Distributed Computing

# James Mahoney T00145442

Code repo: https://github.com/Mahoney0101/DistributedComputingRPC.git

# Protocol redesign from Sockets to RPC.

I am choosing RPC with the presumption that this is an internal API. RPC is quick across a network and has no caching as opposed to REST. This is a benefit for developers to avoid stale data. Using RPC avoids the complications of passing REST arguments and once implemented it is the same as calling any method. With the advantage of being able to call these methods from across a network. When implementing RPC, you are not limited to using HTTP.

## RPC methods pseudo code

### Login

The login method will take one argument which is a string. This string will consist of the username and password separated by a semi colon. This could be designed using two arguments, but I chose to use only one.

**Pseudo code**

Login(String loginString)

Split loginstring into array using delimiter “;”

try

If user loggedIn

return(“104: user already loggeg in”)

if user exists and password exists in Serverfiles/UserFiles

loggedIn == true

user = user in userfiles

return(“101: user logged in true”)

if user exists == false

create user and password files

return(“103: account created”)

catch(Exception e)

return(“102: ” + e)

### Logout

The logout method will not take any argument.

**Pseudo code**

Try

If loggedIn == true

user = null

loggedIn = false

return (“201: user logged out”)

else

return(“203: user not logged in”)

Catch(Exceptione e)

return(“202: ” +e)

### Upload

Upload method will take a String argument.

**Pseudo code**

Try

If loggedIn == false

Return(“303: not logged in”)

Else

Write argument to ServerFiles/MessageFiles/messages.txt

Return (“301: upload complete”)

Catch(Exception e)

Return(“302: ” +e)

### Download

The download method does not take any argument.

**Pseudo code**

Try

If loggedIn == false

Return(“403: not logged in”)

Else

Try

Get messages

Return(“401: ”+messages)

Catch(FileNotFoundException e)

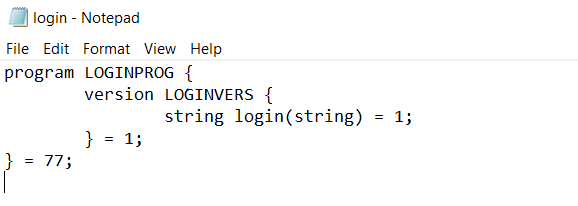
return(“402: an error occurred retrieving messages”)

Catch(Exception e)

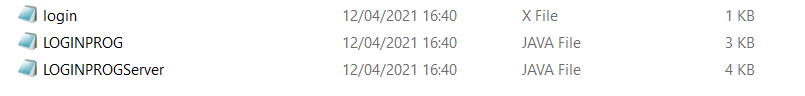
Return(“402: ”e)

# Implement Login method using RPC.

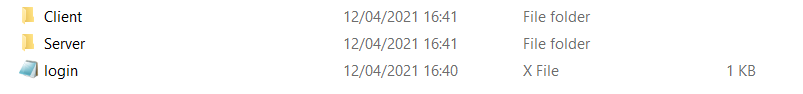
To implement the login method first the XDR IDL protocol specification must be created.



Once this code is saved as login.x the code can be compiled to generate the stubs. One for the client and one for the server. The command to compile login.x is “java Jrpcgen –S < login.x.”. This must be run in CMD as it will not work in powershell. The files generated on compilation can be seen below.



Next Two files are created, one for the client code and one for the server code and the stubs are copied into the folders.



Now that the stubs are in the desired directories code can be written to use these stubs to communicate.

