MFS: M2MI File System

Masters Project Proposal

Ravi Narain Bhatia
Department of Computer Science
Rochester Institute of Technology

Alan Kaminsky
Chair

Hans-Peter Bischof
Reader

Rajendra K. Raj
Observer

Table of Contents

Objective

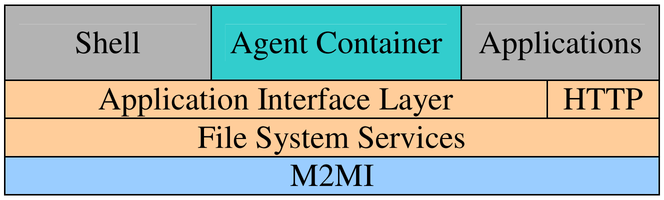
The objective of the project is to build a *Distributed File System* for an adhoc network. The file system would be built using M2MI (Many-to-Many Invocation) [1], which is a middleware that facilitates communication in a proximal network. The file system supports the deployment of mobile code (agents) to enable workflows.

Subject Area

The project study belongs to the area of adhoc networking and File Systems. The project will use Many-to-Many Invocation [1] (M2MI) for message passing between the various software components of the system. Some features of Java RMI [2] will be used to enable dynamic class loading.

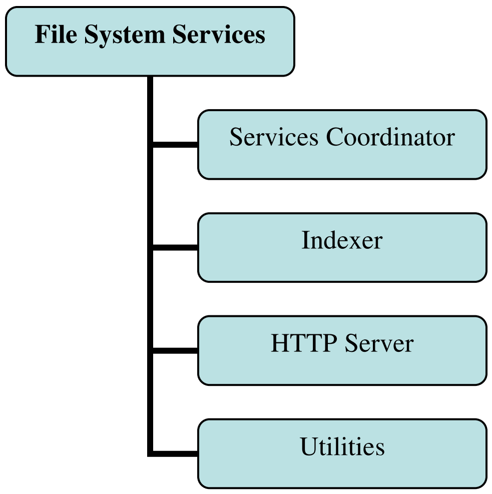
Design Specification

The file system could be divided into the following layers.



M2MI Layer
The M2MI [1] layer provides messaging functionality in an adhoc network. The File System Services layer uses the M2MI layer for coordination between peers.

File System Services
This layer provides the core services of the file system. The File System Services layer provides these services to applications through the Application Interface Layer. The File System Services Layer is divided into the following:



Services Coordinator
This unit coordinates the availability of resources (files) among peers. When a file on a peer is mapped from the Local File System into MFS, the Service Coordinator takes the responsibility to inform other peers. This broadcasted message is received by the Services Coordinator unit on each peer, allowing them to make necessary changes to their file indexing structure.

Indexer
The Indexer takes the responsibility of indexing the available files. It symbolizes the core data model of the system. When new files are mapped into MSF the Services Coordinator is notified. The Services Coordinator asks the Indexer to make the necessary changes to the file indexing structure. All Indexers on each peer will have the same indexing structure for the files available on the network. The Services Coordinator takes the liability of synchronizing this information.

HTTP Server
Applications access resources (files) through the HTTP Server. A HTTP browser can access the HTTP Server to:

1. Obtain a listing of directories and files.
2. Request the contents of a file and pass them to an application.

The HTTP Server monitors the Indexer for changes in the file indexing structure. The HTTP Server and the Indexer follow the Observer pattern where the HTTP Server is the Observer and the Indexer the Observed. Based on the indexing structure it serves the appropriate content.

Utilities
This unit will consist of functions that do not fall into the other specific File System Services. An important function performed by this unit is to monitor the network using techniques like polling to seek changes. Once it observes a change like the absence of a previously available node, it notifies the Services Coordinator to make the necessary changes to the Indexer. Hence the Utilities unit will include several helper functions incurring during project development.

Application Interface Layer
The Application Interface Layer provides an API for applications. Using this layer, applications can seamlessly access files available on the adhoc network. This access will be read-only. The API will provide a set of methods to list and read files. The Application Interface Layer works with the File System Services layer to provide this functionality. The Application Interface Layer follows the Facade pattern providing an interface to core services of MFS.

