COMP30640: Assignment 2 - Bash Project - Exokernel

Adam Ryan - 14395076

December 22, 2020

1 Abstract

In this report, the author describes their solution design to the Bash Social Media System project. This project was written locally using macOS Catalina's Bash terminal rather than using the CS Server, however the author is confident the solution is portable due to it passing the autograder.

Contents

2 Introduction

This project detailed how a simple social media system could be developed in Bash using some of the basic tools which we learned during the tutorials of this module. It covers elements such as checking the existence of directories, the use of variables, creating files, using case statements, using if statements, using input arrays, and the usage of client-server pipe architecture.

This system contains some of the basic functionality of a social media system; in particular:

- The system should have the functionality to create users.
- The system should have a friend list and wall file.
- The system should have the ability to asynchronously add users to a friend list. i.e. 'User A' may add 'User B' to 'User A"s friend list, but User A does not get added to User A's friend list.
- The system should have the ability for friend's to post on somebody's wall.
- The system should have the ability to display what is on a user's wall.

and, following this:

- There should be server functionality, to handle a series of requests or operations.
- There should be client-server functionality, where numerous clients can submit requests to the server which carries out the various operations requested by the clients.

The entire system's functionality is implemented using the following files:

- create.sh
- add.sh
- post.sh
- show.sh
- server.sh
- client.sh

Finally, the project was source-controlled via GIT. The author's username was **AdamPatrick-Ryan** with identifier 14395076. Git pushes were conducted at key development stages, namely after each of the scripts were written, after the server functionality was added, after the client functionality was added, and after some final bug resolution in order to pass in-built assumptions of the autograder.

A 'pipedev' branch on the Git was added to test the functionality of the client changes (which necessitated alteration of the server file) however upon reviewing the Git Tree the author has realised that this branch was not pushed until it was ultimately merged back into origin master.

3 Requirements

As outlined in 2, the system comprised of the following requirements (outlined here in more detail):

create.sh The system should have the functionality to create users. In particular, the system should have the functionality to take a username with/without spaces, and with/without special characters. If one argument is not provided, an error should be returned. If the user already exists, an error should be returned. If the user does not exist, the a folder with the username should be created in the directory, and this should contain two files "wall" and "friends" to facilitate additional functionality.
./create.sh $user

add.sh The system should have the ability to asynchronously add users to a friend list. i.e. 'User A' may add 'User B' to 'User A"s friend list, but User A does not get added to User A's friend list. In particular, the system should have the ability to take two usernames, and add the second username provided to the first user's friend list. The system should appropriately check that the users exist, that only two arguments were provided, and that the user is not already on the friend list. If the user exists and is not on the friend list, they should be added as a line within the friendlist.
./add.sh $userA $userB

post.sh The system should have the ability for friend's to post on somebody's wall. In particular, the system should be able to take two users, a receiver and sender, and a message. The message sent by the sender should be posted onto the receiver's wall so long as the sender is on the receiver's friend list, and the message should be posted into the format "sender: message". As with other scripts, appropriate error checking should be conducted to validate both users exist, three arguments are passed, and that the sender is on the receiver's friend list.
./post.sh $userA $userB $message

show.sh The system should have the ability to display what is on a user's wall. By passing a user, the content of that user's wall should be read line-by-line preceeded by "wallStart" and followed by "wallEnd" (as per the output examples). The number of arguments should be validated, and the existence of the user checked.
./show.sh $user

server.sh There should be server functionality, to handle a series of requests or operations. This script should take a client id, an operation, and the arguments of that operation and execute them concurrently until either a bad request is passed or a shutdown request is passed. This should be implemented via the case structure outlined in the briefing document, and continuously run to allow for a series of operations to be completed. Finally, server.pipe should be created and inputs read from the server pipe, to allow for the server-client architecture, and outputs passed to the client pipe.
./server.sh

client.sh This should allow for the creation of a client pipe named from a passed clientId. It should check that a minimum of two arguments are passed and take a clientId, operation, and the arguments of that operation. It should pass the instructions to the server pipe to be ran, and many different clients can be created.
./client.sh $client_id $operation $args

One important note which should be mentioned is that the briefing document states that the client piece should check the validity of the number of arguments for the appropriate request, however this is actually implemented already within the various operations. My assumption is that the operation itself should handle this (as otherwise it seems redundant to check within the script outside of as a failsafe). If this was needed, it can be implemented via checking user_entries and substracting 2(as client id and operation will certainly be passed, so subtract these two to check the operation argument count and compare appropriately per operation) from this and checking the correct number of arguments are passed.

4 Architecture and Design

To fulfil the requirements outlined in 3, the following broad approach was taken in the design of the solution. This will be detailed more thoroughly in later sections.

- Initially, the four operations were created. After the development of each operation, valid entries were tested and invalid entries were tested (meeting various failstates). The inputs and output examples provided in the briefing documents were used, among others, as test cases to check that the outputs were as required. After each step, a git push was completed with the code as of that state to allow for rollback in the event of data loss.

- Following the completion of the operations, the server functionality was added. In the initial development, this was simply the while statement, read user input directly from the terminal, and responded within the same terminal. This was tested using the script outlined within the briefing document.

