Distributed Computing

Project Part 2

Joe O Flaherty

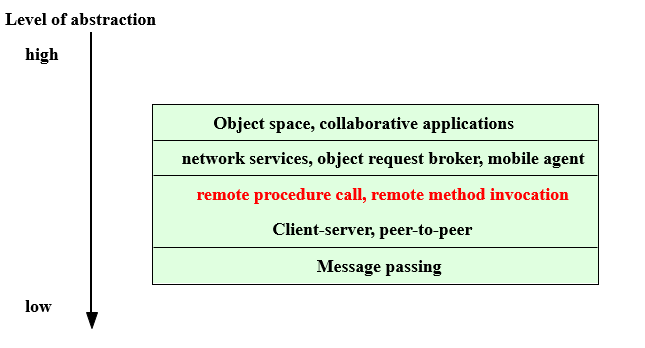
T00155775

## Description

The goal of this part of the project is to implement a Client/Server file management application similar to that developed in part one of the project using a distributed object paradigm that offers a higher level of abstraction than sockets and datagrams. Remote Method Invocation (RMI) is the paradigm chosen to implement the application. The application will allow instantiation of a Server and RMI registry and instantiation of multiple Client processes. Each user will have a personal folder on the Server plus a personal download folder on the Client. The Server will also keep a record of every user actively using the system.

## RMI

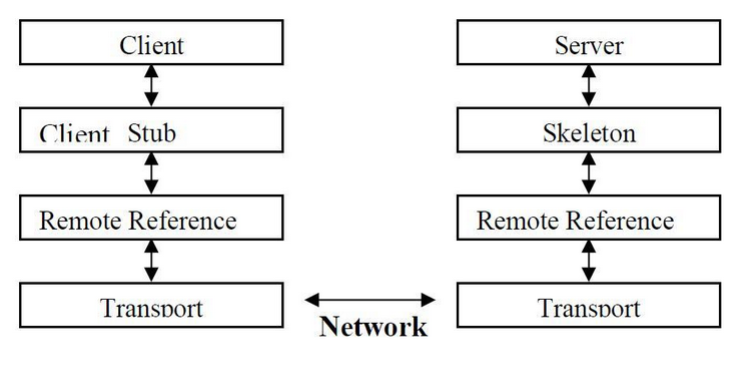
Remote Method Invocation (RMI) sits at a higher level of abstraction than the message passing protocol designed for part one of the project, both paradigms hide lower level network operations.



The architecture of RMI applications follows a standard pattern, when a remote method is invoked by a client communication passes down the layers of the architecture to the transport layer. The transport layer is responsible for any network operations and passes the invocation from the client to the server. The invocation then passes up the layers on the server side to the remote object required then the process is reversed and communications is passed from server to client.

Stubs and skeletons are generated from the implementation of a java interface using the rmic compiler. Stubs and skeletons can be dynamically loaded as required using a process referred to as dynamic class loading. Stubs are client side proxies for remote objects and are responsible for marshalling data and serialized objects before they are transported to the server from the client, and un-marshalling any replies from the server. Skeletons (server side stubs) perform this same functionality on the server side of an RMI based application, they also dispatches calls remote object implementations. The remote reference layer is responsible for passing Inter Process Communication details down to the transport layer.

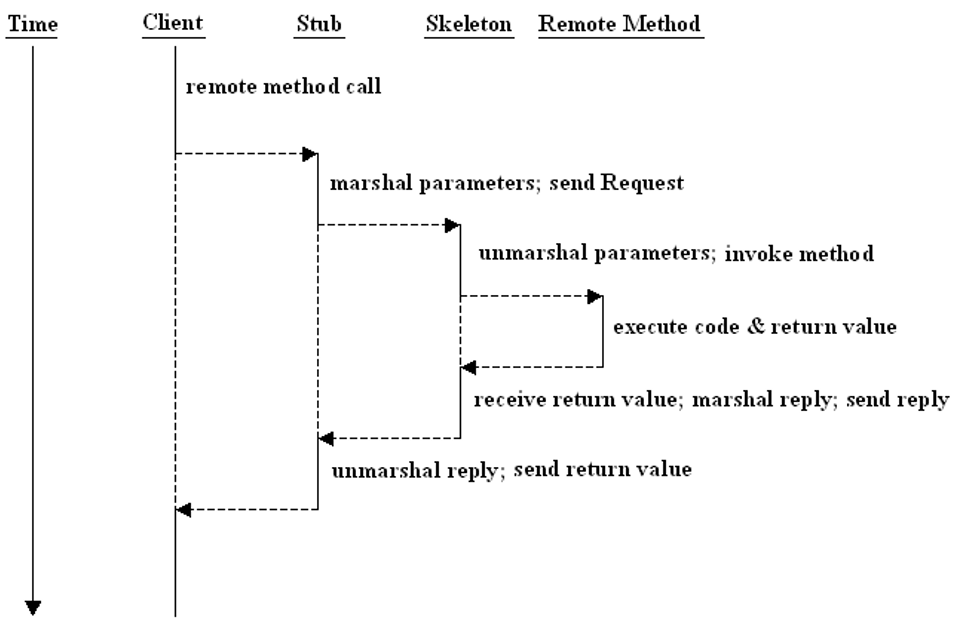
**Note:** use of the rmic compile is not strictly neccesary with newer versions of java, it is still required to make RMI applications backwardly compatible.

  
The sequence diagram is a shows the flow of invocation and response from Client to Server and back through the architecture layers.

**Application Layer**

**Presentation Layer**

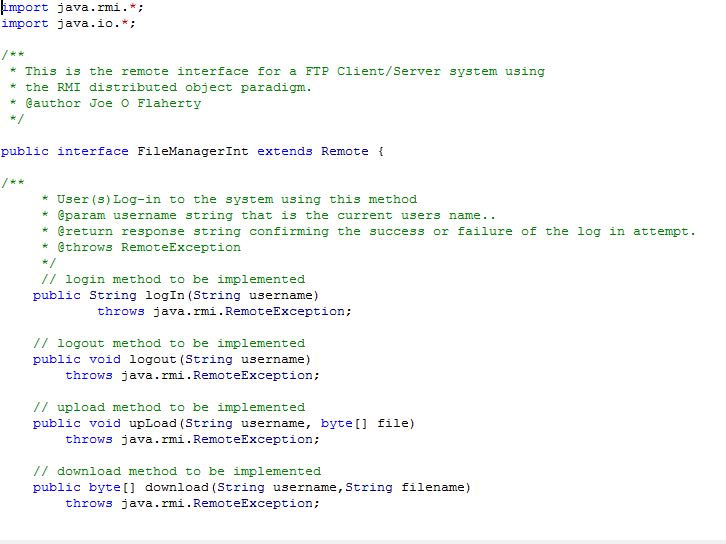
**Transport Layer**



Development of an RMI based application requires the following steps:

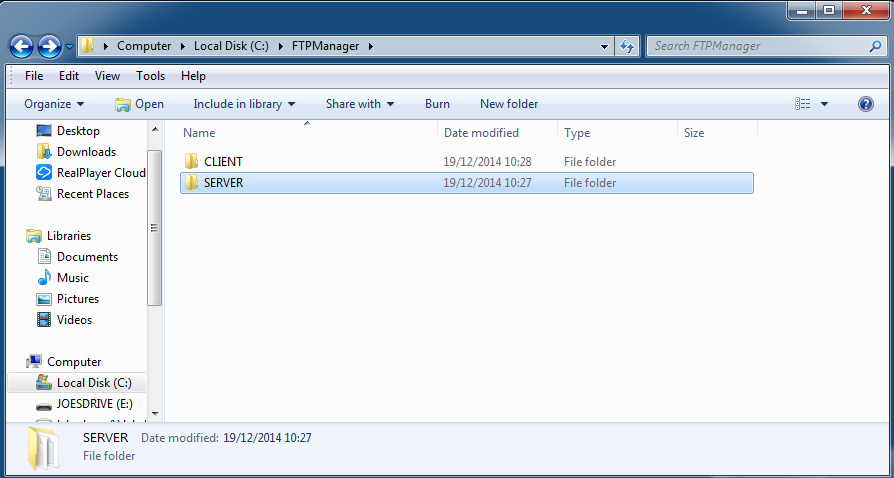
* Code a Java interface file that defines the remote methods.
* Code a class that implements the interface file.
* Define a registry which stores a list of exported objects. Clients look up the registry to check what methods are available. The registry can be shared by any servers running on the same host.
* Generate stubs and skeletons from the implementation class
* Code client and server processes.

