COMP30640 Project Report

Kieran Cosson 16498802

This is a report detailing the requirements and design for the social media application project for the Operating System module this semester. Some anecdotes of challenges faced and solutions found are given also.

Requirements

Create a simple social media application.

The users should be able to create a profile, add friends, post messages to the walls of their friends, and see the messages posted on the wall of another profile.

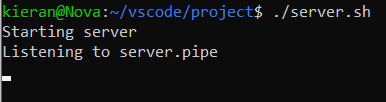
The service is provided by a server, to which the user will connect to via clients. Communication between the client and server will happen via pipes. The user will interact with the client through a terminal.

Interface

I am not sure if this falls under requirements, or design, but I will write a short guide to the interface of the application here. Note that this interface was mostly explicitly given in the assignment description, therefore, this information is mostly a repeat of what is there.

**Server**

To make the service available to users, run the "server.sh" file in the application's root directory. This will allow clients to connect and manipulate profiles, friend lists and walls. It should look something like:



No further interaction is required, and the server should be shut down via the client. It will print debugging information when processing a request. If the server is terminated unexpectedly, it will likely leave behind a file called "server.pipe". This should be deleted before the server is run again.

**Client**

To make requests to the server, "the client.sh" (also in the root directory) is run. It always takes an identifier as first argument, followed by a command, and optionally arguments for that command.

There are five possible commands:

* create (username)  
  This creates a new profile with an empty friend list and message wall. This will fail if the username already exists.  
    
  
* add (user) (friend)  
  This will add "friend" to the friend list of "user", allowing "friend" to post messages to the wall of "user".



* post (user) (friend) (message... )  
  This will write a message from friend to the wall of the user. This will fail if the "friend" is not on the friend list of "user". The user is also allowed to post to their own wall.  
    
  
* show (user)

This will show the message wall of "user".



* shutdown

This will shut down the server. It is the only way to do so gracefully.



Note that in each case, "id1" is given as the first argument. This is used to create the client's pipe, which in this example is named "id1.pipe". "id1" can be replaced with any string input as long as the pipe name remains valid and unique in the root directory.

Same as with the server, if the client is terminated prematurely, the client pipe will be left behind and will have to be deleted if a client with the same id is to run again.

Architecture/Design

Profiles are persisted as folders in the application root folder. Each user folder has the name of the user, and two subfiles - one called "friends" and one called "wall".

* "friends" contains the names of other users allowed to post on this user's wall. Each name is on a new line, and there are no duplicate entries.
* "wall" contains messages posted to this wall. They can be from users on the friends list, or from the user to whom the wall belongs to.

Since these files may be accessed concurrently by different processes, a locking system composed of two scripts is used:

* P.sh takes a name as argument and tries to create a link to a file of that name. If the file does not exist, it exits with an error. If the link cannot be created, it tries again and again until the link can be created.  
  + In retrospect, making a directory might have made more sense than a link, since then a name could be locked without the need to create a file for it. But it suits the purposes of this application as is.
* V.sh deletes the link. It exits with an error if the file or link to the file do not exist.

This system is far from perfect - one unfortunate side-effect of locking using the file system in this way is that if a program terminates unexpectedly during a critical section, the resource becomes locked forever. (Or at least until someone manually deletes the lock link)

There are 4 scripts for manipulating these folders and files:

* "create.sh" takes a username as the only argument and creates a folder for that username, with empty files "friends" and "wall". It returns an error if the folder already exists, or if the folder fails to be created for any other reason.  
  + There is a subtlety here in that the check for file existence comes before the directory creation, and the filename to be created is not locked in any way. If, through concurrency, two processes should go to create the folder at the same time, one will fail as expected. However, the message it outputs will be more general - it will not know why it failed. Given that this is mostly a cosmetic difference, I saw no need to use the semaphore system here.
* "add.sh" takes "user" and "friend" names as arguments and adds the friend to the user's friend list.   
  + Adding a friend simply means appending to the "friends" file. However, the script must first check if the friend already exists - the "grep" command is used to do this.
  + The flag -xqF is used with grep: x so that the full string is matched, q so that there is no output, just the return code, and F so that special characters in names are not treated as regex. (Without F, a name like "joe.\*" would be incompatible in the friends list with any name beginning with "joe", for example)
  + P.sh and V.sh here are used to ensure that two scripts do not enter the check at the same time.
* "post.sh" takes a "user", "sender" and one or many "message" arguments. It appends the message arguments, separated by spaces, as a message from the sender on the user's wall. It will exit with an error if the sender is not the user or a friend of the user, instead of posting the message.  
  + To test if "sender" is a friend of "user", the same grep check is used as with "add.sh".
  + The actual append is protected by P and V calls. This is because it turns out that writes beyond a certain size to files, in ubuntu at least, are not atomic. Some experimentation revealed that on my home system, messages of around 1800+ characters being written by concurrent processes could end up mangled together.
  + Taking many bash words as one message, and assuming that the words were space separated, is not ideal. It would have been nice to take a single, quoted, string. However, I do not know of a way to read text from a pipe that respects something like the quotation escape syntax that bash has.
* "show.sh" takes a "user" argument. It displays all messages on a wall of the given user.  
  + This essentially just amounts to cat-ing the contents of the user/wall file to the pipe and then to the console.
  + This operation is not protected by P and V calls. This means that an incomplete long message could be displayed, in theory, though I was unable to prove this experimentally.

