***Distributed Computing***

***T00171168 – Damien Breen***

# ***Design***

The main objectives of this project were to create a simple file transfer application that followed a protocol, which I would also need to create. The application would use datagrams to transfer message between the client and server and this would help me to become familiar with how datagrams work. This will allow me to identify some of the problems associated with using such a low-level form of programming to transfer data across a network.

To complete the project I will used three layers to implement the code. The code will be broken into Presentation, Application and Service layer. The Presentation layer is the top-most layer and allows users to interact with the more complicated layers of the systems. The Application layer is the middle layer, which deals with all logic within the system and controls all the functionality of the system. The service is the lowest and most complicated layer of the system. The service layer is used to transfer messages across the network between the client and server.

The protocol, which I created, will be implemented in the Application layer on both the client and the server. The logic and steps specified in the protocol will be followed in this layer to ensure that the system can perform the required actions as specified in the protocol.

## ***Client Side Code***

|  |  |
| --- | --- |
| ***Presentation Layer*** | ClientInterface.java/Client.java |
| ***Application Layer*** | ClientHelper.java |
| ***Service Layer*** | MyClientDatagramSocket.java |

On the client side, I started by creating a simple command line program (Client.java) which allowed me to interact with my server and I used this to test that the server was working as intended. Once I had everything working, I created a simple UI (ClientInterface.java) which would allow users to interact with the server in a more user-friendly manner. I kept the UI simple as I did not want to spend too much time on this but instead I concentrated on the functionality if the project.

For the Application layer I created the ClientHelper.java class, which contained all the logic on the client side. The helper class constructed the messages that were sent across the network to the server. The messages are constructed using user inputted data from the user.

On the client side, the ClientHelper.java class will create the messages depending on what the user wants to do. The messages are created dependant on what the user does on the UI.

On the client side, MyClientDatagramSocket.java is used to send messages across the network as datagrams.

## ***Server Side Code***

|  |  |
| --- | --- |
| ***Presentation Layer*** | Server.java |
| ***Application Layer*** | Server.java |
| ***Service Layer*** | MyServerDatagramSocket.java  DatagramMessage.java |

When beginning the project I used the echo client-server example as the basis for my project. I followed the same approach and divided my code into three different layers, Presentation, Application and Service.

The presentation layer contained code that dealt with the user interface or code, which started a service or process.

On the server side, I had a simple command line class called Server.java, which started the server and waited for requests from clients.

The Application Layer deals with how messages are managed across on the client and the server.

On the server side, the messages are manged in Server.java. When a message is received by the server, it is examined to determine what the user wants to do.

The final layer is the Service Layer and this deals with how the messages are constructed so that they can be transferred across the network with datagrams.

On the server side, MyServerDatagramSocket.java and DatagramMessage.java are used to receive messages from clients. The code also enables responses to be sent back to clients by using their port and address.

# ***Protocol***

## ***Login***

***Message:*** "LOGIN <username>"

***Description:*** To login pass the message “LOGIN <username>” will be passed to the server. The parameter <username> is the user who wishes to login and this user will be added to the list of logged in users if successful.

***Codes:*** 200: login successful

201: account created and logged in

401: user already logged in

***Workflow:***

* Check if user <username> has a folder.
  1. If <username> does not have an account create one and log user in (add to list of logged in users).

Return Status: 201

* 1. If <username> has an account, check to see if already logged in.
     1. If not logged in (add to list of logged in users).

Return Status: 200

* + 1. If already logged in (check the list of logged in users).

Return Status: 401

***Sequence Diagram:***

201 (Account created)) d

Client

Server

LOGIN <username>

200 (Login successful)

401 (User already logged on)d

## ***Logout***

***Message:*** "LOGOUT <username>"

***Description:*** To logout of the application, the message “LOGOUT <username>” is sent to the server. This parameter <username> is the name of the user who wishes to logout and if successful the <username> will be removed from the list of logged in users.

