1 Introduction

In this project you will be implementing a basic social media system. The system will have a client-server architecture, and let users (clients) exchange public messages on their "wall" with friends.

We will now describe the system you will have to build and its different features.

2 The Basic Commands of your Server

The server stores information about the different users, their wall, and who they are friends with. Each user, named `$user`, is represented by a folder `$user`. Example values for `$user` are `anthony` or `Dr A.`. Each user directory contains two files:

- `wall`, containing message posted by friends to the user's wall. Each line contains one message and has the following structure: `friend_id : message`
- `friends`, containing the id of the user's friends, one id per line. Only friends can post to a user's wall. Note that the friend relationship is not symmetric in our system: A can be a friend of B (A can write to B's wall) while B is not a friend of A (B can not write to A's wall).

Suppose our server has 3 users, `anthony`, `thomas`, and `aisling`, and that anthony is friends with thomas and aisling. In the server's working directory we would see:

```
$ ls -l
anthony
aisling
thomas

$ ls -l anthony
friends
wall

$ less anthony/wall
thomas: hey, how are you?
aisling: so bored from lockdown, I wanna go out!

$ less anthony/friends
thomas
aisling
```

Note: The user names and message may contain special characters and spaces.

The server needs to implement the following operations:

Create new user Create a new folder and files to store a user's information.

Add new friend Add a user to another user's friend list.

Post a message Add a message from one user to another user's wall.

Show wall Display the contents of a wall.

2.1 Create new user

First create a script `create.sh` that takes 1 parameter, `$user`, and creates the directory and files for the user `$user`. Your script should differentiate the following cases, print a corresponding message (make sure the exit message is the only thing printed in all your scripts, this will matter later in the project) and have an appropriate exit code:

- Too few or too many parameters
- The user already exists
- Everything went well

```
$ ./create.sh
Error: parameters problem
$ ./create.sh anthony
Error: user already exists
$ ./create.sh paul
OK: user created
```

2.2 Add new friend

Now write a script `add.sh` that takes 2 parameters, `$user`, `$friend`, and adds the user `$friend` to the user `$user`'s list of friends. Your script should differentiate the following cases, print a corresponding message (make sure the exit message is the only thing printed in all your scripts, this will matter later in the project) and have an appropriate exit code:

- Too few or too many parameters
- The `$user` user does not exist
- The `$friend` user does not exist
- User `$user` is already friends with `$friend`
- Everything went well

```
$ ./add.sh paul
Error: parameters problem

$ ./add.sh tony anthony
Error: user does not exist

$ ./add.sh anthony tony
Error: friend does not exist

$ ./add.sh anthony thomas
Error: user already friends with this user

$ ./add.sh anthony paul
OK: friend added
```

2.3 Post a message

Write a script `post.sh` that takes 3: `$receiver`, `$sender`, and `$message`. This script posts a message from user `$sender` to `$receiver`'s wall, i.e. add a "`$sender: $message`" line to the `$receiver/wall` file. Your script should differentiate the following cases, print a corresponding message (make sure the exit message is the only thing printed in all your scripts, this will matter later in the project) and have an appropriate exit code:

- Too few or too many parameters
- `$receiver` does not exist
- `$sender` does not exist
- `$sender` is not a friend of `$receiver`
- Everything went well

```
$ ./post.sh anthony
Error: parameters problem
$ ./post.sh tony anthony "Hey there"
Error: Receiver does not exist
$ ./post.sh anthony tony "Hey there"
Error: Sender does not exist
$ ./post.sh anthony aUserThatExistsButIsNotAFriendOfAnthony "Hey there"
Error: Sender is not a friend of receiver
$ ./post.sh thomas anthony "Hey there"
Ok: Message posted to wall
```

2.4 Display a wall

Finally, make a `show.sh` script that takes 1 parameter, `$user`, and displays `$user`'s wall, i.e prints the contents of the `$user/wall` file between a "wallStart" and a "wallEnd" line. Your script should differentiate the following cases, print a corresponding message (make sure the exit message is the only thing printed in all your scripts, this will matter later in the project) and have an appropriate exit code:

- Too few or too many parameters
- `$user` does not exist
- Everything went well

```
$ ./show.sh
Error: parameters problem

$ ./show.sh tony
Error: user does not exist

$ ./show.sh anthony
wallStart
thomas: hey, how are you?
aisling: so bored from lockdown, I wanna go out!
wallEnd
```

