2006

MFS: M2MI file system

Ravi Bhatia

Masters Project Report

MFS: M2MI File System

Committee:

Prof. Alan Kaminsky, Chair	Date
Prof. Hans-Peter Bischof, Reader	Date
Prof. Rajendra Raj, Observer	Date

Ravi Narain Bhatia
Department of Computer Science
Rochester Institute of Technology

Abstract

The M2MI File System (MFS) is a Virtual File System designed for ad hoc networks. It uses M2MI/M2MP[1] to perform ad hoc communication. The file system comprises of several file system instances spread across the network. Each file system instance furnishes a partial state, contributing to the complete structure of the file system. Hence MFS is a union of several file system instances collaborating with each other. The file system facilitates four file operations: ADD, REMOVE, LIST and READ. The operation ADD allows a user to import a file from the local file system into MFS. It also permits the creation of MFS directories. The REMOVE operation helps remove MFS files and directories. Using the LIST operation, users can view the files available on MFS. The READ operation enables a user to read a MFS file. The M2MI File System also caters to the usefulness of mobile code deployment. MFS allows users to deploy mobile code on remote file system instances. This code has access to the MFS files that are a part of an individual file system instance. Mobile code deployment helps process files remotely. Hence the objective of this Masters project is to enable the sharing of file resources in an ad hoc environment by means of a file system.

Table of Contents

1. Introduction

The M2MI File System Masters project was an attempt to create a Virtual File System for ad hoc networks. An ad hoc network is a network whose participants are in proximal range. A group of individuals forming their own wireless network to communicate with mobile computing devices is an example of an ad hoc network. The topology of an ad hoc network is undefined. To build a file system for such a network was the purpose of this Master's project.

A file system is a platform that allows resources to be shared among its users. A distributed file system facilitates sharing of resources in a distributed network. An example of a local file system is the Microsoft Disk Operating System (MS-DOS)[2]. The Sun Network File System (NFS)[3] is an example of a distributed file system. A file system manages resources like data stored on the hard drive, network sockets and even serial and parallel port communication. It considers these resources as files, allowing users to read or write to them.

During the course of developing this project, conventional distributed file system architectures were studied. These file systems follow the client–server style of communication, since this paradigm suits most intranets. In such file systems, most files are located on a central server. A client computer uses a Client Module to communicate with the server computer hosting the files. The server computer contains the Access Control Module and Directory Service that perform the chunk of a file operation. The Access Control Module manages access rights, while the Directory Service maintains information about a file mapped to a unique identifier. These file systems also employ techniques like caching to improve performance. Matured over the years, these file systems are well suited for todays organizational networks. However, they cannot be used conveniently for ad hoc networks.

Hence this project addresses the need to build a file system that could be used for ad hoc networks. It tries to combine required features of conventional file systems with ad hoc computing paradigms. The M2MI File System is a distributed file system that builds on top of the local file system and uses M2MI to share files in an ad hoc manner. M2MI and M2MP were developed for communication between peers in an ad hoc network. M2MI is an invocation mechanism that uses Many-to-Many Protocol (M2MP)[1,]. M2MP is a protocol that directly uses a broadcast medium for inter-peer communication. M2MI and M2MP are part of the Anhinga[1] Project.

2. Architecture

2.1 The Anhinga Project

The Anhinga[1] Project is developing "a new distributed computing infrastructure designed specifically to support collaborative applications running on wireless ad hoc networks of mobile computing devices"[1].

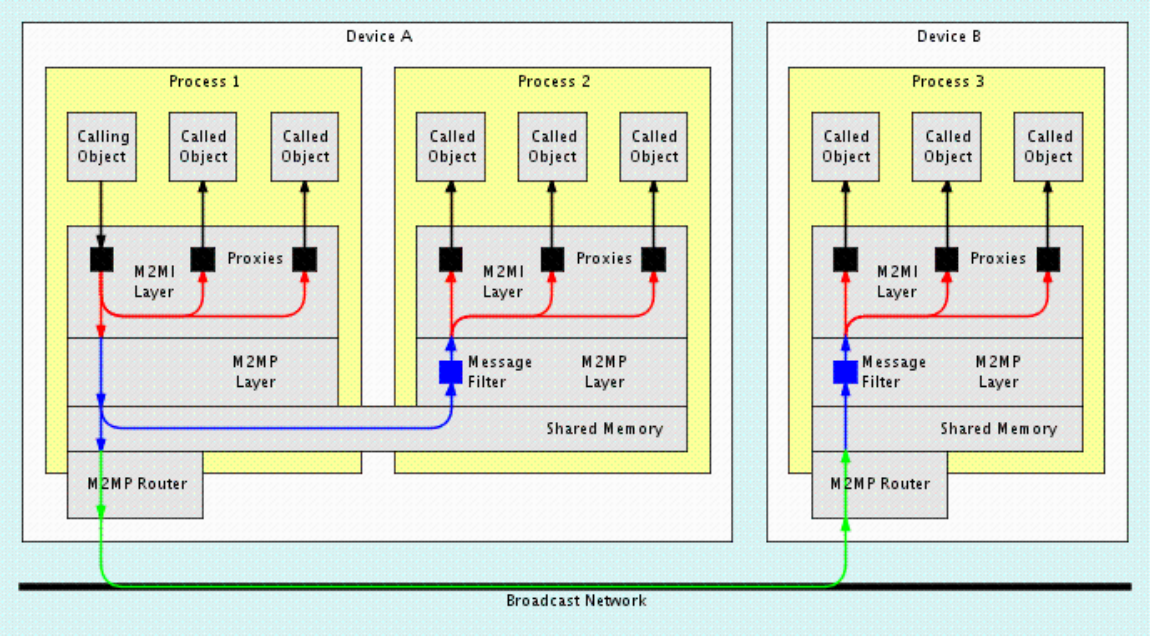

Figure 1: Anhinga Overview[1]

The above figure shows an overview of the Anhinga infrastructure. The diagram shows two mobile devices, Device A and Device B participating in an ad hoc communication. The M2MI and M2MP layer are shown. Messages are broadcasted using M2MP packets. The M2MP Layer has Message Filters registered by applications that choose to use the M2MP Layer for communication. A Message Filter only allows a M2MP packet to be delivered to the application that registered it, if the header of the M2MP packet matches that of the filter. The M2MI Layer registers a Message Filter with the M2MP Layer. A invocation by an object will be broadcasted to all M2MP Layers except the one that sent it. Proxies in the M2MI Layer perform the invocations. Also, the message is partitioned into packets, sent over the broadcast medium, and assembled at the destination. The M2MP Layer can exist on top of any broadcast medium like IEEE 802.11 or Ethernet.

2.2 Characteristics

MFS achieves its distributed nature using M2MI. Data between the peers is transferred using M2MP. Conventional distributed file systems use Remote Procedure Call (RPC)[4]. The M2MI File System exists as a collection of file system instances, spread across the ad hoc network. An application on a peer can spawn a file system instance, such that it shares its local file resources in an ad hoc fashion. Each file system instance talks to another file system instance using M2MI. This collective collaboration forms this ad hoc file system. Conventional distributed file systems use a client-server collaboration paradigm, however such a model fails in an ad hoc network since ad hoc networks are server-less. Hence each instance of MFS is responsible for itself and collaborates with other instances to form the total file system.

MFS is designed for the following file operations:

- Add: Make a file available to all MFS users. An added file is visible to all MFS users.
- Remove: Make a file unavailable to all MFS users. Once a file is removed it is no longer visible to MFS users.
- List: List the available MFS files. Allows users to view the available MFS files.
- Read: Read a available MFS file. Allows a user to read the contents of a MFS file, if the user has the adequate permissions.

The file system has the following characteristics:

- A directory is a file that contains no data, but helps divide the available MFS files into organized tree structures.
- The *root* is a directory, which represents the root of all tree directory structures. The *root* is represented by `/`.
- A MFS file or directory name builds from the *root*. For example, a directory `Alpha` that exists in root will be named `/Alpha/`. All directory names end with `/`. A directory `Beta` that exists in `Alpha` will be named `/Alpha/Beta/`. A file `theta` that exists in `Alpha` will be named `/Alpha/theta`. File names do not end with `/`.
- If two users create a directory with the same name, the directories merge along with their contents. For example, consider a user Charlie that makes a directory named `Reports` containing the file `juneFinancialReport`. Now user Mark creates a directory with the same name `Reports` containing a file `juneSalesReport`. When another user Mary performs a *List* operation, she sees a directory named `Reports` containing the files `juneFinancialReport` and `juneSalesReport`.
- A directory can only be removed if it is empty.
- Files named with same name cannot exist on a individual file system instance, however there can be two files with the same name on MFS.

Mobile code deployment is possible in the M2MI File System. A user in the ad hoc network can deploy code on all file system instances in the network by transferring this code over the network. Such mobile code is also known as an Agent. Agents in the field of computing serve several purposes. In MFS they serve the purpose of processing large remote files. It may be more useful to transfer processing instructions over to a peer that contains a large file rather that transfer the large file for processing. An agent could contain processing instructions that would process a file at its remote location and send only results back to the peer that deployed the agent. An user who wishes to write an agent extends the `edu.rit.cs.mfs.resource.Agent`

class and adds it to the file system as an MFS file. This file should have read and execute permissions. The user asks all file system instances in the network to read and execute the agent. An agent has access to MFS files available on the file system instance that instantiated it. The agent can read and process a file, sending results back to the user that deployed the agent.

The M2MI File System has a simple File Permission Model. A permission for a MFS file is defined by assigning the file:

- An owner, which could be *Individual* or *All.*
- Access rights, which could be *read* or *execute* or both.

Each MFS file has an owner. A MFS file could be owned by the individual user who added the file or it could be owned by all the users in the network. Hence *Individual* and *All* are two types of ownerships. Next the type of access rights have to be associated with the owner. The *read* permission grants the owner the access to read a MFS file. Hence only if a MFS file is associated with an *All* ownership and having *read* permission, can it be read by all users in the network. The *execute* permission is defined for agents. If an agent is to be deployed on all peers in the network, it needs to have *read* and *execute* permission along with an *All* ownership.

2.3 Component View

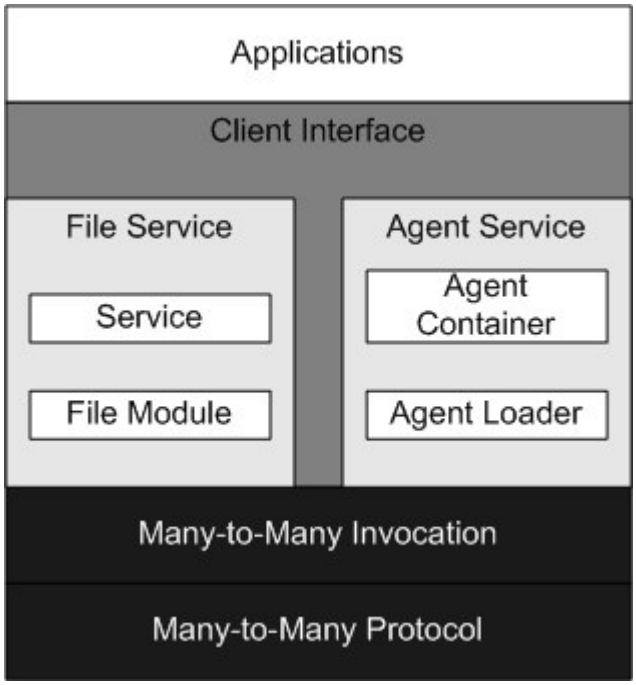

Figure 2 MFS Layer Diagram

MFS consists of these components:

Client Interface
An application that wishes to use the M2MI File System interfaces with the Client Interface. Though the M2MI File System is a set of file system instances existing on various mobile devices, the Client Interface portrays this set as a single file system. The Client Interface provides a single interface for all the functionality provided by the M2MI File System. The Client Interface allows an application to perform the following operations:

- Add, allowing the application to make local files available to the MFS.
- Remove, to remove a file from the MFS.
- List, to view MFS files made available by all network participants.
- Read, to read the contents of a MFS file from another device.
- Execute, for agent deployment.

The application that wishes to use MFS encapsulates an instance of the Client Interface. The Client Interface encapsulates the File and Agent Service.

The File Service
The File Service is responsible for performing file operations. The file operations are:

- Add: Add a file into MFS.
- Remove: Remove a file from MFS.
- List: List the available MFS files.
- Read: Read a available MFS file.

The file service identifies each MFS file with a unique identifier called the **File Identifier** (FID).

The file service uses a Data Store that maintains a mapping between the local filename and the name of the file as it is known in MFS. It is also maintains the permissions of the file.

Unlike conventional client-server based distributed file systems, there is no central server, allowing a client to seek information. Each application on the network has a file service. All such instances of the file service work together to build the total state of the file system. The total state of the system is defined as the organization of all the files available in the ad hoc network. This organization is defined by directories. Each file service maintains a partial structure of the file system. A partial structure corresponds to the files that an application has made available to the file service. An aggregation of all partial states builds the total state of the file system. A file is made available by linking it to its location on the local file system.

MFS is a stateless file system. This means that the total state of the system, is never stored. The hierarchical file organization is built only when the *List* file operation is performed. This stateless model has its own advantages and disadvantages. Being stateless suits the dynamic ad hoc network topology. Maintaining a state requires maintenance. This maintenance relates to keeping consistency using collaborative and synchronization techniques. Being stateless is more simple, which comes at the cost of time and efficiency. Each time the *List* operation is performed the state has to be computed. However, since the state is newly computed, a peer that has left the network or a peer that has joined the network contributes to the state of the file system.

Directories in MFS are maintained unlike conventional file systems. MFS file services that contain a directory with the same name, cause the contents of all such directories to be merged in the aggregated total state of the file system. Hence two directories named *X* on different peers, one having a file *y* and the other *z*, result in the directory *X* containing both *y* and *z* for the total state of the system.

The File Service comprises of two sections as shown in the figure below:

- **The Service Interface** that listens to the Client Interface. This interface broadcasts a request from the Client Interface to all File Module Interfaces residing on all instances of the file service over the ad hoc network. The results of the request are sent back to the Service Interface, which are unified and sent to the application. This interface processes *Add*, *Remove*, *List* and *Read* operations. This interface is realized by the `edu.rit.cs.mfs.file.FileServiceImplementation` class.
- **The File Module Interface** listens to requests from any Service Interface. A *List* or *Read* request to the Service Interface is broadcasted to all File Module Interface modules. The File Module Interface is only concerned about the files, its file system instance makes available to the MFS. This module only process *List* and *Read* operations. This module is realized by the `edu.rit.cs.mfs.file.FileModuleImplementation` class.

This division suits the ad hoc nature of the file service. The File Module Interface is responsible for the collaborative operations that can be performed over the ad hoc network, while the Service Interface is responsible for operations that can be performed by an application. The Service Interface and File Module Interface help create a secure divide. The Service Interface permits an application to *Add* and *Remove*, since only the application can access the Service Interface. The File Module Interface is accessible universally and allows the Service Interface to *List* or *Read* its files. The Service Interface is responsible for unifying the results it obtains from the all the participants of the ad hoc network.