The application developed followed the above steps firstly defining an interface and following each subsequent step as outlined.

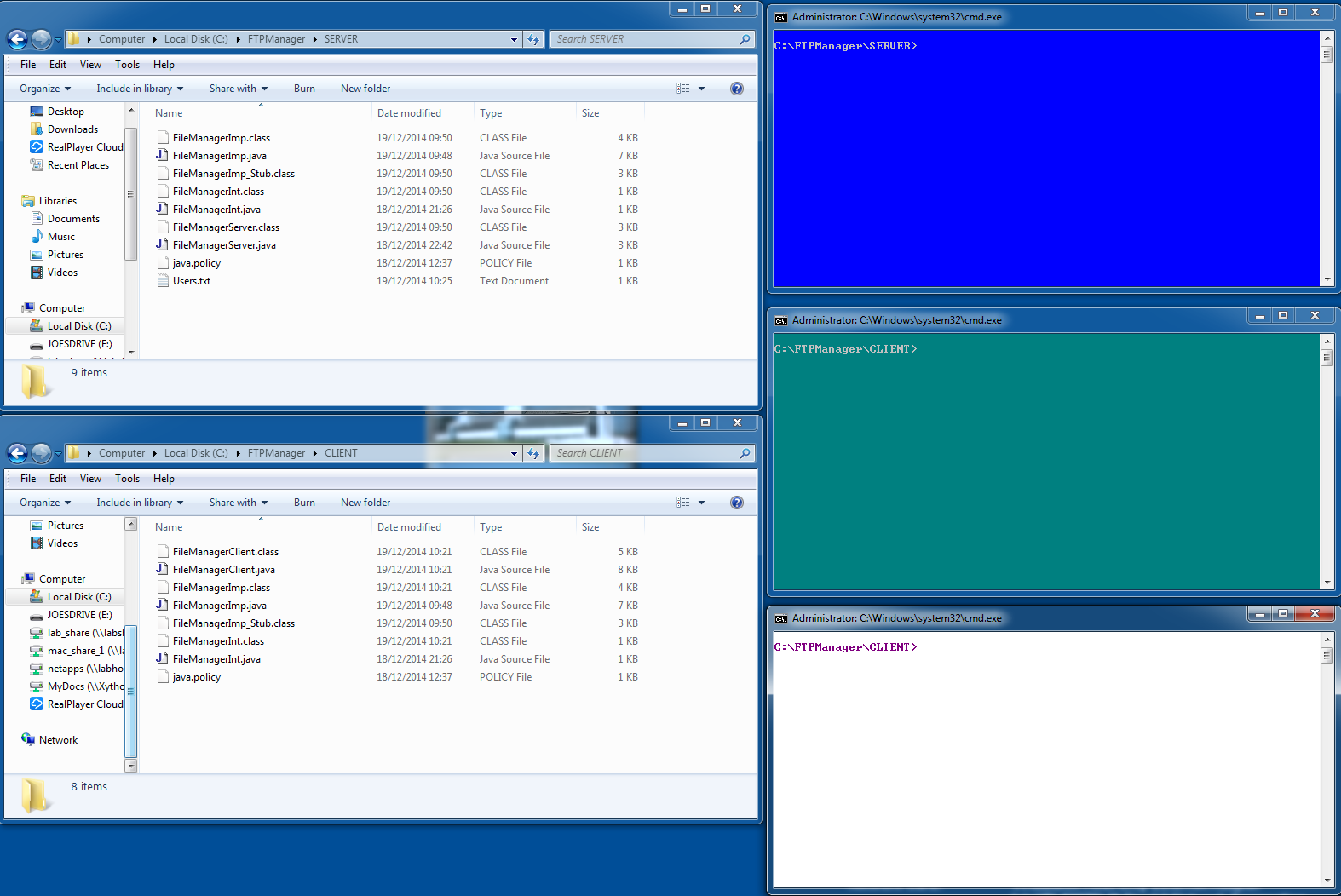


## System Execution

The system is a client/server file management system and as such has two folders a Client (users) and a Server folder.



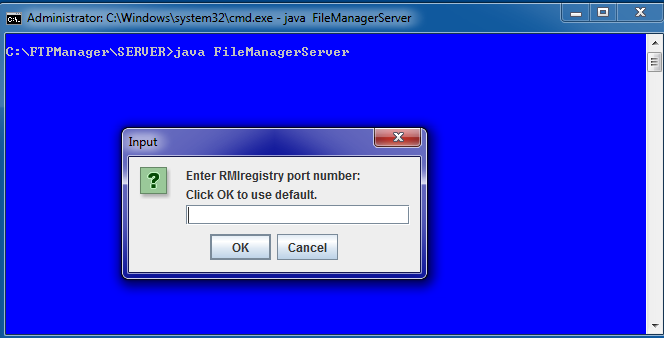
Initially both folders contain just the classes they require to run plus a java security policy file that contains all permissions required by both Client and Server.



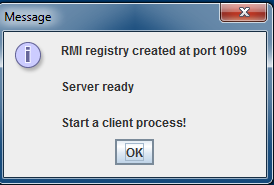
**Client Folder**

**Server Folder**

Start the Server using **java FileManagerServer** (.bat file written to bypass this step) having navigated to the Server folder via the command line interface (CLI)



The user is asked to enter a port number used to start the RMI Registry. If no port number is supplied the RMI Registry is started on the default port 1099 and a confirmation message is displayed.

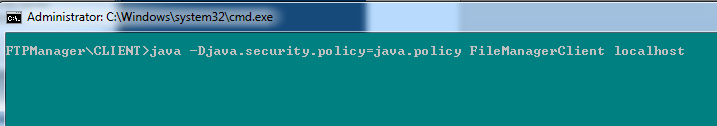


The user is also prompted to start a client process.

Client processes are started via the CLI.

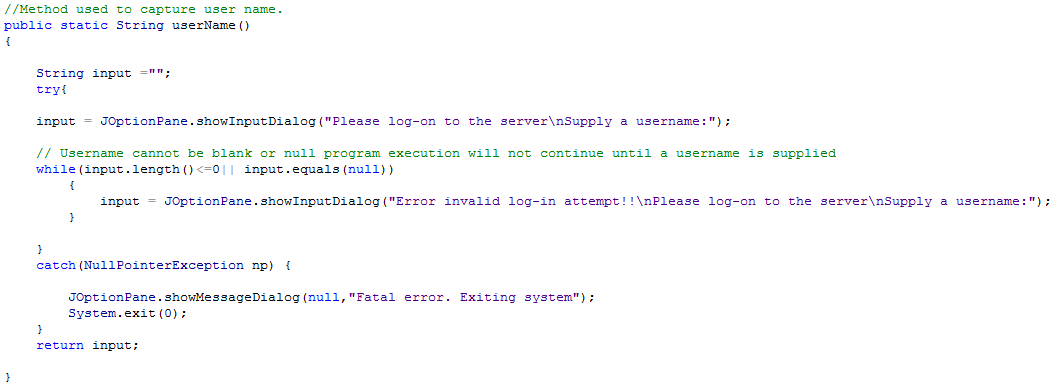
Navigate to the Client folder and use the **java –Djava.security.policy=java.policy FileManagerClient localhost** command. (.bat file written to bypass this step).