***Codes:***  200: User successfully logged out.

402: User cannot logout, as they are not logged in.

***Workflow:***

* Check if <username> is logged in (check the list of logged in users)
  1. If logged in, log out <username> (remove from list of logged in users).

Return Status: 200

* 1. If not logged in, user cannot be logged out.

Return Status: 402

***Sequence Diagram***

Client

Server

LOGOUT <username>

200 (Successfully logout)

402 (Unseccessful logout)d

## ***Upload***

***Message:*** "UPLOAD <filename> <file>"

***Description:*** To upload a file the message “UPLOAD <filename> <file>” is sent to the server. The parameter <filename> is the name you want to save the file as and the <file> parameter is the file you want to upload to the server.

***Codes:***  200: File successfully uploaded.

405: A file already exists with that file name.

***Workflow:***

* Check users existing files for file of same name
  1. Check if file already exits with the same name (check users existing files)

Return Status: 405

* 1. If file doesn’t exist, save the file on the server(write the file to the users folder)

Return Status: 200

***Sequence Diagram:***

Client

Server

UPLOAD <filename> <file>

200 (Successful upload)

405 (file already exists with name)d

## ***Download***

***Message:*** "DOWNLOAD <filename>”

***Description:*** To download a file the message “DOWNLOAD <filename> <download\_location>” is sent to the server. The <filename> parameter is the name of the file you wish to download (check users list of files).

***Codes:*** 200: File successfully downloaded.

404: Unable to locate file with that name.

***Workflow:***

* Check to see if the file exists in the users’ folder (check users folder)
  1. If file not found

Return Status: 404

* 1. If file can be downloaded(return file to user)

Return Status: 200

***Sequence Diagram:***

Client

Server

DOWNLOAD <filename>

200 (Successful download) + file

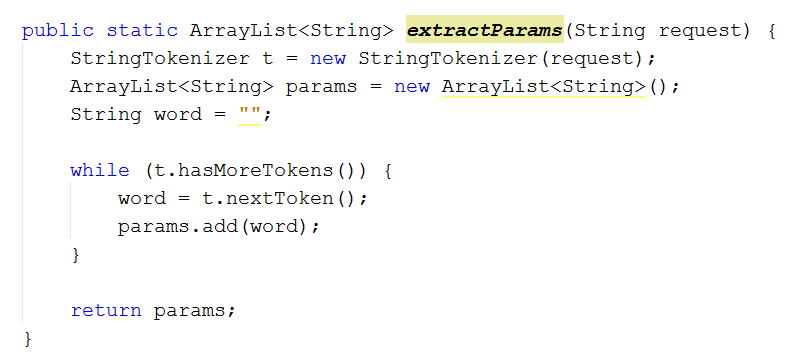
404 (file not found)

# ***Code***

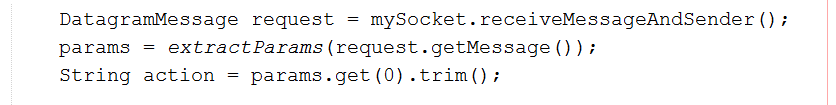
## ***Server***

When deciding how I would interpret the messages on the server side I decided that my messages would all need to follow a similar pattern or else it would have been impossible to know what request the client has sent. I decided that the messages would contain different parameters where each was divided by a space or empty string. By doing this, it allowed me to get each parameter from the string that was sent in the datagram. In the message, the first string would be the action that the client wanted to execute. This was important, as the number of parameters would vary depending on the request. By letting the action be the first parameter I could then set the code up to perform different actions to the request depending on the action.

