting the remote file or directory.

3.1 Exporting and Un-Exporting

A process starts and stops sharing data by using the Export mechanism. When data is exported to the middleware, a message is published to notify peers that new data is being shared. The export message includes an ID uniquely identifying the shared object, which can be a regular file or directory, name of the file or directory, and ID of the device generating the message. When a process wishes to import shared files, data encapsulated in export messages is utilized to perform a bind operation, as discussed later. However, publishing of messages only when data is exported poses a problem for a newly joined device, as processes on this device will not have received export messages for data already exported. To resolve this issue, a process wishing to discover files that are currently being shared in a network, can send a message to other processes requesting their export information. An export information request triggers all processes sharing data to send export messages to the requesting process.

In AFS, the smallest exportable object is a regular file. If a directory is exported, the content of the exported directory are automatically shared. However, only locally stored data can

be exported, data imported into a virtual file system cannot be exported to AFS. Figure 1 shows three different devices running process *A*, *B* and *C*, and the virtual file systems visible to each of the three different processes. In Figure1-a by process *A* is exporting a regular file. In Figure1-b directory *b21* is exported by process B.

To stop sharing a file or directory, a process can use the Un-Export operation. Similar to Export, Un-Export generates a message that informs others that a currently shared object is being unexported. This message allows a process to gracefully remove imported files from its virtual file system.

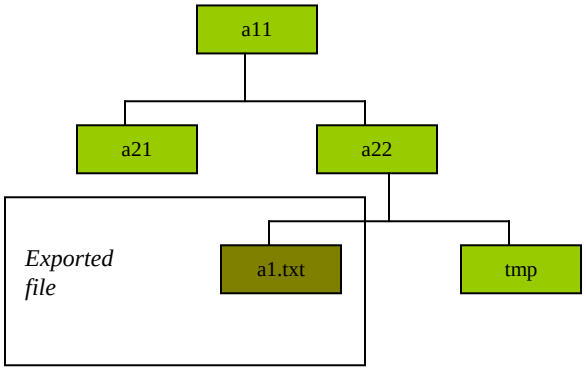


Figure 1-a: Device 1 running process A. Regular file a1.txt is exported by A.

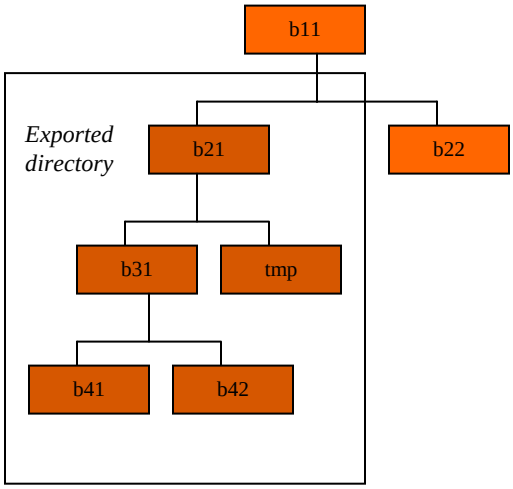


Figure 1-b: Device 2 running process B. Directory b21/ is exported by B.

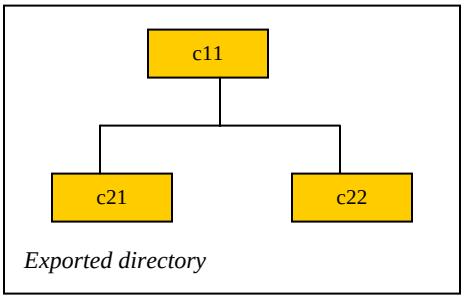


Figure 1-c: Device 3 running process C. Directory c11/ is exported by C.

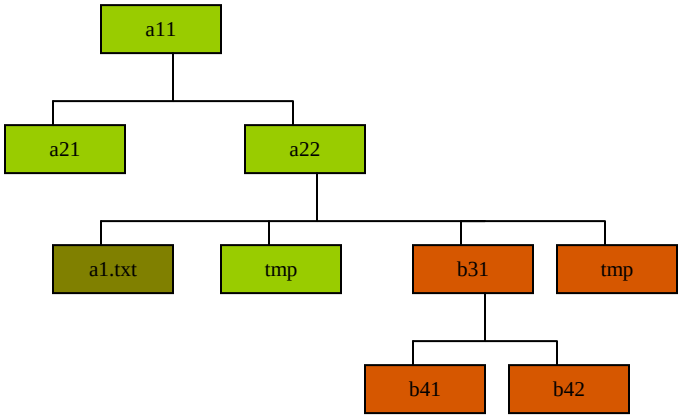
3.2 Bind

Once data has been exported to AFS, the Bind mechanism allows peers to import the shared data into a process' virtual file system, modifying the file system namespace. The imported entity, generally a tree structure by forming a union directory. Contents of multiple directories merged together create a union directory. Union directories provide an advantage over simply attaching the imported file to the virtual file system. For example, if a process is exporting a directory */arch/cpu/bin*, another process can bind the directory to its */bin* directory and access contents of */arch/cpu/bin* as if they were local to */bin*.

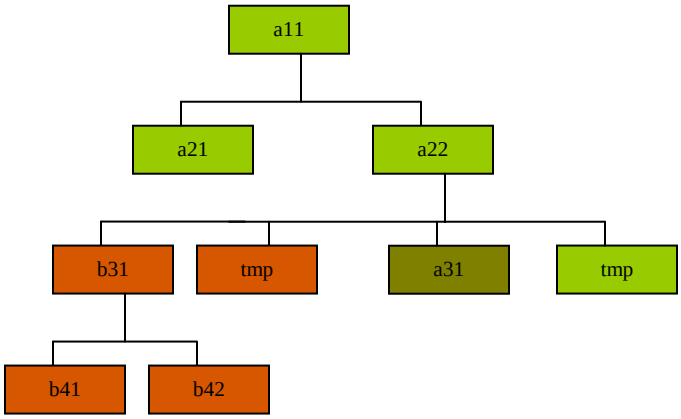
The Bind operation requires three arguments, a binding point, a bind point, and a bind flag. Binding point refers to the remote object being imported, and bind point is the directory where the binding point is attached. Bind point can be a directory physically located in a device, or it can be a remote directory in the virtual file system. The third parameter, bind flag, with possible values of *Before, After,* and *Replace,* defines how the union directory is created, giving the user different options to modify the virtual file system. *Before* attaches the remote file at the beginning of the union directory, *After* places the contents of the remote file at the end of the union directory, and *Replace* option replaces the bind directory with binding point. The figures below demonstrate union directory formed by using the three different bind flags. The order of binding imported file is important in AFS, specially for file lookup, as we will see in the next section.

Figure2-a shows *A*'s file tree after remote directory *b21* has been attached to directory *a22* with bind flag *After*. Now contents of b21 can be accessed using directory a22. In Figure2-b, remote directory *b21* is imported with bind flag *Before.* Figure2-c shows modification to *A*'s virtual file system after bind operation is performed with *Replace* option.

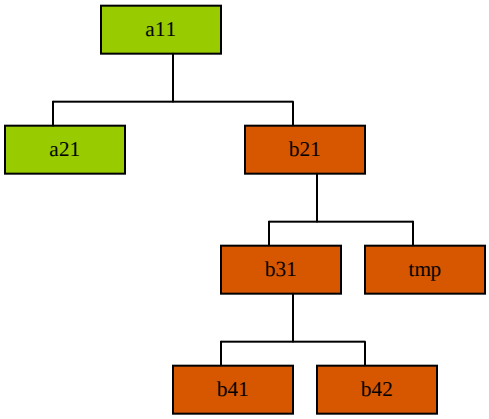
Union directories introduce an issue for name resolution in AFS. How should the contents of a union directory be searched? For example in figure 2-b, when b21 is imported two different file with the same name might exist in directory a22 and b21, as is the case. Next section discusses how name resolution is performed in AFS.



a) Device *A*'s local file system after binding b31 to a22 with bind flag *After*



b) Device *A*'s local file system after binding b31 to a22 with bind flag *Before*



c) Device *A*'s local file system after binding b31 to a22 with bind flag *Replace*

Figure 2: Union directories.

3.3 Name Resolution

Union directories in AFS can lead to conditions where a path in a virtual file system cannot be resolved to a single file. In traditional file systems, such as the Unix file system, an absolute path can only be resolved to one physical file. In AFS, use of union directories creates circumstances where an absolute path can point to more than one file, as an imported directory and the directory it is attached to may contain files with same name. For example in Figure 2-b, the path *a11/a22/tmp* can be used to refer to two different directories.

To resolve this problem of name resolution, contents of union directories are searched from left to right. In Figure-2b the search for a file a11/a22/tmp/f1.txt, the instance of the tmp directory under b21 will be searched first for the file f1.txt. If f1.txt is encountered in this directory, the search will conclude, and return the instance of f1.txt under first tmp directory. Otherwise, the second *tmp* directory is searched for f1.txt. The same lookup method is applied during a file open operation. Open with write mode will return the first file encountered with write permission, and open with read mode will return the first file with read, or read and write permission. In the case of create or copy file operations, if the create path points to more than one directory, AFS will attempt to create the file or directory in the first writable directory. For example creating */a11/a22/tmp/file2.txt,* AFS will attempt to create the file in the first *tmp* directory with write permission. If the user does not have write permission in the first *tmp* directory, AFS will proceed to create the file in the second *tmp* directory, under *b21*. In the event that file creation fails in the first directory with write permission, AFS will abort the operation.

