CT213 – Assignment 2 Report

BashBook V2.0

Report and Project by Alasdair and Tim

Introduction

As the assignment brief lays out this project implements basic Facebook like social media server using Bash. This version of BashBook successfully integrates a client.sh script which can be run on a user's individual terminal to interact with the server. The transfer of data between the server and clients is done exclusively through pipes. Each client uses a common pipe to enter their command into the server but a unique pipe to receive the output. The server receives data from each of the scripts by reading each script's 'echo'. In this case, echo acts like a return statement. Using the echo command, the server script is able to receive the output of each script and then transfer it to the client. Temporary files such as locks and user-based pipes are removed once the program exits. The client.sh script and the changes made to the server script meet the criteria required by the brief for a functioning version of BashBook.

System Organisation

As mentioned above the majority of this new version of BashBook revolves around pipes. When the client script is run it takes one argument, their username. A pipe is created based on this username. This pipe will be used to receive output from the server. A different pipe called server_pipe is used to send inputs to the server. This pipe is shared with all other clients. When a user runs the client script it waits for user input. This waiting for input is not using any of the pipe's time and the input from the user is only transferred to the pipe once information has been received and the user has pressed enter. This helps reduce any overlap between user inputs over server_pipe. When the user presses enter the input to the server is immediately transferred to the server. The second that the input from server_pipe is read, the pipe is then free again to take another input.A basic link lock is implemented to ensure that there is absolutely no overlap between information on the pipes. This lock is only used for server_pipe as it is the only common pipe.Once the request has been transferred to the server the respective command is run. The server receives the output from each command through the use of 'echo', as mentioned above this acts similar to 'return'. The output variable in the server script reads the echo output from each command. This ensures that individual scripts do not communicate with user pipes directly and instead it is done only through the server. The output from the function is then received by the server and transferred to the individual user's pipe. This pipe is read by the client script and the output is printed to the terminal. Of course each script has slightly different outputs and accommodations have been made to ensure that they print correctly to the terminal.

The information read into the pipes is read in the format specified in the brief 'req 'id' 'args'. This information is then relayed to the server and stored in a series of variables. The user is not required to retype their name for the invocation of any of the commands. The original client argument of the user's name is transferred to the server and used as an argument in each of the scripts automatically without user intervention. It is important to note here that I have added a check to ensure that a user cannot login as a user which does not exist. I have added this check as logging in as a fake user was causing many errors in the different scripts and was often leading to unforeseen outputs. Each individual pipe waits until it has either been written to or read from. i.e. If you write to a pipe in a bash script the script will not continue until the pipe has been read from and if you read from a pipe in a bash script that script will not continue until there is information to read from that pipe. This means that when running the server script, the code will not continue past the ‘read *** < server_pipe’ line until information has been passed from the client to the server. When you combine this with the infinite while loop it basically means that the server is constantly able to receive information. This is compounded by the lock in the server_pipe meaning that the server will always finish its current command before receiving information from another client. From the client’s perspective it means that any command can be typed at any time with a maximum delay of a few milliseconds.

Another important point of this implementation is how the server receives information from each from the individual scripts. As the brief shows in the diagram of BashBook’s desired communication architecture the output from each of the functions must go from the server to the client directly and not from the individual function to the client. This is done through the use of ‘echo’ commands and reading the output of that echo command with specific bash syntax. When one of the scripts such as ‘post’ or ‘add’ is called the variable called ‘output’ receives the text which that script echoes. This is extremely useful for transferring data from the individual functions to the server without having to create a series of pipes. It also eliminates the risk of pipe overlap or functions being called with the wrong arguments. The server, once an output has been received from the individual script then sends that output through the individual user pipe(ensuring the output is private) where the client is waiting to read the data from the user’s pipe. Lastly the appropriate output is printed in accordance with the specific request in the brief. Note that the server receives the original output from each of the functions. Basically this means that the server retains the information of the original output but through the use of if statements still prints the updated response to the terminal. My point here being that I am not changing the original responses of each of the functions I am simply changing what the client sees printed on the terminal.