Localhost could have been hard-coded in this instance but was decided against as doing so restricts the usability of the system.

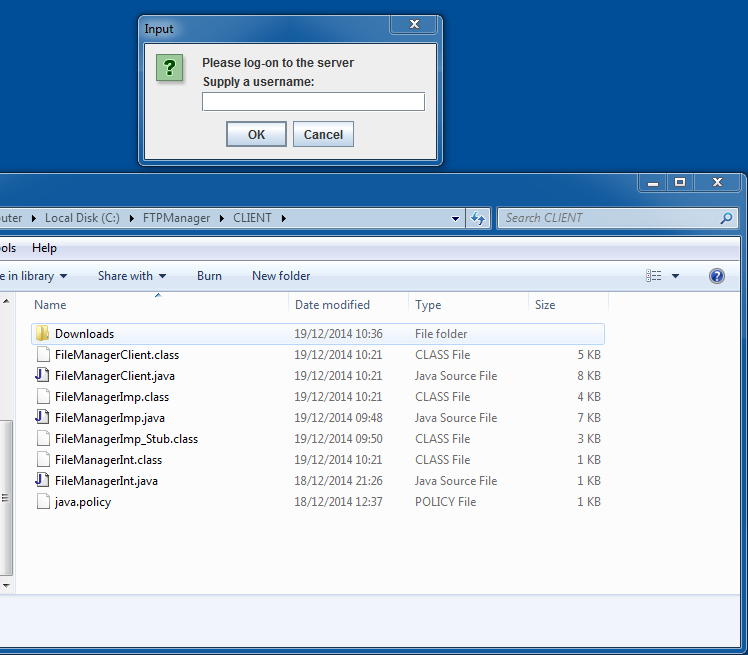


Once a Client is started a request to log-in to the server is issued, a username MUST be entered and a simple validation loop is used to ensure no blank entries are allowed.

A simple re-useable helper method was written to capture log-in details from users.



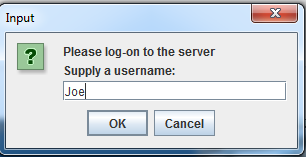
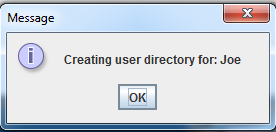
A downloads directory is created in the Client folder at this point which will be used to store download folders individual to each user.



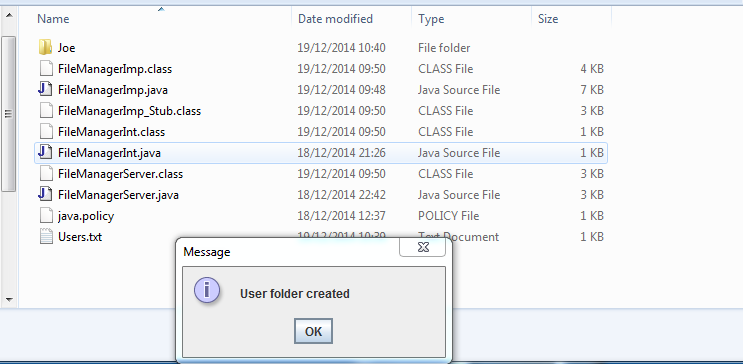
**Downloads Directory**

A username is supplied, if this is a new user a folder in their name is created on the Server, this is used to store any files that user uploads to the server.

A series of messages are displayed that indicate progress.

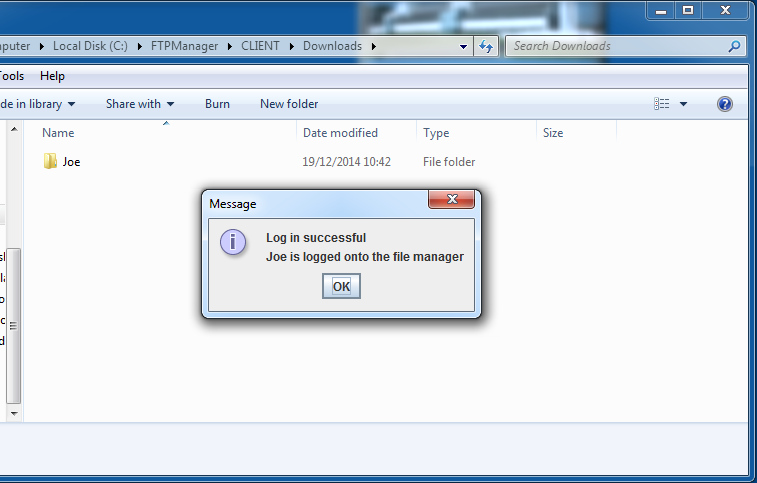
In the event that a user folder already exists in the user name a message is displayed confirming this and execution continues.



**Confirmation Messages**

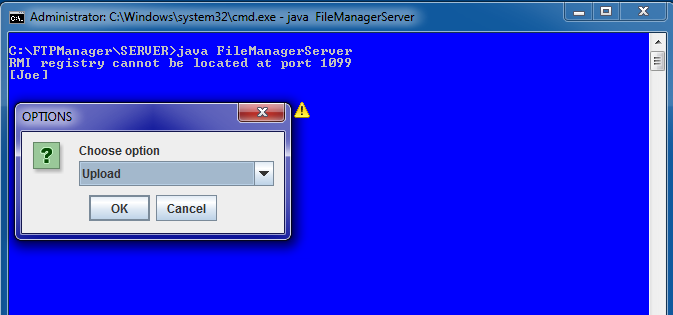
**User Folder on Server**

Following successful log-in a folder specific to the user is also created on the Client, this is created in the Downloads directory



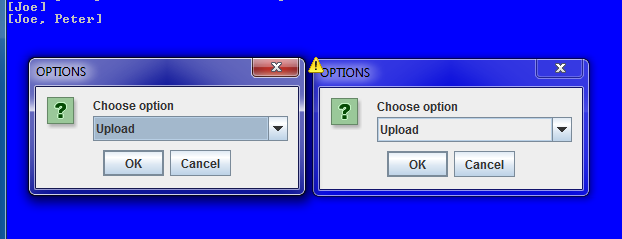
**User download folder**

The user is added to a list of logged on users which is displayed in the Server CLI (this is for confirmation purposes). The user menu is also displayed at this point. When a different user starts a Client process they are added to the list of logged in users and all operations can be performed concurrently.

