# Chapter 4

# SYSTEM DESIGN AND IMPLEMENTATION

## 4.1 Introduction

In our system, the files are kept on a centralized server that hosts a master agent. The file transfers are made by slave agents that migrate to designated destination, bringing the required files.

This chapter discusses the design principles, algorithm used and the implementation of the agent system.

## 4.2 Agent Design Principles

The design of this system requires two types of agents: a Master Agent and a Slave Agent. The design uses the extension of Aglet class in Aglets. The Master Agent maintains the graphical user interface (GUI) which enables us to start or launch the system. The Master Agent also creates and sends Slave Agents to destination hosts. These slave agents perform jobs on behalf of the Master Agent. The Slave Agents would do the necessary steps to bring the file and write it to secondary storage at destination hosts.

## 4.3 Algorithm of Agent System

As specified above, this agent system requires two agents, a Master Agent and a Slave Agent. Two nodes are required to run this system, one as a server (the origin node) and another as a client (the destination node). Both Master and Slave Agents are created on the server nodes. Figure 4.1 shows that the master sends the slave to client while Figure 4.2 shows the classes involved to transfer a file from a server to a client. TransferFile class represents the file which is to be transferred.

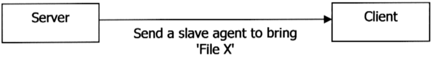


Figure 4.1
Transfer of 'File X' from Server to Client

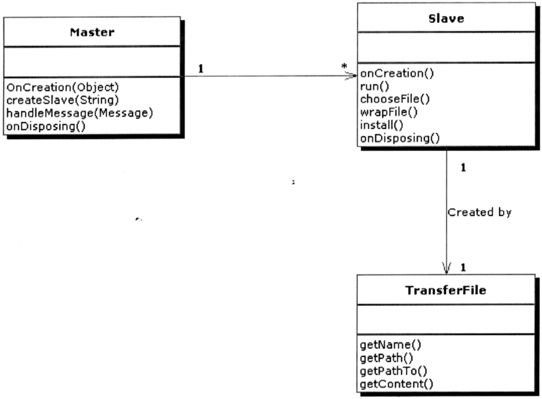


Figure 4.2
Classes Involved in the Agent System

The overall algorithm involved is as follows:

1.0 Create master agent on server node

2.0 Create slave agent

 2.1 Choose client node

 2.2 Choose 'File X'

3.0 Send slave agent to destination together with 'File X'

4.0 Write 'File X' on client node's secondary storage

 4.1 Update results on master agent

5.0 Slave agent returns to master agent

6.0 Dispose of slave agent

## 4.4 Implementation of Agent System

The implementation of this agent system, shown in Figure 4.3, can be divided into the following three phases:

### 4.4.1 The Master Agent (Connection Initiation)

The steps involved in this phase are:

1. Creation of Master Agent
2. Master Agent creates Slave Agent
3. Slave Agent gets the file information
4. Slave Agent wraps the file into a serializable object
5. Master Agent dispatches Slave Agent to the destination host.

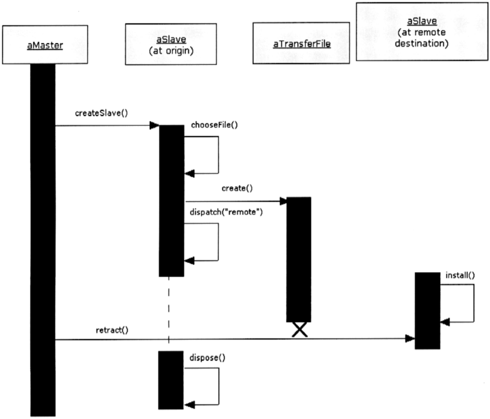


Figure 4.3
Sequences of Operations

In this system, the file is converted to a serializable object of *TransferFile* class. This is to enable the Slave Agent to bring it from origin to destination. This means that the file was changed into its sequential byte representation at origin, together with the Slave Agent, and moves to the destination node where it would be constructed back into its original state.