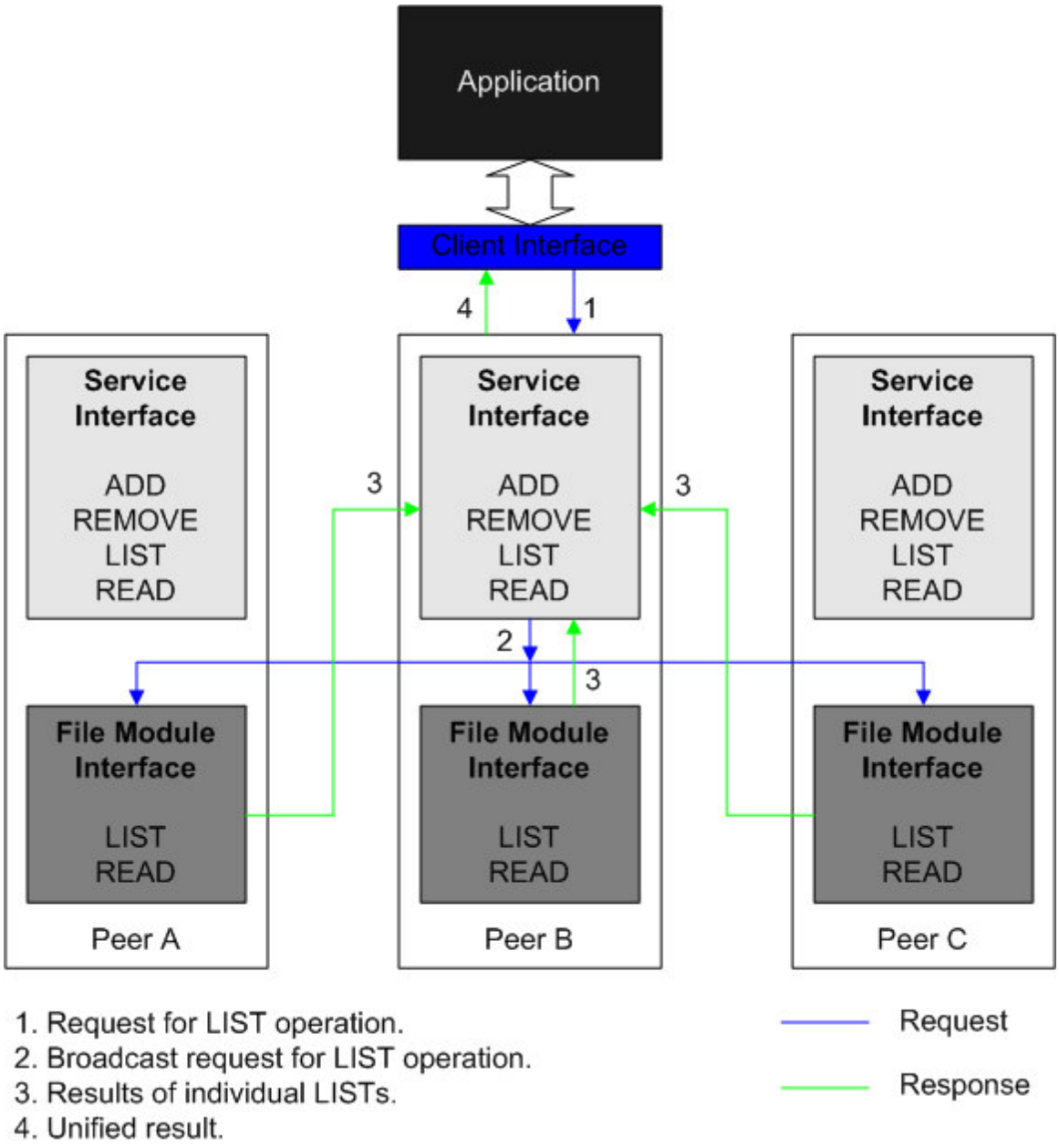

Figure 3 File Service Components

Agent Service

The file system allows an application to deploy mobile code on all instances of the file system that have enabled their Agent Service. An application that wishes to deploy an agent defines its location as a MFS file. The agent class file is read by all Agent Service enabled instances of the file system from the location defined by the application. Once each instance reads the agents class file, an agent object is instantiated. The agent object is able to communicate with the application that deployed it. Agents are instantiated under strict control, with the help of the Java Security Manager[5]. An agent is only able to list and read the files made available by an instance of the file system. Bi-directional communication between the agent and the application that deployed it is possible.

The Agent Service has two components:

- Agent Loader
- Agent Container

When an application wishes to deploy an agent it informs its Client Instance. The Client Interface broadcasts a message to all file system instances asking them to load the agent. If the agent service is enabled for an file system instance, the Agent Loader steps in and reads the agents class file from the applications defined location. Since this class file is available as a MFS file, the M2MI File System is used to transport it. Once the Agent Loader receives the agent class file, an agent object is instantiated. The instantiated agent is initialized with the reference of the application that deployed it and a reference of the Agent Services Agent Container. The Agent Container has methods that allow the agent object to *list* and *read* files available on its file system instance. The Agent Container also allows the agent to define a reference that can be used by any remote application to communicate with the agent.

2.4 Users and Application Programs

The MFS serves two audiences. One that are developers that wish to build applications on top of the MFS infrastructure. The MFS provides an API for developers to use. The other are users that wish to share their files in an ad hoc fashion. The Console and HTTP daemon are available applications that allow users to share their files. External applications that wish to read MFS files would have to do so using the Console or HTTP daemon.

- The MFS Console allows users to perform all MFS file operations. With a set of commands users can
    - Build directories and add files, allowing other users to make use of them.
    - Change directories, such that users can move within the file structure.
    - List available files, enabling a user to view files made available by other users.
    - Saving, Opening or Streaming MFS files. These three are variations of the read operation. Saving a file allows a user to fetch and save a MFS file to the local file system. Opening a file enables the user to fetch and view the fetched file in a local application. Streaming a file permits a user to fetch and pipe the contents of a MFS file to a local process.

The HTTP daemon allows a browser to view file listings and fetch any MFS file. The browser can choose to open the fetched file in a suitable application.

3. Developing Applications

All interaction with MFS is done though the class `edu.rit.cs.mfs.service.MFS`. A developer that wishes to perform file operations uses an instance of this class.

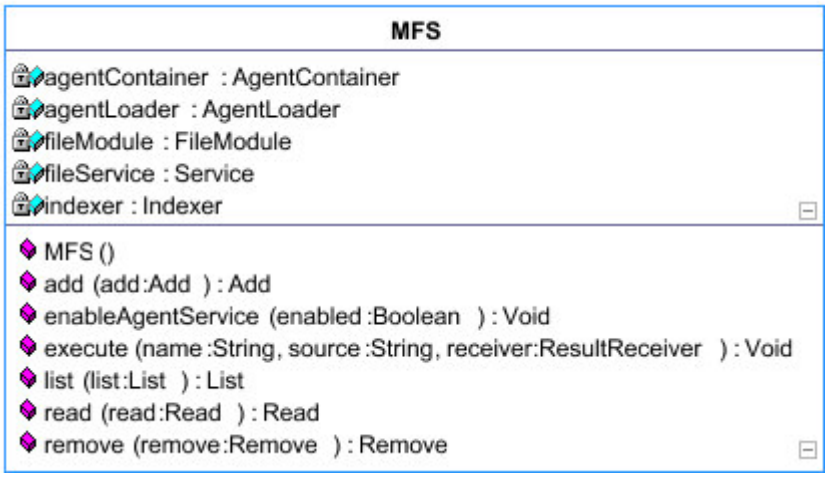

Figure 4: edu.rit.cs.mfs.service.MFS

The methods *add*, *list*, *read* and *remove* allow a user to perform file operations. A file operation is defined by an object whose classes are found in the `edu.rit.cs.mfs.operation` package. Each operation encapsulates the parameters required to complete the operation. It also contains the results of the operation once the operation is complete. Hence a user who wishes to perform the *add* operation, instantiates an object of the type `edu.rit.cs.mfs.operation.Add` with the required parameters like the name of the new file or directory. This object is passed to the `MFS.add()` method, which passes it to the File Service. The File Service performs the operation and returns the object. The results of the operation are encapsulated in this object. In this case, the success of the operation is its result. This object is now returned to the user by `MFS.add()` method.

A user should be aware of the following methods in the `edu.rit.cs.mfs.service.MFS` class to perform file operations:

- `MFS.add()`
- `MFS.remove()`
- `MFS.list()`
- `MFS.read()`

However, before performing file operations, one should understand how a file is represented in MFS.

3.1 The MFS File

The `edu.rit.cs.mfs.resource.File` class is the object representation of a MFS file. A user will not have to instantiate an object of this class. `File` objects are instantiated by the File Service as a result of a file operation.

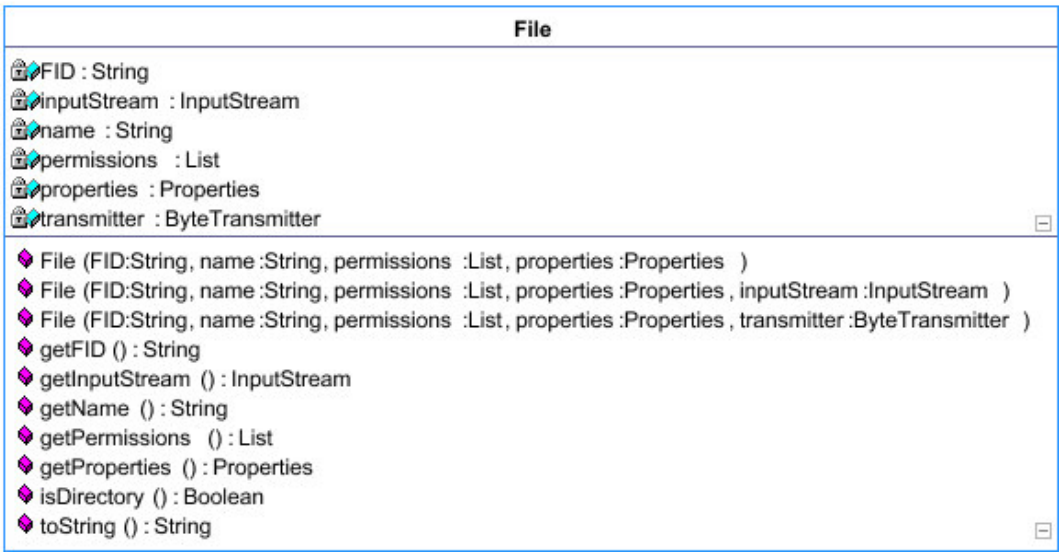

Figure 5: edu.rit.cs.mfs.resource.File

A `File` object always encapsulates its:

- File Identifier (FID). `File.getFID()` returns the file's unique identifier.
- Permissions. `File.getPermissions()` returns the permissions associated with the file.
- Properties. `File.getProperties()` returns any properties combined with the file.

The `File` object has different constructors to encapsulate separate sets of data for different operations. The *List* operation uses the first constructor, hence the result of the operation would contain `File` objects that contain the above mentioned properties. The *Read* operation uses either the second or third constructor. The second constructor is used when a file has to be read from a remote location and the third constructor is used when the file has to be read locally by agents.

The following properties are defined for an MFS File:

Key	***Value Type***	***Definition***
`File.SIZE`	`java.lang.Long`	Defines the size in bytes of a readable MFS file.
`File.HASH`	`java.lang.String`	Hash value. For a readable MFS file the MFS path name and its contents are hashed. For an MFS directory the path name is hashed.

3.2 An Operation

The `edu.rit.cs.mfs.operation.Operation` class is the base class for all file operations. All file operations are sub-classes of this class. As a user, you would not use this class.

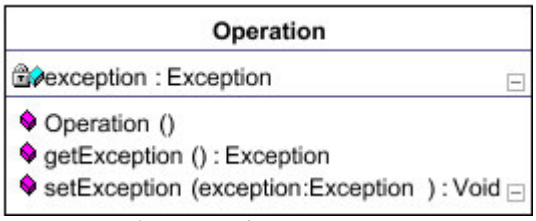

Figure 6: edu.rit.cs.mfs.operation.Operation

Its has two methods:

- `Operation.setException()`: This method allows the File Service to set an exception for a file operation.
- `Operation.getException()`: Allows the user to read the exception set by the File Service, while performing a file operation.

When the File Service encounters an error in processing a file operation it sets the exception into the operation using `Operation.setException()`. A user can retrieve this exception using `Operation.getException()`.

3.3 Add Operation

The *Add* operation allows a user to add a file. This operation has two constructors.

- Add a directory. Allows the user to create a directory with a set of permissions.
- Add a file. This constructor links the MFS file with a file found on the local system. It also allows the user to associate a set of permissions with the file. The user can define a set of desired properties which should be associated with the file.

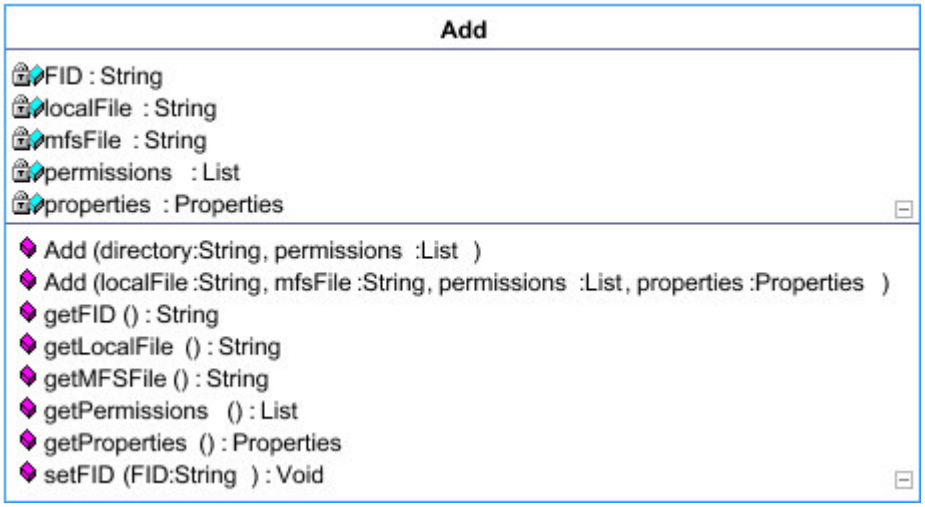

Figure 7: edu.rit.cs.mfs.operation.Add

Once the `MFS.add()` method has been called with an instance of the `Add` object, the `Add` object is passed to the File Service. The File Service makes a new entry in the MFS Data Store with the information provided by the `Add` object. A File Identifier (FID) is assigned for the new file and set into the `Add` object using `Add.setFID()`. Once the object is returned by the `MFS.add()` method, the user can obtain the FID, using `Add.getFID()`.

Example:

```
MFS mfs = new MFS();
java.util.List permissions = new ArrayList();
permissions.add(new Permission(User.ALL, true, false));


//Adding a directory.
Add operationAdd = new Add("/Report/", permissions);
operationAdd = mfs.add(operationAdd);

if (operationAdd.getFID() == null) {
      System.out.println(“Unable to add directory.”);
      System.exit(1);
}


//Adding a file.
operationAdd =
new Add("expense.doc",
“/Report/expense.doc",
permissions, new Properties());
operationAdd = mfs.add(operationAdd);

if (operationAdd.getFID() == null) {
      System.err.println("Unable to add file.");
      System.exit(1);
}
```

3.4 Remove Operation

The *Remove* operation allows a user to remove a file from the file system. Only a file that has been added by the user using *Add* can be removed using the *Remove* operation.

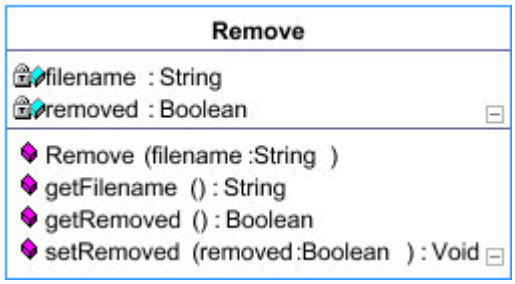

Figure 8: edu.rit.cs.mfs.operation.Remove

The *Remove* operation constructor requires the name of the MFS file, which could be a file or a directory. If the operation is successful, the `Remove.getRemoved()` method returns `true`.

Example:

```
MFS mfs = new MFS();
Remove operationRemove =
new Remove("/Report/expense.pdf");
operationRemove = mfs.remove(operationRemove);

System.out.println("Removed: " + operationRemoved.getRemoved());
```

3.5 List Operation

The list operation allows any user to obtain the files available on the MFS file system.