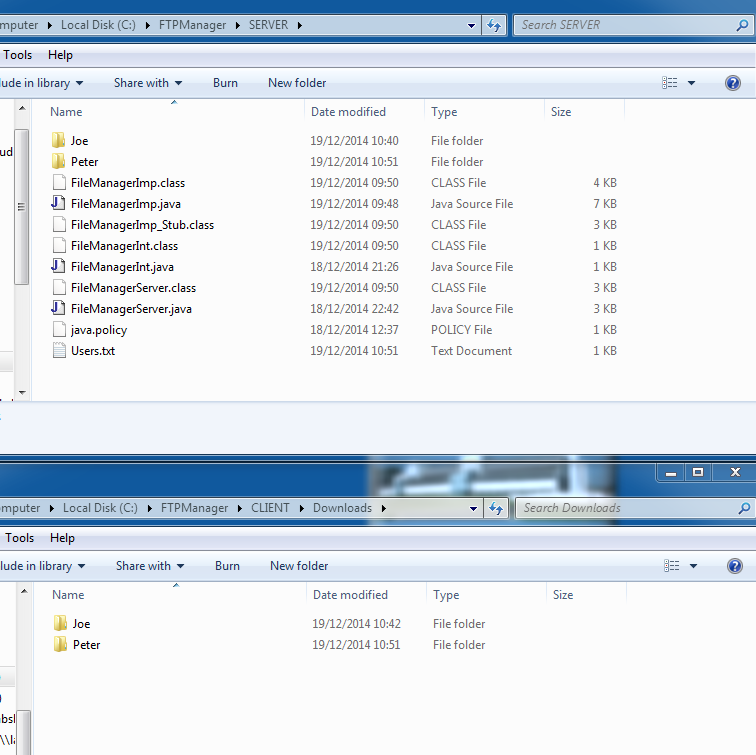


**User menu**

**Logged on users**



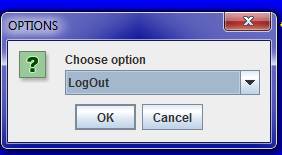
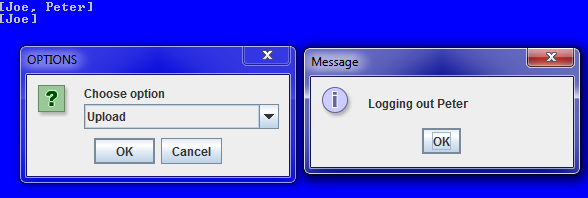
**2 users logged on**

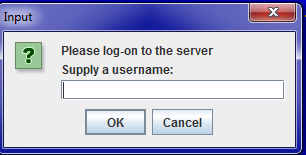
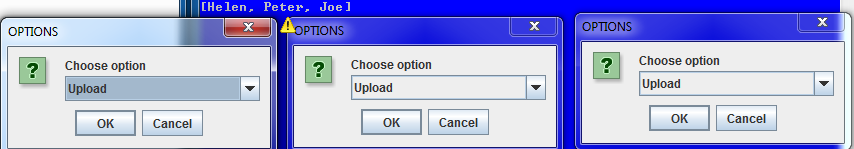


**All users have upload and download folders**

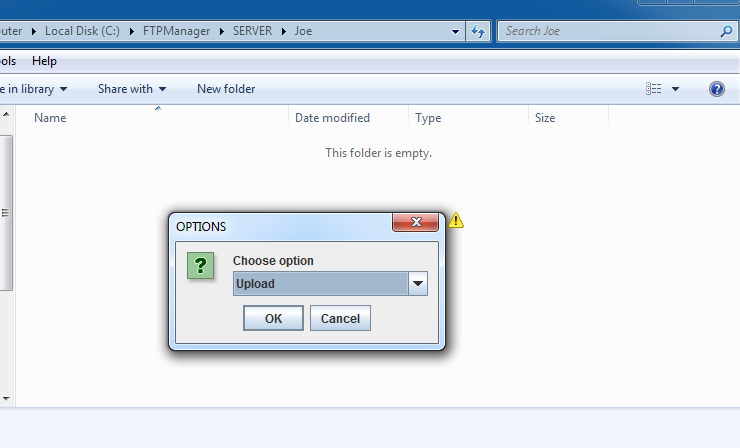
When a user logs out they are removed from the list of logged in users and the username request window is displayed for use by a new user. A log out confirmation message is also displayed to the user.

**Peter has logged out and is removed from user list**

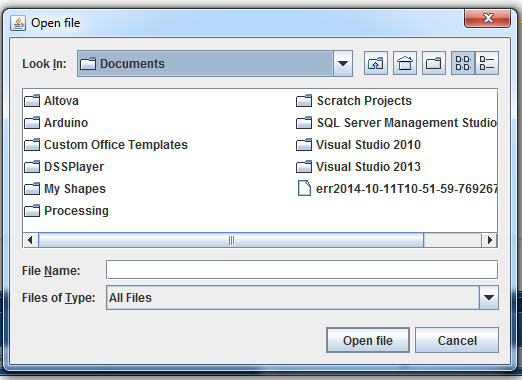
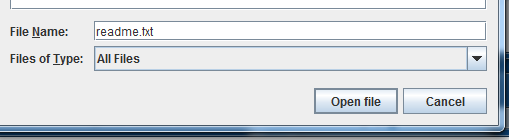
 

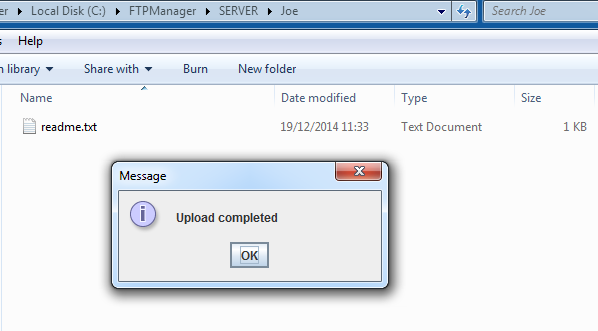
**3 concurrent users are logged in.**

**Upload** is selected from the user drop down menu and OK clicked – a file chooser is displayed to the user. Due to security issues some folders (C) can be inaccessible, the Documents folder or Desktop prove the best demonstration folders. 

**Upload is chosen, the user folder on the Server is empty at this point**

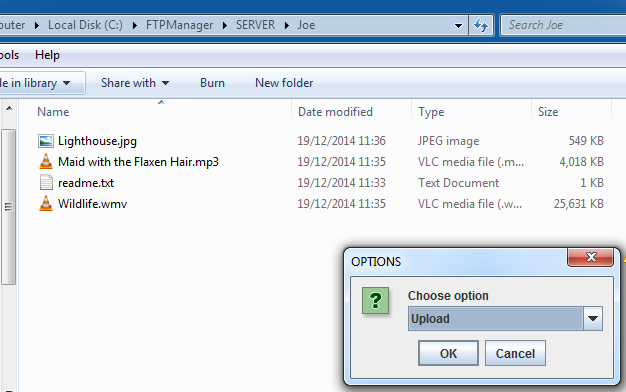
 

**The user selects a file to upload and clicks Open file option.**



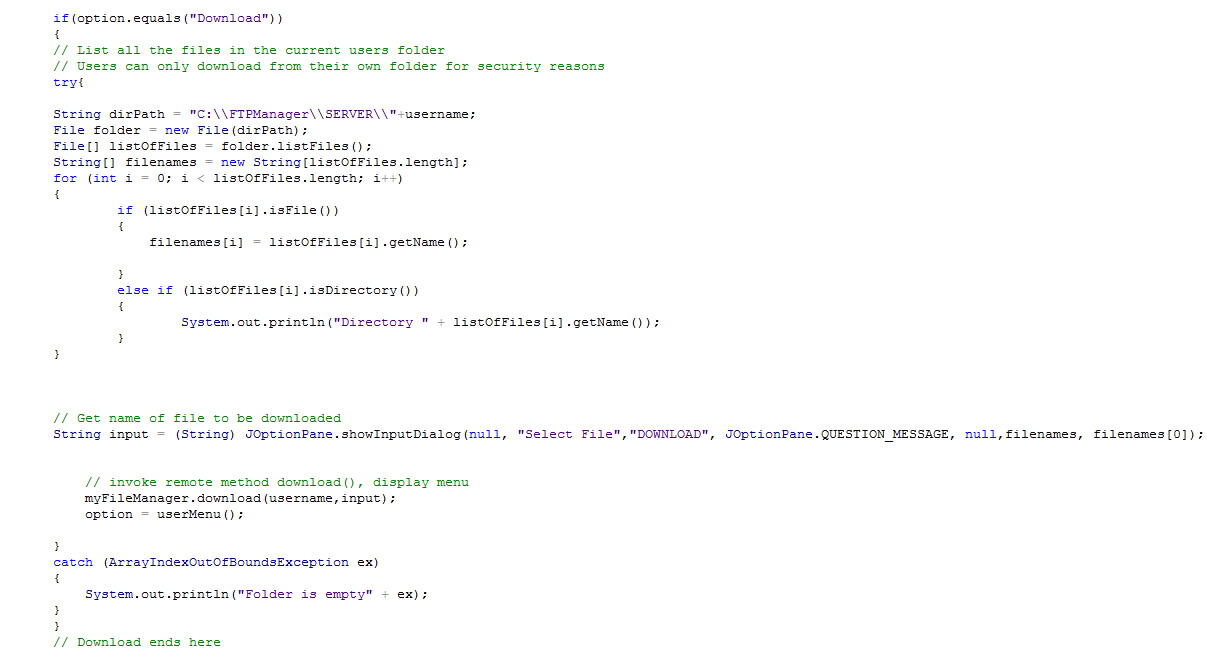
**Upload is successful. The file now appears in the user folder on the Server**

There is no restriction on the type or size of file that can be uploaded using the system.