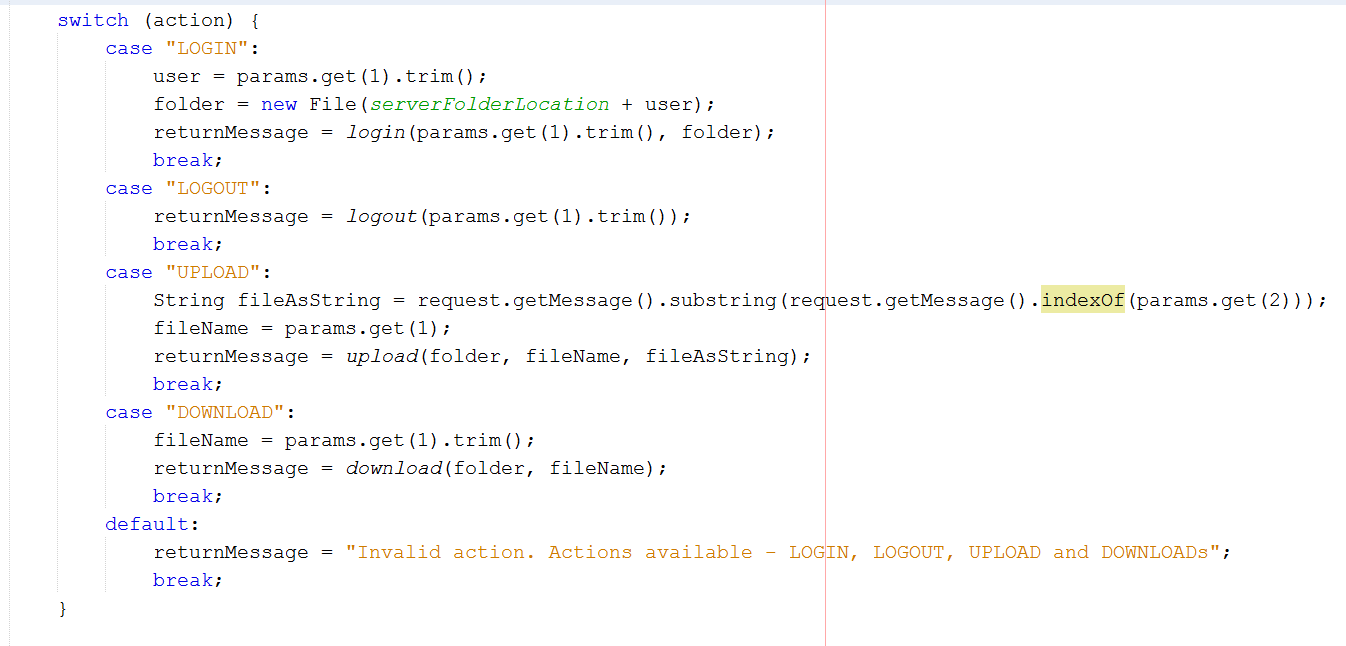
To be able to determine the action that was sent to the server I needed a method, which would extract the parameters from the message that was sent to the server.



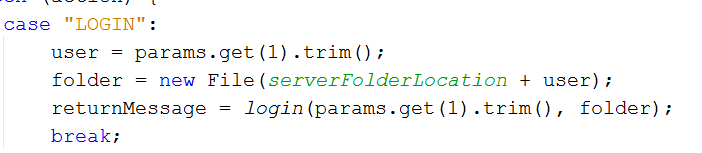
The method extractParams takes in the string request that the server has just received and extracts each of the words from it. Once extracted the words are added to the ArrayList params. This ArrayList is returned from the method and this can then be used to get all of the different parameters from the request.



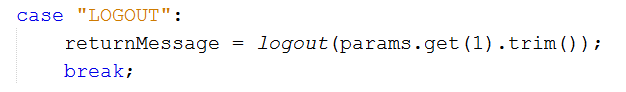
In this code, the request is the Datagram that has been sent to the server and I then pass the message to the extractParams method. This method returns the params, which is the ArrayList of parameters that are in the request. I can then get the action by getting the first parameter from the ArrayList. This value will always be either LOGIN, LOGOUT, UPLOAD or DOWNLOAD as this is how each of the messages start according to my protocol.

As I now know what action the user wants to complete I can call the appropriate method using a simple switch statement.