Figure 9:
edu.rit.cs.mfs.operation.List

It is important to know the use of the `List.setFilter()` method. The argument of this method is a MFS file or directory name. If a filter is not set, the `MFS.list()` method will return a `List` object containing all files available on the MFS file system. If the filter represents a MFS file, the result of this operation will only contain this file, if available. If the filter is a directory, the result of this operation will contain the tree (files and directories) which has this directory as its root. The results of the *List* operation are contained as a `java.util.List` of `File` objects, retrievable from `List.getFiles()`.

Example:
Fetching all MFS files:

```
MFS mfs = new MFS();
List operationList = new List();
operationList = mfs.list(operationList);

java.util.List files = operationList.getFiles();
java.util.Iterator iterator = files.iterator();
while (iterator.hasNext()) {
      System.out.println(files.next());
}
```

Fetching MFS files in the directory `/Reports/June/`:

```
MFS mfs = new MFS();
List operationList = new List();
operationList.setFilter("/Reports/June/");
operationList = mfs.list(operationList);

java.util.List files = operationList.getFiles();
java.util.Iterator iterator = files.iterator();
while (iterator.hasNext()) {
      System.out.println(files.next());
}
```

3.6 Read Operation

The *Read* operation allows a user to read the contents of a MFS file as a stream of bytes. The file should provide the user with the necessary permissions to read the file.

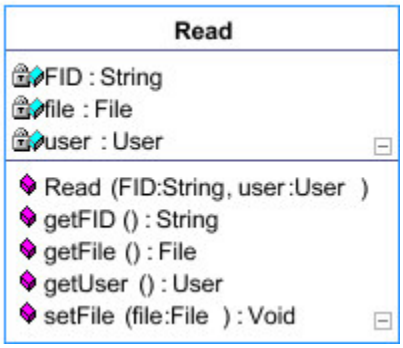

Figure 10:
edu.rit.cs.mfs.operation.Read

To initialize a *Read* object, the user should have the file's FID, the user wishes to read. Also, the user should provide the ownership, the user represents. The result of this operation is obtained using the `Read.getFile()` method. The `File` object returned by this method contains the `java.io.InputStream` that can be used to read the contents of the file. To obtain the `java.io.InputStream` use the `File.getInputStream()` method.

Example:

```
MFS mfs = new MFS();


//Perform operation list to get the file FID.
List operationList = new List();
operationList.setFilter("/Report/expense.pdf");
operationList = mfs.list(operationList);


//Check if the file exists.
java.util.List files = operationList.getFiles();
if ( files.size() == 1 ) {
      File file = files.get(0);
} else {
      System.out.println("Unable to fetch file.");
      System.exit(1);
}


//Perform operation read.
Read operationRead =
new Read(file.getFID(), new User(User.ALL, null));
File file = operationRead.getFile();


//Read the contents of the file
java.io.InputStream inputStream = file.getInputStream();
```

3.7 Using Agents

This section discusses on writing and deploying agents. An agent is code that could be migrated and deployed on a remote machine. The MFS allows simple agent deployment such that a user wanting to process a large remote file can do so without transferring the entire contents of the file. A user who wishes to deploy an agent does so by writing an application that deploys the agent on all remote file system processes. A file system process contains an Agent Loader and an Agent Container. The Agent Loader is responsible to load an agent after fetching the code from the application that wishes to deploy the agent. An agent can only access the files made available by the Agent Container's file system process. The agent uses the Agent Container's methods to access these files.

Writing an agent is simple. A user who wishes to write an agent should extend the class `edu.rit.cs.mfs.resource.Agent`.

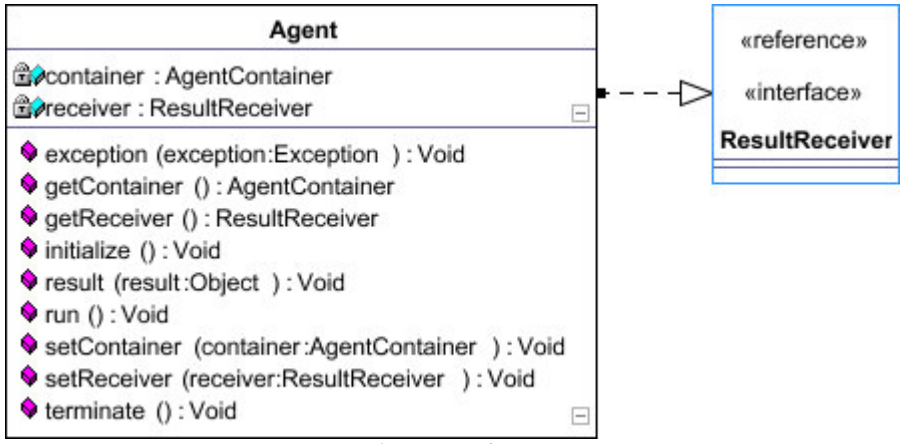

Figure 11: edu.rit.cs.mfs.resource.Agent

An MFS Agent has a life cycle:

- `Agent.setContainer()`
- `Agent.setReceiver()`
- `Agent.initialize()`
- `Agent.run()`
- `Agent.terminate()`

A user should only override `Agent.initialize()`, `Agent.run()` and `Agent.terminate()`. The `Agent.initialize()` should contain initialization code, while the `Agent.run()` must contain the main processing instructions. The `Agent.terminate()` would contain clean up code. This method is certain to be called even if `Agent.intialize()` or `Agent.run()` fail.

The `Agent.setContainer()` and `Agent.setReceiver()` are set by the Agent Loader, such that the agent could use these values from the `Agent.getContainer()` and `Agent.getReceiver()` respectively. The Agent Loader sets the Agent Container's reference using the `Agent.setContainer()` to allow the agent perform *List* and *Read* operations.

An application that wishes to deploy an agent, implements the `edu.rit.cs.mfs.agent.ResultReceiver` interface. This interface is responsible for communication between the application and the agent. For an agent to send processing results to this application, it has to have the `ResultReceiver` reference of the application. For an application to talk to its agent, it needs the `ResultReceiver` reference of the the agent. The agent gets the `ResultReceiver` reference of the application from `Agent.getReceiver()`. Using the Agent Container method `AgentContainer.getReceiverReference()`, an agent can obtain its `ResultReceiver` reference. This reference will have to be sent to the application.

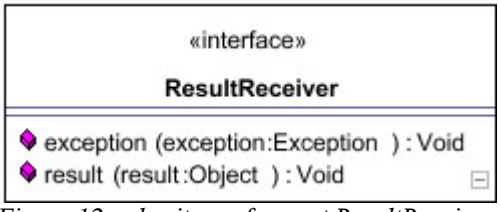

Figure 12: edu.rit.cs.mfs.agent.ResultReceiver

The `edu.rit.cs.mfs.agent.ResultReceiver` interface contains two methods. One to send results and the other to send exceptions to the receiver. The argument of `ResultReceiver.result()` should be a object that can be serialized.

An agent is added as an MFS file. An agent file should have *read* and *execute* permission with an *All* ownership to permit deployment. The application which wishes to deploy the agent calls the `edu.rit.cs.mfs.service.MFS.execute()` method. This method takes the full quantified name of class, its source location and the reference of the application that wishes to receive results from the instantiated agent. Hence if an agent is called `edu.rit.MyAgent` and exists in the MFS directory `/Agents/`, then the full quantified name of the agent would be `edu.rit.MyAgent` and `/Agents/` its source location. The MFS filename of the agent would be `/Agents/edu/rit/MyAgent.class`. If your application demands more classes then they would also need to exist as MFS files. Hence if the `edu.rit.MyAgent` class uses the `edu.rit.Helper` class, the `edu.rit.Helper` class would need to exist as `/Agents/edu/rit/Helper.class` in MFS. Note that for each source location a separate Class Loader is assigned.

Example:
See Section 5.4

4. Design

4.1 M2MI Interfaces and Sequence Diagrams

4.1.1 `edu.rit.cs.mfs.service.Service` and `edu.rit.cs.mfs.file.FileModule`

The `edu.rit.cs.mfs.service.Service` interface is responsible for representing a service. A class that implements this interface can broadcast a query message across the network and wait to receive results. The broadcasted message contains a reference (M2MI Unihandle) of this interface. A module that processes this broadcasted message, uses this reference to send back results. As shown in the figure below, this interface is implemented by `edu.rit.cs.mfs.file.FileServiceImplementation`.

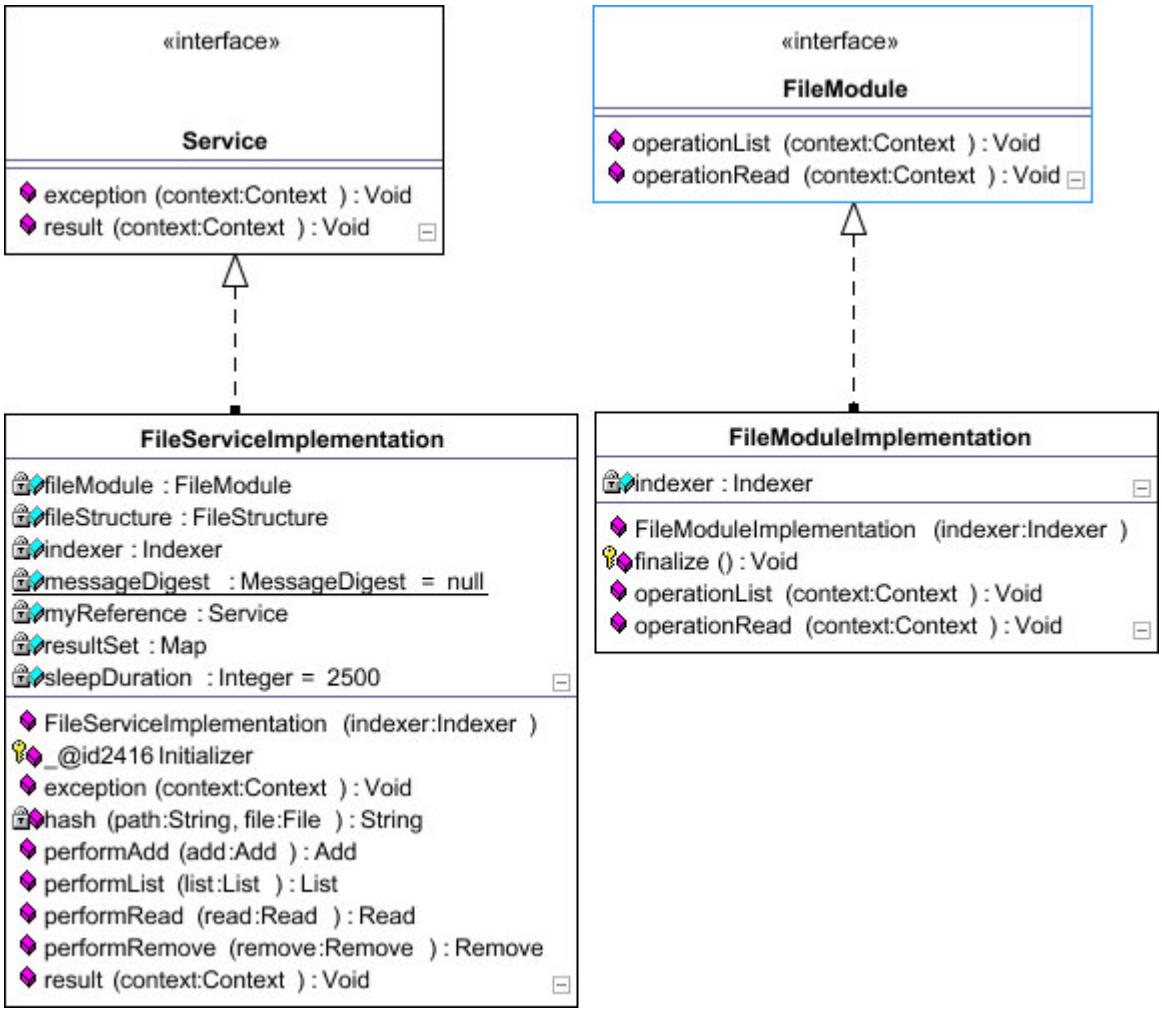

Figure 13 edu.rit.cs.mfs.service.Service and edu.rit.cs.mfs.file.FileModule

The edu.rit.cs.mfs.file.FileModule represents a module that can process file operation queries made by a Service object. Hence a Service object makes requests to all FileModule objects in the network, while a FileModule object reports the result of a request to a Service object. The FileModule interface is implemented by the edu.rit.cs.mfs.file.FileModuleImplementation class.

Also note that edu.rit.cs.mfs.resource.Context objects are used to facilitate messaging between the Service and FileModule objects. The Context object can pack the M2MI Unihandle reference of the Service object along with the file operation (edu.rit.cs.mfs.operation.Operation) that has to be performed by a FileModule object.

The above diagram shows that this interface has only two methods. One to receive exceptions and the other to receive results. The FileServiceImplementation broadcasts a file operation to all edu.rit.cs.mfs.file.FileModuleImplementation objects. The file operation is packed in the container object, edu.rit.cs.mfs.resource.Context. This container object also holds the M2MI Service Unihandle of the FileServiceImplementation object. Once the FileModule processes the file operation, it sends back the results contained in the same Context object to FileServiceImplementation. The results from all the FileModule objects are then processed by FileServiceImplementation. This unified result is passed to the user.

The diagram below shows the flow sequence when a user performs a *List* operation using edu.rit.cs.mfs.service.MFS.list(). The sequence diagram shows two peers Alpha and Beta. The user creates a MFS file system instance on Alpha. The MFS.list() method takes an edu.rit.cs.mfs.operation.List object which represents a *List* operation. The MFS object instance asks the FileServiceImplementation to perform the *List* operation by passing it the List object. The FileServiceImplementation implements edu.rit.cs.mfs.service.Service. It packs the List object in a edu.rit.cs.mfs.resource.Context object and broadcasts it to all edu.rit.cs.mfs.file.FileModule instances. The Context object also has the M2MI Service Unihandle for the FileServiceImplementation object.

In the figure FileModule instances on Alpha and Beta receive the Context object. The FileModule process the *List* operation by publishing its MFS files into the List object. The Context container objects containing the List operation objects are sent back to the FileServiceImplementation using FileServiceImplementation.result(). The FileServiceImplementation unifies the results from the List objects received from FileModules at Alpha and Beta. A List object containing the unified results is returned to the user.

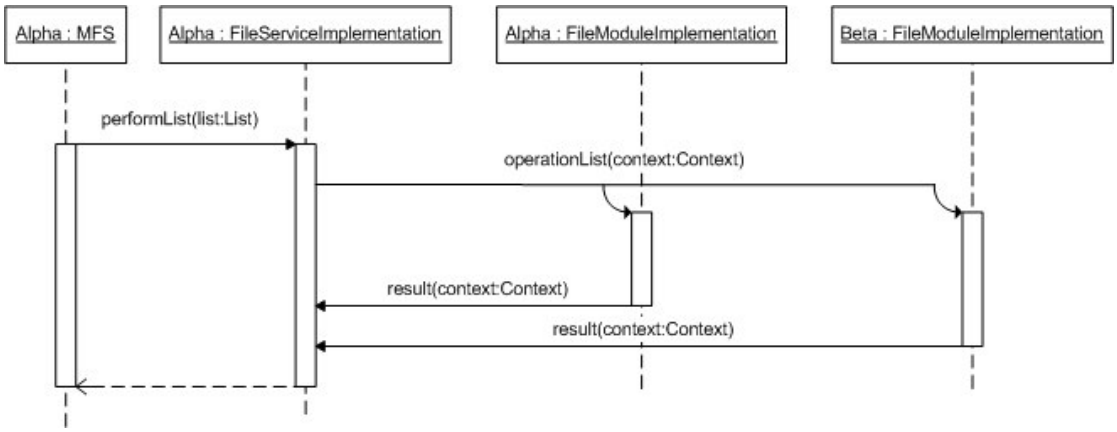

Figure 14: Performing the List operation.