3.4 Unmount

The unmount operation allows users to remove an imported file from a virtual file system. In the trivial case, where the remote file or one of its children do not form a union directory, as in Figure 2-a, Figure 2-b, and Figure 2-c, removing an imported object returns the union directory to the its previous state. For example in Figure 2-c, directories *a31* and *tmp* will be accessible to the user again after unmounting *b21*. In Figure 3-a, unmounting imported directory *c11* will still allow the user to access the contents of *b31*. However, unmount operation becomes more complicated when a child of the directory being removed is also a union directory. For example if directory *c11* is removed from the file structure in Figure 3-b, *b31*, *tmp*, *b41*, and *b42* will become inaccessible. The next section, 4.5, discusses how AFS provides a mechanism for unmounting a remote file that avoids creating inaccessible file structures.

3.5 Shadow Directory

An imported file or directory is removed from a virtual file system if the process exporting the imported file stops exporting the file, or a user chooses to unmount an imported object.

Sometime unmounting a file can lead to dangling tree structures. If *c11* is removed from process A's virtual file structure, as shown in Figure 3-b, directories *b31, b41, b42,* and *tmp* become inaccessible. Shadow directories are utilized in AFS to maintain file systems when removing an imported directory can lead to forest(s). Using the previous example, when process detects that *c11* is unexported or unmounted, a pseudo file structure constituting of shadow directory representing the path from *a22* to *b21* is added to its virtual file tree (Figure 3-b). Directory *c22* will be part of the file system until directory *b11* is unmounted. The sole purpose of the shadow file tree is to provide access to data that can become inaccessible due to unmount operation. In AFS, certain restrictions appliy to shadow directories. Files and other directories cannot be created in a shadow directory. However, users can import remote data to a shadow directory.

To implement the proposed mechanism of pseudo-file structure, a locking mechanism is required to prevent a process from accessing a segment in the virtual file system that is being replaced with shadow directories. The mechanism will lock the affected segment's parent node, denying access to its subdirectories. The affected segment is any part of a virtual file system that and has to be represented by a pseudo-file structure. In Figure 3-b, the proposed locking mechanism would lock *c22*'s parent denying access to the whole sub-tree rooted by the node *a22*. Once the pseudo-file structure is in place, the lock on *a22* will be removed.

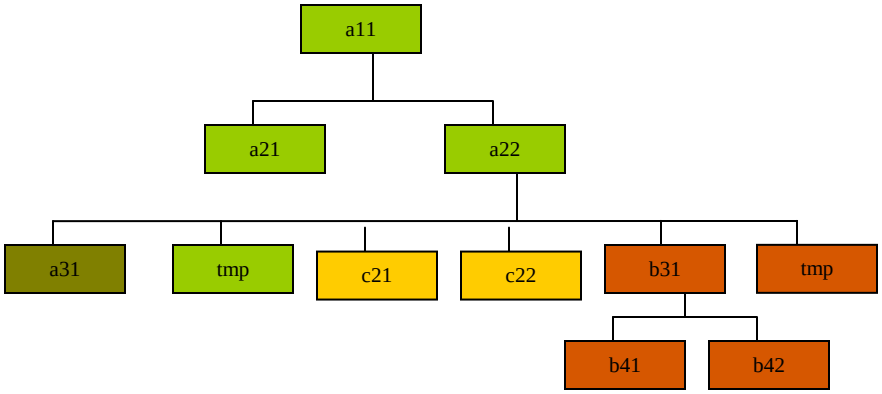


Figure 3-a: A virtual file system with remote directories c11 and b21 bound to directory a22.

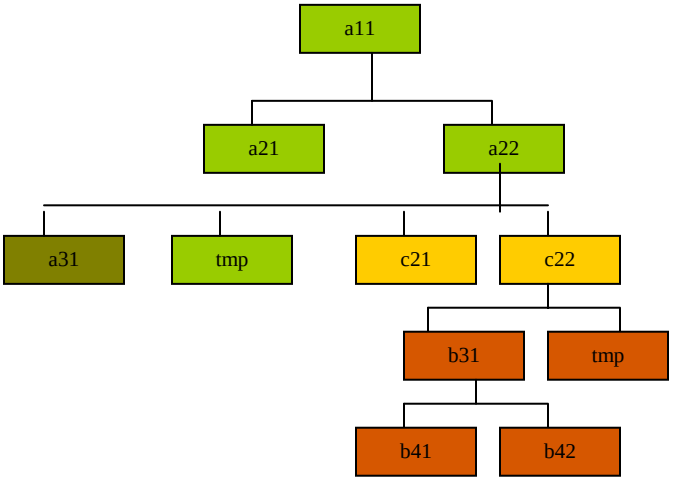


Figure 3-b: A virtual file system with remote directory c11 bound to directory a22, and remote directory b21 bound to directory c22.

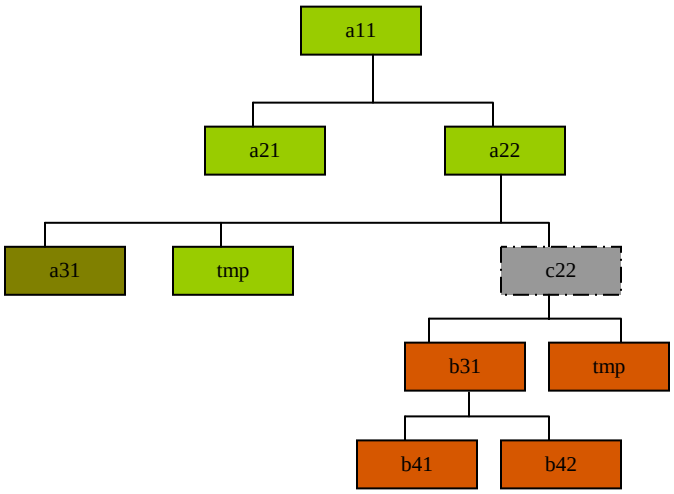


Figure 3-c: A virtual file system from figure 3-b with shadow directory c22.

3.6 File Operations

Basic file operations supported by the AFS middleware are listed in Table 1. Modifications to an imported file structure resulting from any basic operation will be reflected in the process exporting their respective data. As in any other file system, multiple processes can access the same file in AFS.

Operations	Description
Create	Create a file or directory
Open	Open a file
Close	Close a file
Read	Read from a file
Write	Write to a file
Remove	Remove a directory or file
Move	Move a file or directory
Ls	List directory content
Link	Create a symbolic link to a file.

Table 1: File operations supported by AFS middleware.

4 Architecture

4.1 M2MI

Figure 6 shows the architecture of M2MI developed under Anhinga project [1]. M2MI is a new paradigm that facilitates development of new applications for collaborative systems. Here two devices are participating in an ad hoc network, where device A is running two processes, Process 1 and Process 2, and device B is running a single process, Process 3.

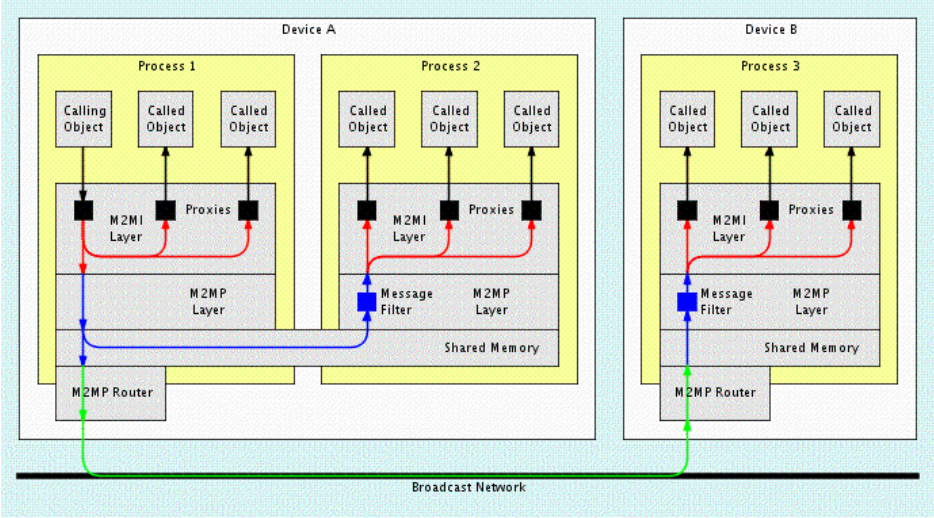


Figure 4: M2MI Architecture [1]

M2MI allows inter process communication, whether the processes are resident in the same device or running in different devices. Communication in M2MI can happen in a one-to-one, one-to-many, or many-to-many fashion. Unihandle, multihandle, and omnihandle objects generated by the framework facilitate these tree types of communication capabilities, respectively. A handle in M2MI, generated for a target interface, defines all the methods of the interface. When a method on an omnihandle is invoked, all objects implementing that interface also invoke the same method. Invoking a method in a unihandle invokes the same method in the unihandle's target object.