This implementation also successfully deletes temporary user files when using ctrl+c to exit the program.This is done through the use of a function and linux command. When ctrl+c is pressed it runs the ctrl+c function. Importantly this function has the ‘exit’ command in it. Above I have typed a piece of code which runs a linux command if the exit command is run. This means that the exit command essentially acts as a call for the command ‘rm $1_pipe’ once it has removed the pipe the exit command fulfills its original purpose and exits the program.

Alasdair & Tim

For the sake of this PDF, we are dividing the server and client scripts into our own individual sections. In reality both members of this group worked on both scripts. The client and server scripts were both so dependent of each other that going off individually and doing our own versions would have been impractical. Instead we coded both scripts side by side and tested them as we went. Obviously without a server we could not test the client and vice-versa so it was important that this work was done in close collaboration with one another.

Alasdair

server.sh

The majority of the work I did was on the server.sh script. The changes to this script mainly revolved around ensuring that data was transferred correctly from the client to the server and passed into the individual functions correctly. After working out how data is read from pipes and that the data could be read into individual functions I went to work on the transfer of data from individual functions. This by far took the most time out of all the coding I had to do for server.sh. Figuring out the exact format of $('./script.sh arg arg) was a challenge as I could not find the answer readily online.

I began by attempting to find a Bash equivalent of the java 'return' statement but quickly found out that variables could not be shared by individual bash scripts. Knowing from using the 'read' command previously that the terminal could be read for user inputs. I began searching for ways to read an echo output from one script to another script. This quickly yielded results and I soon learned that running a sh script in bash as though you were running it in the terminal allowed me to read its output into a variable. The use of the $ sign and brackets seems to allow bash to read any outputs that the function echoed into a variable. This implementation was necessary as the communication architecture diagram clearly shows that there are no pipes between the server and individual functions therefore another method of information transfer had to be used. The transfer of the individual output back to the client using pipes was simple and the only real concern was ensuring that the client was correctly reading the input. I was also aware that the output of the display_wall function would need to be edited slightly within the client.

Tim

client.sh

The main purpose of this script is to serve as the main point of communication between the server and the user of Bashbook. It works by utilizing the main, shared server pipe to send specific requests to the server for it to interpret. The client script also retrieves output from the server through a special pipe that is created specifically for the user so other users cannot access/read the output generated for that particular user.

The script starts by running validity checks on the user. First it checks if any information was sent to the script file, if there was, it checks the relevant Bashbook directories to find an existing user. After the checks the user is welcomed to the service and is informed of the commands that are available. The user's very own pipe is also created in the background. The script enters an endless while loop, where it will continue looping until the user exits the script using ctrl+c. I have implemented a method to ensure all pipes and/or locks are removed once the user exits the script so the pipes are not left running or locks remain held for an indefinite amount of time. Upon entering the loop, the script awaits for a request to be entered. Once it receives a request, a lock is created to ensure no other requests are run until that one is finished. This link is deleted inside the server script file once the request is finished. The request is in a particular format to ensure all the necessary information is passed into the relevant script files. The format $request $id1 $id2 $message is used to achieve this. Every request sent between the client and server script files are of this format.

After the server script file interprets the request and runs the necessary script files, and output is generated which is fed back to the client file. I designed the client file to parse the output received and appropriately format the output in a user friendly manner. If statements are used to parse the relevant data and determine the appropriate output from the client file. I did run into trouble attempting to display user walls in a user friendly way. Printing the wall without the start of file and end of file statements proved to be challenging but through manipulating full stops cleverly placed within the post_messages script file, I found a method to overcome this issue. All full stops were replaced by new line escape characters.

Locking Mechanism

The locking mechanism for this assignment is mainly implemented in the client script which creates a link to the file from the client called 'clientserverlink'. This file is only created if it does not already exist. Once this file is created it is not deleted until the server script has finished processing and it is clear that the data has been read to the client correctly. The 'clientserverlink' is deleted inside the client script once the data has been read from the server correctly. This is done to ensure that there is no overlap between functions inside the client script. If the server script is midway through a function and a client enters a command then they will enter a while loop and sleep for a quarter of a second. Due to the speed of the program it is highly unlikely that the user will have to wait for any more than one quarter second sleep cycle. From the user's perspective this is instant. The while sleep loop is exited when the other client deletes the link file once it has received output from the server. Other implementations of this locking mechanism may use separate and acquire and release functions but we felt this was unnecessary because there was only once common pipe between the clients(server_pipe) this means that locks will not have to be used elsewhere so making a common function that can be used in all scripts seemed unnecessary. You will also notice that the link file is only deleted after the data has been read from the individual user pipe and not the server_pipe this is to absolutely guarantee that the server has finished its process before another client can run the server script. This is because the server will only send data over the user pipe if it has totally completed its current process. If speed and performance was a serious concern and we had millions of users calling functions then it may make more sense to release the lock after the data has been read from the server_pipe but with our implementation it reduces risk of errors at the cost of performance.