4.1.2 `edu.rit.cs.mfs.resource.ByteReceiver` and `edu.rit.cs.mfs.resource.ByteTransmitter`

The `ByteTransmitter` and `ByteReceiver` are responsible to set up a protocol for transferring the contents of a file from one file system instance to another. Hence they allow users to read remote files. The `ByteTransmitter` is implemented by `edu.rit.cs.mfs.resource.InputChannel` and `ByteReceiver` by `edu.rit.cs.mfs.resource.InputChannelProxy`.

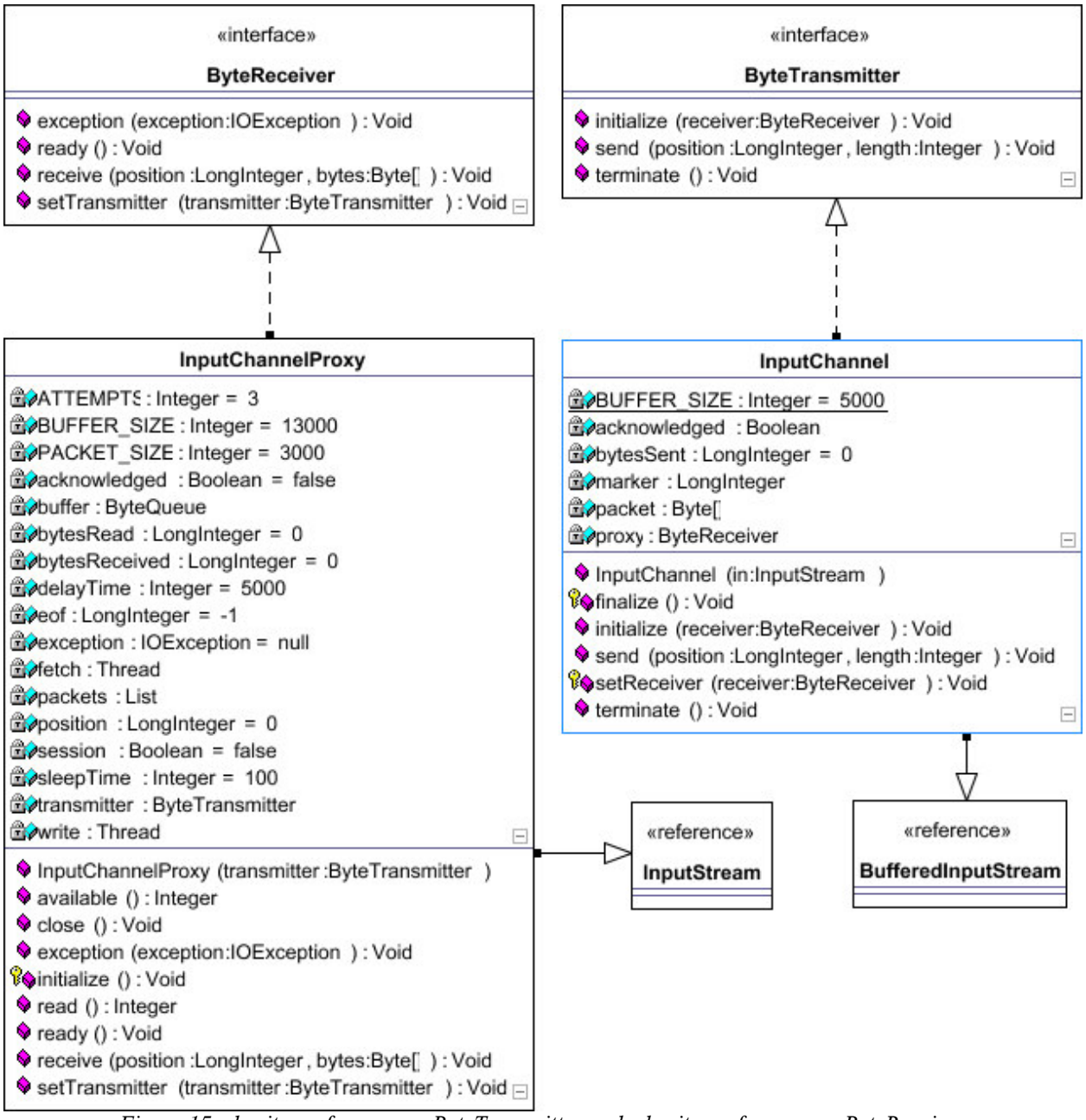

Figure 15 edu.rit.cs.mfs.resource.ByteTransmitter and edu.rit.cs.mfs.resource.ByteReceiver

The `InputChannelProxy` is a `java.io.InputStream` which can be used by the user to read a remote file while the `InputChannel` is a `java.io.BufferedInputStream` which buffers the remote file to be read. The transfer of bytes from the `InputChannel` to the `InputChannelProxy` is defined by the protocol explained below.

The sequence diagram below shows how data in a file is transferred from one peer to another. Consider two peers, the *Host* and the *Reader*. The *Host* contains a MFS file `report.doc` that is available to all network users to read. The *Reader* is a peer that wishes to read this file. The *Reader* uses the *Read* operation to obtain the MFS file from the *Host*.

Once the *Read* operation is performed the *Reader* has an instance of the MFS file from which it can obtain a `java.io.InputStream` to read the file. The *Reader* uses the `File.getInputStream()` to obtain the `java.io.InputStream`. Using this input stream, the *Reader* will be able to read the contents of the file. This input stream is an instance of `InputChannelProxy`, which implements `ByteReceiver`. Hence the *Reader* has a `ByteReceiver` since it wishes to read the bytes of the MFS file `report.doc`.

The *Host* has an `InputChannel` which is a `java.io.BufferedInputStream`. A buffered input stream is an input stream that buffers contents from a contained input stream. In this case, this contained input stream is a `java.io.FileInputStream` that reads the contents of the file `report.doc`, which is available at *Host*. Note that `InputChannel` implements `ByteTransmitter` since it wishes to transmit bytes from the file `report.doc` to `InputChannelProxy`.

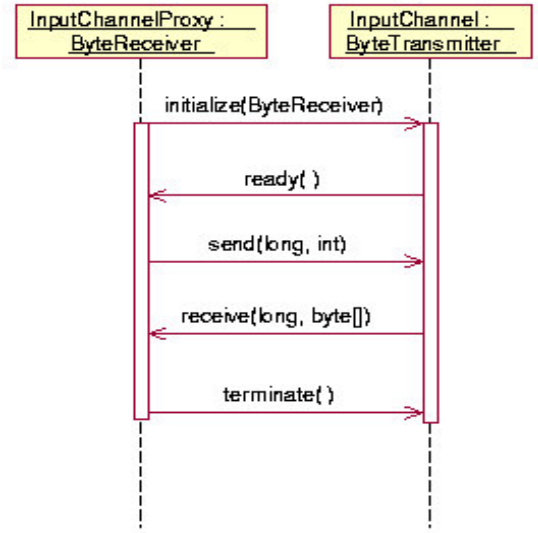

Figure 16 Protocol for Data Transfer

Hence the *Reader* has a `ByteReceiver`, while the *Host* has a `ByteTransmitter`. To initiate the transfer, the `ByteReceiver` *initializes* the `ByteTransmitter` with its reference. In response, the `ByteTransmitter` informs the `ByteReceiver` that it is *ready*. The `ByteReceiver` can now read bytes from the file `report.doc`. It asks the

`ByteTransmitter` to *send* it a set of bytes. The `ByteTransmitter.send` method takes two parameters. One, the position in the file from which it wishes the bytes are read, and second, the number of bytes that are to be read from that position. The `ByteTransmitter` creates a byte array containing bytes from the file `report.doc`, read from the specified position. The size of this array is defined by the number of bytes that are to be read. If the bytes available at the *Host* are less than the specified length, the `ByteTransmitter` creates an array whose size is equal to the number of the bytes available at the host. The `ByteTransmitter` maintains a marker that records its current read position in the file. The `ByteTransmitter` now asks the `ByteReceiver` to *receive* the packed byte array, along with the new marked position. The `ByteReceiver` can acknowledge the end of a file when it receives a byte array of ZERO length and no increase in the marked position. The transfer can be terminated or closed at any time by calling `ByteTransmitter.terminate()`.

4.1.3 `edu.rit.cs.mfs.agent.AgentLoader`

This interface enables agents to be instantiated on remote file system instances.

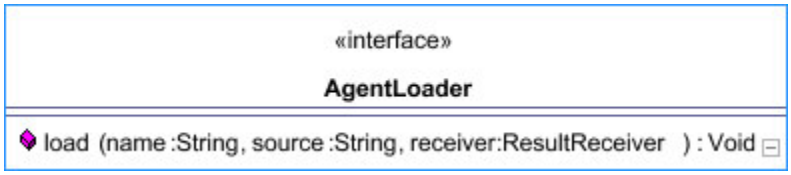

Figure 17: edu.rit.cs.mfs.agent.AgentLoader

The following sequence diagram shows how agents are loaded on to remote file system instances. Consider two peers Alpha and Beta forming an ad hoc network. An application on Alpha wishes to execute an agent on Beta. Beta has an enabled Agent Service. The application uses a MFS file system instance (`edu.rit.cs.mfs.service.MFS`) to execute an agent. Each MFS instance contains an `AgentLoader`, if the Agent Service is enabled. The applications MFS instance broadcasts a request to all `AgentLoader`s to execute the agent. Once an `AgentLoader` receives the request it instantiates and initializes the agent. Since the application implements `edu.rit.cs.mfs.agent.ResultReceiver` it can receive results from the instantiated agent.

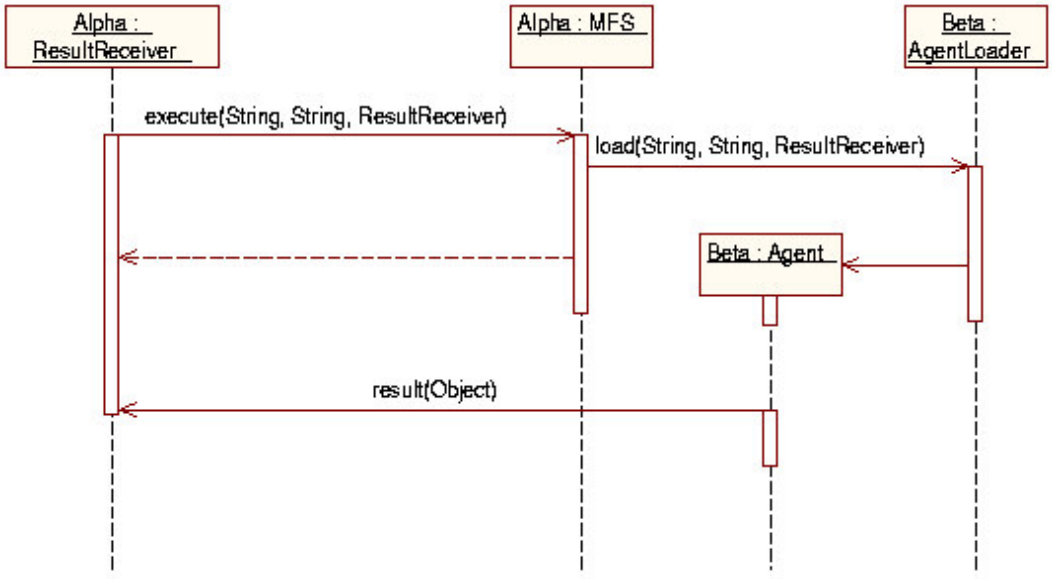

Figure 18: Agent Loading Mechanism

4.2 The Console

The MFS Console is a text based shell that allows a user to input commands to perform file operations in MFS. The Console encapsulates an instance of the MFS file system. Each command that can be executed in the console is represented by a separate class that extends `edu.rit.cs.mfs.console.Command`. The `edu.rit.cs.mfs.console.Command` class encapsulates the arguments passed to the command and the reference of the Console. A reference of the Console allows commands to print to the Console and set/get Console properties. A class that represents a command follows a particular naming convention, where the name of command is placed after the name `Command`. Hence the Add command would be called `edu.rit.cs.mfs.console.CommandAdd`. The Console loads these classes at run time depending on the command issued by the user.

The Console provides a prompt that waits for the user to input a command. Once a command is entered, it instantiates the appropriate command class passing the arguments that were provided along with the command. For example, let us assume that the user has just issued the following command: `add juneReport.pdf /Reports/june.pdf`. This command statement consists of two sections,

- Command: `add`
- Arguments: `juneReport.pdf /Reports/june.pdf`

The console loads the `edu.rit.cs.mfs.console.CommandAdd` class passing it the arguments and its reference. The `edu.rit.cs.mfs.console.CommandAdd` class can now process the arguments to perform the *Add* operation.

4.3 The HTTP Daemon

The figure below shows the design and work flow of the HTTP daemon. When the browser connects to the `HTTPServer`, the `HTTPServer` class intializes:

- `HTTPConfig`: Responsible to hold the HTTP daemons configuration information which it reads from the `httpd.properties` file.
- `HTTPLog`: To log HTTP requests and errors.
- `MimeConverter`: Assigns a file type based on the requested documents extension.

The `HTTPConfig`, `HTTPLog` and `MimeConverter` exist independent of other classes such that they can be referenced from other classes.

The `HTTPRequest` is created by the `HTTPServer` to serve an incoming HTTP request. The `HTTPRequest` parses the HTTP request headers which are encapsulated in `HTTPMessageHeaders`. CGI execution information is stored in the `HTTPInformation` object. Based on the HTTP request, one of the following `HTTPHandler` objects will be created:

- `HTTPGetHandler`: HTTP Get Request
- `HTTPHeadHandler`: HTTP Head Request
- `HTTPPostHandler`: HTTP Post Request

The `HTTPHandler` creates `HTTPResponse` and `HTTPObject`. The following `HTTPObject` objects can be created:

- `HTTPFileObject`: HTTP request for a file or directory listing.
- `HTTPProcessObject`: HTTP request for a CGI program execution.

The `HTTPResponse` object will contain response entity headers like `Content-type` added by the `HTTPObject`.

The browser now receives a response based on the information in `HTTPResponse` and `HTTPObject`.