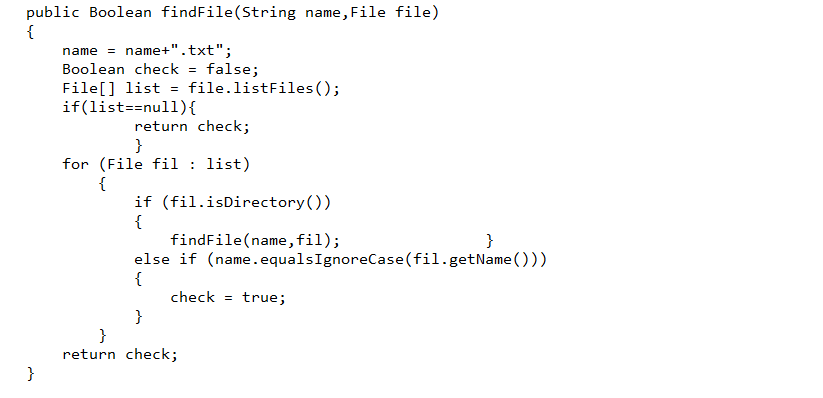
## Server

In the MyServer.java class there are two important methods, findFile and login\_1. findFile will be used to search through the ServerFiles directory for a specific UserFile by the login\_1 method. The login\_1 method is the abstract method generated by compiling login.x.

### FindFile Method

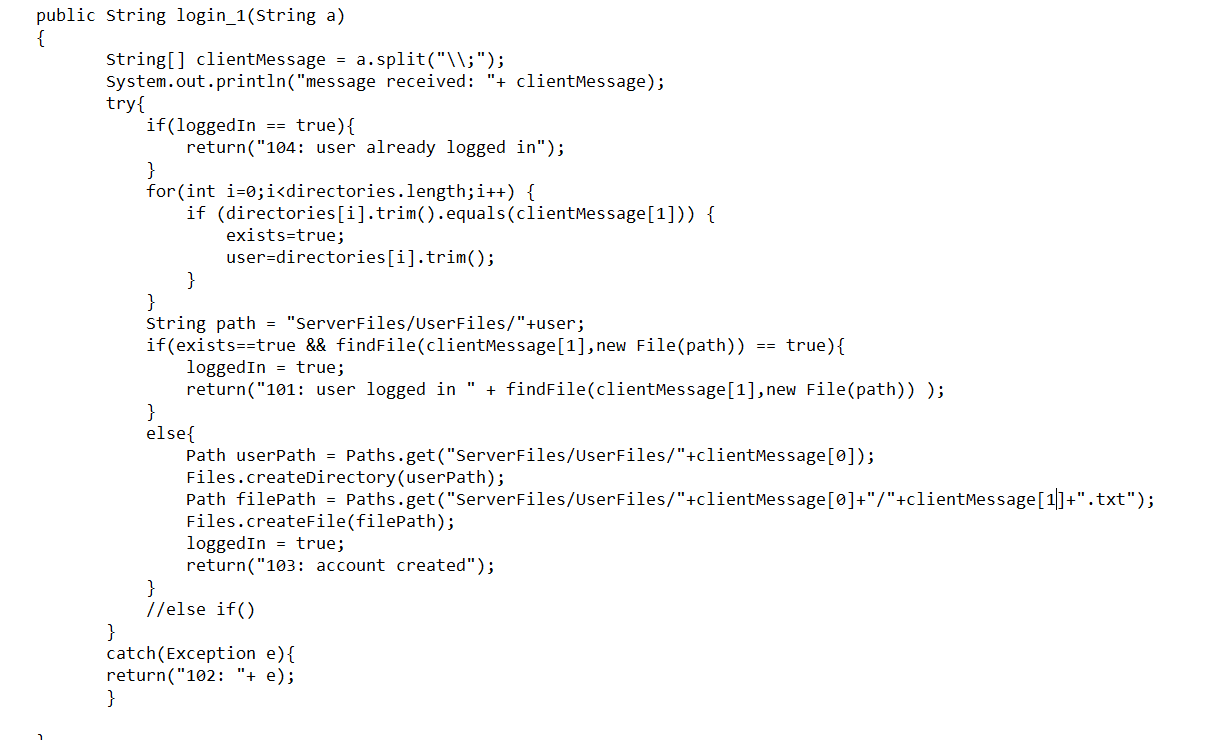
This method is the exact same as the method created for the Twitter Messaging Protocol project.