4.2 Ad hoc File System

AFS middleware developed using M2MI allows for a process using the middleware to communicate with other peers in the network. Unihandle and omihandles classes in M2MI are instrumental in providing AFS its distributed nature, as unihandles allow one process to talk to

another specific process in a network, and omihandles allow one process to send messages to every other process in the network.

AFS is divided into three main layers, as shown in Figure 5. The top layer consists of the file system API, which provides a uniform interface to the middleware. Processes interested in using the distributed file system use the API to perform export, unexport, bind, unmount, and all file operations. Each method in the API calls the appropriate modules in the middle layer to perform the required tasks. All export and unexport operation are forwarded to the Export Manager. Bind and unmount calls are channeled to the File System Updater module. File operations executed by a user are forwarded to the File Operation component. As shown in the diagram, components in the middle layer do not invoke methods defined in the AFS API.

The middle layer of the middleware consists of the Export Manager, the File System Updater, the File Operation Handler, and the Virtual File System. The Export Manager encapsulates export and unexport file information in a export message, which are forwarded to the communication layer for broadcast over the network. This module also functions as a repository as it maintains a list of export messages generated by other processes, referred to as the Remote Exports.

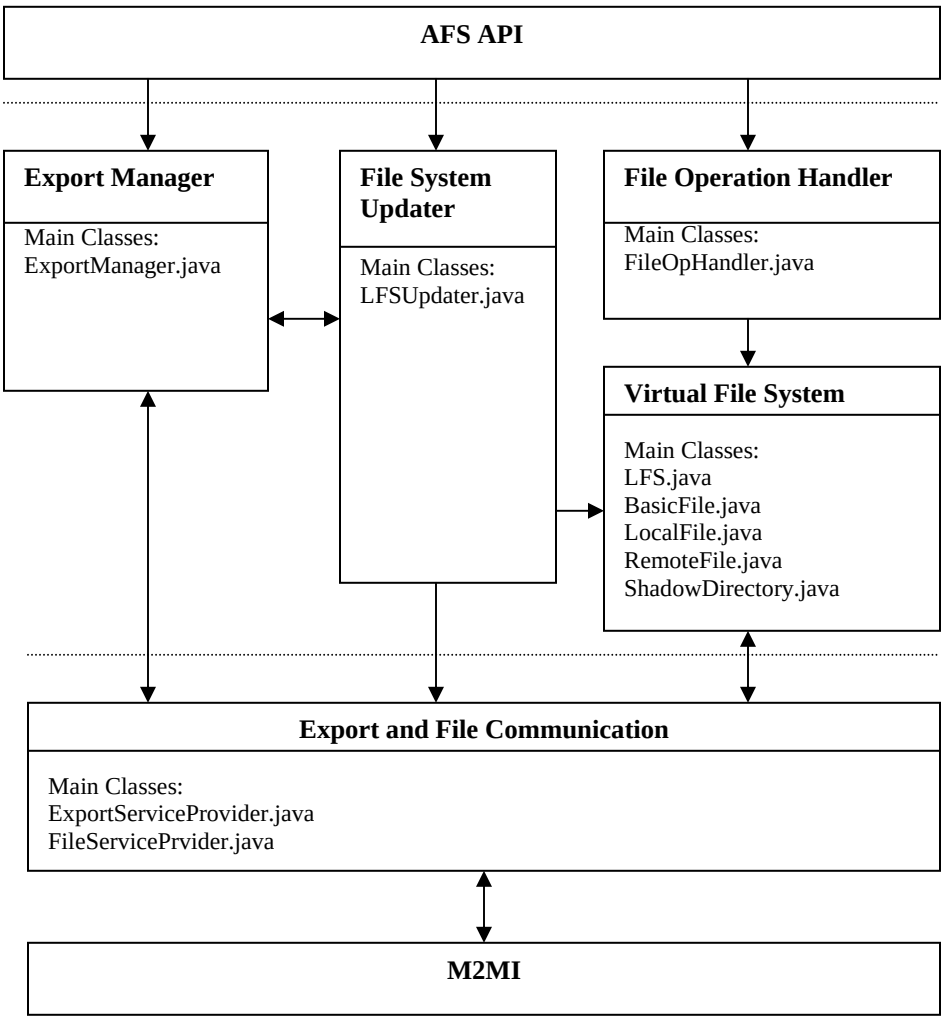


Figure 5: AFS components and their main classes. The arrows portray the follow of information between different components in AFS, and between AFS and M2MI layer.

During a bind or umount operation, the File System Updater is responsible for modifying the file tree maintained in AFS. For bind, the Remote Export list is looked up by File System Updater to retrieve the export message corresponding to the binding point. Information stored in the export message allows the File System Updater to import and integrate the file structure representing the remote file. The communication layer facilitates importing of the remote file structures for the File System Updater. For the unmount operation, the File System Updater locates and removes a remote file from the virtual file system.

The File Operation Handler, as the name suggests, executes all basic file operation, such open, close, read, and write. The File Operation Handler accesses files via the Virtual File System component, which represents the file structure generated by the middleware, maintains directories and files visible to AFS. The complete hierarchical file structure in Virtual File

System is not constructed when AFS is launched; it is built on demand. When a process starts AFS, only the node representing the root is created, and rest of the file structure is built as required, being added as a user start accessing different children of the root. The virtual file system is maintained in memory and needs to be constructed whenever AFS is launched. This file structure is build to resemble a part of the native file system that exists on a device. File permission checks and resolving a path to the appropriate file are performed by the File Operation Handler.

The third layer of AFS consists of the Export and File Communication component, which forms the backbone of the file system's communication capabilities. It directly interfacing with the M2MI layer to receive and send messages to other processes. This layer is responsible for filtering incoming messages and forwarding them to appropriate components in the middle layer. When export and unexport messages are received, these messages are forwarded to the Export Manager. When file operation messages are received, the messages are forwarded to the Virtual File System. File operation messages include opening, closing, reading, from and writing to a remote file.

5 Design

Now that the major components of AFS have been discussed, lets look at classes that make up these components: AFSApi, ExportManager, LFSUpdater, FileOpHandler, LFS, ExportServiceProvider and FileServiceProdiver. AFS employs singleton instances of these objects with AFSApi providing interface for processes interested in using the file system. Other important classes implemented are BasicFile, LocalFile, RemoteFile and ShadowDirectory. The functionality of each class is discussed along with how they interact with each other within the middleware. A code example is provided for the API as a guide on how the distributed file system can be used, and sequence diagrams are included to show interaction between different peers on a network.

5.1 AFS API

The *edu.rit.cs.afs.AFSapi* class allows users to utilize services and features provided by the distributed file system. The class constructor requires two parameters, a process name and an absolute path to a directory in the device's native file system. The virtual file system in AFS is rooted by this directory.

Exporting data: The export method allows users to share data on the network. The first parameter for the method must be an absolute path for the data to be exported data. The second parameter is a string used to describe the exported data. The method first verifies that the file or directory to be exported exists on the native file system. If the path is valid, an *ExportedFSDescriptor* object is created with the following information, export device name, share descriptor, and export object path. The newly created object is then used to call `exportFileStructure` in ExportManager class for exporting.

Export method syntax:

AFSapi.export (<exported file path>, <share description>);

Example code of export the directory */home/sdatta/Picutres/Vacation*:

```
AFSapi.export ("/home/sdatta/Pictures/Vacation",  "Vacation Pictures");
```

Un-Export data: A user can un-export the above referenced directory by calling the API's unexport method, which requires the absolute path of the file or directory to be un-exported. This method calls `unexportFileStructure` to stop sharing the exported file.

Unexport method Syntax:

```
AFSapi.unexport (<un-exported file path>);
```

The following code can be used to unexport the above referenced directory:

```
AFSapi.unexport (“/home/sdatta/Pictures/Vacation”);
```

Importing remote data: Two bind methods implemented in the class allow users to import data exported by process in a network. Both the methods require a path to the bind point, which must be a directory, and a bind flag. The two methods differ in how the remote data, the binding point, is identified. The first method requires the unique export ID for the remote object that is to be imported. The second method requires the process name, and remote object name as arguments.
Bind methods Syntax:

```
AFSapi.bind (<process ID, <binding point>, <bind point>, <bind flag>);
AFSapi.bind (<Unique export ID>, <bind point>, <bind flag>);
```

The following code can be used to import a remote directory exported by process *A*.

```
AFSapi.bind (“A”, “Vacation”, “/home/sdatta/Friends_Pics”, Bind_After);
AFSapi.bind (“UniqueID”, “/home/sdatta/Friends_Pics”, Bind_After);
```

Un-importing data: Imported data can be removed from the virtual file system by using the *umount* method. Umount method requires two arguments, the path to the union directory and the unique ID of the remote directory that is to be removed from the file system.
Umount methods Syntax:

```
AFSapi.umount(<union directory>, <Unique export ID>);
AFSapi.umount (<union directory>, <bound file>);
```

The following code can be used to remove a remote file form a virtual file system:

```
AFSapi.umount(“/home/sdatta/Friends_Pics”, “UniqueID”);
AFSapi.umount (“/home/sdatta/Friends_Pics”, “Vacation”);
```

Network share: The `updateNetworkShareInfo` method allows a user to send a request to other process to learn what data others are exporting. The share information sent by process is maintained in Remote Export list of the Export Manager.