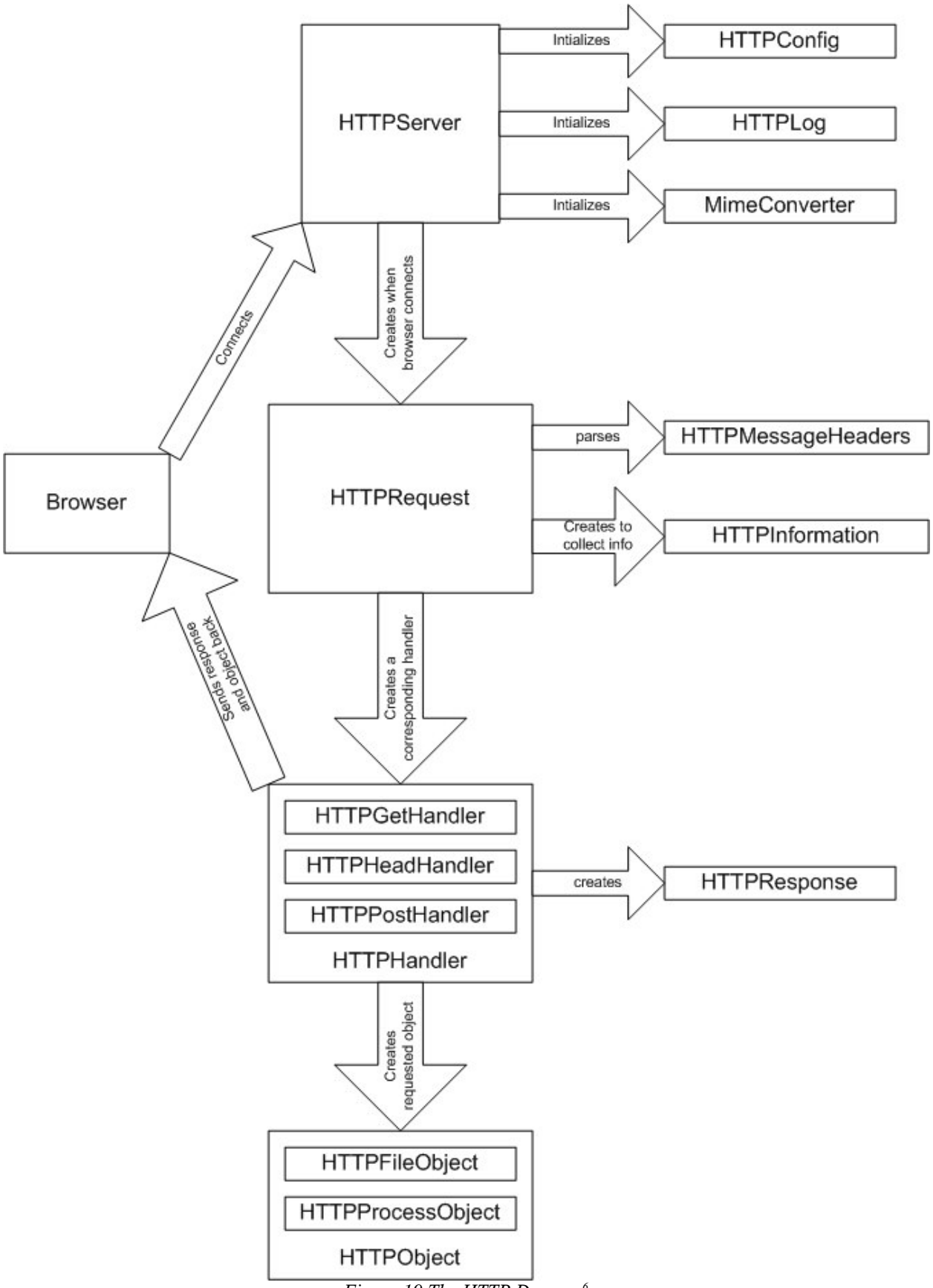

Figure 19 The HTTP Daemon[6]

4.4 The Score Board Application

The Score Board Application is created to show the mobile code (Agent) deployment capability of the M2MI File System. The application demonstrates a Score Board that displays scores awarded by two judges for a group of contestants. To explain the application let us consider three peers: *judge1*, *judge2* and *Host*. The two judges ONE and TWO reside on peers *judge1* and *judge2* respectively. Each judge maintains a score file. Judge ONE and TWO maintain the respective MFS files `/Contest/judge1.txt` and `/Contest/judge2.txt`. Each file contains the score of all the contestants awarded by an individual judge. A score file looks like:
Daffy 54
Dexter 77
Tweety 66

The Score Board Application consists of two components:

- The agent program that is supposed to read the file `/Contestant/judge1.txt` or `/Contestant/judge2.txt` that contains the score for each contestant.
- The main program that displays the Score Board and deploys the agent. The agent program sends results back to this main program for display.

The agent program is called `ContestantFetch` and the main program is called `ContestantScoreBoard`.

Before the `ContestantScoreBoard` program can be executed each judge ONE and TWO should have added their score files as `/Contest/judge1.txt` and `/Contest/judge2.txt` respectively. Once the `ConstestantScoreBoard` program is executed it adds the `ContestantFetch` agent to the MFS directory `/Contest/`. Hence the agent program now exists in MFS as `/Contest/ContestantFetch.class`. Now the `ContestantScoreBoard` application asks all peers on the network to execute the agent by calling `MFS.execute()`.

The agent is instantiated on peers *judge1*, *judge2* and *Host*. The agent looks for either `/Contestant/judge1.txt` or `/Contestant/judge2.txt`. Once it finds either file, it starts to read the file continuously like the Unix `tail` command. The agent parses each entry and sends them to the `ContestantScoreBoard` program. Once it reaches the end of the file it waits until a new entry has been added by a judge. The `ContestantScoreBoard` program maintains a table to which it adds each entry that it receives from an agent.

4.5 Classes

MFS is structured into a set of Java packages:
- `edu.rit.cs.mfs.resource`: Core data structures.
- `edu.rit.cs.mfs.service`: The Client Interface.
- `edu.rit.cs.mfs.file`: The File Service.
- `edu.rit.cs.mfs.operation`: The file operations.
- `edu.rit.cs.mfs.agent`: The Agent Service
- `edu.rit.cs.mfs.util`: Utilities

MFS Applications:
- `edu.rit.cs.mfs.console`: The MFS console application.
- `com.wrox.httpserver`: The HTTP Daemon.
- The Score Board Application.

4.5.1 `edu.rit.cs.mfs.resource`

Let us begin with the `edu.rit.cs.mfs.resource` package. The figure below shows the class diagram of this package. These classes represent the core data structures used for MFS. The `edu.rit.cs.mfs.resource` classes:
- `File`: Responsible for representing a MFS file. This class has separate constructors to encapsulate separate sets of data. The constructor used is based on the required operation. Notice that one constructor takes an `java.io.InputStream`. This is required to pass a `java.io.FileInputStream` to an agent that wishes to read a locally available file. The other constructor takes a `ByteTransmitter`, which is an interface that helps transmit bytes from a local file to be read remotely. The `ByteTransmitter` argument is actually a M2MI Unihandle[1] reference for the actual `java.io.BufferedInputStream` that reads the locally available file. Bytes read from this `java.io.BufferedInputStream` are transmitted to a remote location. Of course, this `java.io.BufferedInputStream` encapsulates a `java.io.FileInputStream` to read the locally available file which is to be read remotely.
- `ByteTransmitter`, `ByteReceiver`, `InputChannel` and `InputChannelProxy`: The `ByteTransmitter` and `ByteReceiver` are two classes that compliment each other to setup a protocol for byte transfer from one file system process to another. The `ByteTransmitter` is responsible to transmit bytes to the `ByteReceiver`. In MFS, the `InputChannel` implements the `ByteTransmitter` and `InputChannelProxy` implements `ByteReceiver`. The `InputChannel` is a `java.io.BufferedInputStream` that encapsulates a `java.io.FileInputStream`. The `java.io.FileInputStream` reads the file whose bytes are to be transmitted. The `ByteTransmitter` reference of the `InputChannel` is transmitted, where it is used by the `InputChannelProxy` to read the contents of the file. The `ByteTransmitter` reference is transferred in an `File` object.
- `Permissions` and `User`: The `Permission` class encapsulates an ownership, which is defined by the `User` class and *read*/*execute* access rights. Ownership refers to the type of user that owns a file. There are two types of `User` instances: *Individual* and *All*. Semantically, an

Individual `User` defines ownership by a single user, while *All* `User` defines ownership by all the users of the network. When an *Individual* `User` object is instantiated it encapsulates the user name defined by the under lying operation system. Two Individual `User` objects are equal only if their operating system defined user names are equal. *Individual* `User` object instances for separate individuals is assumed always to be different for this project. A `Permission` object has two access rights: *read* and *execute*. *Read* allows an owner to read a file and *execute* allows agent execution on an owners file system instance. A `Permission` object is associated with a file using the *Add* operation.

- `Agent`: An Agent represents mobile code that can be migrated to a remote location. An agent can only use the methods provided by an Agent Container to perform MFS file operations. The agent can obtain the reference of the Agent Container using `Agent.getContainer()`. The reference of the Agent Container is set by the Agent Loader while initializing the agent using `Agent.setContainer()`. An agent can talk back to the application that deployed it. This application should implement the `edu.rit.cs.mfs.agent.ResultReceiver` interface. The M2MI Unihandle[1] of the application is set by the Agent Loader using `Agent.setReceiver()`. This is the Unihandle for `edu.rit.cs.mfs.agent.ResultReceiver`. The agent can retrieve this reference using `Agent.getReceiver()`. An agent can get an M2MI Unihandle (`edu.rit.cs.mfs.agent.ResultReceiver`) reference for itself using `AgentContainer.getReceiverReference()`. A user extending the `Agent` class should override `Agent.initialize()`, `Agent.run()` and `Agent.terminate()`. The first is used for initialization, the second for processing and the third for clean-up. The Agent Loader calls `Agent.setContainer()` and `Agent.setReceiver()` before `Agent.Initialize()`, hence any calls to `Agent.getContainer()` or `Agent.getReceiver()` will return `null`, before `Agent.initialize()`.
- `Fetch` and `Write`: Internal classes of `InputChannelProxy`. `Fetch` fetches bytes from `InputChannel` and `Write` writes theses bytes to an internal buffer in the `InputChannelProxy`.
- `Context`: A `Context` object is a container. An object of this class is used for communication between a `edu.rit.cs.mfs.file.FileServiceImplementation` and network-wide available `edu.rit.cs.mfs.file.FileModules`. It contains the address of the sender and a file operation.

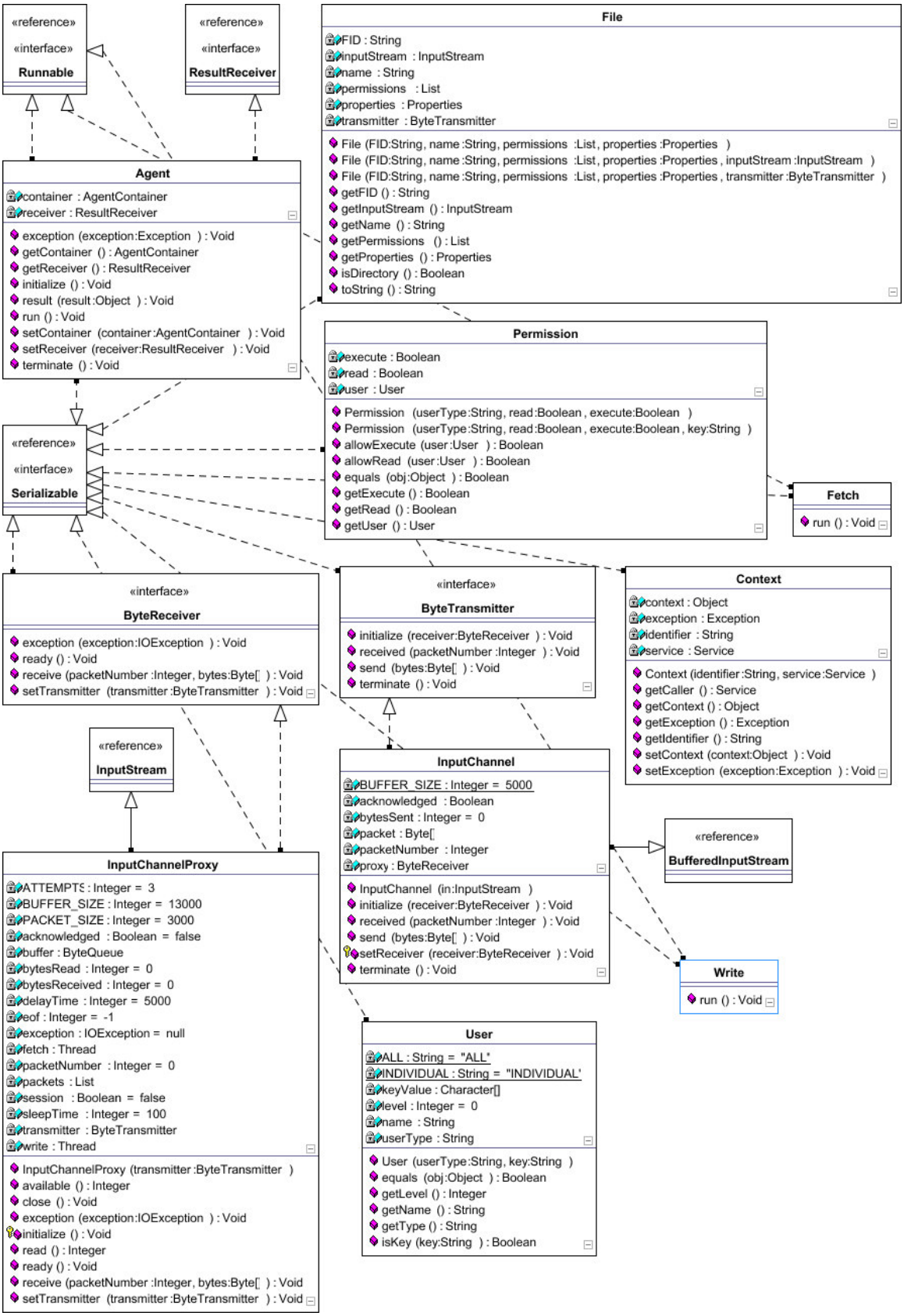

Figure 20: edu.rit.cs.mfs.resource

4.5.2 edu.rit.cs.mfs.service

The `edu.rit.cs.mfs.service` package consists of:

- `Service`: An interface that represents a service. This interface is implemented by the `edu.rit.cs.mfs.file.FileServiceImplementation` class to receive results or exceptions from a `edu.rit.cs.mfs.file.FileModule`.
- `MFS`: The Client Interface class that is used by applications to perform file operations.

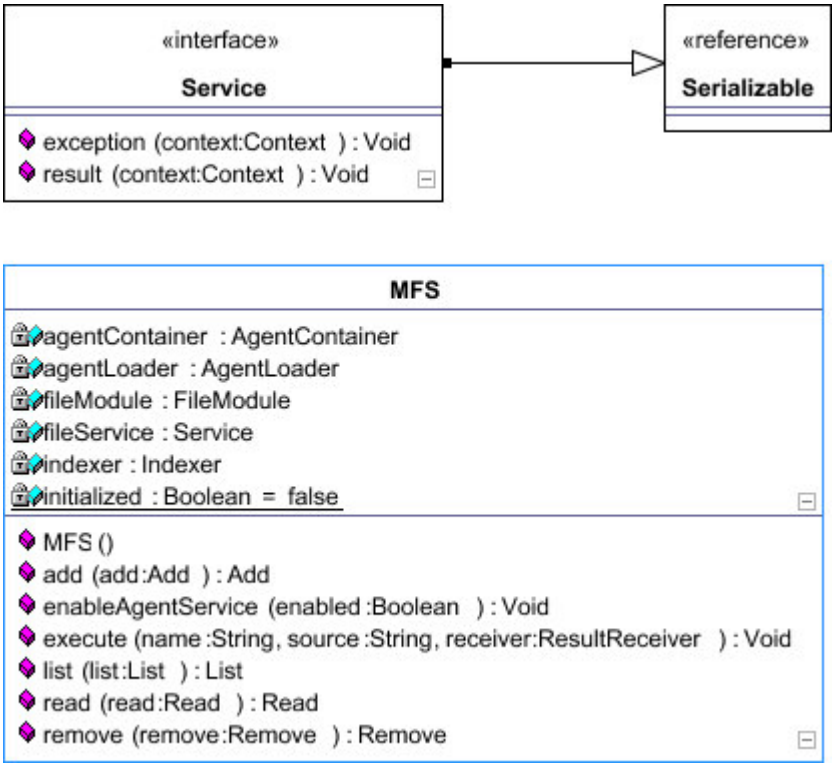

Figure 21 edu.rit.cs.mfs.service

4.5.3 edu.rit.cs.mfs.file

Let us now discuss the edu.rit.cs.mfs.file package. This package contains the File Service, which is responsible to perform file operations. The classes are:

- FileStructure, File and Directory: The FileStructure maintains a file structure that has been created by a file system instance. As mentioned earlier each instance of the file system maintains its created file structure. An aggregation of all available file structures across all the network is the MFS file structure. It uses internal classes File and Directory to build an in memory tree. It contains methods that help parsing this file structure tree.
- FileModule, FileModuleImplementation: FileModule is an interface, implemented by FileModuleImplementation. A FileModule is a unit which is aware of the file structure that is created by its file system instance. It permits the *List* and *Read* operation on the files made available by its file system instance. A FileModule listens to requests from a FileServiceImplementation.
- FileServiceImplementation: FileServiceImplementation is an implementation of edu.rit.cs.mfs.service.Service. The FileServiceImplemetation permits file operations to be performed on the file system as a whole. It accomplishes this by talking to FileModules spread over the network. This communication takes place with the help of edu.rit.cs.mfs.resource.Context. FileServiceImplementation packs a file operation in a Context object and broadcasts this container to all FileModules over the network. Each FileModule performs the operation and sends the results back to the FileServiceImplementation. These results are unified by FileServiceImplementation. This flow is shown by Figure 14.
- FileInformation: A FileInformation object is used to store all information about a MFS file. This class extends edu.rit.cs.mfs.util.BasicRecord.

