FACEBOOK SERVER

COMP30640

NOVEMBER 13, 2016

DANIEL MCMAHON
09556362

Contents

Introduction:

The following report outlines the development of a set of scripts that interact with each other to form a simplistic localized version of how a Facebook client/server system might be developed. Throughout the project many challenges were faced and the final product delivered can still be optimized. All will be outlined in the following pages.

Requirements of the System:

The overall requirements for this system relies on a set of basic commands for the server which consist of a create.sh, add_friend.sh, post_messages.sh and display_wall.sh. After setting up these scripts it was necessary to develop a server.sh that will read commands from the user continuously and execute the basic commands when certain strings were entered by the user. The next challenge of this system was to setup a client.sh that allows multiple users to access the server simultaneously. To ensure this works the implementation of pipes and semaphore locks was necessary. The formation of these scripts will be broken down in the following sections.

1. Create.sh

Create.sh accepts one argument from the user and assigns it to the variable id. It then takes on a check to state if the argument is empty then respond with the error message “nok: no identifier provided”. If the system detects a user has already been created, by checking if a directory with the value of id exists, then it will return an error “nok: user already exists”. In the event the value entered for id is not an existing directory then the system will send a message saying “user created!” and will make a directory named after the value of id. It will then make two files within that directory called wall and friends.

2. Add_friend.sh

Add_friend.sh accepts two arguments from the user and assigns them the value of id and friend. If the directory of $id doesn’t exist (i.e. that user has not been setup with the create.sh script) then it will return an error stating “username $id does not exist”. It will conduct a similar check for the $friend value and return a similar error message stating “nok: user $friend does not exist!”. If the $friend is already on the $id friends file, then it will print an error stating “User already in Friends list!”. If the $friend is not already on the $id friends file, then it will add the $friend’s name onto the bottom of the $id’s ‘friends’ file that is kept with the $id directory.

3. Post_messages.sh

Post_messages.sh accepts three arguments from the user and assigns them the value of sender, receiver and message. The script will check if the value of sender matches a directory (i.e. that the user exists) if it finds none it will return “nok: user $sender does not exist”. It will then conduct a similar check for the $receiver value and return “nok: user $receiver does not exist” if there is no matching directory. If the $sender is on the $receiver’s friends list, then the script will write a message saying “$sender: $message” to the end of the $receiver’s wall file. (the wall file within the $receiver’s directory). Otherwise if the $sender is not a friend of the $receiver it will return an error message pointing this out.

4. Display_wall.sh

Display_wall.sh accepts 1 argument and assigns it the value of id. It conducts a check to see if the directory $id does not exist – if it does not exist it will return an error saying “nok: user $id does not exist”. If the user does exist it will print out a message stating “start of file” – contents of wall file – “end of file”.

5. Server.sh (server script)

The overarching purpose of the server.sh file is to be able to run all 4 of the basic scripts depending on the users input. To achieve this an infinite while loop is used to continuously read the user input. When the users input matches certain strings it will match the corresponding scripts and execute them – this is achieved using a switch statement. The key words that are checked are ‘create’, ‘add’, ‘post’, ‘display’. The wildcard value * is used to print back a statement to the user if they put it any other input – this prompts them to use create | add | post |display commands.

Initially this script was designed to run on its own however with the development of the client.sh file to allow multiple users to run the server at once this was changed. The server.sh in its current state only accepts input that comes from the client.sh script.

The code at the very top of the server.sh creates a server.pipe if one does not already exist. This is the primary means of communication between the client and the server. More information on pipes included on point 6 below.

Note: ‘server(loops but read sync issues).sh’ file has been included – this was an initial developed version of this script that read in and exported user ids rather than take in multiple inputs at the time the user ran the basic scripts. This continued reading of multiple variables was however leading to synchronization issues and thus was scrapped and replaced with the updated server.sh file.

6. Client.sh (client script & communication)

The client.sh is intended to be the main script run by users. This script should take 1 argument assigned to the variable id. It first checks if there is a pipe setup for $id.pipe. If there is not then it will make a pipe.

The next code in the script traps the control and c function to essentially remove the users pipe once they disconnect from the script using ctrl + c command.