Viewing shared data: To view data exported by other process the `getAllNetworkShare` method a can be used. This method retrieves remote data information from Export Manager’s Remote Export list. `Enumeration` objects returned by the method contain *ExportInfo* objects, which encapsulates export data information.
The example below retrieves remote export information and displays the name of the exporting process:

```
Enumeration elements = AFSapi.getAllNetworkShare();
while( elements.hasMoreElements() ){
```

```
            ExportInfo eInfo = () elements.nextElement();
            String procName = eInfo.getDeviceName();
            System.out.println("Exporting process: " + procName);
      }
```

Export data for a specific process name be retrieved from Export Manager by using `getNetworkShare` method that takes name of the process as an argument.

Directory list: The `listDirectory` method allows a user to list contents of a directory or file information. The method takes a file path as an argument, and returns enumeration of *FileListDescriptor* objects. An enumeration object with zero elements is returned if the argument to the method does not point to valid a directory or a file.

The example below shows how to list a file or content of a directory:

```
      Enumeration elements = AFSapi. listDirectory(path);
      while( elements.hasMoreElements() ){
            FileListDescriptor fld = () elements.nextElement();
            String name = fld.getName();
            System.out.println("Name: " + name);
      }
```

Open file: A file in the virtual file system can be opened using `openFile` method. The method takes path to the file to be opened as an argument. If the file is opened successfully, an object of type *FileHandle* is returned. If the path points to a directory *null* is returned.

Close file: An already opened file can be closed with `closeFile` method call. Close file takes an object of type *FileHandle* as an argument. The *FileHandle* must have been returned by a `openFile` method call.

The code below opens and close a file:

```
      FileHandle fh = AFSapi.openFile ("/home/sdatta/Friends_Pics/Pic1.jpg");
      AFSapi.closeFile(fh);
```

Write file: `writeFile` method allows users to write data to a file. The method takes a *FileHandle* object and a byte array as arguments. The byte array is written to the file pointed to by the file handle object.

Read file: `readFile` method reads data form a file. The method takes a *FileHandle* object and a byte array as arguments. *FileHandle* object must point to an already opened file. Data read from the file is placed in the byte array and returns number of bytes read.
The code example below opens file Pic1.jpg. If the file is opened successfully then 50 bytes are read from and written to the file. After read and write operation the file is closed:

```
byte[]  data = new byte[50];
String fName = "/home/sdatta/Friends_Pics/Pic1.jpg"
.
.
FileHandle fh = AFSapi.openFile (fName);
if( fh != null){
       AFSapi.read(fh, data, data.length);
       AFSapi.write(fh, data, data.length);
       AFSapi.closeFile(fh);
}
else {
       System.out.println("Failed to open " + fName);
}
```

Create file: The *createFile* method allows users to create a local or remote file. Since a new file can be created under an exported or imported directory, multiple file system might require update due to a single create operation.

When a local file is created, if the file's parent directory has been exported, then all process importing the parent file adds the new file to its file system. In the case where a file is created in an imported directory R, first the process exporting R adds the new file to its virtual file system, then other processes importing R adds the new file to their file system.

The create method requires three parameters, a file path, file permission, and a file type. The file type parameter defines if the file to be created is a directory or a regular file. File permission has the following possible values, 1 (read), 2 (write), and 3 (read write).

Delete file: The *deleteFile* method allow a user to delete local or remote files. Delete operation follows the same principle as create operation, where if the deleted file is exported or imported, the delete operation is reflected in all file structures accessing the file. Since this operation removes files from file structures, removing a file that is a union directory can lead to orphaned file trees. In the case were delete operation can produce orphan file structures; shadow directory

is used to replace the deleted file in the virtual file system. The shadow directory is generated with the same name and permission as the deleted file.

Copy: The copy operation allows a user to copy a file or directory. This operation follows the same principle as create operation. This method requires two parameters, the source path and the destination path. The user must at least have read permission to the file pointed by the source path, and write permission to the destination path.

Move: The move operation allows users to rename or move files across a file system. Move follows the same principle as delete operation. For example, if a user moves a local file that is exported, the move operation is reflected in the file systems importing the moved file. If a user moves an imported file, the moved operation is reflected in the virtual and native file system of the process exporting the file.

MoveFile method requires two parameters, source and destination path. User executing the operation must have write permission to the file pointed by both the paths. If the source path points to a union directory, only the file structure root by the source path is moved. All remote files or directories bound to the union directory are not moved. Similar to delete operation, move can also lead to orphaned file structures. Shadow directories are used to provide access to files that can become inaccessible due to a move operation.

One restriction is placed by the *moveFile* method; a user cannot utilize this operation to move a mounted file. For example, in figure 2-d, where directory b21 is mounted using replace option, b21 cannot be moved to another directory. However, the user can move contents of the directory, such as subdirectory b31.

Add listeners: The API provides mechanisms for users to be notified when a virtual file system is modified. Using the *addBindListener* method, a user can register an object implementing interface *IBindListener* with the API. In the event of a file or directory being bound or unmounted from the virtual file system, the API notifies all objects registered using *addBindListener* method that the virtual file system has been modified. If a user wants to notified when a file is created or deleted, user can use the *addCreateDelListener*.

5.2 Export Manager

Export and unexport functionality in AFS is provided by the ExportManager class. This class implements two primary tasks: generating export messages for locally shared data, and serving as a repository for export message generated by other processes. The messages generated by this

class are encapsulated in ExportFSDescriptor objects, which include path of the exported file, unihandle to export manager's communication object, and a unique ID, constructed from the shared file's path and unihandle's EOID. EOID is a unique value produced by the M2MI layer for handle objects.

When a process starts sharing a file, the export manager broadcast a message notifying other peers that new data is being shared. Export managers not generating the message, store the message in its repository, allowing a user to view what data is being shared in a network. Figure 6 below shows process A sending a message to process B and C notifying them that new data is now available for import.

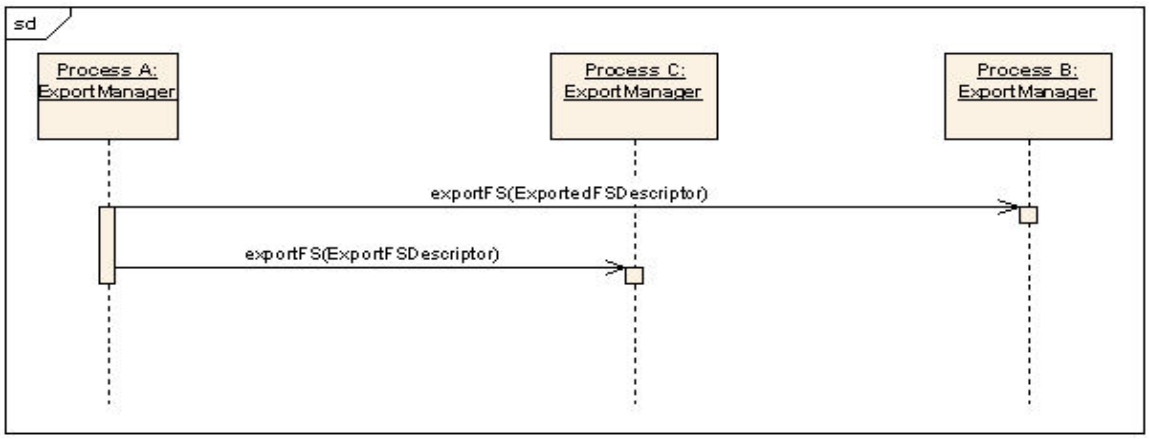


Figure 6: Process A exporting a file.

A process can stop sharing a file by unexporting the file from AFS. When unexporting a file, the ExportManger broadcasts massage intended to inform others that a shared file is no longer available for import. ExportManager receiving this message remove the corresponding saved export message from its Remote Export list. Figure 7 below shows process A notifying others that a shared data is being unexported.

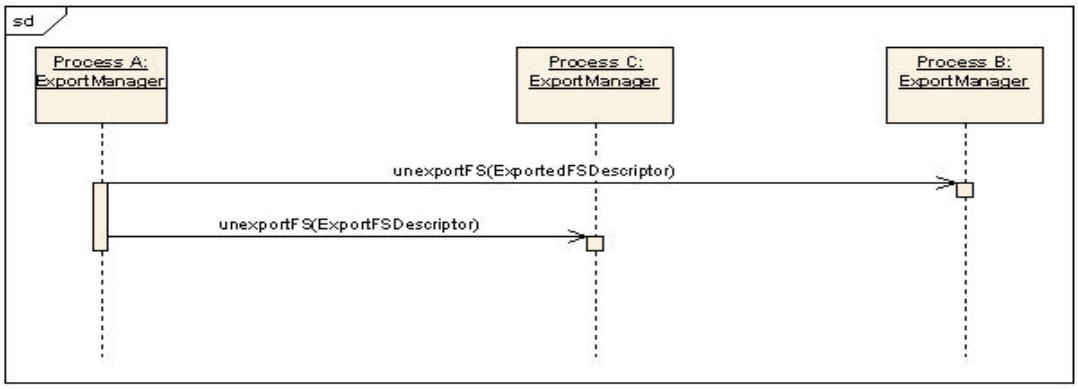


Figure 7: An exported file is being unexported by process A.