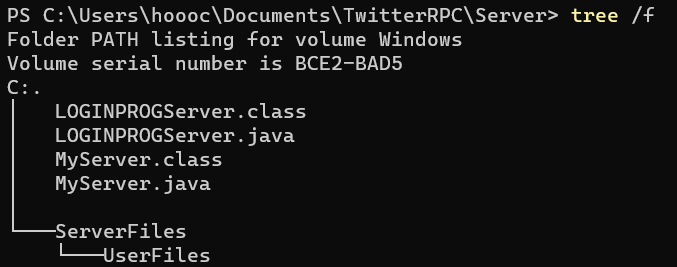


### Login\_1 Method

This method must overwrite the login\_1 method in the LOGINPROGServer.java class. The code in this method will be like the login code for the Twitter Messaging Protocol as it takes a string value and splits the string into an array of strings. These strings will be the login message followed by a username and a password separated by a semi-colon. The message will look similar to the following message “user;password”. The code for the login\_1 method can be seen below.

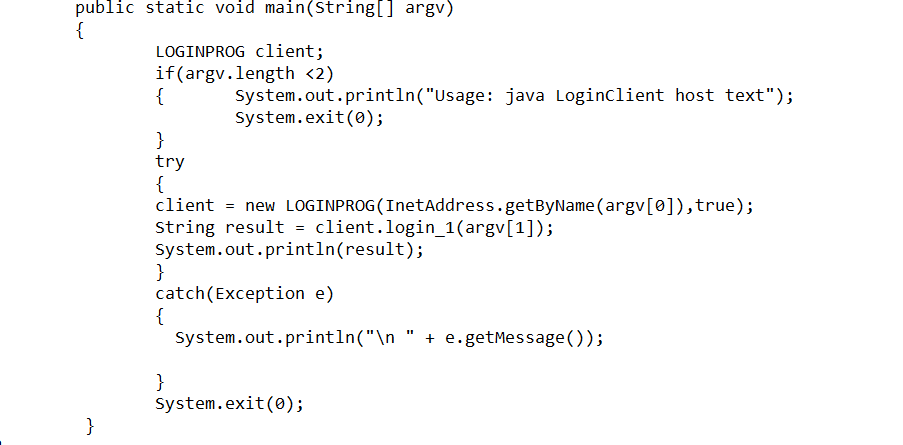


There is also a folder for ServerFiles with a Subfolder called UserFiles in the Server folder. The file structure can be seen below.

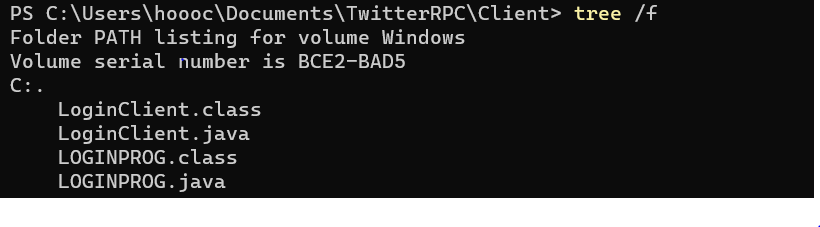


## Client

In the Client folder a java class called loginClient.java is created that has a main method. The main method will declare a variable of type LOGINPROG and will first check if the arguments sent are the correct number of arguments. Next the LOGINPROG constructor is called using the first argument passed from the command line when calling this class and a true setting for TCP. Next the login\_1 method is called, and the response is printed to the console. This response will be one of the codes followed by a message.



The folder structure can be seen below.



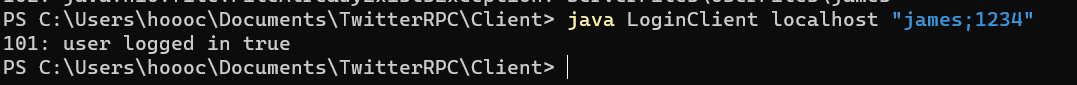
## Code in use

To run the code all java files must first be compiled by changing directory into the directories where the java classes exist and running the command javac \*.java. The classes are ready to be run now.

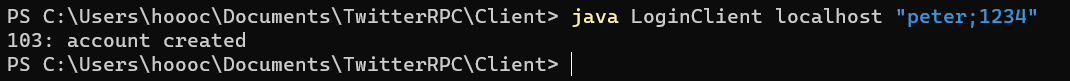
Run the server by running the code “java MyServer” as below.



Next change directory into the Client directory and login by running the code “java LoginClient localhost “username;password”” as seen below.

