# Java-Based

# Simple Mobile Agent System

**Hyon S. Kim**

**CSS499**

**Project Report**

# Table of Contents

# 1. About UW Mobile Agent

## *a. Project Topic*

Implement a Java-based simple mobile agent system that will be used for Prof. Fukuda's grid computing later on.

## *b. Technologies used*

The implementation of a mobile agent system requires the up-to-date Java, XML, and Servelt technologies that are the central of IS industries.

Currently I used three Java networking technologies: serialization, RMI, and class loader. Using the Java serialization, a mobile agent will capture their execution state. With RMI, the agent will transfer its state to a remote site. And finally, it will resume its execution with the support of the Java class loader.

# 2. Background Knowledge

## *a. Serialization*

Serialization is the process of writing objects to a stream and reading them back. To be serialized, your objects must implement the *Serializable* interface. This interface has no fields, constructors, or methods – it just shows that an object is serializable.

## *b. RMI*

Remote Method Invocation (RMI) enables the programmer to create distributed Java technology-based to Java technology-based applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts. A Java technology-based program can make a call on a remote object once it obtains a reference to the remote object, either by looking up the remote object in the bootstrap naming service provided by RMI or by receiving the reference as an argument or a return value. A client can call a remote object in a server, and that server can also be a client of other remote objects. RMI uses object serialization to marshal and unmarshal parameters and does not truncate types, supporting true object-oriented polymorphism.

## *c. Class Loader*

A class loader is an object that is responsible for loading classes. The class ClassLoader is an abstract class. Given the name of a class, a class loader should attempt to locate or generate data that constitutes a definition for the class. A typical strategy is to transform the name into a file name and then read a "class file" of that name from a file system.

# 3. Directory Structure and Files

## *a. Directories*

As you can see in Figure 1 below, root directory contains mnode1, mnode2, mnode3 and mnode4 directories and other necessary files.

- mnode1, monde2, mnode3, and mnode4 directories
  This is the arbitrary folders that I made since medusa and other networked computers(mnode#) are in NFS. This way I can verify that the RMI really works or not. Each of these folders contains all the files in root directory, except 'Inject.class', 'Inject.java', 'TestAgent.class' and 'TestAgent.java'