A TransferFile object has the following information about the file that it represents: its name, directory of origin, directory of destination and content of file. This information can be obtained using four methods : getName, getPath, getPathTo and getContent, shown in the following code fragment.

```
public class TransferFile implements Serializable{

        public String getName();

        public String getPath();

        public String getPathTo();

        public byte[] getContent();

}
```

### 4.4.2 The Slave Agent (Arrival at Client)

The steps involved in this phase are:

1. Slave Agent notifies Master Agent of its arrival.
2. Slave Agent deserializes file.
3. Slave Agent writes the file into secondary storage.
4. Slave Agent notifies Master Agent of status of transfer.

The Master Agent and Slave Agents communicate to each other using messages. Slave Agents would inform Master Agent about their arrival at destination host and also the status of transfer. This communication is handled by *handleMessage* method in *Master* class and *Slave* Class.

For example, when Slave Agent arrives at destination, it would send a message to Master Agent through its proxy, shown in the following code fragment.

```
public void onArrival() {

        masterProxy.sendMessage("arrived");

                ....

}
```

The Master Agent would then handle this message using its handleMessage method. This is shown in the following code fragment.

```
boolean handleMessage(Message message) {
        if (message.sameKind("arrived")
                ....
}
```

### 4.4.3 Results of Installation

The steps involved in this phase are:

1. Master Agent retracts Slave Agent
2. Master Agent disposes of Slave Agent

After the file transfer completes, the Master Agent would retract the Slave Agent through the Slave Agent's proxy, and then dispose it. This is shown in the following code fragment.

```
SlaveProxy.retract();
SlaveProxy.dispose();
```

This agent system was developed and implemented based on the specifications in Table 4.1 below. The findings for testing and evaluation of this agent system in terms of communication cost for various sizes of files. are presented in Chapter 5.

Table 4.1
Agent System Specifications

| | |
|---|---|
| **Platform** | Windows 98 or Windows 2000 |
| **Programming Language** | Java 2 and Java Aglet Software Development Kit 2.0.0 |
| **Nodes** | Server node and client node(s) |

### 4.5 Implementation of Java Streams Application

In order to evaluate the performance of the agent system, we developed a client-side and server-side application for file transfer. This application uses Java streams to communicate between client-side and server-side. For comparison purposes, this application also uses TransferFile class to represent the file being transferred. Figure 4.4 below shows the server-side application while Figure 4.4 shows the client-side application.

```
public class Server {
private ObjectOutputStream output;
private ObjectInputStream input;
private ServerSocket server;
private Socket connection;
connection= server.accept();
output = new ObjectOutputStream(connection.getOutputStream());
...
TransferFile tf= new TransferFile(fileFr, dirFr);
output.writeObject(tf);
output.flush();
}
```

Figure 4.4
Server-side Application

```
public class Client {
private ObjectInputStream input;
private Socket client;
private TransferFile tf;
...
client=new Socket(InetAddress.getByName(server), 5000);
input = newObjectInputStream(client.getInputStream());
...
tf=(TransferFile) input.readObject();
String file = tf.getName();
FileOutputStream fos = new FileOutputStream(file);
ObjectOutputStream oout = new ObjectOutputStream(fos);
oout.writeObject(tf);
oout.close();
}
```

Figure 4.5
Client-side Application

## 4.6 Implementation of Java RMI Application

We also developed a Java RMI application which is able to transfer files between two machines. We chose RMI because it is widely used for building distributed applications. As stated in Section 2.2, RMI also supports CS paradigm. RMI allows a Java object executing on one machine to execute methods on other machines. Six files were used in this application, shown in Table 4.2. The findings and results for the agent system and Java applications are presented in Chapter 5.

Table 4.2
Files Used in RMI Application

| Files on Server Machine | Files on Client Machine |
|---|---|
| FileServer.class | FileClient.class |
| FileImpl.class | FileImpl_Stub.class |
| FileIntf.class | |
| FileImpl_Stub.class | |
| FileImpl_Skel.class | |