5.3 LFSUpdater

The *edu.rit.cs.afs.LFSUpdater* class is responsible for performing all bind and unmount operations. When a bind is executed, LFSUpdater first verifies if a remote process is currently sharing the file identified by the binding point. This verification is performed by ExportManger object by comparing the binding point against stored messages in its Remote Exports list. If a matching export message is found, LFSUpdater proceeds to import the tree structure representing the binding point. The tree structure is generated by the process exporting the remote file, and transported to the process performing the bind operation. If the import process does not successfully receive the tree structure, the bind operation is aborted. Figure 8 below shows process A importing a remote file structure from process B.

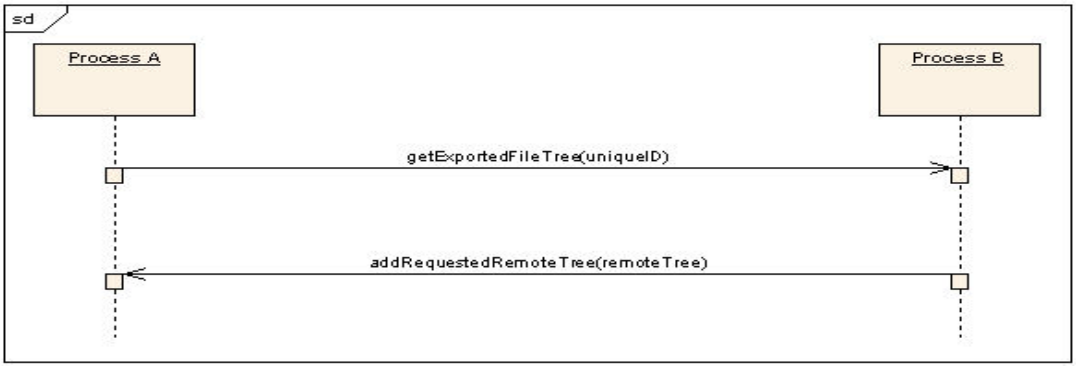


Figure 8: Process A retrieving a remote file structure from process B. The remote file structure is generated in process B.

For umount operation, LFSUpdater removes a remote file structure from the virtual file system. The removed file structure must have been integrated to the virtual file system using a bind call. Suppose, directory R is imported by a process, children of R cannot be unmounted from the file system.

5.4 File Classes

Classes that model regular files, directories, and shadow directories in AFS are BasicFile, LocalFile, RemoteFile and ShadowDirectory; instances of these classes form the virtual file system. BasicFile implements properties and functionalities common to all files, such as a file has name, a path and a permission, a file can have children, can form union directory, and can be removed for the virtual file system. Main functionality provided by instances of this class is a file's ability to create union directory.

During bind operation, the remote file structure imported by LFSUpdater is integrated to virtual file system by BasicFile object. If the bind flag is *Before* or *After*, the remote file is simply added at the front or end of bind point's directory structure. However, importing with bind flag *Replace*, the remote file is superimposed on the binding point temporarily removing the bind point from the virtual file system.

Local File

The LocalFile class, which extends BasicFile, represents files that exist on a device's native file system. Each LocaFile object, which has a corresponding physical file, implements file operation that allows a process to access and modify the file object's underlying physical file. For example, writing to a local file results in array of byte being written to disk. Open, read, write, and closer, are defined for regular files. Create and delete operations are supported for both directories and files.

Unlike, read and write operation, creation and deletion of a LocalFile can affect more than one file system, especially if a parent of a newly created file has been imported into multiple virtual file systems. Thus, when a file is successfully created in an exported directory, parent of the new file notifies other processes that a file has been added to an exported directory. Process importing the exported directory, adds a remote file representative of the new file to their virtual file system. The same is true for delete operation; deleting an exported local file that has been imported by other processes, removes all remote instances of the file.

RemoteFile

During a bind operation, file structure imported by process constitutes of RemoteFile objects, which serves as surrogates for LocalFile objects. RemoteFile class facilitates access and modification to files imported into a virtual file system as it implements the same set of basic file operations as the LocalFile class. When a user performs an operation on a remote file, an object encapsulating the requested operation is forwarded to the process exporting the remote file. The local file corresponding to the remote file executes the operation. For example, when a remote file is written to, a local file in the target process writes an array of bytes to its underlying physical file. However, file operations get a little more complicated with creation and deletion of remote files.

Create and delete operation behaves differently than other file operations, as an exported file might be imported by more than one process. To crate a remote file, parent of the remote file sends a message to its corresponding local file to add a new file to the native file system. If the new file is successful created, the LocalFile object sends a message to other processes, informing them a new file has been created under an exported directory. Processes that have imported the remote directory, generate a new file upon receiving the create message. This ensures that a remote file is only added when its corresponding local file exists. The same holds true when a remote file is deleted. A remote file is only removed from a file system when its corresponding local file has been deleted.

The sequence diagram below shows a remote file created by process A in target process B. First A sends a create operation message to B. In response to a successful creation of the new file, process B sends a message to all other processes notifying them that a new file has been created. Processes importing the exported directory create the new file in its virtual file system.

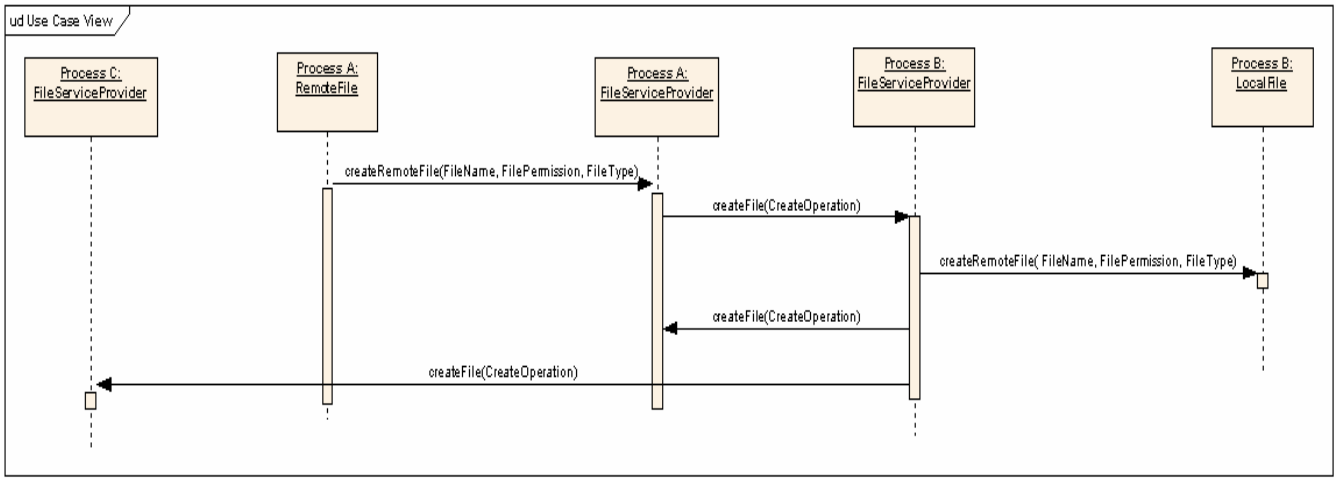
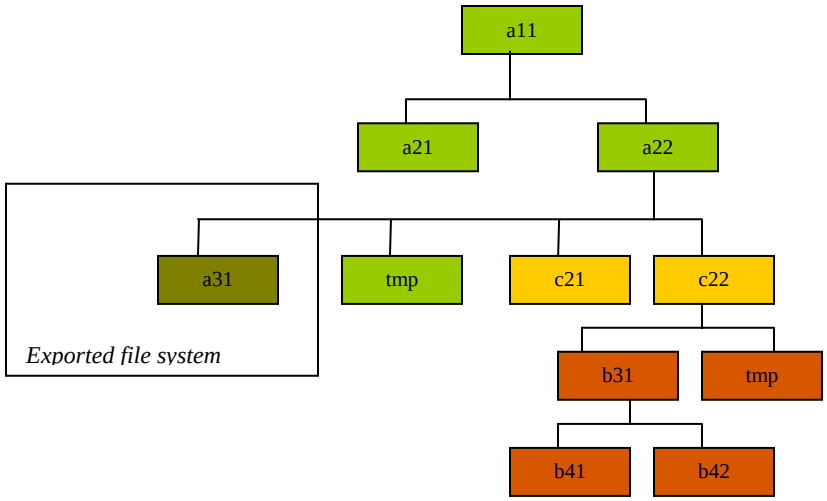
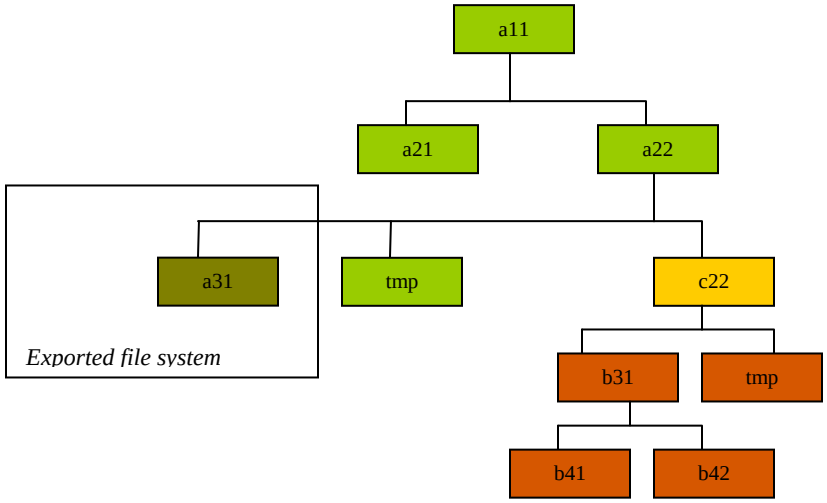


Figure 9: Process A is creates a file in a directory exported by process B. The exported directory is also imported by process C. Once the file is successfully created in B, the remote file representing the new file is created in process A and C.