- After the server functionality was able to replicate the I/O outlined in the briefing a git push was completed to record the project at this step, as the final section necessitates not only an additional file but changes to the server section also. A git branch was created at this point for the development of the client functionality, however in reviewing the git tree the author can see that the git branch actually wasn't pushed until it was complete and merged back to origin master, however this was used for the development of this feature.

- The server-client architecture was implemented. The server code was modified to create the server.pipe, and client.sh was created using a similar outline to server.sh, first checking the number of inputs and then using an if-elif-else structure to validate which request was implemeneted (the same case statement could have also been used which, in hindsight from writing this report, the author realises it probably would have been nicer in allowing a copy and pasting from server,sh and create a more consistent solution, but given the autograder has passed the current implementation I'm not going to restructure this).

- As a penultimate step in the process, a series of final testing was conducted, with mixed results which will be outlined further below. While the functionality works, the author notes that sometimes there isn't great consistency in whether the server hangs. The final three tests conducted passed successfully which is why the project was pushed, however the changes were very minor making it unclear (outside of potentially getting caught in a loop reading from the pipes) what fixed these errors and if they'd reoccur.

- Finally, a final round of bug-checking and tidy-up was conducted to modify the solution to pass the autograder. This involved a series of minor changes such as changing tabs to spaces, changign the friends and wall file from text files to files without extensions, and similar alterations

- The final deliverable involve four operation scripts, a server script, and a client script satisfying the requirements outline in 3. To run the file, the server script should be run in one terminal, while in a second terminal client operations are sent to the server.

5 Solution Detail

This section delves into the various scripts in more detail, but still kept relatively light, and describes briefly how each script works and difficulties I faced in developing them. To summarise, I'm familiar with using Bash scripts in a production data pipeline for basic automation however the pipe aspect required by server.pipe and client.pipe is totally new and as such there were significant challenges to the authors in developing this. The structure of most scripts is relatively clear to follow, and pseudocode has been implemented in each script's audit log to document aspects of the code, while in-line comments are present to provide more technical detail. This section describes in more general pseudocode the operation of the script.

5.1 create.sh

There were not many difficulties associated with this script. The script works by checking how many inputs the user passed. If it is a valid number, then it checks if the user directory exists. If it does, the user exists so it returns the appopriate message. If it doesn't exist, then the user is new so it makes the user directory and creates the friends and wall file. One aspect of this which posed a minor problem were the files were not meant to be text files, and I initially designed them as text files which meant I had to ultimately come back and remove the extension from all of the scripts which references these files.

5.2 add.sh

There were not many difficulties associated with this script. The script works by checking how many inputs the user passed. If it is a valid number, then it checks if both user directory exists. If it doesn't, it checks which of the user directories don't exists and returns an appropriate message. If they do, then it checks the friends file and sees if the to-be friend is already on the friend list. If they are, it returns an error otherwise it adds that line of the file. To avoid partial friend matches e.g. 'ant' in 'anthony' where both are users the -Fxq is used to eliminate this case.

5.3 post.sh

There were not many difficulties associated with this script. The script works by checking how many inputs the user passed. If it is a valid number, then it checks if both user directory exists. If it doesn't, it checks which of the user directories don't exists and returns an appropriate message. If they do, then it checks the friends file and sees if the sender is already on the friend list. If they aren't, it returns an error otherwise it adds the message to the wall file. One aspect of this which caused continuous errors is that the autograder checks for a space after the "sender:" portion, however I had implemented a colon and tab initially as a delimiter which caused multiple fails for the autograder. This and the text files were the key reason why a number of rapid pushes to the git were conducted after the project development was concluded, to troubleshoot this, using the test_script.py file to identify what was being checked.

5.4 show.sh

This invovled checking the argument count, checking if the user directory exists, and if it does reading the wall file line-by-line to output it to the terminal

5.5 server.sh: Section 3

The first part of the development of the server section was not too complicated. The overall outline of the structre from the briefing document made it relatively clear how it should be approached, and as it was specified it should run continuously it was clear that the read would have to be incorporated inside of the loop. As each of the operation scripts handle error checking, very little additional development was needed for this as all lines simply involved calling the scripts already written and letting those scripts handle errors.

5.6 client.sh and server.sh: Section 4

This bit was a lot of struggle. I knew the structure would broadly follow what was implemented previously in the server section, primarily checking if the operation matches one of the valid operations and passing these inputs to the server pipe while also creating the client pipe. Similarly, it was clear to me that the server pipe would have to read from the server.pipe and output to the created client pipe, however what was really challenging for me was identifying where it was and wasn't working. In initial development, many of the operations would hang or run endlessly, and it was not entirely sure what was causing it outside of potentially looping through the pipes, and some of the functions would be less consistent than others (e.g. post and add). The key challenge which was posed was corrrecting these errors. Although I did ultimately manage to series of working tests, I'm not certain where additional optimisation can be complete to eliminate some of these errors. One area of potential optimisation is in implementing a user.lock on each of the scripts to protect concurrent running (e.g. writing to the same text file by two users simultaneously).

6 Conclusion

Overall, this submission details a Bash social media system with client-server architecture. Although the author admits the client-server architecture implementation is... likely highly optimisable and potentially limited by bugs which the author is not sure how to resolve, the key components and core structure has been implemented.