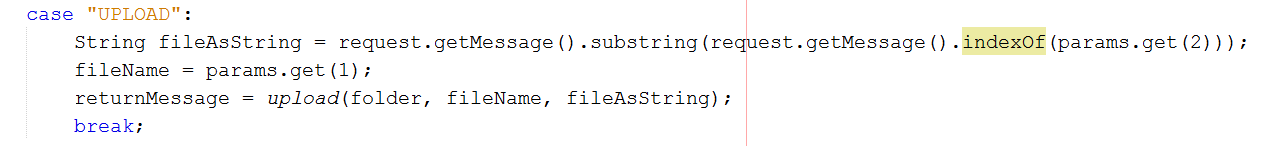
Within the case statements, different steps need to be completed, as each action will have different parameters.



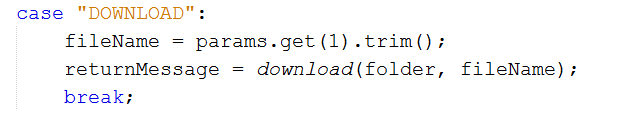
In the “LOGIN” case I know from my protocol that the second parameter will be the username if the person who wishes to login. Here I also set the folder variable to be this new users folder. This will allow me to upload and download to this folder later on.



In the “LOGOUT” case, I simple call the logout method and pass it the second parameter witch will be the username of the user who wants to logout.



In the “UPLOAD” case, I know that the second parameter is the name of the file that the user wants to upload. In my protocol, I know that this message will contain the file that the user wants to save. To get this I get a substring of the request. This substring is assigned to the String fileAsString. This string is then passed to the upload method along with the users’ folder and the filename.

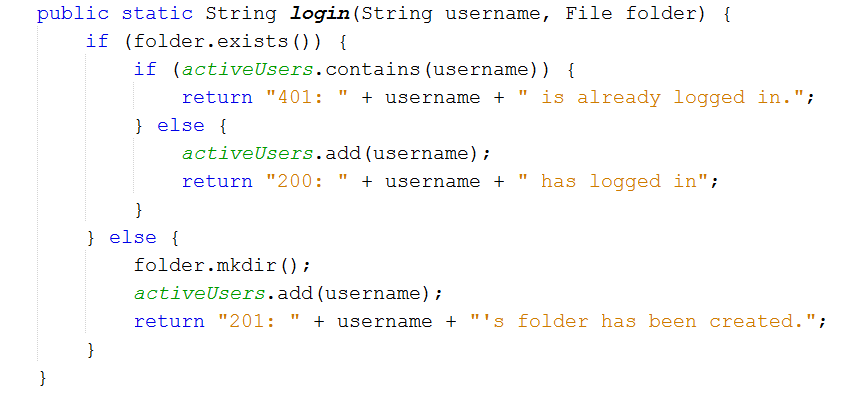


In the “DOWNLOAD” case, I know that the message will contain the filename of the file that the user want to download. This filename can be then passed to the download method.

Once the request has reached this switch statement, the server will have established what the user wants to do. From here, the code can follow my protocol and perform the action in the appropriate method.

### ***Login***

The login method will follow the protocol and executes code, which will log the user in.

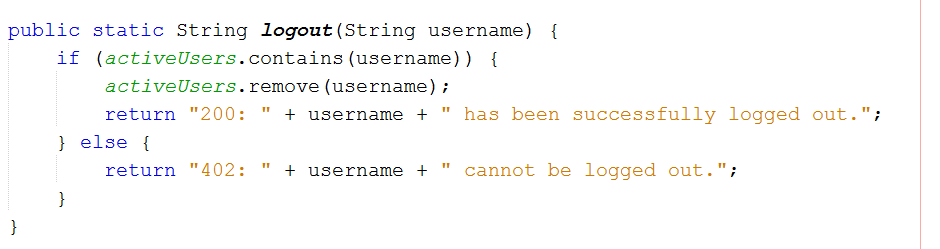


As per my protocol, a check is done to see if the user already has a folder. If they do a further check is done to see of the user is on the list of active users. If they are, a 401 response is returned, if not on the list, the user is added to the list and a 200 is returned. If the user doesn’t have a folder I know that they have never logged in before so I create their folder and add them to the list of active user and finally return a 201 response.