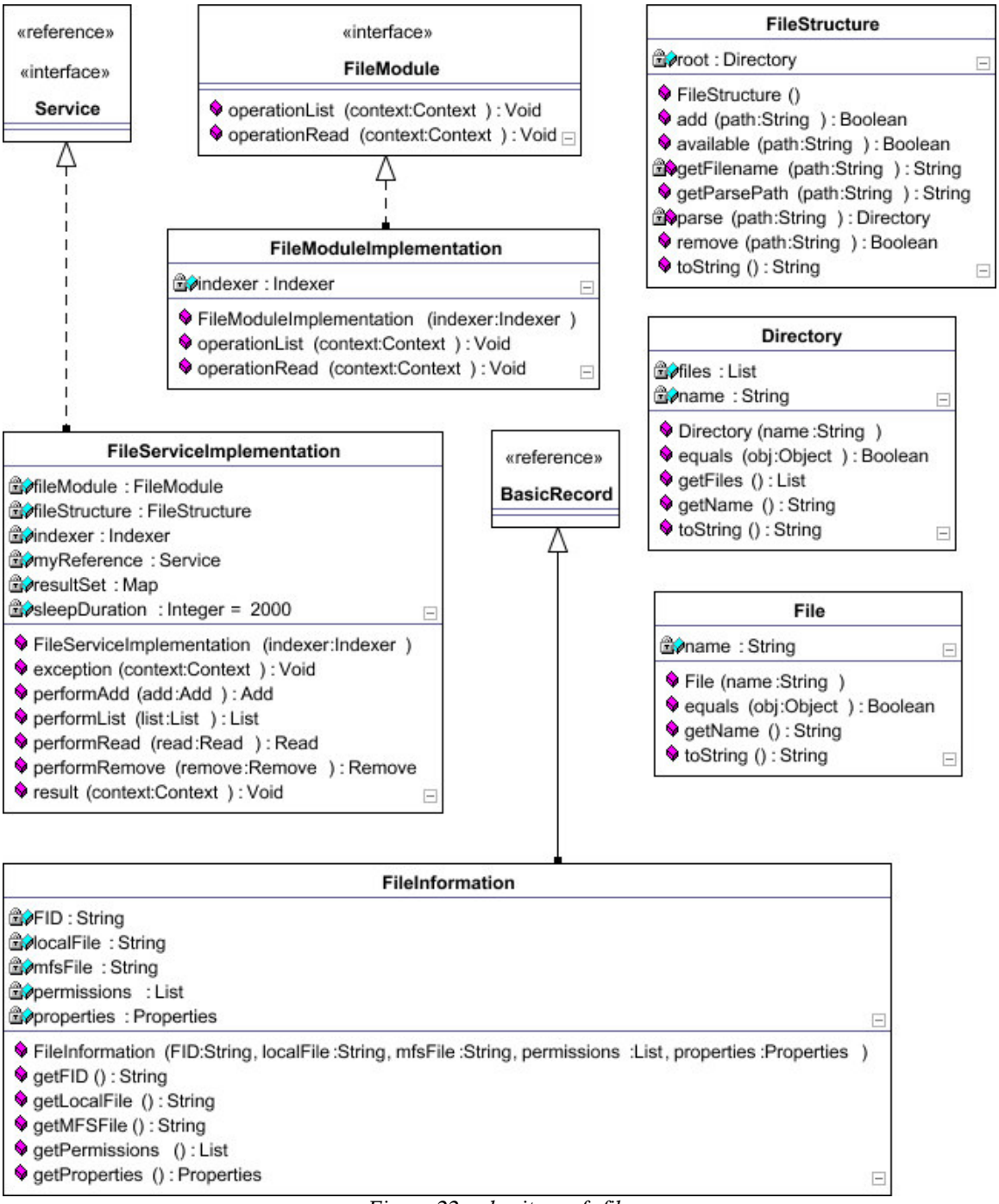

Figure 22: edu.rit.cs.mfs.file

4.5.4 `edu.rit.cs.mfs.operation`

The next package is `edu.rit.cs.mfs.operation`, which representations all the file operations that can be performed. An operation object also stores the results of the operation.

- `Add`: An object that allows adding a file to the M2MI File System. It allows a user to create a link between a locally available file and a MFS file.
- `Remove`: Facilitates removal of an MFS file.
- `List`: List files available on the M2MI File System.
- `Read`: Allows a user to read a file.

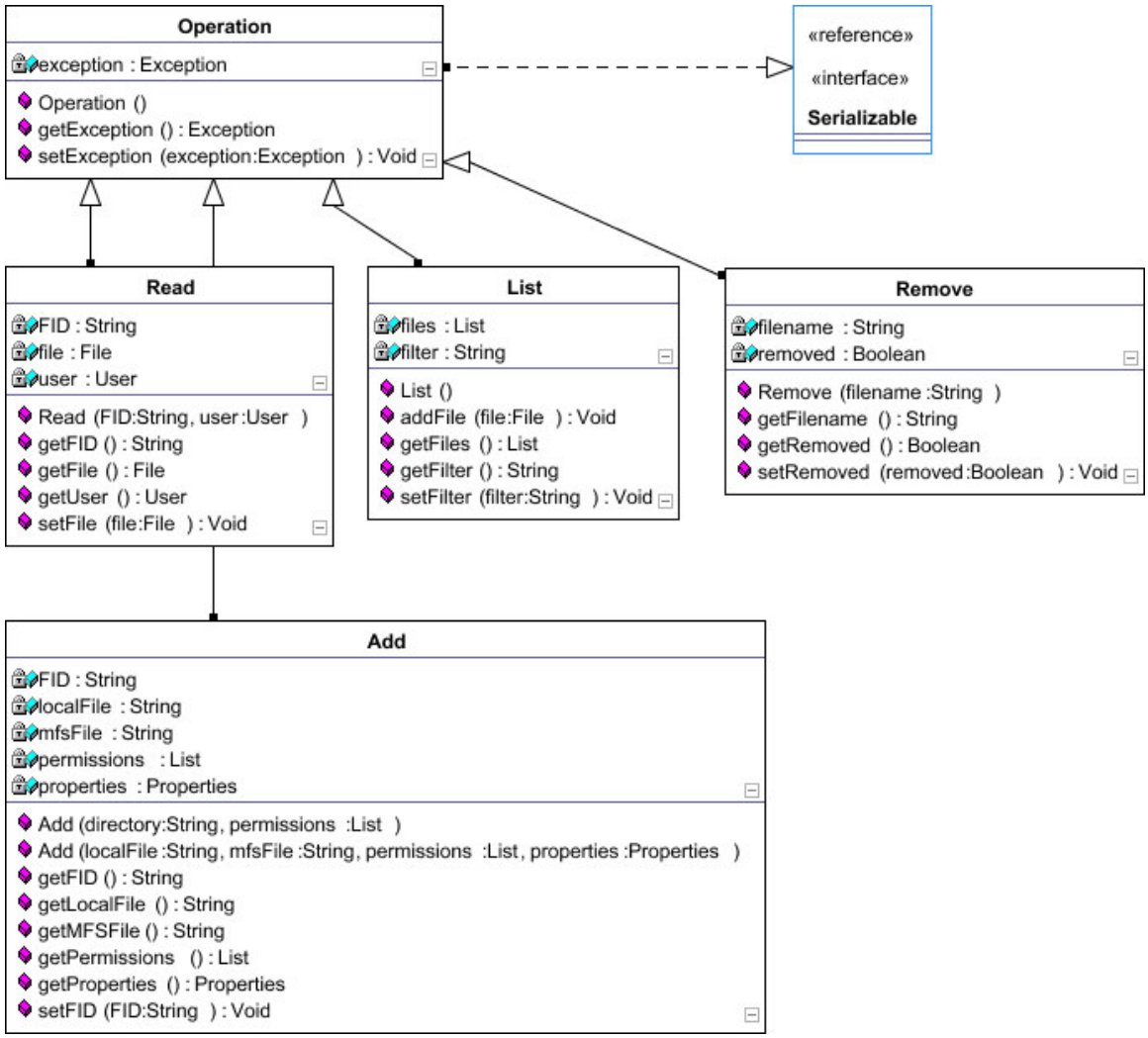

Figure 23: edu.rit.cs.mfs.operation

4.5.5 `edu.rit.cs.mfs.agent`

This package comprises of the Agent Service provided by MFS.

- `MFSClassLoader`: A Java Class Loader that finds classes from the M2MI File System. Its constructor takes a MFS directory as an argument. The class loader finds and loads classes from this directory. This directory contains the full-quantified directory structure of the agent.
- `AgentContainer`: Provides an deployed agent access methods to perform file operations on files available on the `AgentContainer`'s file system instance.
- `AgentLoader`: Provides a mechanism to instantiate agents onto remote file system instances. The `AgentLoader.load()` method accepts the full-quantified class name of the agent, along with the name of the source directory where the agent class exists on MFS. This source directory contains the full-quantified class directory of the agent. The `AgentLoader.load()` method also accepts an instance of an object that implements `ResultReceiver`. This object is usually the application that deployed the agent. It creates a M2MI Unihandle[1] for this object. This Unihandle is used while initializing the agent using `Agent.setReceiver()`, so the agent can send results to this object. Creates a new `MFSClassLoader` for each source directory.
- `AgentServiceImplementation`: Implements `AgentLoader` and `AgentContainer`.
- `ResultReceiver`: A class defined to allow communication between an agent and the application that deployed the agent.

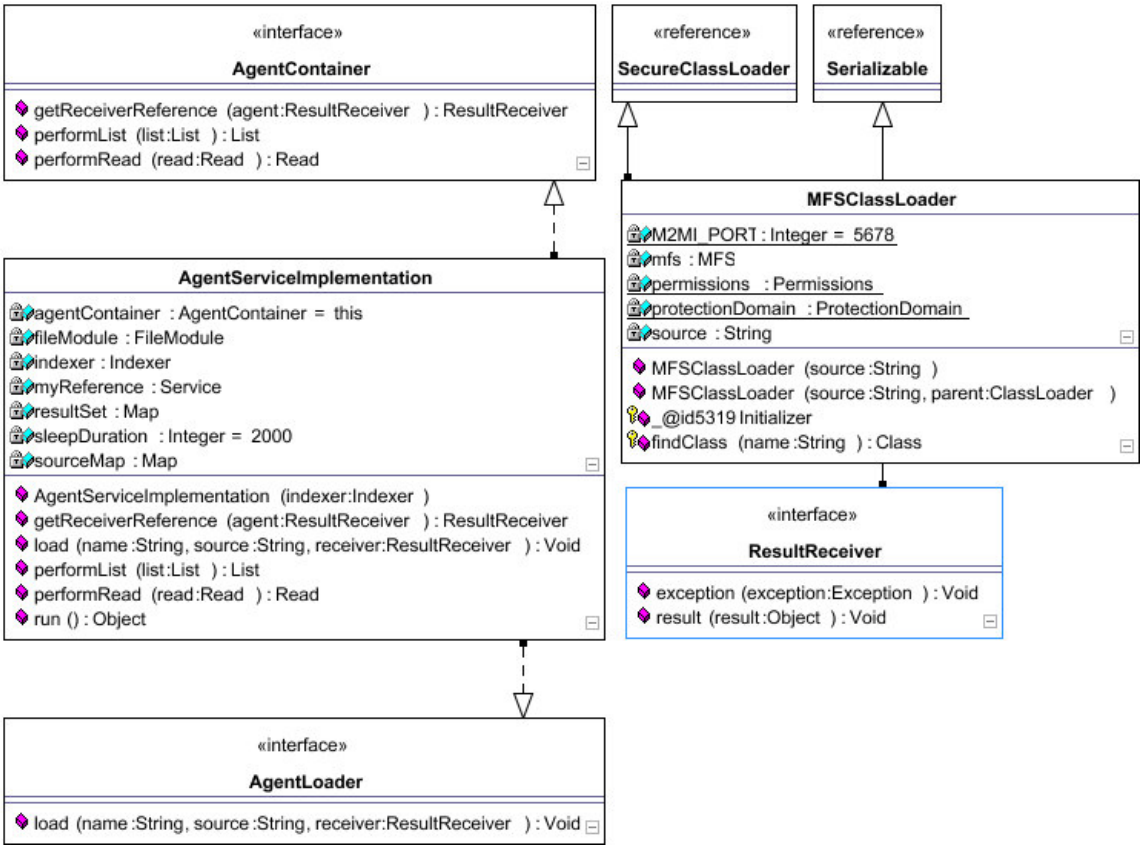

Figure 24: edu.rit.cs.mfs.agent

4.5.6 `edu.rit.cs.mfs.util`

This package contains utility classes.

- `Record`, `BasicRecord` and `Indexer`: The `Indexer` is a data store that can store and retrieve objects of the type `Record`. `BasicRecord` is an abstract implementation of `Record`. The `Indexer` is used to store objects of the type `edu.rit.cs.mfs.file.FileInformation`. The `Indexer` classifies and stores its records based on their class type. Each `Record` has a unique identifier.
- `Properties`: This class is used to store key-value pairs, similar to `java.util.Properties`. This class was written for two primary reasons. One, to allow `java.lang.String` keys to be associated with values of any object type. Two, to provide a mechanism that allows values to be updated automatically over the ad hoc network. The latter has not been implemented.
- `Identifier`: A class that provides unique identifiers.

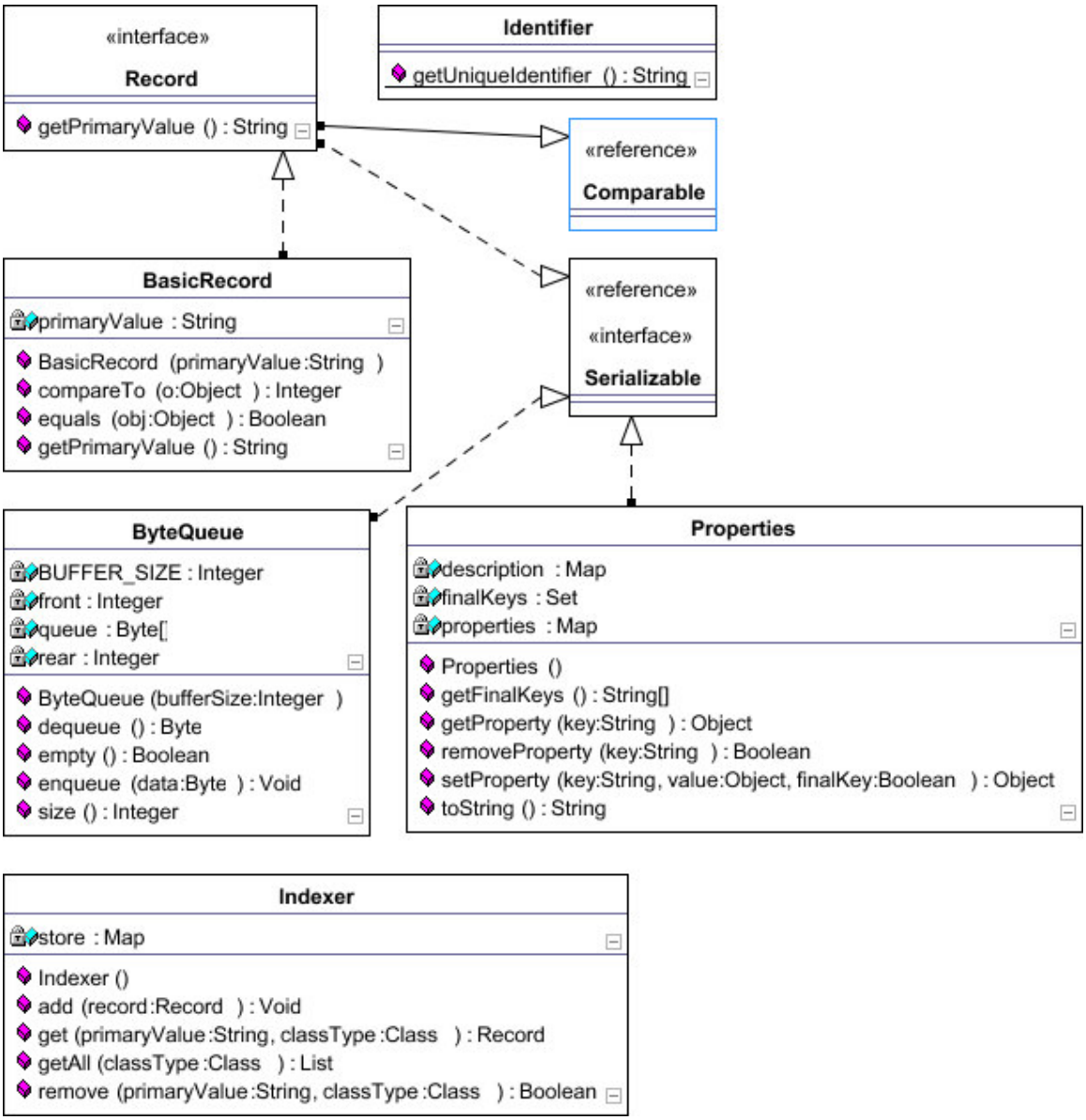

Figure 25: edu.rit.cs.mfs.util

4.5.7 `edu.rit.cs.mfs.console`

This package contains the MFS console application.