Challenges **Encountered**

Alasdair

I faced a series of challenges while working on both scripts. After understanding how pipes work my main problem became ensuring that the data was read to the client from the server correctly. This became a serious issue when running the 'display_wall' function which required the data to be printed to the terminal in an organised fashion. I learned that when calling a variable with text in it special characters such as \n are only saved if the function is called using quotes. My problem was that the special characters were not being transferred over the pipe. Using cat to read in the data from the text file \n is automatically applied to the end of a line. This \n was successfully transferred from a function to the server but was not being transferred from the server to the client through the pipe. When I printed the data to the clients terminal from the display wall the data was printing all in one large unorganised block as opposed to the nicely formatted text which the 'cat' file read in. We both worked on the solution to display_wall. To fix this problem we added a full stop to every line that a user adds to a text file using a post message. My partner could then use bash variable manipulation to replace every full stop with a \n this was done inside the client which allowed us to print the output of the display wall correctly. We also used Bash variable manipulation to replace the lines 'start_of_file' and 'end_of_file'.

The second set of challenges I faced was in regard to the optional ctrl+c command trap. The trap command traps the ctrl+c command and runs the ctrl+c function but after testing this I was unable to figure out how to pass arguments to this function thus I was unable to delete the pipe of the logged in user(user $1). Therefore I had to trap the exit command and run a linux command when exit was run. I attempted to use the same code format with ctrl+c but it did not work. My final solution was trapping ctrl+c and running the ctrl+c function with exit in it which deleted the pipe of the logged in user.

Tim

I faced many challenges in regard to properly ensuring the client script file printed the correct output to the screen. Throughout this development process, I had to experiment with various methods to try to find a solution. At first, I had trouble comparing the output variable to a specific server script file output. Using the if statement to compare strings proved quite troublesome but eventually I managed to figure it out.

Eventually I've started to run into problems regarding where the output was being printed. Especially with regard to running the add friend and display wall scripts. If a user was to add a user who wasnt on the system, the client file would refuse to print any output but instead the server script file was printing the output. I managed to find the cause of this issue which was due to the error output identifier ">&2" which was still within those script files.

Implementing the lock also proved quite challenging as locating the critical zone was harder than expected. After attempting to run the program with the lock in different locations and a lengthy discussion with my lab partner, we settled on a location we believe is good. It ensures that the chances of two processes interfering with each other is mitigated. This is vital when multiple users can connect to the server and request/send data. Processes will collide with each and effective locks are a must have in service like Bashbook.

Conclusion

Overall we feel as though our project was a success. As opposed to Assignment 1 of BashBook the work submitted in this assignment was more equal. The dependent nature of the client and server scripts as well meant that we both had to work closely together to ensure suitable integration between the scripts. We believe that our pipe implementation in this assignment ensures quick and correct transfer of information between the client and server.. Pipes were the core of the communication architecture as so it was crucial that they function correctly or the whole program would break down. Our locking system helps to ensure that our pipes function without error even at the cost of some performance. Minor changes were made to ensure that error codes were consistent with the criteria in the brief. Error codes were also added for logging in as a user which does not exist and adding a friend who is already your friend. These error codes help ensure that in all scenario's something is transferred through the pipe so the program is not stalled.

Finally we wish to highlight how much our bash skills improved throughout our work on this assignment. Firstly before this assignment we had no knowledge of pipes in Bash or how they work but now I would feel confident using them for any assignment. I also learned many useful tricks like how an echo statement can be used to output data to another function and how to trap commands. The close communication required to ensure suitable integration between the two scripts meant that our teamwork and overall presentation skills improved considerably.