For my project, I decided that I wanted a user to be only able to log in on one device at a time and this is why I check the list of active users. This is something that is not necessary but is something I wanted to include as it meant added more steps to my protocol.

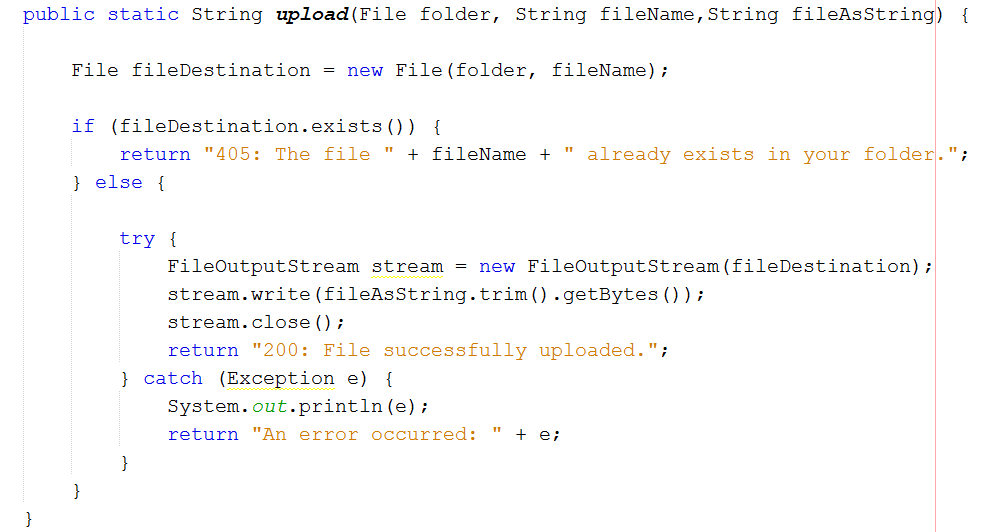
### ***Logout***

The logout method will execute some simple code that will logged out a user from the system.

This simple logout method will follow the direction of my protocol and will check if the user is logged in by checking that the username is in the list of active users. If the user is on the list, they are removed and a 200 response is returned to the server. If the user is not on the list a 402, response is returned to the client.

### ***Upload***

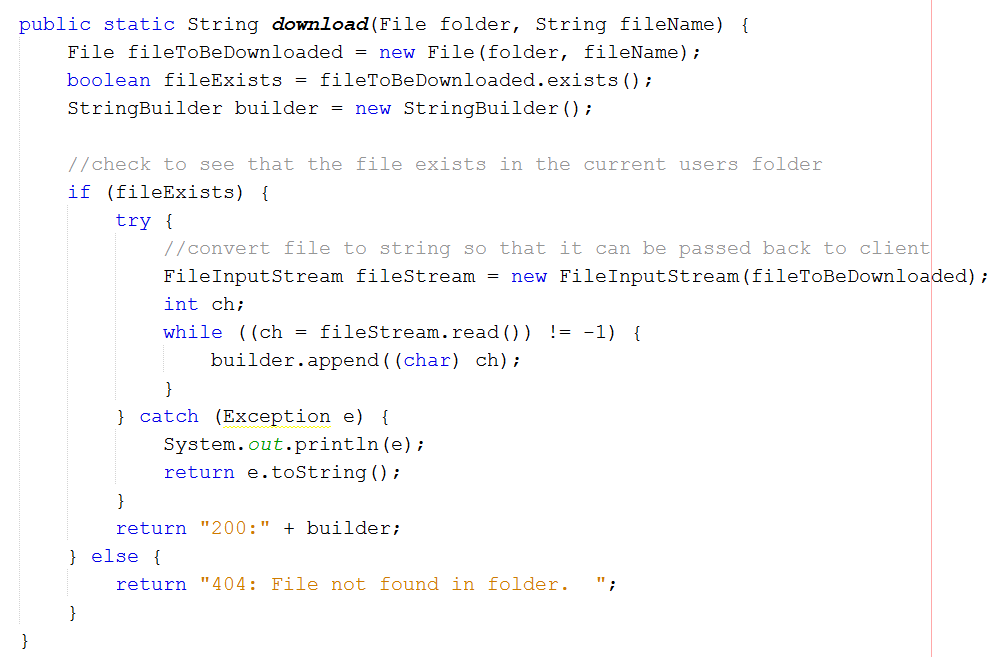
The upload method takes a file and stores it in the current users’ folder by following the steps defined in my protocol.