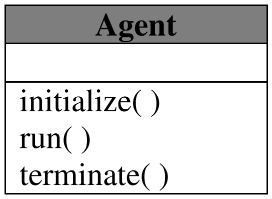
Shell
The Shell provides a console-based interface for users to:

1. Add/Remove files from MSF
2. List files on MSF.
3. Access (read-only) files, available on MSF

Applications
This section comprises of applications that access the file system using either the Application Interface Layer or HTTP. Most applications can access files using HTTP, hence applications will be able to access files through a peer's inbuilt HTTP server. The web browser is an example of an application that can retrieve a file and pass it to the chosen application.

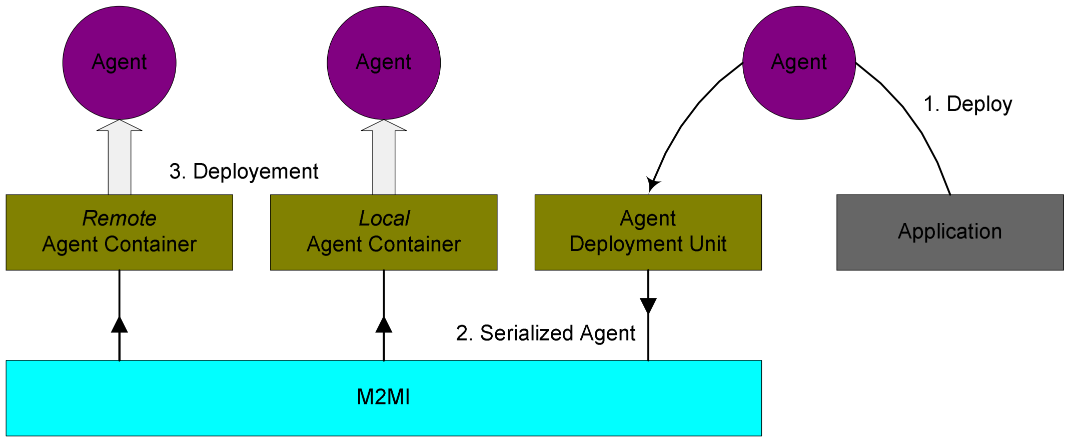
MFS will support the deployment of Agents. An agent is mobile code that is sent from one system to other to perform specialized functions. Agents provide useful functions. Since the file system is distributed, the transfer of larger files to access their contents is an expensive operation over the network. Smaller mobile programs will be allowed to execute on remote peers whereby they can read-access the contents of remote files. Thus agents can be used to shape specialized queries. Agents also enable workflows. A group of agents can synchronize their messaging to achieve workflows. Since MFS will allow read-access on remote peers, agents can perform data-collection, processing operations and notify results. Agents can be broadcasted to peers even before the required file is available. Once the file becomes available the agent can start to work on it. An agent performs the functions that its developer has intended for it. For e.g. an agent can be sent from peer X to peer Y to monitor the changing contents of a file at peer Y, sending back results to peer X.

The following is the class diagram of an agent:



The initialize( ) operation should contain initialization code. The run( ) operation encapsulates the work an agent performs and the terminate( ) operation allows an agent to complete any finishing jobs it wishes to perform. An agent is executed in the sequence: initialize( ), run( ) and terminate( ).

Agents will be deployed using the **Agent Deployment Unit** (ADU). This unit provides an API to deploy agents in **Agent Containers**.



The ADU holds references for the *Local* Agent Container (M2MI:Unihandle) and *Remote* Agent Containers (M2MI:Omnihandle). Hence it allows agents to be deployed in the *Local* Agent Container or *Remote* Agent Containers. An agent that has the reference of a particular *Remote* Agent Container can choose to be deployed in it, using the ADU API.

Most users that will write agents will extend the *Agent* class and each agent will be different. Hence when an agent is being executed on a remote peer, the class file will have to be obtained from its source peer location using *Dynamic Class Loading*. The Java Virtual Machine which receives an agent, obtains the agent's class file from the MSF HTTP server located on the agent's source location. The RMI *MarshalledObject* class will help serializing agents. The serialized agents will contain the *codebase* URL. The *codebase* represents the location of the remote class file. The java.rmi.server.RMIClassLoader loads the *Class* file for the agent class using the *codebase* URL. The Java System Security Manager will be used to prohibit downloaded agents to write/execute files on the disk or display user interfaces. Agents can only use the MSF API to list and read local files.

Functional Specification

The M2MI File System servers files and maintains a directory structure to organize files like traditional file systems. However there are significant differences. A file that has been exported to the M2MI file system has read-only access. Also, when users create directories with same names, the directories will merge. Files with the same name will be represented differently i.e. with a suffix added to their name. Any changes to represent different file names will only be for the purpose of representation. For MSF, internally files are represented by unique identifiers. The uniqueness of these identifiers will be a function of either the file location on a local file system or the message (file content) digest. A unique identifier will help distinguish duplicate files. MSF provides a platform to share files available on the Local File System. It does not provide the functionality to create new files.