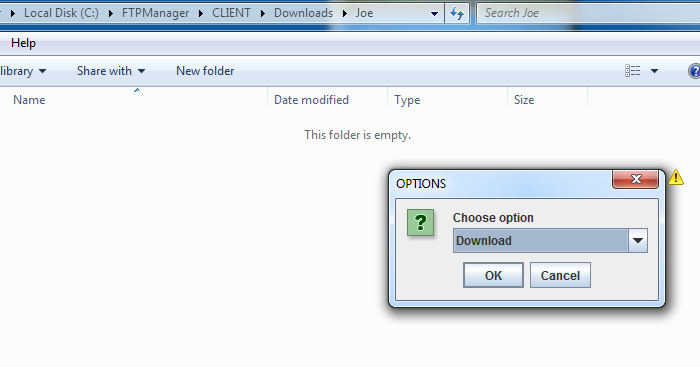
**This user has uploaded multiple file types to their folder**

The user should only have download access to files stored in their personal folder on the Server. When download is selected a list of available files in their Server folder is displayed to the user, this is accomplished using the following code.

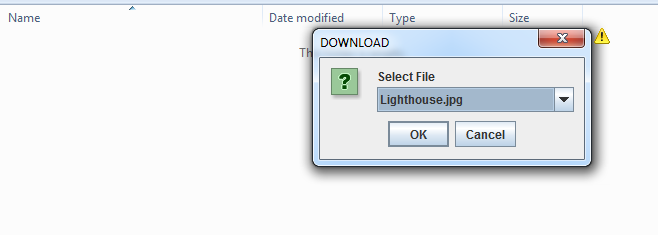
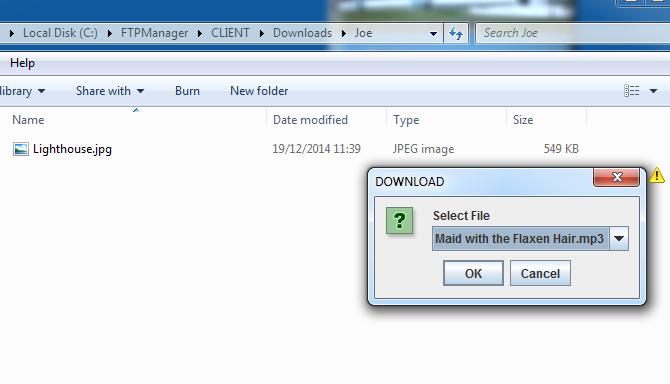


**Select file here**

**The user chooses the file to download from a list created here**

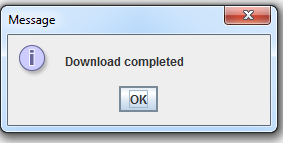
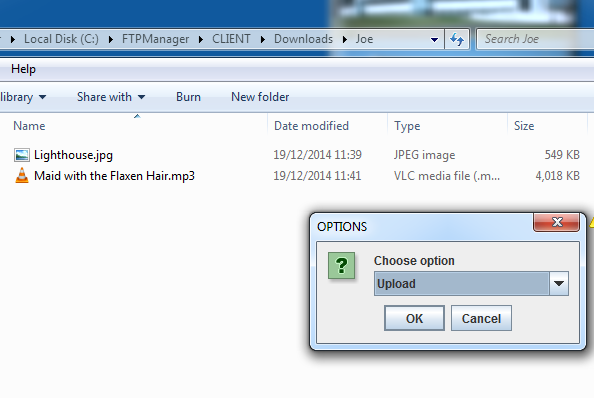


**User download folder is empty at this point**

**Select file**

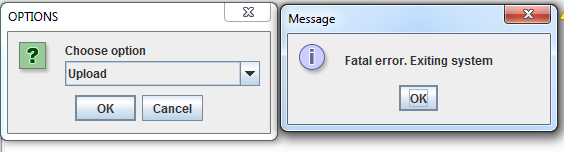
**Downloaded file in user download folder on Client**

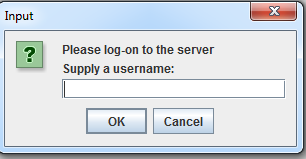
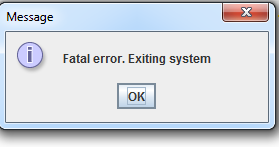
 

**No restriction to file types or size**

**Confirm success**

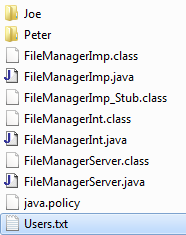
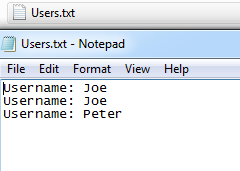
Following each successful operation the user options menu is displayed to the user, in the event that cancel is selected on either menu a fatal exception is thrown and the client process exits



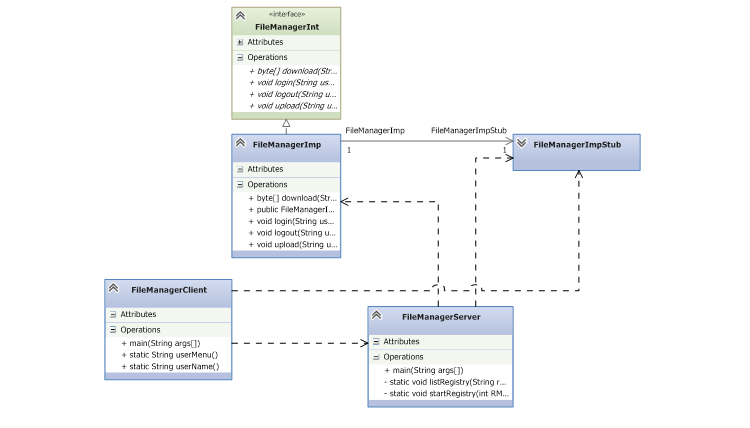
 

**Fatal errors. Cancel is selected from user menus**

The Server stores a textfile of every user that has logged onto the system for recording purposes.