In this, upload method a check I done to see if a file already, exist with the same name as the new file. This is so that a user cannot save two files with the same name. If one already exists a 405 response is returned to the client. If it does not already exist, a stream is used to write the file to a file in the users’ folder. If this is successful a 200 response is sent back to the client. However if an error occurs a message is sent with the exception that occurred.

### ***Download***

The download method allows a user to search for a file in their folder on the server and then download it to their client machine.



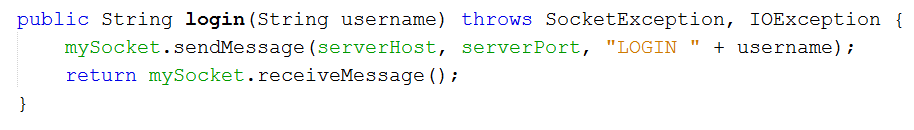
In the download method, a check is done to see if the users’ folder contains the desired file. If the file does not exist a 404 response is sent back to the client. If the file does exist a file, input stream is used to read its contents into a String Builder, which will be returned to the client. This will allow the file to be downloaded to a file on the clients’ machine.

## ***Client***

On the client side most of the logic takes place in the ClientHelper.java class, as this is where the messages are constructed and sent across the network.

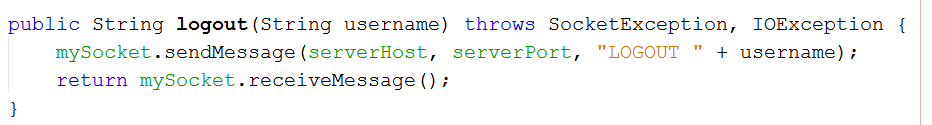
### ***Login***

The login method is very simple as all that need to be sent to the server is the message “LOGIN <username>”.



This method simply takes a username and this is sent to the server along with the server host and server port. This method is called from the presentation layer and when a response has been received from the server, it can be returned to the presentation layer.

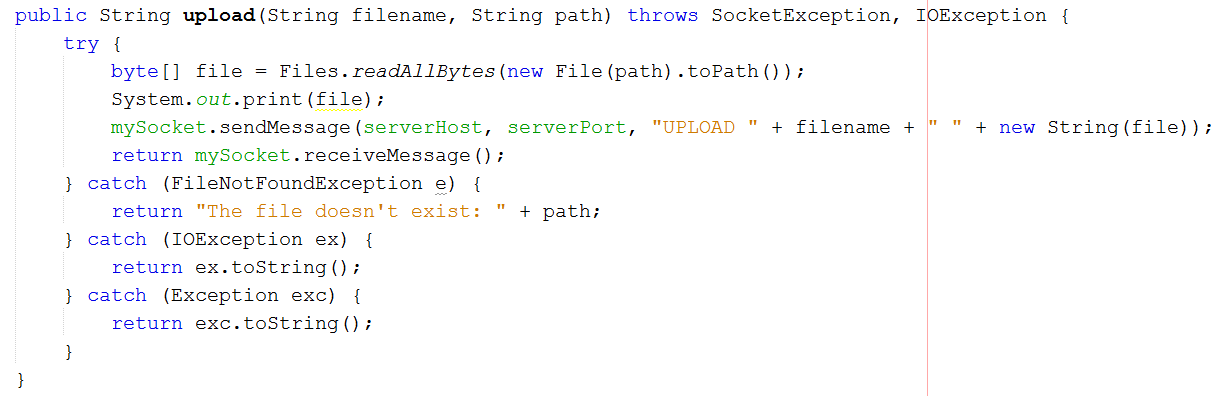
### ***Logout***

The logout method is just as simple and the only difference is the message that is sent. 