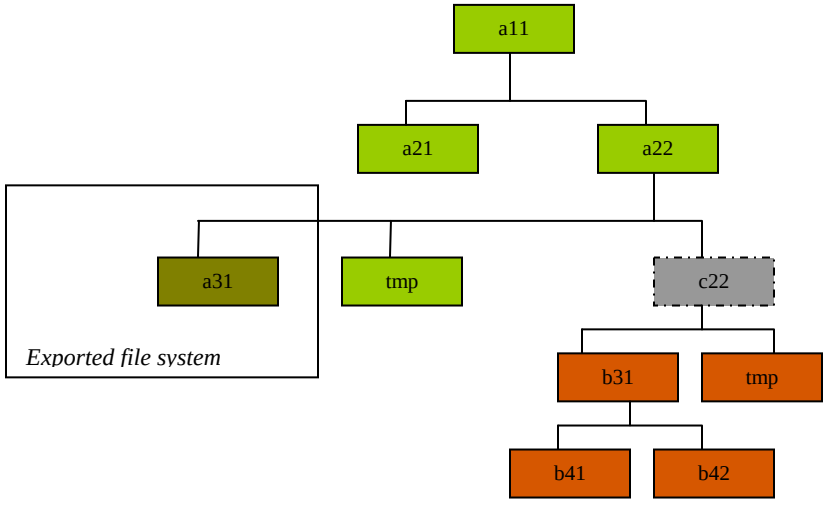
Importing of remote directories raises a question, what happens when a remote dirctory to be unmounted has a child that is a union directory. Take the case in Figure 10 below, unmounting directory *c11* from *a22* will render files *b31* and *b21/tmp* unreachable. To resolve the problem of possible orphaned file structures, AFS utilizes ShadowDirectory objects. During a unmount operation, a remote file first removes children that do not form union directory, as shown in Figure 10-b. Next, children not removed from the file structure are recursively replaced with instances of shadow directory objects, generating a traversable path to possible union files. Figure 10-c shows union directory c22 after being replaced with instance of ShadowDirectory object.



a) Structure of directory a11



b) Children of c11 not forming union directory is deleted.



c) Remote directory c22 replaced with shadow directory

Figure 10: replace operation

ShadowDirectory

ShadowDirectory objects represent the pseudo-file structure generated by AFS to provide access to file that otherwise would become inaccessible in a virtual file system. Object of this class do not support any basic file operations, however they are allowed to form union directories. A unique functionality implemented by ShadowDirectory class is the ability to remove itself from the virtual file system. A shadow directory that is providing access to a remote file is no longer required if the remote file is removed from the virtual file system. For example, in Figure 10-c, when the remote directory b31 is unmounted, directory c22 is no longer required in the tree structure. Thus, when b21 is unmounted, the ShadowDirectory removes itself from the file system.

5.5 LFS

The *edu.rit.cs.afs.tree.LFS* class serves as the interface between the virtual file system and other components in AFS. Classes such as FileOpHandler use the LFS object to gain access to LocalFile or RemoteFile object. Main tasks performed by LFS class are name resolution and delegation of file operations to the appropriate file object.

One of the rudimentary services implemented in a file system is file lookup. Lookup is required in AFS when a user lists a file or executes a file operation. The LFSUpdater supports two form of file lookup, one using absolute path and the other using open file handles. Absolute path lookup is necessary for file operations such as open, create or delete. File handle lookup is required when a user is writing, reading or closing a regular file. A user can obtain a file handle by opening a file, which remains valid as long as the file is kept open. Open file handles are maintained utilizing a simple map, where each handle points to its coreesponding unique file, remote or local.

To look up a file by absolute path, the LFS traverses the virtual file system starting from the root node. Existence of union directory in AFS complicates this task, as an absolute path can be resolved to more than one file, as explored in section 3.3. Thus, LFS searches in parallel multiple branches in the file tree, returning more than one file object associated with an absolute path, if possible. FileOpHandler class in AFS is the main client of this functionality.

Aside from lookup service, the other main responsibility of LFS class is delegating file operations to the appropriate file object, operations that can be initiated by a local or remote process. The FileOpHandler object generally initiates local file operations. In case of a remote process, remote file operations are channeled to LFS from the file communication object. In

either case, once a target file object is identified by LFS, the appropriated method is invoked on the file object to perform the required task.

5.6 Communication

AFS employs two classes, the *edu.rit.cs.afs.comm.ExportServiceProvider* and *edu.rit.cs.afs.FileServiceProvider*, which directly interface with the M2MI layer to facilitate communication between processes running AFS. The first class implements interface *IExportServices* and the second class implements interface *IFileServiceOperations.*

ExportServiceProvider class, as the name suggests, sends and receives messages required for export and un-export operation. Main client of this class are ExportManager and LFSUpdater. Export message generated by ExportManger object is published in the network by this class. During bind operation, LFSUpdater uses this class to import remote file structures. To communicate with other process, *ExportServiceProvider* class exports itself to the M2MI layer, and retrieves a unihandle and an omnihandle for target interface *IExportServices.* The omnihandle allow this class to send to and receive message to all other ExportServiceProviders in a network, and unihandle allow the class to perform one-to-one communication with a specific process.

The *FileServiceProvider* class provides AFS the capability to send and receive file operation messages. RemoteFile and LocalFile objects directly interface with this class, to communicate with target objects in other process. All remote file operations are wrapped in *Operation* object by this class and send to the target process.

File operations supported by this class, can be classified into two categories, blocking and non-blocking. For blocking operations, open and read, this class waits for a predetermined period of time to receive a response for the remote file operation. Open returns a file handle, and read returns bytes read from a file. The non-blocking operations are close, write, create, and delete, in which case AFS does not wait to verify if the file operation was successful. However, special attention needs to be paid when a process reads from or writes to remote file. *FileServiceProvider* class enforces a predetermined limit on number of bytes that can transported be during a single read or write call. This is implemented so that a user cannot perform an arbitrarily long read and write operation. For example, if a read operation reads more than the predetermined limit L, the read bytes are broken into packets of size L before being returned to the target process. Breaking the returned bytes into multiple packets requires a mechanism for assembling the arrived bytes in correct order, as the returned packets can arrive out of order in the

target process. Assembling of returned bytes from read and write operations are provided by the *BufferTracker* class, which is uses a bit map to determine right sequence of the arriving packets.

The Figure below shows, process A writing to a remote file exported by process B. Data written by A is broken into two parts and transported to B. Once the two pieces of data arrive in process B, the data is assembled and written out to the local file.

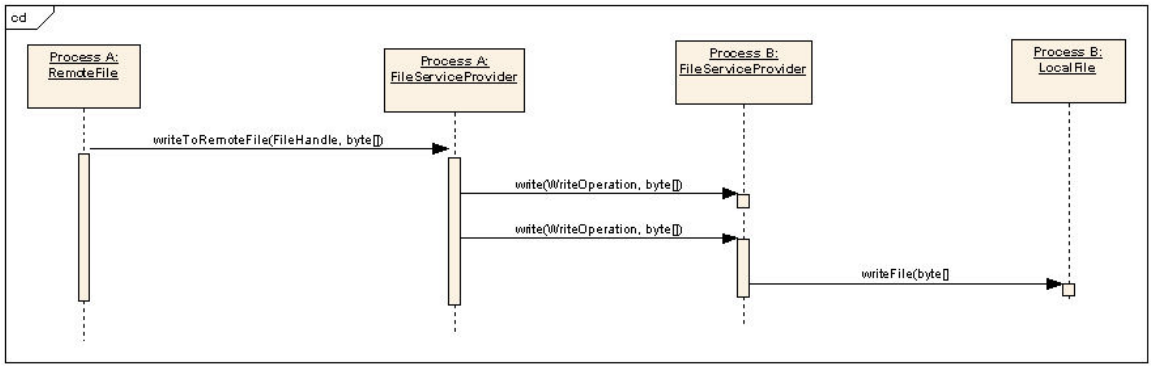


Figure 11: Process A writing to a file imported from process B

6 User Programs

As part of the project two applications have been developed that utilize the Ad-hoc File System. The first, a console application, provides user functionality similar to the Unix shell. The second application, a visualization tool, allows a user to observe how the virtual file system in AFS is modified by bind and umount operations.