Users will interact with MSF through the shell or a web browser and developers through the Application Layer Interface. The Shell will provide functionality through a set of console-based commands. The shell commands will provide the following functionality:

- Add and Remove files (add/remove)
  add <*local*> <*remote*>
  remove <*remote*>
  where, *local* is the location of the file on the local file system.
  remote is the location of the file on MSF.
  A file can only be removed by a user if the same user has added the file.

- Create and remove directories (mkdir/rmdir)
  mkdir <*name*>
  rmdir <*name*>
  where, *name* is the new directory name.
  A directory with files cannot be removed.

- Listing files and directories (list)
  list <*optional-parameters*>
  where, *optional-parameters* is to format the output of the **list** command.

- Retrieve files (get)
  get <*remote*> <*local*>
  where, *local* is the location of the file on the local file system.
  remote is the location of the file on MSF.
  A file that has been obtained using the **get** command can be changed since it now exists on the local file system. However the original file available on MSF will remain read-only.

Using a web browser a user can view and retrieve files. A file can be retrieved for reading by clicking on its link on the browser. Once the browser downloads the file from the remote peer's HTTP server it opens the designated application for the downloaded file's format type. Therefore a file with a *doc* extension will likely open a word-processing application. The web interface will not provide the complete functionality of the shell. The web interface will only allow users to view the MSF directory/file structure and retrieve files. Additional functionality could be introduced into the web interface using applets/servlets.

Files are addressed using the following scheme:
Shell: */<directory>/…/<directory/<file>* which is similar to the syntax in C shell or MS-DOS.
Web browser: *http://<localhost>:<port>/<directory>/…/<directory/<file>*

An agent can access the local MFS through the Agent Container and Application Layer Interface and perform M2MI calls. Using the API provided by the Agent Container, an agent can browse the files available on the local MFS, read them, send results to other agents and send itself back from its source location. Using policy files and permissions in the Java Security Model, an agent's functionality will only be limited to the above. Data acquired by agents, can be processed and displayed only by the local application that initiated the workflow. A local application uses the API provided by MSF (ADU) to deploy their agents. The Agent Container provides a platform for agent execution where the ability of an agent is limited by the API of the Agent Container.

Workflows are performed using agents. Agents can reside on peers and monitor the changing contents of a file. Based on the contents, agents can send results to a master agent which forwards the results to the application that initiated the agents. An application uses the ADU to place its agents on peers. An agent on execution then chooses its flow of action. A combined synchronized effort of a group of agents results in a workflow.

Project Deliverables

1. M2MI File System, including Shell and Web Browser GUI
2. Demonstration examples
   - File Access: This demonstration will be carried out to demonstrate remote file access. Microsoft PowerPoint will be running on one peer and the file that has to be read will be located on another peer. Using a URL, the peer running PowerPoint will access its local HTTP Server. The HTTP Server fetches the file from the remote peer using M2MI and delivers it to the PowerPoint Application.
   - Workflow: To illustrate the usefulness of workflows based on agents, an example that simulates a contest will be demonstrated. In this contest two judges allocate points to participants based on their performance. However the two judges are not allowed to interact with each other and make changes to their mark sheet when a participant has finished a performance. The host of the contest, who publishes the results, has to fetch marks from both the judges and display them to the audience. In this example, the host deploys a master agent. The master agent creates two more agents that are deployed on the judge's peer nodes. When any judge makes a change to his/her mark sheet, the agent on the judge's node sends back the results to the master agent. Hence the host is able to view the marks given by each judge.

Schedule

Project :	Winter 2003-2004
Defense:	January 30, 2004

September 25, 2003	Completed UML (Class/Sequence) diagrams for File System Services, Application Layer Interface. Partial diagrams for other units.
November 30, 2003	Completed implementation of File System Services, Application Layer Interface and Shell.
December 30, 2003	Completed Agent Container. Testing, Report and demonstration examples work begins.
January 30, 2003	Defense

References

1. The Anhinga Project. http://www.cs.rit.edu/~anhinga/
2. Java Remote Method Invocation. http://java.sun.com/products/jdk/rmi/
3. M2MI File System. http://www.cs.rit.edu/~rnb1914/