- `Console` and `ConsolePanel`: This is the main class that builds the MFS console application. A `Console` object executes commands. A command in this console application is represented by `Command`. The `Console` uses a `ConsolePanel` for its text Input Area. It provides methods that allow commands to write to the Input Area and the Status Bar. It instantiates a new `Command` object based on the text input. When instantiating a new `Command` object, the `Command` object is set with the arguments of the inputed command and reference of the `Console` object.
- `Command`: A `Command` is an abstract class that is overridden to write commands for the MFS console application. A `Command` sub-class should be named using the Command{Name} convention. The command name should be all capital. A sub-class can get the `Console` reference using `Command.getConsole()` to write to the Input Area and Status Bar. Also, the sub-class can get the inputed command arguments using `Command.getArguments()`. The `Command.resolveFilename()` is a utility method, which when given a MFS filename returns a list (`java.util.List`) of file (`edu.rit.cs.mfs.resource.File`) objects. The actual MFS filename or the FID enclosed in square brackets can be used as an argument to this method.
- `CommandADD`: Used to add a file in MFS.
- `CommandREMOVE`: Used to remove a file from MFS.
- `CommandSAVE`: To save a MFS file onto the local file system.
- `CommandOPEN`: Opens an MFS file with a local application.
- `CommandLIST`:List the available MFS files.
- `CommandSTREAM`: Pipe MFS file contents to a process.

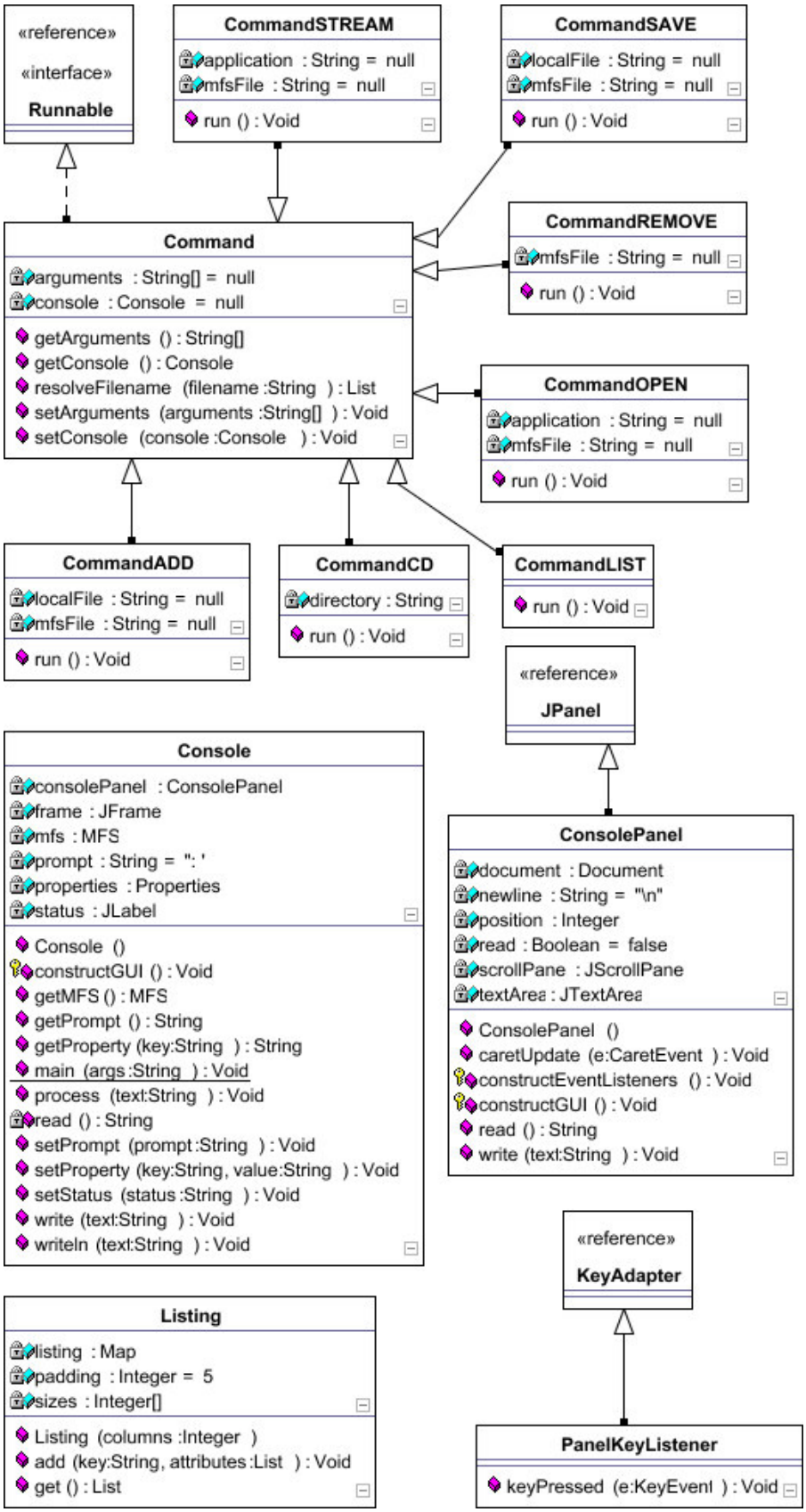

Figure 26: edu.rit.cs.mfs.console

4.5.8 com.wrox.httpserver

This package contains the classes that form the HTTP daemon. This package is a text[6] book example where some classes were modified.

- `HTTPConstants`: This class contains constants defined for the HTTP daemon such that these constants can be accessed by several other classes.
- `HTTPServer`: The main class for the HTTP daemon responsible to listen for HTTP requests. It spawns a new `HTTPRequest` object for every incoming HTTP request.
- `HTTPLocalizedResources`: Class required to read property files.
- `HTTPConfig`: Contains the HTTP daemons configuration information.
- `ConfigFileException`: Exception thrown due an error in the configuration file.
- `MimeConverter`: Provides the required MIME type for a given filename extension.
- `HTTPGMTTimestamp`: Represents an immutable, current time stamp.
- `HTTPLog`: Maintains a log of errors and successful HTTP requests.
- `HTTPException`: Signals an error for a given HTTP request. It provides the necessary HTTP error status code.
- `HTTPStatus`: Encapsulates all HTTP status codes defined by HTTP 1.0.
- `HTTPRequest`: Class responsible to process an HTTP request.
- `HTTPBufferedInputStream`: Needed to read HTTP request information line by line in the defined encoding. For example to read HTTP request header information in ISO 8859-1 encoding.
- `HTTPMessageHeaders`: Encapsulates the headers required for the HTTP request.
- `HTTPInformation`: CGI program execution information.
- `HTTPHandler`: Base class to handle a HTTP request methods like GET, HEAD and POST. `HTTPGetHandler`, `HTTPPostHandler` and `HTTPHeadHandler` are sub-classes of this class.
- `HTTPGetHandler`: Processes HTTP Get request method.
- `HTTPHeadHandler`: Processes HTTP Head request method.
- `HTTPPostHandler`: Processes HTTP Post request method.
- `HTTPObject`: Abstract class to represent a response entity. `HTTPFileObject` and `HTTPProcessObject` are sub-classes of this class.
- `HTTPFileObject`: Response entity to represent a file request. This class is used when a URI requests a file or directory listing. This class has been modified such that instead of representing locally available files it can represent files available on MFS.
- `HTTPProcessRequest`: Response entity to represent a CGI program execution.
- `HTTPResponse`: Encapsulates an HTTP response.

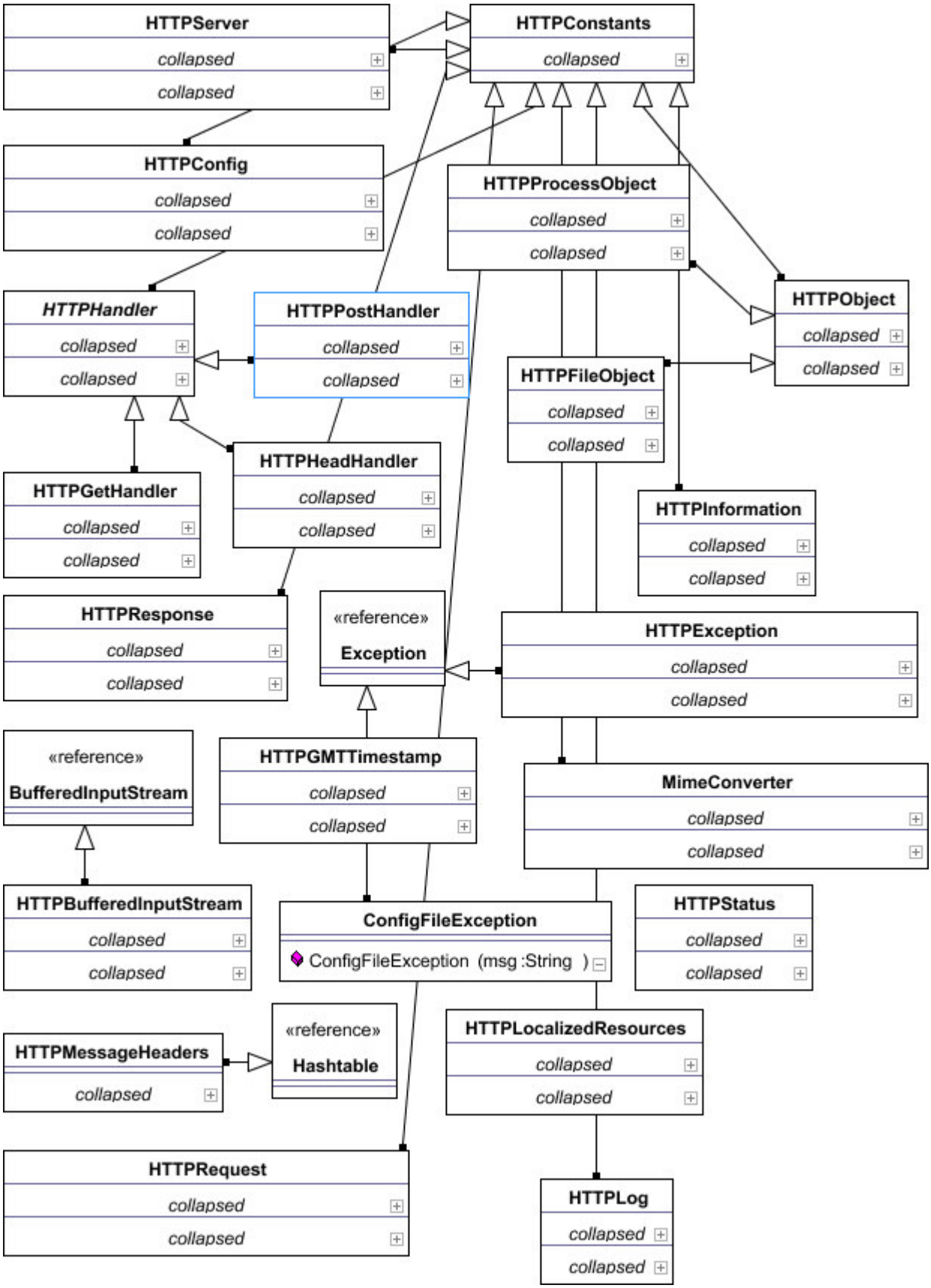

Figure 27 com.wrox.httpserver

4.5.9 The Score Board Application

The Score Board Application consists of two classes:

- `ContestantScoreBoard`: The main program that displays the Score Board and deploys the agent program on all peers in the network.
- `ContestantFetch`: The Agent program that gets deployed on all peers in the network. Once deployed it sends results to `ContestantScoreBoard`.
- `ContestantTableModel`: Table model to represent the Score Board.

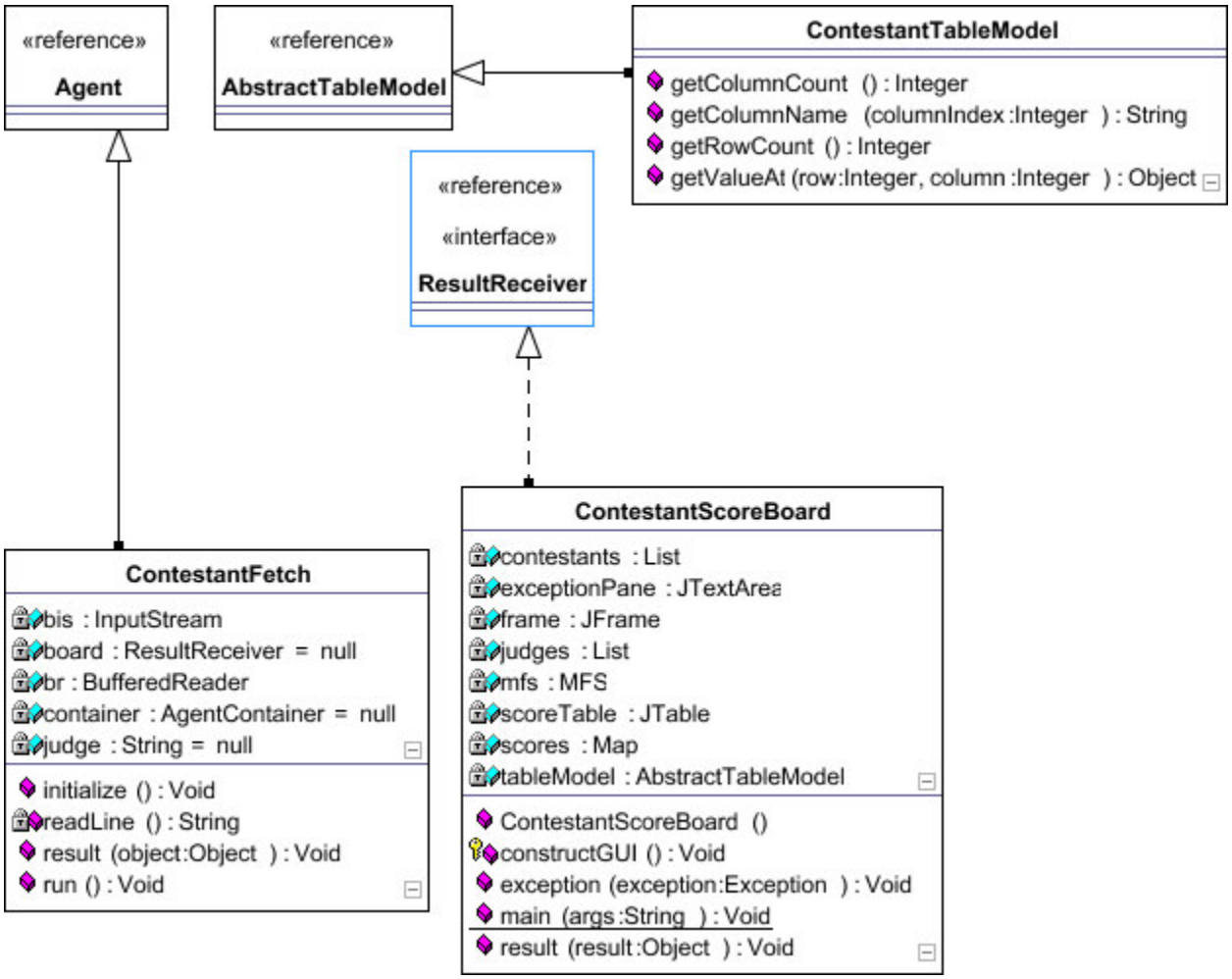

Figure 28 Score Board Application

5. Users Manual

5.1 Installation and Execution

The needed files to run the file system are located in the `MFS` directory. Let the absolute path to the sources be `<MFS_DIRECTORY>`.