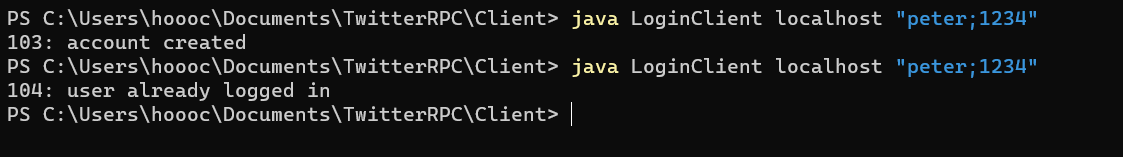


It returns 101 status. The user is logged in because the user already existed. If it is a new user the response would be as below.

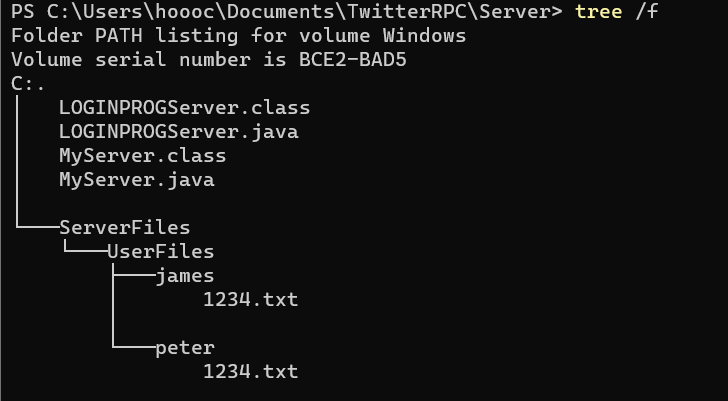


Here a new account is created.

If there is another attempt at an already logged in account, the message will inform the user that they are already logged in as seen below.



Files are created as seen in the image below.



# Compare and contrast Protocol solution with new RPC solution.

When Writing the RPC version of the Twitter service. The first thing I noticed was that it was far simpler to implement this version as a lot of the details are abstracted away. Where you would need to write classes for threading and helper methods for the old version, the RPC version is as simple as generating the stubs and then calling them as if they were any other class. The RPC version is more developer friendly and requires far less knowledge to implement. Below the files from the original TMPServer can be seen along with the RPCTMPServer. There are less files needed in the RPC version.

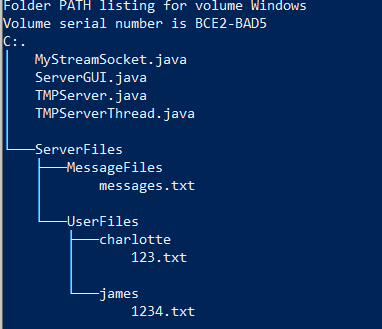


Figure 1 Original Stream socket version

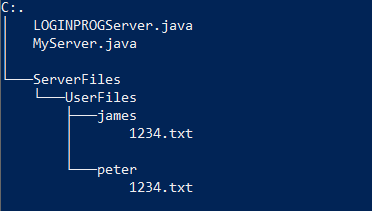


Figure 2 RPC version

RPC uses dynamic port allocation whereas the original version the port needed to be input when making a call to the server. If you do not know which port or make a mistake when adding the port. You will not be able to access the server using the stream socket version. When calling the RPC version, the developer or user of the software does not need to know anything about the port address as it is all handled already. This makes it less like to make a mistake.

Instead of choosing between datagram and stream sockets for UDP and TCP all the developer must do to choose between either when using RPC is add true or false as a second argument when creating an instance of the client stub. True will initialise a TCP connection, false will initialise a UDP connection an example can be seen in the code below.



Figure 3 Initialising a TCP connection

If using the original version new classes would have to be written to use datagrams.

When using RPC over sockets since the lower-level coding has been abstracted away you could be sacrificing speed for simplicity. Also, with the RPC version the portmapper or RPCBind uses a fixed port number (111). This means that if there are multiple servers running on the system and the original server that started the RPCBind, all other servers will lose the ability to bind new clients.

With the original stream socket version, the ports were mapped individually so this would not be an issue.