These four scripts correspond to four of the possible commands that can be sent to the server - the fifth is just a signal that the server should shutdown.

The two remaining scripts are "client.sh" and "server.sh".

When "server.sh" is started, it creates a pipe called "server.pipe". If this file already exists, the script fails. Currently there is no way to configure the name of the pipe, so only one server can be active at one time. This is for the purposes of simplicity - having multiple pipes would be complicated. The client would need to know them all and choose which one to communicate with. Meanwhile, having multiple servers acting on one pipe might cause concurrency issues, I am not sure as of writing this.

"client.sh", as mentioned before, takes a "clientID", "command", and one or more extra arguments. The ID is used to create the pipe, if the file already exists, the script fails.

The communication between the client and server occurs as follows:

* The client appends a its arguments to the pipe, space separated. There is no client-side checking of the command beyond making sure that there is a clientID with which to create the pipe. Note that P,V are used here - the client must get a lock on the pipe before appending.
* The server is in a loop reading from the pipe. It reads each line and splits it into an array. The first element of the array allows the server to locate the client pipe, the second is assumed to be the command, remaining words are assumed to be arguments.
* The server attempts to recognize and execute the command. Output from this step is redirected to the client pipe. If the command is not recognized as "shutdown", the server continues to read/wait on the next line from the server.pipe. The commands (barring "shutdown") are executed asynchronously. If the command is not recognized at all, the server sends a message saying so through the client pipe.
* The client receives any output from the executed command. This can include error messages regarding arguments or the format of the command - this is how the parameter checking is done. There is no parameter checking for "shutdown" - any additional arguments are ignored.

Challenges and Solutions

Checking for friends

I immediately turned to grep for checking the friends file for a given username. The flags gave me little bit of trouble - originally, I just had q and x, to quiet output and match the full string.

I didn't notice that grep had a flag to disable regex special characters, and at first tried to use string replacement to escape them. This was a little awkward since some of those characters double as bash special characters as well. It was a relief to reread the man page and discover the -F flag.

Reading commands, word by word

Reading commands from a redirection in server.sh gave me (and the others in the breakout room at the time) some pause. Zammad was involved too.

At first, I tried to look for ways to split a string in bash (after read). I don't remember where that lead, only that I switched to using "read -a" instead. After that, it was a case of experimenting with bash array syntax while glancing at the bash manual.

A third option presented itself that would probably have been more elegant in our specific case, and that is simply to use "read id cmd arg1 arg2 arg3". Almost coincidentally, it coincides perfectly with our use case - the commands can have up to three arguments, except in the case of "post" where all the remaining arguments should be concatenated. This is exactly what read does too. However, for the sake generality and because I already had a working solution, I stuck to "read -a".

Client-side timeout

If the server were to make a mistake, or be terminated prematurely, and leave the client hanging, the client currently never times out. It just sits there until the user terminates it. A more natural behaviour would be to have it wait a certain time (and perhaps let the user adjust the time in a config file though that is beyond the scope of the project).

One approach I tried for that was to use the "timeout" bash command - this works to limit the time a command takes, but alas, it seems to not be made to work with redirections. And it was the redirection from the client pipe that was hanging.

A second, finickier approach, was to use the timeout on the read command instead of a redirection. That lead to a fair bit of awkwardness, especially where the "show" command was concerned - show can send multiple lines and read needs to be configured to accept multiple lines. Now that I'm writing this here, I imagine this route was feasible, or at least worth a zammad ticket, but at the time I dropped it in favour of keeping things simple and moving on.

Multi-line Response

My original client script relied on the "read" built in to read the server response, a legacy of attempting a timeout system. This didn't work when the "show" request returned multiple lines, since read would take in the first line and leave the rest. Swapping the read out for a redirection into the "cat" command fixed that.

It does show how brittle this program is at the moment - one mistake in sync leaves the server unable to respond and/or the client hanging indefinitely.

Locking the Server Pipe

I asked this as an academic question in Zammad, after it came up as a discussion in discord, but I did not really think that locking the server pipe would be important. However, later, when running a concurrency test in which many clients submitted requests at the same time, I found that about half the clients would never get an answer. In fact, only about half the messages (sometimes much less) where making it through the server pipe.

Adding a P/V lock to the server pipe solved this, all clients started working. I still do not know why. The commands being sent were well below the buffer length, and I was unable to replicate this with simple file tests.

Cleaning Up

In many tests, such as the one above, I would be left with stray pipes and locks and processes left after something minor having gone wrong and broken everything. For this reason, there is the "clean.sh" script included with the application - it will kill processes related to this application, delete .lock and .pipe files, and delete subfolders (user profiles) of the current directory.

Conclusion

In conclusion, the requirements given by the assignment have been met (I hope!).

However, this application is not one I would want to use any time soon - the dated interface aside, it is very brittle. Of course, I understand the intention was not necessarily to give us a practical exercise here. No one in their right mind would program a new social media system for real in Bash, and this was for learning purposes only.

On the subject of not using bash for something like this, one thing that was driven home to me while running tests on this application is that bash is very, very slow. It is interesting to able to do for loops and comparisons and functions and so on, but ultimately, its purpose seems more to link and organize and delegate heavy lifting to other programs. Redirections and file-name expansions are quite something, though.