The files are organized as:
- `<MFS_DIRECTORY>/Source/anhinga/`: The M2MI/M2MP libraries.
- `<MFS_DIRECTORY>/Source/freemarker/`: The FreeMarker[7] libraries.
- `<MFS_DIRECTORY>/Source/httpd/`: The HTTP Daemon application.
- `<MFS_DIRECTORY>/Source/mfs/`: The MFS source code and compiled classes.
- `<MFS_DIRECTORY>/Source/examples/`: The Score Board application using agents.

The CLASSPATH variable should have the following:
- `<MFS_DIRECTORY>/Source/mfs/`
- `<MFS_DIRECTORY>/Source/anhinga/`
- `<MFS_DIRECTORY>/Source/httpd/`
- `<MFS_DIRECTORY>/Source/freemarker/freemarker.jar`

Create a directory `.mfs` in your home directory. This directory contains MFS configuration files. Place the file `list.ftl` into this directory. This file is found in `<MFS_DIRECTORY>/Source/httpd/` directory. This file is required to build HTML directory listings on-the-fly, used by FreeMarker[7], a template engine that produces HTML output.

The HTTP daemon reads a configuration file `httpd.properties` which must be accessible through the CLASSPATH variable. `<MFS_DIRECTORY>/Source/httpd/` contains this configuration file. Since `<MFS_DIRECTORY>/Source/httpd/` is available through the CLASSPATH variable, the `httpd.properties` file is accessible too. Please make the following changes to the `httpd.properties` file:
- Port: `12358` The local port used by the HTTP daemon to listen for HTTP requests.
- DocumentRoot: `<MFS_DIRECTORY>/Source/httpd/` The directory from were the HTTP daemon serves documents.
- DirectoryIndex: `index.html` Default file for a HTTP directory request.
- LogFile: `<MFS_DIRECTORY>/Source/httpd/httpserver.log` Log file for the HTTP daemon.
- CGIPath: `<MFS_DIRECTORY>/Source/httpd/cgi-bin` Directory where CGI scripts are stored.
- TemplateDirectory: `<USER_HOME_DIRECTORY>/.mfs` Needed to build directory listings on-the-fly. This property should indicate the directory containing `list.ftl`. Note we have placed the `.mfs` directory containing `list.ftl` in the your home directory.

Also, you would have to create necessary changes to your `.java.policy` file which should be placed in your home directory. This file may or may not be present in your home directory. A sample file is present in the directory `<MFS_DIRECTORY>/Source/`. If you are using this sample file, you would have to make changes depending on where the MFS sources are placed. You can use Java's `policytool` to make the necessary changes. The entries in the policy file

give all permissions to the MFS and Anhinga library.

The `m2mp.properties` and `m2mi.properties` file are required by the Anhinga library. These files could be placed in several locations as defined by the Anhinga documentation. You could place these files in your home directory. The directory `<MFS_DIRECTORY>/Source/` contains sample versions of `m2mp.properties` and `m2mi.properties`. Both these files contain a property `edu.rit.m2mp.deviceid`, which should equal to the MAC address of the Network Interface Card you would be using for MFS. You can safely leave the other properties the same. A complete description of these properties is available in the Anhinga library documentation.

The Console application executes with `java edu.rit.cs.mfs.console.Console`. To run the HTTP daemon use `java com.wrox.httpserver.HTTPServer httpd`.

5.2 The Console

The MFS Console is an application that provides access to MFS files using text based commands. The set of commands are:

- `list`: List MFS files. Based on the current directory, it lists all the file that branch form the current directory. If the root directory is the current directory, this command will list all the MFS files.
  Usage: `list`
- `add`: Used to add a local file into the M2MI File System. Also, used to create a MFS directory.
  Usage:
  - `add {DIRECTORY_NAME}`, where DIRECTORY_NAME is the name of a new directory to be created.
  - `add {LOCAL_FILENAME} {MFS_FILENAME}`, where LOCAL_FILENAME is a locally available file and MFS_FILENAME is the name of the file as it should exist in MFS.
- `remove`: Allows the user to remove a MFS file. Only an empty directory can be removed.
  Usage: `remove {MFS_FILE_NAME}`, where MFS_FILE_NAME is the name of the MFS file or directory.
- `cd`: Change the current directory. This command changes a console property, which allows other commands to convert relative pathnames to absolute pathnames.
  Usage: `cd {MFS_DIRECTORY_NAME}`, where MFS_DIRECTORY_NAME is the name of a MFS directory.
- `open`: Fetches an MFS file's contents such that it can be opened by a local application.
  Usage: `open {MFS_FILENAME} {LOCAL_APPLICATION}`, where MFS_FILENAME is the name of the MFS file and LOCAL_APPLICATION is a locally available application.
- `save`: Fetches an MFS file's contents such that it can be saved as a local file.
  Usage: `save {MFS_FILENAME} {LOCAL_FILE}`, where MFS_FILENAME is the name of the MFS file and LOCAL_FILE is a new local file.
- `stream`: Fetches an MFS file's contents such that it could be streamed to a local process.
  Usage: `stream {MFS_FILENAME} {LOCAL_PROCESS}`, where MFS_FILENAME is the name of the MFS file and LOCAL_PROCESS is a locally initiated process.

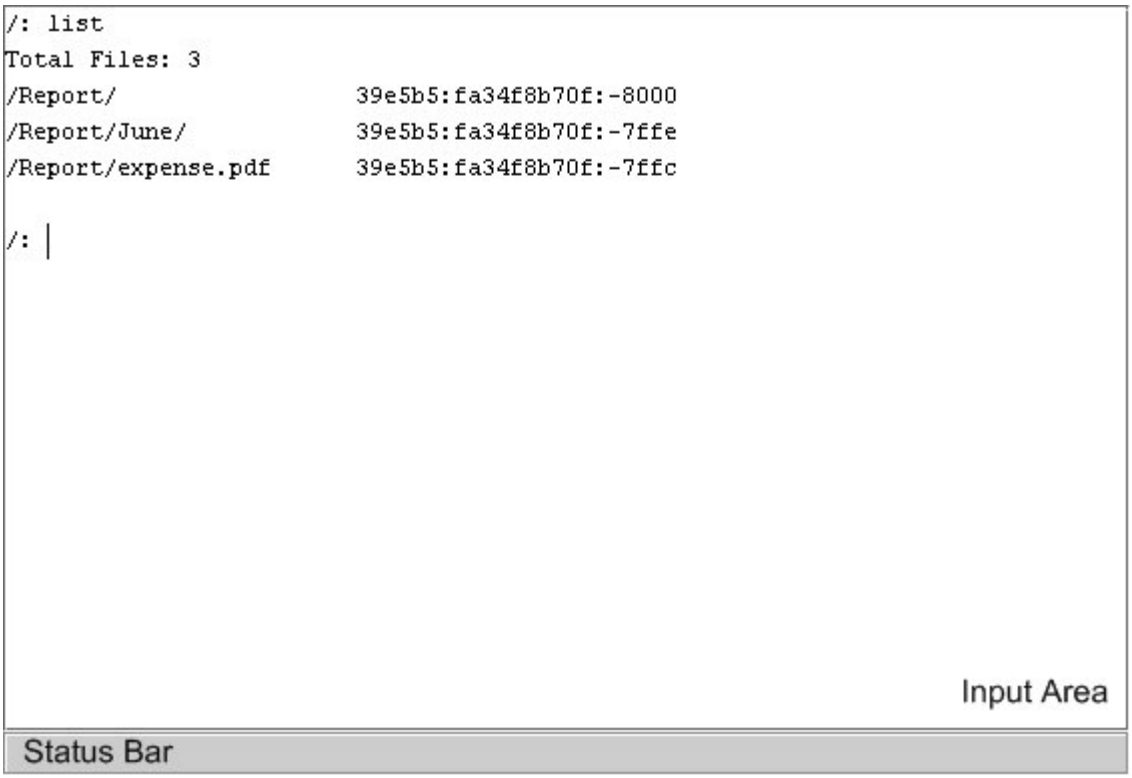

Figure 29: The MFS Console

The above figure shows the console. There is a text Input Area that allows the user to input commands. The Status Bar displays status messages.

5.3 The HTTP Daemon

The HTTP Daemon is an application that provides a HTTP based user interface for the files available on MFS. The HTTP daemon is borrowed from a text on Java Networking[6]. The code was modified to enable build directory listings on-the-fly. Also, when fetching a file, the contents of a file are directly streamed to the browser from the remote location.

Let us assume the daemon is now listening for HTTP requests on the specified port 12345. The URL `http://127.0.0.1:12345/` will take your browser to the root directory. The HTML page will show all the available MFS files. You could directly point to a directory by specifying the directory name after the daemons address.

Example:
The URL `http://127.0.0.1:12345/Report/June/` will display the MFS files in the directory `/Report/June/`.
Similarly, `http://127.0.0.1:12345/Report/June/expense.pdf` would fetch the contents of the file `/Report/June/expense.pdf`. The browser handles the contents of this file using an external application.
The URL `http://127.0.0.1:12345/[39e5b5:fa34f8b70f:-7ffc]` would fetch the file `/Report/June/expense.pdf`.

5.4 The Score Board Application

See `<MFS_DIRECTORY>/examples/`. This directory contains an application `ContestantScoreBoard` and an agent `ContestantFetch`. The application represents a Score Board that sends out agents to judges that are scoring contestants. For this example we would assume that there are only two judges, `judge1` and `judge2`. You would find two directories:

- `<MFS_DIRECTORY>/examples/judge1` containing `judge1.txt`, and
- `<MFS_DIRECTORY>/examples/judge2` that contains `judge2.txt`.

We need to open two MFS Console sessions one in each directory and add each file to MFS. These files need to be added to a directory called `/Contest/`.

Let us set this up:

1. Change to the local directory `<MFS_DIRECTORY>/examples/judge1/`
2. Execute the Console: `java edu.rit.cs.mfs.console.Console`
3. In the Console create a MFS directory `/Contest/`: `add /Contest/`
4. Add `judge1.txt` to `/Contest/`: `add judge1.txt /Contest/judge1.txt`
5. Change to the local directory `<MFS_DIRECTORY>/examples/judge2/`.
6. Execute the Console: `java edu.rit.cs.mfs.console.Console`
7. Add `judge2.txt` to `/Contest/`: `add judge2.txt /Contest/judge2.txt`
   Note that there is no need to add `/Contest/` again, since it has been created by another file system instance.
8. Change to the local directory `<MFS_DIRECTORY>/examples/`.
9. Execute the application: `java ConstestantScoreBoard`
   The application opens a Score Board organized by the judges, judge1 and judge2.
   Open an editor to add contestant names and scores to the files `judge1.txt` and `judge2.txt`. Each contestant should be placed on a new line. The contestant name and score should be separated by a white space. Note that the last contestant should not be terminated with a new-line character before saving the file. The agents work like the Unix `tail` command. As new contestants are added, the Score Board will reflect the changes. An example of a judge's file:

   Daffy 65
   Tweety 44
   Bugs 76

6. Future Work

The MFS infrastructure can be made more useful by other expansions. Some of them are:

- Permit users to write to files remotely. The `edu.rit.cs.mfs.resource.File` object can provide the user with an `java.io.OutputStream`. A user that wishes to write to this file can write to this stream. However, unlike reading, where there can be multiple readers, while writing, there can be only one writer.
- MFS needs a mechanism for self updating properties. For example, an MFS file object contains `File.SIZE` as a property, that defines the size of the file. What if the size of the file changes. All MFS file objects that represent this file will still have the old value.
- There may be a need to abstract lower level IO functionality from the MFS working mechanism since Java IO packages might not be the same on mobile platforms.
- The Console commands utilize only the essential part of the functionality provided by MFS file operations. These Console commands can be made more useful and flexible. Also, a new set of commands can be added with respect to permissions.
- It could be possible to write a new URL Protocol Handler using `java.net.URLStreamHandler`. The `URLStreamHandler` is a stream protocol handler that knows how to open a connection for a given protocol type for example like HTTP, FTP. By writing a URL Protocol Handler it would be possible to provide a special URL to read a MFS file.
- It could be possible to save the state of the files made available by a file system instance, which could ease the user of not adding a same local file every time she initiates a MFS Console session. An effort in this direction could be possible be using a small memory footprint (embedded) database to maintain the file information. The task of maintaining file information is currently done by `edu.rit.cs.mfs.util.Indexer`.
- Agent deployment should be possible using Java Jar files. A developer should be able to pack all his files for mobile code deployment into a single Jar file. MFS currently requires that all required class files be added onto MFS instead of a single Jar file.

7. Conclusion

The project was aimed at enabling sharing of file resources in an ad hoc environment. It also supports the usefulness of mobile code deployment. During the course of building this project some conventional file systems were studied. This project combines required features from conventional file systems with the ad hoc paradigm obtained by the Anhinga project. In MFS, each participating peer is a file system in itself, willing to share its information with peers in the network. This synergy forms the M2MI File System.

The project helped better my understanding of designing applications for ad hoc networks. I also realized that using agents is a useful computing paradigm. Though not used extensively, I believe this model of computing to be very flexible. Agents could be used to write complex operations or queries, deployed on a framework that provides simple operations.

Finally, I would like to thank my project committee members Alan Kaminsky, Hans-Peter Bischof and Rajendra K. Raj[8] for their support and the opportunity to grow in this field.

References

1. The Anhinga Project
   www.cs.rit.edu/~anhinga
2. Microsoft Disk Operating System (MS-DOS)
   www.microsoft.com
3. Sun Network File System (NFS)
   www.sun.com
4. Remote Procedure Call
   http://en2.wikipedia.org/wiki/RPC
5. Java Security Manager
   http://java.sun.com/j2se/1.4.2/docs/guide/security/
6. Chad Darby, John Griffin, Pascal de Haan,
   Alexander V. Konstantinou, Sing Li,
   Sean MacLean, Glenn E. Mitchell II,
   Joel Peach, Peter Wansch, William Wright
   Beginning Java Networking
   Wrox Press Inc
   First Edition, 2002
   http://www.wrox.com
7. FreeMarker
   http://freemarker.sourceforge.net/
8. M2MI File System Master's Project Committee
   - Alan Kaminsky [Chairperson]
     http://www.cs.rit.edu/~ark
   - Hans-Peter Bischof [Reader]
     http://www.cs.rit.edu/~hpb
   - Rajendra K. Raj [Observer]
     http://www.cs.rit.edu/~rkr
9. George Coulouris, Jean Dollimore, Tim Kindberg
   Distributed Systems, Concepts and Design
   Addison-Wesley Publishers Limited
   Third Edition, 2001
   http://www.cdk3.net/