6.1 Console

The AFSConsole is a text-based application that allows a user to interface with the API provided by AFS. Commands supported by the console are listed below in table 2.

Command	Description
ls	List file or directory
rqexp	Request export information from remote devices in the network
expinfo	Display export information
bind	Bind a remote file
umount	Un-mount a remote file
echo	Print to the screen
cat	Print content of a file
>	Redirect output
clear	Clear console screen
cd	Change directory
exit	Exit console
cp	Copy file
mv	Move file
Rm	Delete file
mkdir	Make directory

Table 2: Commands supported by AFS Console application.

The application requires two command line parameters. The first parameter is an ID used to identify a process using AFS and the second parameter is an absolute path to a directory. This directory becomes the root node for the file tree maintained by AFS. Only children of the root node are visible to the AFS. After the console is launched, a prompt is displayed and the application waits for user input. Once the user enters a command, a check if made to verify if the command is supported. If so, the appropriate method in file system API is called. Once the

command is completely executed, a result is displayed if necessary and the console prompt is displayed again for user input. If the command is not successfully completed, an error message is displayed.

User guide

Commands in AFSConsole follow similar syntax as a shell under Unix. This specially holds true for cp, mv, mkdir, clear, cd, echo, cat and output redirect commands. A complete list of commands syntax is listed below. This list can also be obtained by typing help at the prompt in the console window.

Command Syntax	Syntax Meaning
ls [-l|-d]	-l use long listing -d display file type, remote or local
rqexp	None
expinfo [device name]	device –name: Name of exporting device
bind <export device> <exported file> <union directory> <bind flag>	export device: Name of device sharing exported file exported file: Name of file to be imported union directory: binding directory bind flag: Bind flag, with possible value 0 – After 1 – Before 2 – Replace
umount <union directory> <umount file>	union directory: directory from which file is unmounted umount file: File to be un-mounted
echo “text”	Text: Text to be displayed
cat <file name>	file name : Content of file to be displayed
Clear	N/A
cd <directory name>	directory name: change to directory
exit	N/A
cp <src file> <dst file>	src file: Source file name dst file: destination file name
mv <src file> <dst file>	src file: Source file name dst file: destination file name
rm <file name>	file name: File to be removed
mkdir <directory name>	directory name: Name of directory

Table 3: Syntax for commands supported by AFS Console application.

6.2 File Viewer

The second application FileViewer, allows the user to visualize the file tree maintained by AFS, and view JPEG images. It supports core functionalities available in AFS, bind, un-mount, export and un-export. This example uses two classes named *edu.rit.cs.asf.test.FileViewer* and *edu.rit.cs.asf.test.FileModel*. The first class displays and handles all user input. The second class interfaces with the file systems API.

The application requires two command line arguments, a process ID, and an absolute path to the directory that serves as the root of AFS file tree. The main window for FileViewer application is shown below (Figure 12). The left side of the window is a scrollable pane that displays the virtual file system visible to AFS. As a user modifies the underlying AFS file tree, the application automatically updates the displayed tree. The right side of the window allows the user to display pictures.

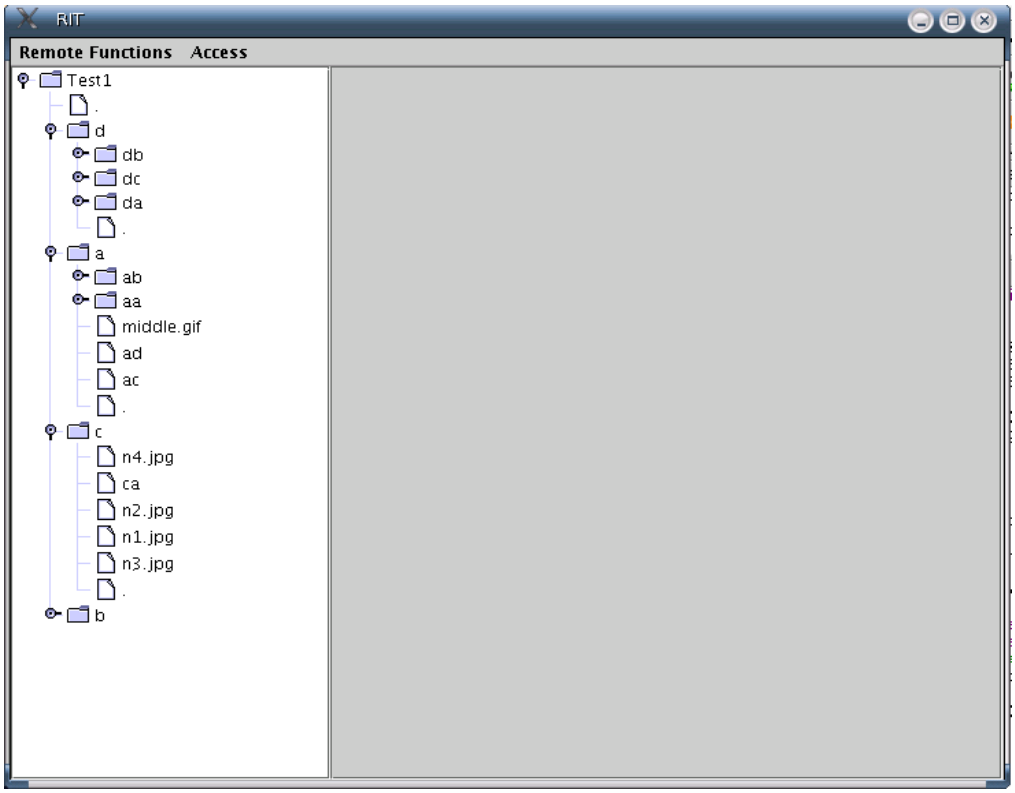


Figure 12: File Viewer User Interface

Different colors are used to depict local, remote and shadow files in this application. Local directories are displayed in blue shade and local regular files are displayed in white shade. The color black is used to indicate shadow directories. All colors other than blue, white, and black are used to display remote files. Colors assigned to remote files are based on process names, with each process being associated with a unique shade. This convention allows a user to differentiate imported files based on processes. Colors to remote files are assigned automatically and are reused once files from more than nine different devices are imported. The figure below shows remote files bound to union directory /Test1/a/aa/aab.

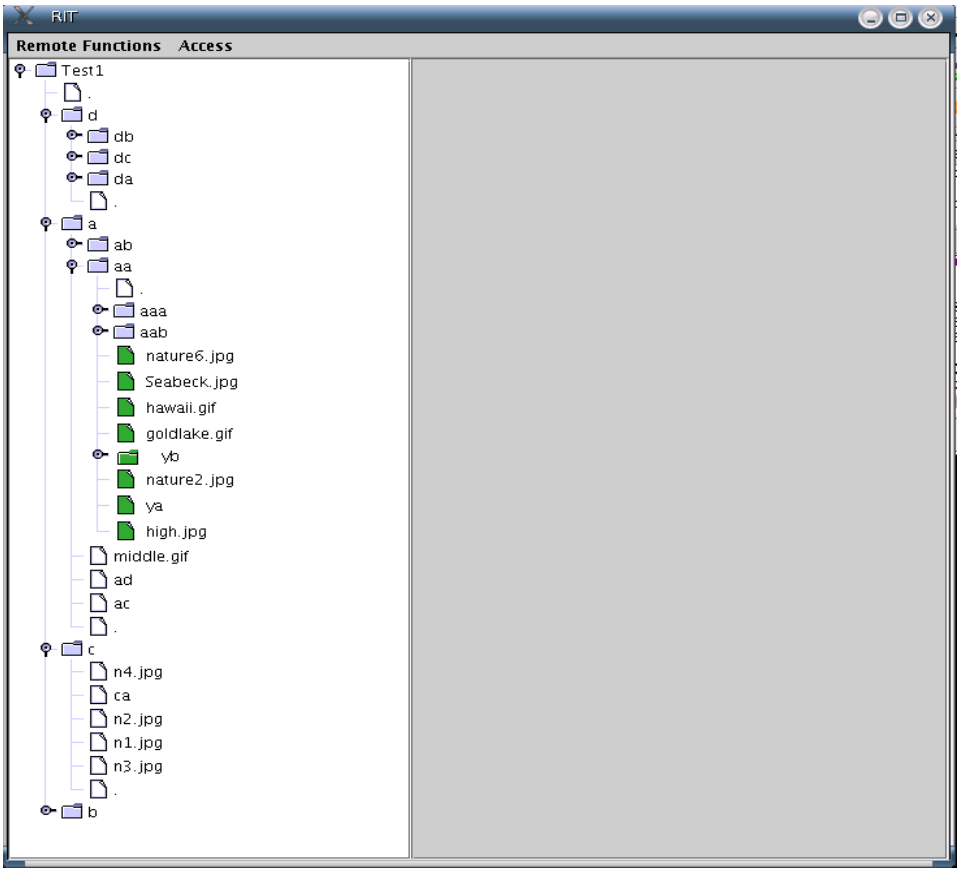


Figure 13: File Viewer Interface with imported files. The imported files are in green.