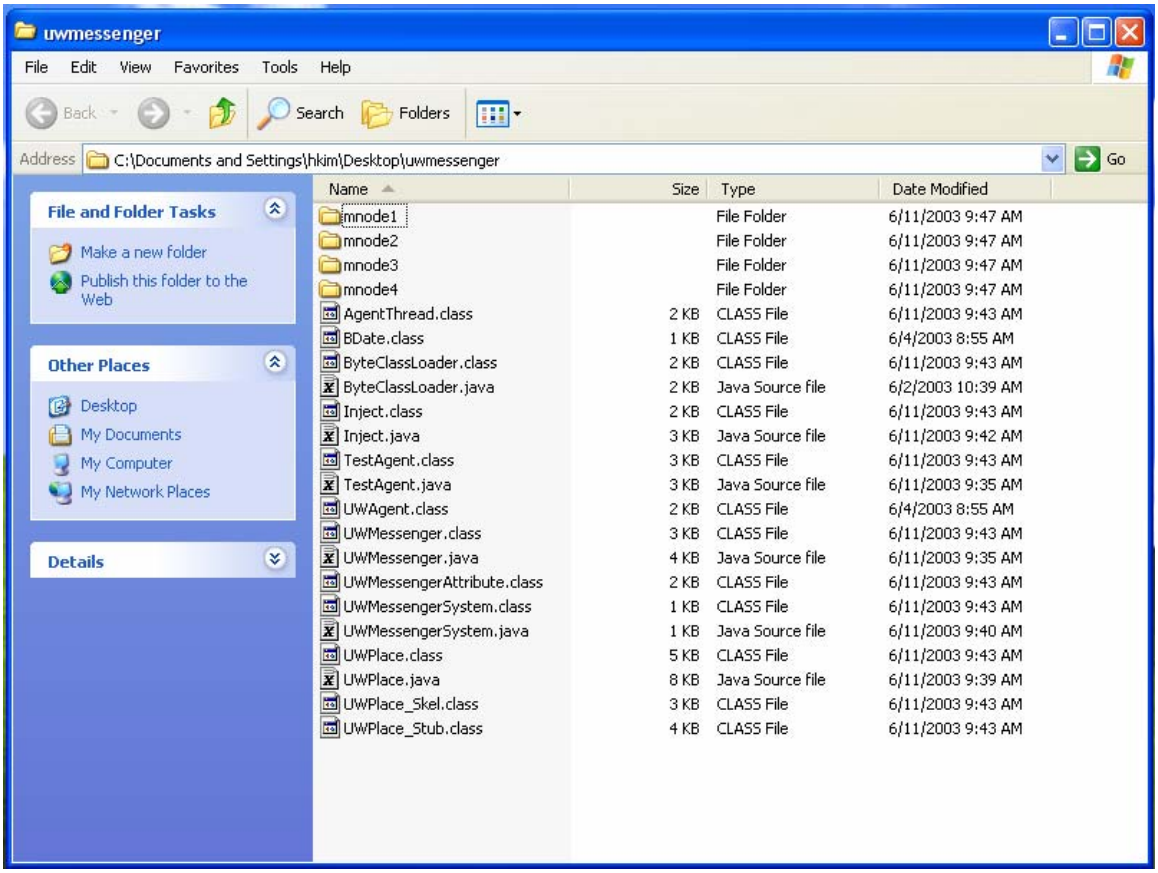


***Figure 1: Directory Structure and Files***

## *b. Files*

- ByteClassLoader.java: Used to load a class from a byte array.

- Inject.java: Used to initiate launching the mobile agent.
- TestAgent.java: Used to create a mobile agent.
- UWMessenger.java: Used as an abstract class for TestAgent.java
- UWMessengerSystem.java: Used to call the remote function (RMI)
- UWPlace.java: Used as a stationary object that the mobile agent comes and goes.

# 4. How to use

This is based on the example directory structure I explained on Section 3. This explains step-by-step instruction for a practice. You might want to shorten it in the future for your convenience.

I updated the final codes in “uwmessenger” directory (login as uwagent)

## *a. rmiregistry %port_number%*

Open command line.
Ssh –l hyonkim mnode1
%your password%
Go to //root directory/mnode1
rmiregistry 35563

Open command line.
Ssh –l hyonkim mnode2
%your password%
Go to //root directory/mnode2
rmiregistry 35563

Open command line.
Ssh –l hyonkim mnode4
%your password%
Go to //root directory/mnode4
rmiregistry 35563

## *b. UWPlace %place_name% %port_number%*

Open command line.
Ssh –l hyonkim mnode1
%your password%
Go to //root directory/mnode1
java UWPlace mnode1 35663

Open command line.
Ssh –l hyonkim mnode2
%your password%
Go to //root directory/mnode2
java UWPlace mnode2

Open command line.
Ssh –l hyonkim mnode4
%your password%
Go to //root directory

java UWPlace mnode4

### *c. Inject agent*

Open command line.
Ssh –l hyonkim mnode4
%your password%
Go to //root directory
java Inject %place_name% %port_number% %class_name% %owner_name% %func_arg1% %func_arg2% %func_arg3%
ex)java Inject mnode4 35663 TestAgent Justin 2 2 3

# 5. Improvement

- ClassLoader
  - Version 1 : I made “FileClassLoader” that loads classes from files in the hard drive.
  - Version 2 : I directly copied the class file into the remote computer’s root directory and instantiated the class. I did not need a special class loader to accomplish this.
  - Version 3 : Finally I made “ByteClassLoader” that loads classes from byte array. In order to hop around the UWMessenger, I moved around the byte array and stored it in vector.
- The incoming UWMessengers will populate in vector array as byte array.
- Port number can be specified by the use. However, all the UWPlace and Inject Place have to have the same port number to communicate each other