

# Distributed Computing Assignment 2.

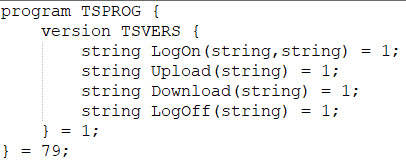
Emma Fitzgerald

BSc(Hons) Computing with Games Development

# Question 1:

For the redesign of the Twitter Service to a microservice I decided to use ONC RPC as the API. RPC (Remote Procedure Call) involves executing a procedure call on a local machine (client stub) and sending this marshalled data over the network to a local kernel then to a remote kernel. Here the data is unmarshalled on a server stub and will call the actual function on the server. The server stub will send back the results to the client stub over the network and is then shown to the client.

ONC RPC is a very simple RPC API to use. The application Jrpcgen makes creating an RPC API very simple. Jrpcgen allows us to create client and server stubs with our chosen methods without having to write complicated code. The only thing needed to create these stubs is an IDL file to state the name of the program and version and the methods that will be implemented. Once this file is created the next step is to run the file in the command prompt to create the stub files. These files will then be linked to the client and server files for easy data delivery over the network.



*Figure 1.1*

Figure 1.1 shows the IDL file for the new ONC RPC Twitter Service. The program will have the same methods as the Twitter Services Protocol: LogOn, Upload, Download and LogOff. For the Download and LogOff there is no need to pass in parameters into the methods since they are just calling certain features that don’t need user input. When I tried to create the methods without parameters it gave me an error so for the redesign these methods will pass in a string but it will be empty and not used within the method.

**Pseudocode**:

1. LogOn:

Function LogOn\_1(username, password)

Read in users from users.csv into a list called users

Declare returnMessage

for each user in users do

if the username and password match a username and password from users do

Set returnMessage -> 110;Login successful! Welcome username

else

Set returnMessage -> 120;Username or Password is incorrect please try again!

End if

End for

return returnMessage

End function

1. Upload:

Function Upload\_1(msg)

if the message isn’t empty do

write to the users file with the message

returnMessage -> 210; Upload Successfully

else do

returnMessage -> 220;Message was empty. Upload Failed!

End if

End function

1. Download:

Function Download\_1(emptyMsg)

Read in messages from users file into a list called clientMessages

if clientMessage is not empty do

Declare messages -> “All messages:”

for each message in clientMessage

Add the message to messages

End for

return 310;messages

else do

return 320;There are no messages available for download!

End if

End function

1. LogOff:

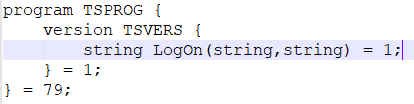
Function LogOff\_1(emptyMsg)

return “410;Log Off Successful”

End function

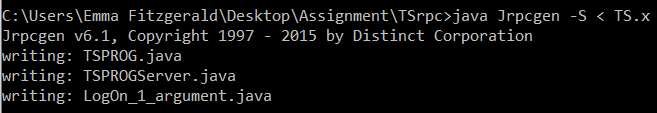
# Question 2:

For the code redesign I decided to redo the LogOn method.



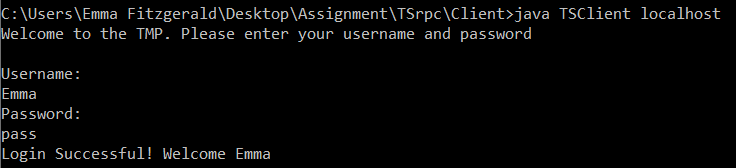
*Figure 2.1*

The first step was to create the IDL file, shown in figure 2.1, to create the stub client and server code for the LogOn method. The method will pass in two string variables: the username and password of the user. The method will also return a string similar to the return message for the LogOn method in the protocol version of the service.



*Figure 2.2*

Figure 2.2 shows the running of the TS.x file created above to produce the client and server stubs of the API. It also created another file called LogOn\_1\_argument. This file will be used within the stub files to pass in both the parameters to the LogOn method.



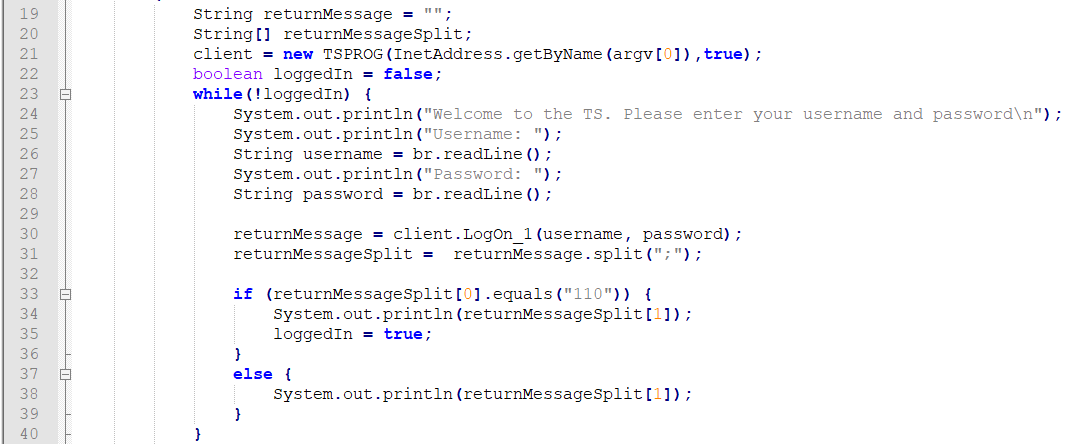
*Figure 2.3*

Figure 2.3 shows the client asking the user to log on using their username and password. Once they have successfully logged on the client side will end since there is no other methods implemented.