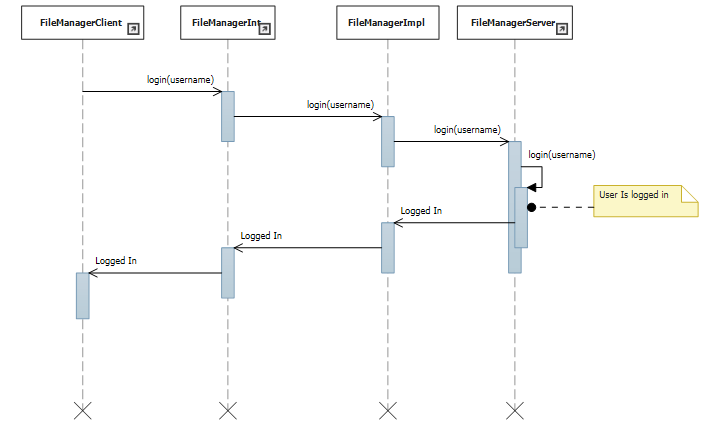
 

## Class Diagram

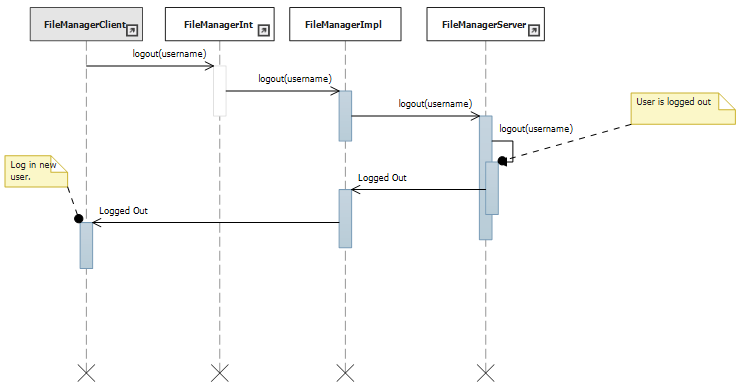


## Sequence Diagrams

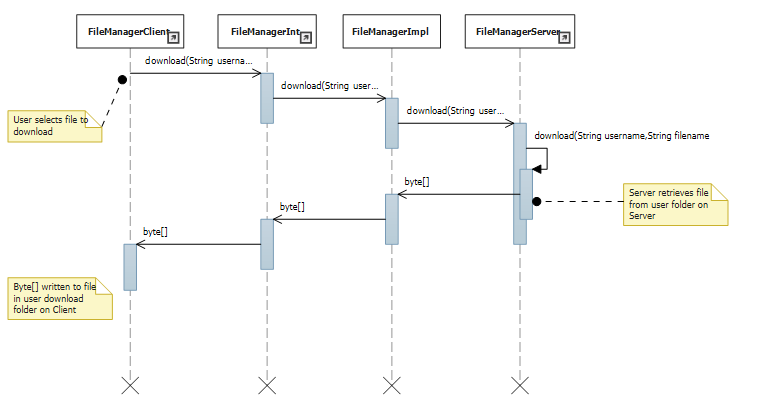
### Log-in



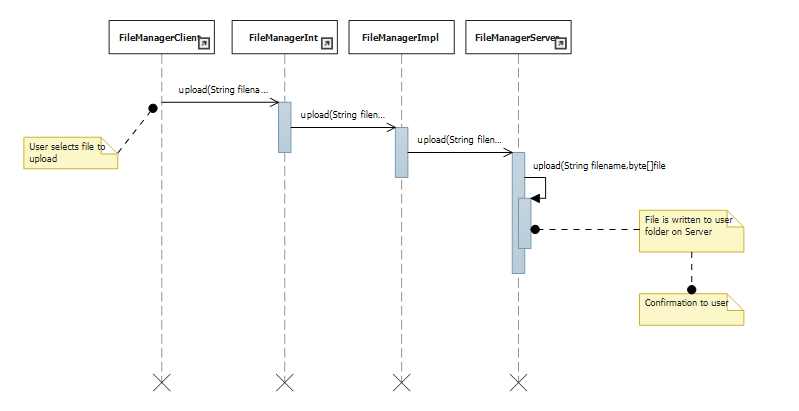
### Log-out



### Download

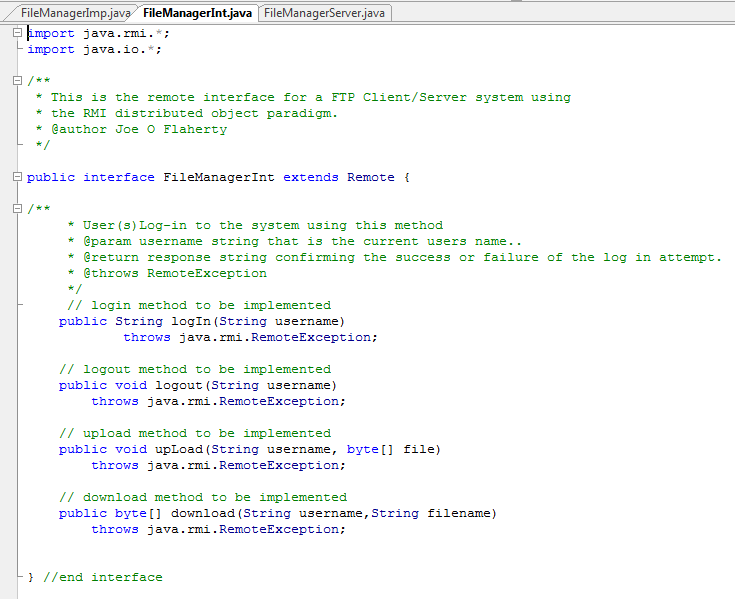


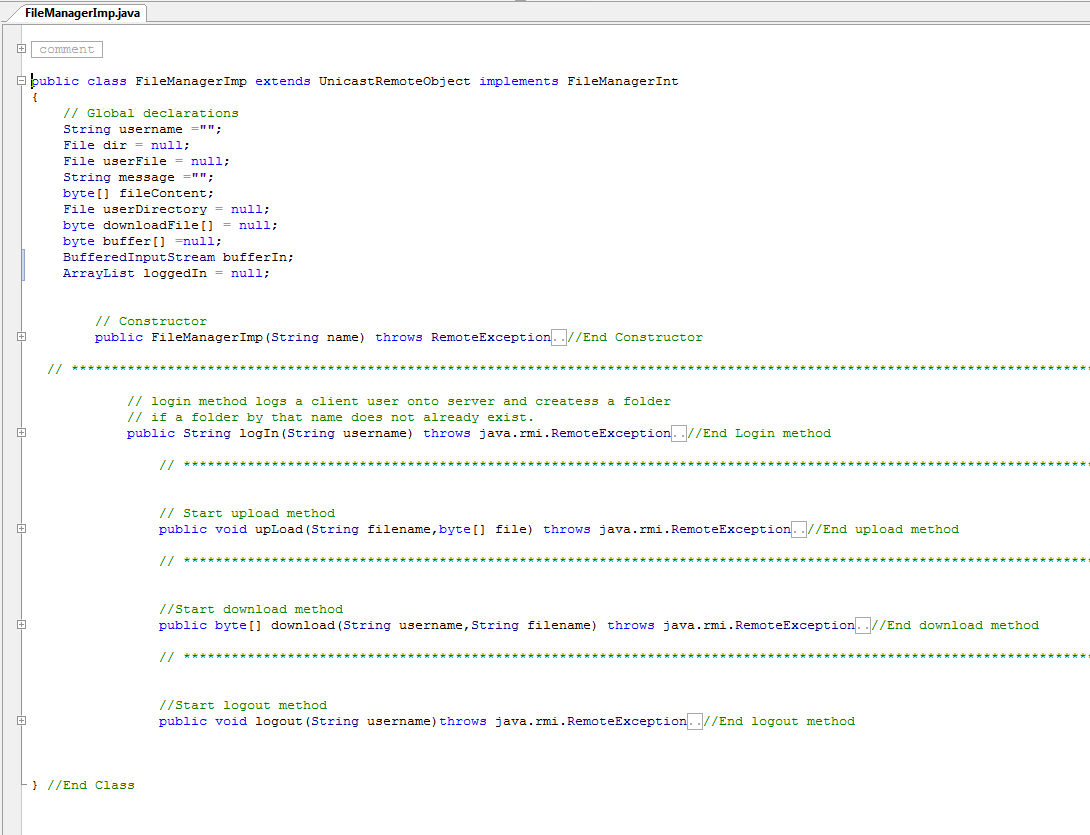
### Upload

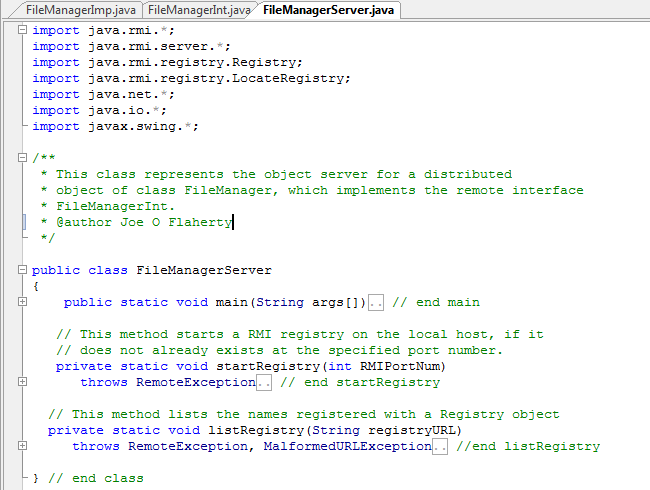


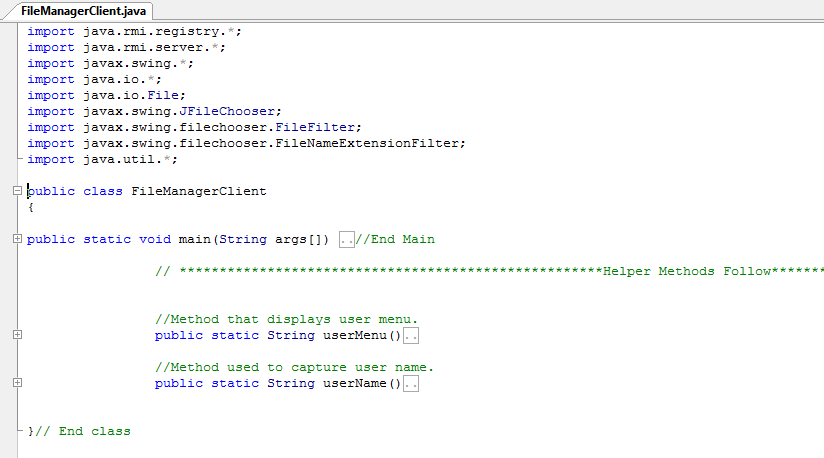