This method sends “LOGOUT <username>” to the server.

### ***Upload***

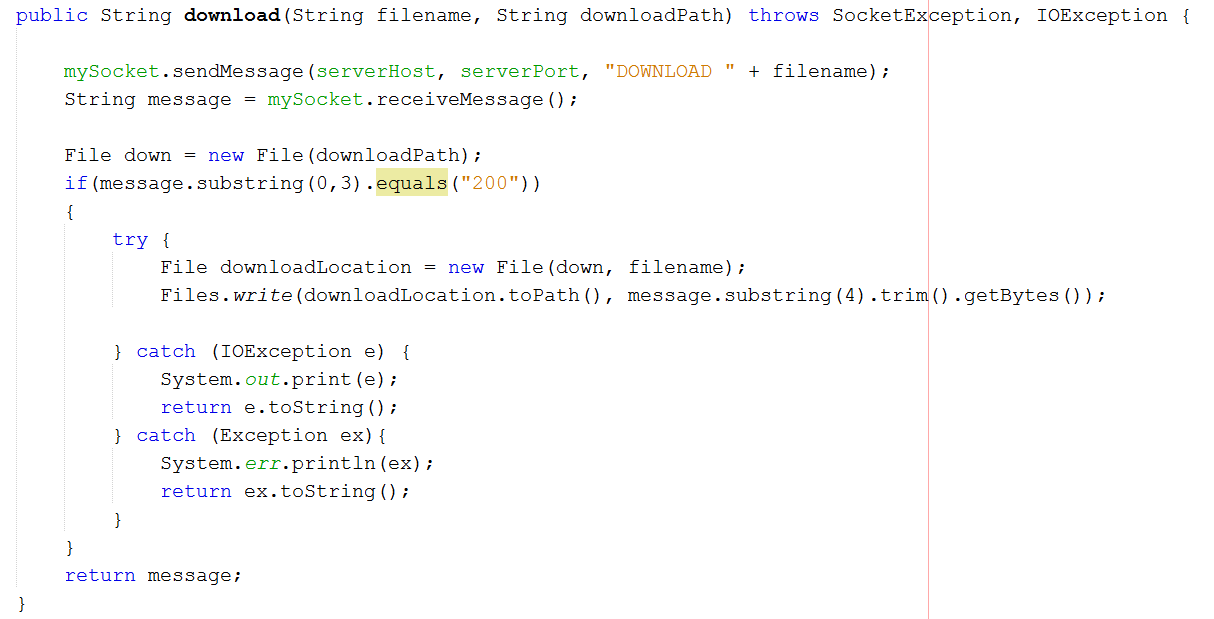
The upload method is a bit more complicated as it involves files and sending them to the server.



To send the file to the server a filename and a file needs to be sent to the server as per the protocol. This method takes the path to the file and uses this to create a file and then converting the file to bytes. Once the byte array has been created, it can be sent to the server as a string. The message is passed as “UPLOAD <filename> <file>” where file is a string. This method also catches some exceptions to ensure that the file exists in the path specified.

### ***Download***

The download method allows users to download files that are in the users’ folder on the server.



This method will send a “DOWNLOAD <filename>” message to the server and it can be written to the clients machines. A file is created in the location that the user picks. The downloaded string is then written to this file. The string returned from the server is written to this file if the response code returned is 200, otherwise an error has occurred and the file was not returned from the server.