Usage

All user interaction for the application is provided using two menus, Remote Functions and Access. Remote Functions allow users to export and unexport files, request export information, and bind and unmount remote files. The second menu allows users to display images, which can be local or remote to the device.

To export a file, first select the file click in the left panel. Once a file is selected choose the *Export* menu item from the Remote Functions menu. To unexport a file, select an already exported file in the left panel and then choose the *Unexport* menu item from the Remote Functions menu. If the file selected to be unexported is not already exported, the unexport request is disregarded.

To request export information from other devices, select the *Network Exports* menu item from the Remote Functions menu.

A remote file can be imported by using the *Bind* item from Remote Functions menu. First select a bind point from the file tree in the left panel and then choose *Bind.* A new window, Bind Panel, shall be displayed with a drop down menu and a radio button group. The drop down menu, which lists all shared data in the network, allows the user to choose the file to be imported. The radio button group allows users to specify the bind flag. Once an import file and bind flag are selected, clicking OK in the *Bind Panel* will import the selected file into the device.

To unmount a remote directory, select a union directory and choose the *Unmount* item from the Remote Function menu. A new window, *Unmount Panel*, shall be displayed. The new window provides a drop down menu that contains all remote files bound to the selected union directory. Selecting a remote file from the drop down menu unmounts the file from the selected union directory.

To view an image, select an image file from the AFS file tree, and choose View from Access menu. Upon successfully reading the file, the application will display the image in the right pane of the application window.

7 Ad Hoc File System Comparison with M2MI File System

M2MI File System (MFS) [2] is a stateless distributed file system for sharing and accessing data in an ad hoc network. Similar to AFS, MFS too was developed using M2MI Framework, where processes communicate with each other using handle objects generated by M2MI layer. In MFS, a "virtual file system" is created on top of a device's native file system, allowing processes to share local files and access networked data.

MFS was designed to function as a collaborative file system, where multiple file structures appear as one, referred to as the MFS file system. Each instance of MFS maintains a private file structure. Contents of these local copies, present on separate peers, is merged to represent the unified file system. The MFS file system is not maintained in memory and needs to be built as required, mainly during file lookup. When a process executes the list operation, every process running MFS executes the operation, and the results of multiple list operations are composed together in the process initiating the operation.

The MFS file system is constructed by the client module of MFS, which is responsible for providing a single interface to the middleware. The individual file structures themselves are not maintained by the client module but by a separate component, Data Store. Data Store maintains the files visible in MFS in a non-tree data structure, and provides mappings between MFS files and files in a host's native file system. During a file operation, items in the Data Store are looked up to identify the target files.

File operations supported by MFS are Add, Remove, Read, and List. The Add operation allows a user to add a "virtual" directory or file to the MFS file system. When a user adds a directory, a new entry is created in the Data Store indicating a directory has been added to the files system. When a user adds a regular file, a link is added to the Data Store, which points to a physical file on the local system [2]. Thus, when a user looks up a file in MFS, the Data Store resolves the link between the MFS file and its corresponding physical file.

The remove operation allows a user to remove a file from the MFS file system. Only the user adding a file using Add operation can remove the same file from the file system.

The List operation lets a user access files available on the MFS file system, whether the file is local or remote. By default the operation provide access to all files in MFS, but user can utilize a filter to access a desired directory or regular file. The filter identifies the file or directory of interest. Since list operation combines result from multiple processes, situations can arise where two or more directories with the same name exist in more than one peer. When MFS detects such a case, the contents of those directories with same name are merged together and appear under a

single directory. MFS also allows two different regular files with the same name to exist under the same file system. Contents of files returned by list operation can be accessed using Read as a stream of bytes.

File System Comparison

Comparing Ad hoc File System and M2MI File System, illustrates some similarities and differences. Both the file systems utilize M2MI and sit on top of the M2MI layer, which provides their communication capabilities. Both middleware maintain a virtual file system to allow process to share and access networked data. The major difference between the two distributed file systems is the collaborative nature of MFS. Each process using AFS is provided an instance of a local file system, visible to the process. File residing in remote devices are not transparent to a process and remote files can only be accessed after importing. In MFS, files constituting a local file system are visible to all the process.

Differences also exists on the level of how files are shared by the two middleware. AFS provides the export function as a mechanism for users to start and stop sharing data. The export function also provide users the ability to inquire other peers for shared information. Analogous to the export and unexport function in AFS are the Add and Remove operations in MFS. Files added in MFS are stored locally where export information is stored by all processes running AFS. Once information is shared in MFS, the List operation permits users to access remote data. List in MFS composes all information regarding shared files, and provides user access to the file of interest. Meanwhile, in AFS processes are required to use a Bind operation to access remote data. An advantage of binding remote file is the ability to dynamically change the structure of the virtual file system, as discussed earlier. Once remote data is imported, AFS allow a process to create, delete, read from and write to a remote file. A file operation on a remote file is visible to the process exporting the file and also to all processes importing the remote file. MFS only allows users to read from a local or remote file. Files visible in the MFS file system cannot be deleted or modified in any nature by the user.

Feature	MFS	AFS
File System	MFS is a collaborative file system. Each host maintains a virtual file system, contains of these file structures are merged together to generate MFS.	Each host maintains an individual virtual file system.

Feature	MFS	AFS
	Files part of the virtual file system visible to any host running MFS.	Files in virtual file system have to be export in order for a host to access remote data. This allows a host to explicitly indicate which file other hosts can access using import mechanism.
	File part of the virtual file system is accessible to any host running MFS.	Exported files have to be imported into virtual file system in order for a user to access a specific remote file.
	The virtual file system is not maintained in a tree like structure	Virtual file system is maintained in a tree like data structure.
	N/A	File can be unmounted from the virtual file system.
	The collaborative file structure is not maintained in memory. MFS is built each time a user executes the List operation.	The virtual file system is built as required and maintained in memory.
	N/A	Remote files part of a virtual file system, which become unavailable due to change in the network topology, are automatically added to the virtual file system again once become available.
File Access	File existing on a host can be added to MFS using Add operation.	Files existing on a host are automatically added to the virtual file system.
	Files in MFS can be removed from MFS using Remove operation, where is remove file is no longer part of the virtual file system. The removed file is not deleted from the host.	File on virtual file system can be deleted. The deleted file is removed for the host's native file system.
	Files cannot be created or deleted in MFS.	Files can be created on the virtual file system. The created file is also generated on the host's native file system.
	Files in MFS can be read using Read operation.	Files in AFS can be read using read operation
	File cannot be written to in MFS.	Files, local or remote, can be written to in AFS.
	Files cannot be copied or moved in MFS.	Files can be copied and moved across file systems in AFS.

Table 4: Comparison of features available in MFS and AFS.

8 Future Work

The Ad Hoc File System permits users to perform basic functions found in current distributed file systems, such as accessing and modifying remote data. However, additional functionality is desirable to make AFS sounder, such as synchronization.

Many viable distributed file systems provide a locking mechanism, which is used to avoid the adverse effect of multiple processes accessing the same file. AFS does not employ a file-locking mechanism; so multiple users could potentially modify a file simultaneously. A locking mechanism based on leasing can be implemented in AFS. A process wishing to access a file can be required to acquire a lock on the file, with the lock being leased for fixed period of time. If a lock's lease is not renewed within the fixed period, the lock will be released. Another case where synchronization is desirable is to let two processes P1 and P2 import the same remote directory R into their virtual file system. After importing R, P2 leaves the network for a while, during which time P1 creates a file in R. When P2 joins the network again, the newly created file will not be visible to P2.

9 Conclusion

The AFS middleware enable peers to share, retrieve, and modify networked data in an ad hoc fashion. Two fundamental issues arising from lack of both central authority and fixed infrastructure, the ability to discover shared data and the need to gracefully handle remote data due to change in network topology, are addressed in AFS by employing the Export function and a pseudo-file structure using Shadow Directory. Exporting data to the AFS middleware allow a process to share files with other processes in a reliable and robust manner. This mechanism removes the need for prior knowledge of shared files and directories. A device can leave a network for period of time and still be able to access data exported during its absence from the network. Shadow Directories address the issue of gracefully handling imported data that can become inaccessible due to changes in network topology, to which an ad hoc network is very susceptible. The pseudo-file structure prevents the virtual file system in AFS from becoming disconnected.

10 Acknowledgments

I want to thank Prof. Hans-Peter Bischof for his invaluable guidance and support in this project, and Prof. James E. Heliotis for his comments and corrections on the Project Report. I also would like to thank Prof. Alan R. Kaminsky and Prof. Roxanne Canosa for their assistance.

11 References

[1] Bischof, H.and Kaminsky, A., "Many-to-Many Invocation: A new framework for building collaborative applications in ad hoc networks". CSCW 2002 Workshop on Ad Hoc Communication and Collaboration in Ubiquitous Computing Environments, New Orleans, Louisiana, USA, November 2002.

[2] Bhatia, R., "MFS: M2MI File System". MS Project Report, Rochester Institute of Technology, Department of Computer Science, April 2004, *http://www.cs.rit.edu:8080/ms/static/ark/2003/2/rnb1914/index.html*, Retrieved September 2005.