## Code snippets

### Class overview



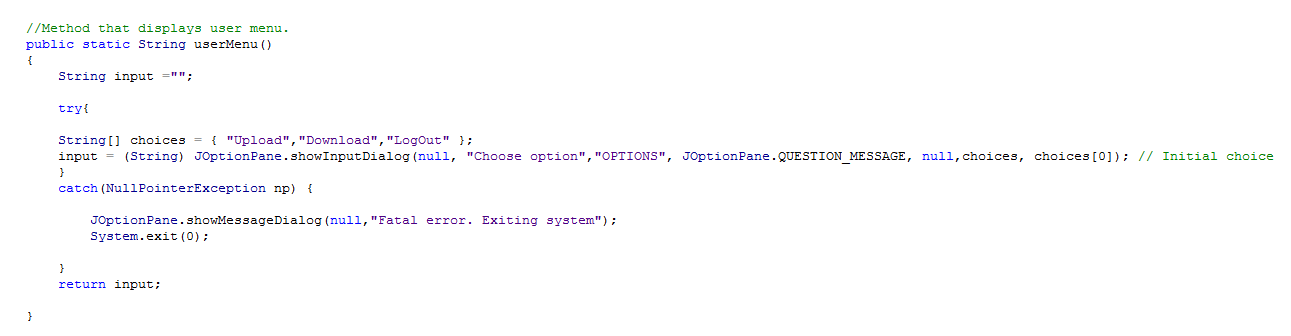






### User Menu Helper Method

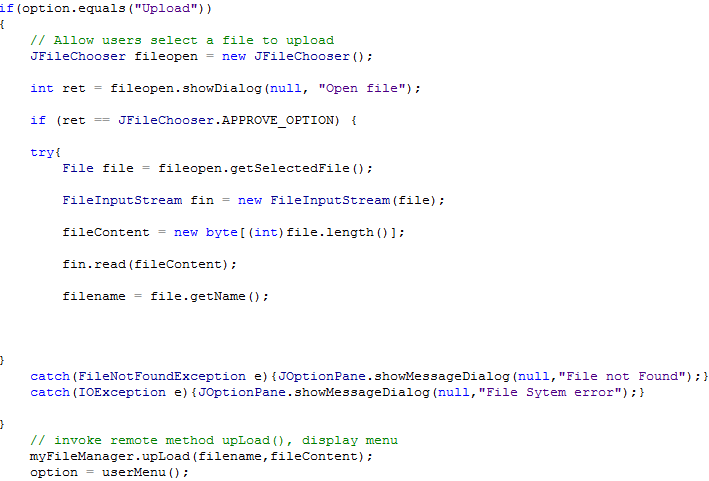
This method is used to display the user menu when appropriate.



### Upload

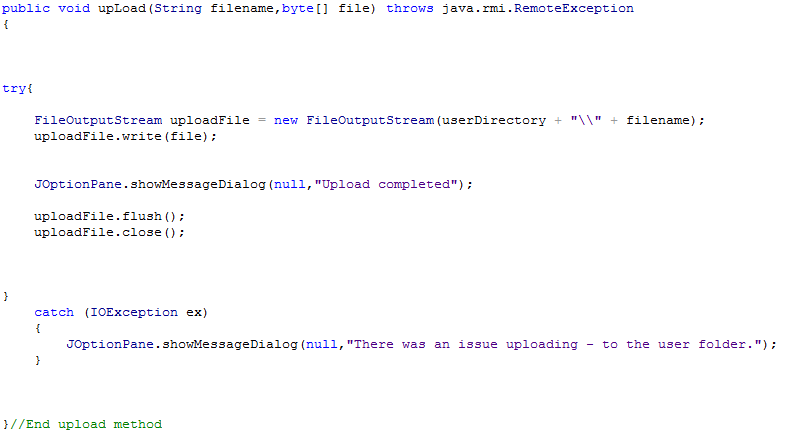
Upload client side code that displays a file chooser allowing the user select a file, writes the file contents to a byte array and sends the filename and byte array to the Server.