The next thing that will happen in the script is that it will check that the number of parameters given to the client is at least 1 (i.e. that a value has been given to the script). If there is a value then it will set the value of x to true and send the $id to the server through the server.pipe. If the arguments is less than 1 then it will set x to false and not send input to server.

When the value of x is true it will read input from user and assign them to variables command, arg1, arg2, arg3 and send them across to the server through the server.pipe. The server will be listening for the arguments through the server.pipe. When the server has read the arguments from the server.pipe it will enter its switch statement and will pipe back the scripts to the client through the $id.pipe. The client.sh will be reading the serverResponse from the $id.pipe and will echo back the message received. This process will continue indefinitely until the user has disconnected from the client.sh using the ctrl+c method which will close its unique $id.pipe and exit the script.

Problems Encountered

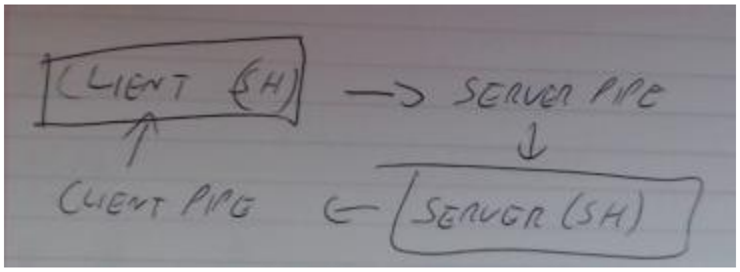
The following is a list of issues that were encountered throughout engagement with the project:

1. Differentiating between passing variables with the script e.g. typing "./create.sh Michael" and setting Michael as the value of $id as opposed to running './create.sh' and prompting the user to type in their name and reading that input as the variable $id. As mentioned above I had initially undertaken the project and landed at the server (loops but read sync issues).sh solution – only after discussing with peers and the demonstrators did I see the synchronization this was causing by requiring multiple reads within the switch statement.

2. Outputting the full contents of the user's wall through pipes – it works fine when outputting directly to the server but when piping through to the client it currently only specifies the first line of the text.

3. Lack of familiarization with echo syntax and structure. As with any new programming language it was difficult initially to wrap my head around the various syntax and structures associated with Bash. I initially wrote the server.sh as a bunch of if statements and then rewrote into the switch syntax. This was an invaluable learning experience as I had not used switch statements prior to this.

4. Pipe redirection was confusing at first. There was some initial brute forcing deciding between which pipe structure to follow. After drawing out the basic structure below I understood it better:



5. Error Redirection. To handle the final part of the assessment brief I took out some initial >&2 redirects so error messages will display directly to the client user. May not be the optimal solution

6. Early in development the server.sh file encountered a weird bug whereby after reading the id from server.pipe it would pick up a random empty character and read that as a command. To try

counteract this error a sleep 1 was inserted. This issue may have been localized to my machine and only occurred randomly – I left this sleep command in as it doesn't hinder the overall performance of the script and I would figure it's better to be overly cautious than underly.

7. Synchronization issues. Use of P.sh and V.sh implemented to act as semaphores to help prevent synchronization issues in the critical sections of all the scripts. I have omitted these from the server.sh file. These can be easily put in place if necessary – would need multiple users logged in simultaneously to check if an issue as semaphores could restrict only 1 user being able to create an account at a time etc. if needed.

Known Bugs

The following is a list of bugs within the scripts:

- Add_friend.sh will echo out the name of the user adding a friend before printing ok.
- Display_wall.sh will not output full contents of wall – only first line. A solution is commented in the client.sh file (cat < "$id.pipe") however this resulted in errors when reading the wildcard item in the switch statement so this was commented out.

Conclusion

Overall the project itself should meet the criteria as outlined in the brief. There were some challenges along the way due to unfamiliarity with Bash however all in all it was an enjoyable assignment to go through the process of building all the scripts from scratch and form them into a more significant project. The issues encountered throughout the project are all detailed above. The tutorials in the lead up to the project were all beneficial and relevant towards the assessment and provided a tremendous help in understanding the key concepts that underpin this type of project.