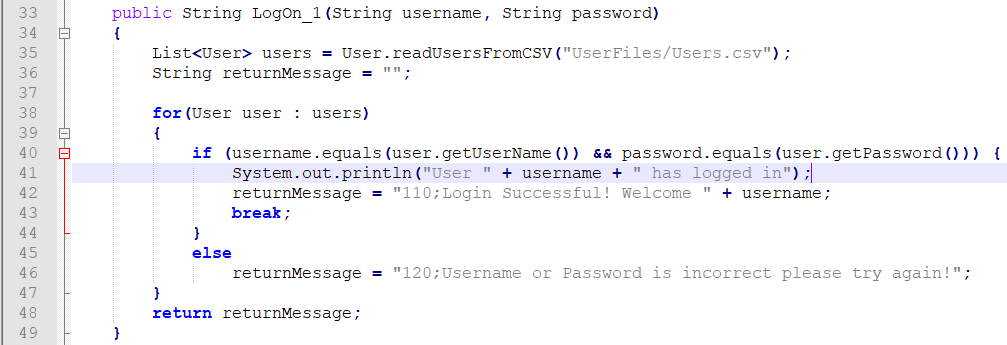
*Figure 2.4*

Figure 2.4 shows the server side of the application. There is confirmation shown to the user when the server is up and running and what user logs into the server.



*Figure 2.5*

Figure 2.5 shows the code created for the client-side Log On functionality. Most of this code was created in the protocol phase and just reused since the Log On functionality doesn’t need to be changed for the ONC RPC version. The TSPROG client stub is called and saved in the client variable. This constructor call passes in the localhost when the class is executed. The user will then be presented with an interface to allow them to enter their username and password. These user inputs will then be passed in the client stub LogOn\_1 method and the return message from the server will be saved in the returnMessage variable. This variable will then be parsed to see if the user was logged in or not.



*Figure 2.6*

Figure 2.6 shows the code created for the server-side Log On functionality. This code is the same as what was used in the TMPServerThread in the protocol version of the service. The only different is instead of receiving and parsing a message from the client to gain access to the username and password, the username and password are passed into the method directly. The method will read in the users from the User file and store them in a list. Each user in this list will have their username and password compared to the username and password supplied by the user. If there is a match a success message will be sent back to the client stub then to the client. If there is no match a failed message will be sent to the client.

# Question 3:

The development speed for creating the API compared to the protocol Twitter service was quite significant. With the protocol approach there was a lot of alterations needed to get the server and client connected to each other. There were two classes for both the client and the server which took a long time to create so that they were able to communicate with each other. With the ONE RPC API the connection code for communication between the server and the client was generated automatically once the IDL file was created. This made setting up an environment where data could be passed between the two connections quicker and simpler. There were very little issues creating the API compared to the protocol since the code for the API was easy to understand once you know the basics. The only issue that arose when I was creating the API was that you can’t create a method name in the IDL file without a parameter. Whereas with the protocol that were many issues trying to get the server up and running and getting the messages to pass accurately between the client and the server.

With the protocol version of the service we implemented SSL to create a secure connection between the client and the server. This SSL would allow us to encrypt the data that was being sent over the stream that connects the two ends of the application. This was done by creating a keystore that would have to be accessed by the client and the server. Both the client and the server would have to create an SSL socket connection by using the keystore we created to create a secure connection. From here we were able to create our own encryption code i.e. reversing the data being passed through the socket. In the ONC RPC version this was not done. We did not implement any way to encrypt the data being passed through the network. This could cause issues where the data could be accessed by outside sources if not taken care of. There are authentication methods that can be implemented into ONC RPC like Diffie-Hellman authentication and Kerberos-based authentication.

Maintenance for ONC RPC would most likely be simpler than doing maintenance on the protocol approach. The reason for this is since in the RPC approach half the code we used to create the application was created for us using Jrpcgen. If something was to go wrong in the RPC version of the application it would make it easier to narrow down where the problem could be accruing since it would most likely not be occurring in the stubs created by Jrpcgen. With the protocol-based solution it would be harder to maintain since there is a lot more code needed to create the connection between the client and the server. All the code created for the protocol was created from scratch which could lead to having issues finding the break point in the code if something was to happen to cause the connection to crash.