#### Upload Client



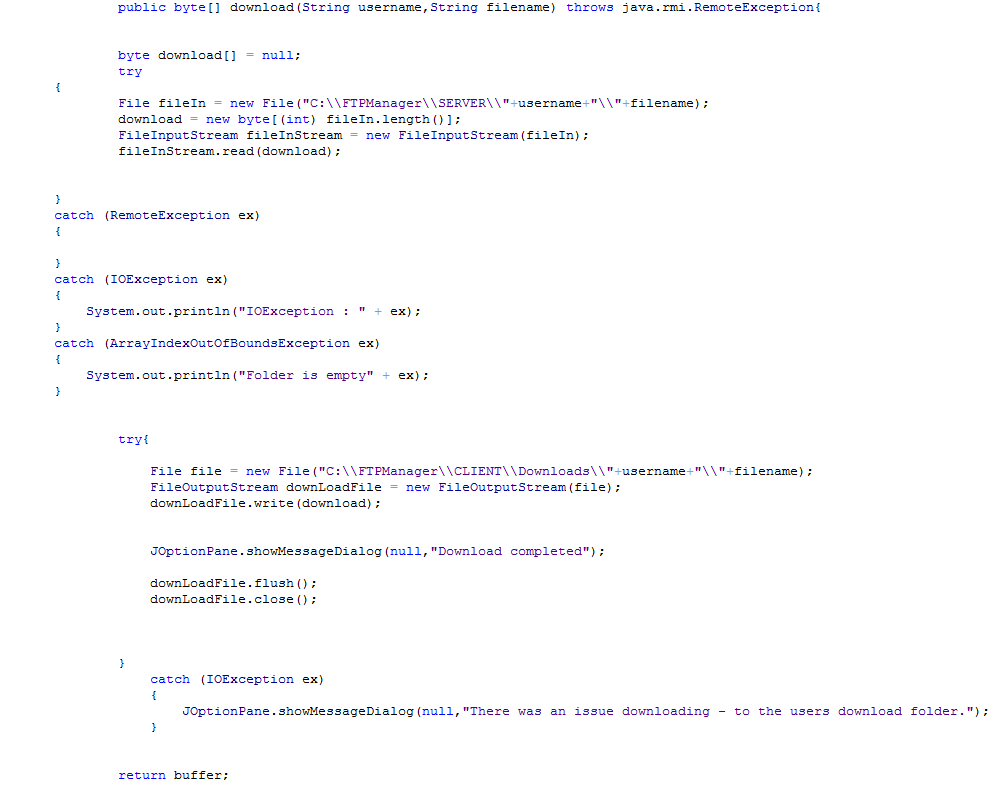
#### Upload Implementation

Implementation of the upload method that takes the byte array sent by the client and writes it to a file in the user’s folder on the server.



#### Download Implementation

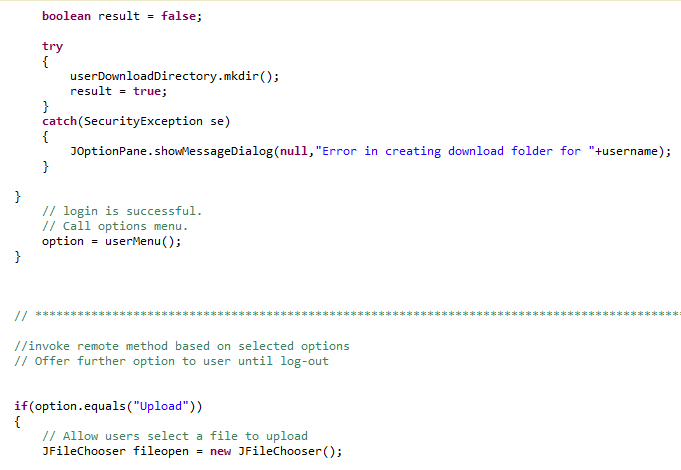
The client has selected a file from their folder on the Server to download (shown in System execution section). The method writes the contents of this file to a byte array. The contents of this byte array are written to a file in the users download folder.

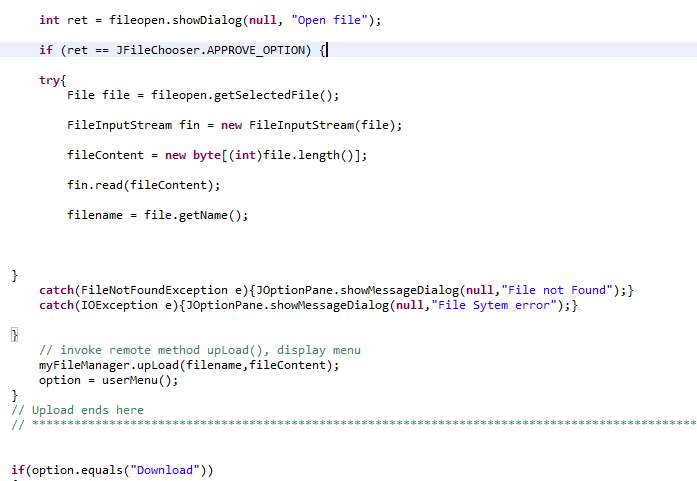


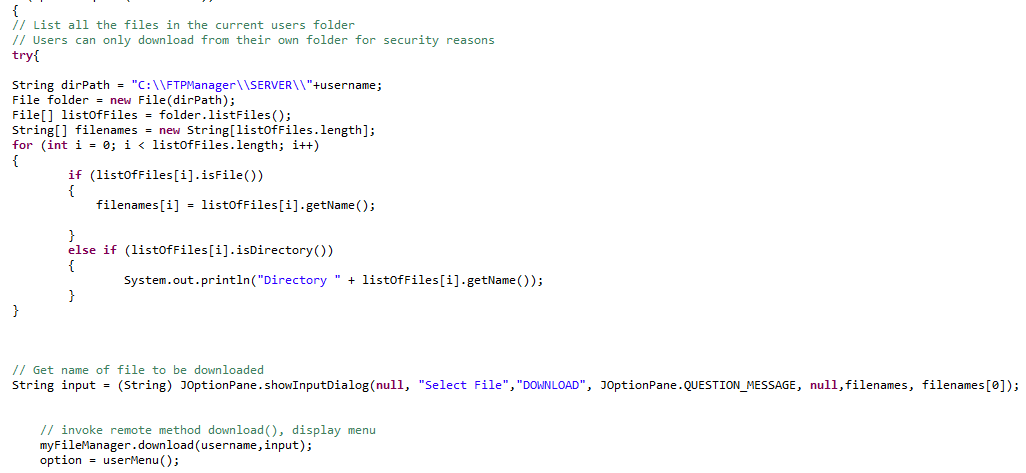
#### Client Loop

The client is controlled using a while loop plus a series of if statements which allow for continuous use by a client until log-out. The client immediately offers a log-on option following a successful log-out, this allows for new users to log in to the system.











## Evaluation

Part 1 of this project was implemented using Sockets and Datagrams and as such required a protocol which had to be designed and implemented. The protocol was responsible for establishing and maintaining a connection. The protocol was also responsible for agreeing the format of messages passed from Client to Server and back. Part two of the project has seen the need for design of a protocol eliminated, this has greatly reduced the level of complexity experienced. Many errors made in part one were a result of mistakes in protocol design and logic not actual coding of the application. Passing request and confirmation codes can easily become overly complex and it is very easy to lose track of what code should be sent and when. Without the need to track what messages and codes were passed and when implementing part two of the project much more straight forward and much quicker.

Coding an RMI application has proven more satisfying and successful than what was experienced when programming with sockets. The fact that RMI is method oriented not data oriented takes the responsibility of deciding on a format that data takes away from the developer. There was no need to decide on what format data and messages would take the middleware in this case RMI is responsible for the formatting of messages. Again this removed much complexity from coding the application essentially all that is required with the RMI solution is code the interface and implement those methods.

The only frustration found with this part of the project was as a result of choosing RMI as the middleware. RMI is a java only middleware and I found this a little restrictive, it would have been nice to implement the system in a different language. The dependency on the rmic compiler is also a little irritating when developing the application, once complete and batch files were written this issue was less of a concern. I found it a little easy to get lost at times towards the end of development when classes were in separate Client and Server folders. Any changes to the implementation class and regeneration of the stubs caused the application to fail regularly for no apparent reason which proved frustrating especially as the error was so simple.

It could be argued and I would agree that the best feature of RMI is ‘out of the box’ concurrency. There is/was no requirement to make any changes to code or include any extra functionality or complexity to the RMI application. Achieving the same result with the sockets used in part one would have been a complex operation particularly as the process was bound to a single port. The RMI solution simply requires a registry and server to be running and any number of client process can run and invoke the remote methods.

Finally I found coding the application by following the basic template of

* Interface.
* Implementation.
* Define a registry and bind/rebind to it.
* Generate stubs/skeleton
* Write client server application code

straightforward to a large degree and very satisfying. The only design difficulty encounterd was in writing the method headers for the interface i.e. deciding on what parameters and return types were required. Also the fact that getting basic functionality could be implemented extremely quickly made iterative development reasonably easy, with a little functionality added on each iteration. The fact that all functionality is delivered in this part of the project has also been very satisfying.

.