3 The server

Now you will set up the server of your application. As a first step, your server will read commands from the prompt and will execute them. Write a script `server.sh`. This script is composed of an endless loop. Every time the script enters the loop it reads a new command from the prompt. Every request follows the structure `req [args]`. There are four different types of requests:

- `create $user`: which creates user `$user`
- `add $user $friend`: which adds user `$friend` to user `$user`'s list of friends
- `post $receiver $sender $message`: which adds `$message` to `$receiver`'s wall from `$sender`
- `show $user`: which displays `$user`'s wall
- `shutdown`: the server exits with a return code of 0

If the request does not have the right format, your script prints an error message: `Error: bad request`. A good structure for your script could be the `case` one. Your script will look like that:

```
while true; do
    case "$request" in
        create)
            # do something
            ;;
        add)
            # do something
            ;;
        post)
            # do something
            ;;
        show)
            # do something
            ;;
        shutdown)
            # do something
            ;;
        *)
            echo "Error: bad request"
            exit 1
    esac
done
```

At this point you should probably test that your server works well by sending requests to it and checking that the files are created and modified accordingly. For example, run the following scenario (assuming a blank system at the start) and make sure things looks right after each step:

```
$>./server.sh
create anthony
OK: user created
create anthony
Error: user already exists
create thomas
OK: user created
add anthony thomas
OK: friend added
post thomas anthony hello
Error: Sender is not a friend of receiver
post anthony thomas hello
Ok: Message(s) posted to wall
post anthony thomas "hello there"
Ok: Message(s) posted to wall
show thomas
wallStart
wallEnd
show anthony
wallStart
thomas: hello
thomas: hello there
wallEnd
shutdown
$
```

Eventually your server will be used by multiple processes concurrently (the various clients accessing the server). You then need to execute the commands concurrently and in the background. The problem is that with concurrent execution comes the risk of inconsistencies; for instance when two commands accessing the same file run concurrently. You then need to update all of your scripts to avoid these potential inconsistencies. An option is to use a lock: $user.lock, whenever your server's scripts try to access user $user's folder or files (you can also have more fine-grained locking). Modify the server script `server.sh` to start the commands in the background and update the scripts to run concurrently.

4 The Clients

The last component of your system is the client application, which will be the interface for the users and will send requests to the server. You will write a script `client.sh`, which will be used with the following syntax: `./client.sh $clientId $req [args]` where $clienId is the unique identifier of this client and $req is one of create, add, post, show or shutdown.

First your script has to check if it received enough arguments (at least 2), and then that the request is a valid one. Depending on the request, your client should do different things:

create check that a client id and user name were given and send a create request to the server

add check that a client id, user name and friend name were given, and send an add request to server

post check that a client id, user name, a friend name and a message were given, and send a post request to the server

show check that a client id and a user name were given, send a show request to the server and print the result.

```
$ ./client.sh client1 show anthony
thomas: hello
thomas: hello there
thomas: Something important
thomas: Something silly
```

shutdown send a shutdown request to the server

How will the clients send the requests to the server and receive its answer? Your system will connect the client(s) and server with named pipes. The server will create a named pipe called `server.pipe` and each client will create a named pipe `$id.pipe` with $id the id given as parameter of the client.sh script (the client must send its $id to the server along with the requests).

Modify the scripts `server.sh` and `client.sh` to create those named pipes (and delete them when you are done!).

Now, clients need to send their requests on the server's pipe and, likewise, the server needs to read the requests on its named pipe. Modify both scripts accordingly. We recommend you open two terminals, one for the client and one for the server in order to test your scripts.

Note: Parsing arguments sent through a pipe can be a bit of a challenge. To make this easier you can consider that the user names never contain a space. Your individual scripts still need to handle this case but you can ignore it for the client/server communication. Otherwise you can choose a special character that you will use as a delimiter and assume that this character is never in the user name or service name. If you choose this solution please make it clear in your report.

The server now needs to send the replies to the clients' named pipes and not on the terminal. The client script needs to receive what's been written on the client's named pipe and process it accordingly.

Figure 1 shows a simple architecture diagram of the communication between processes using named pipes.

Conclusion

This is a complex project – yet every component is made of simple scripts/commands that you are/will be familiar with. As for every large project, do not be scared by the complexity but try to understand the overall picture and start coding small parts that you understand. For instance Section 2 should be ok so start with these scripts and focus only on them – later on you can integrate small pieces together, they should make more sense then.

Good luck!

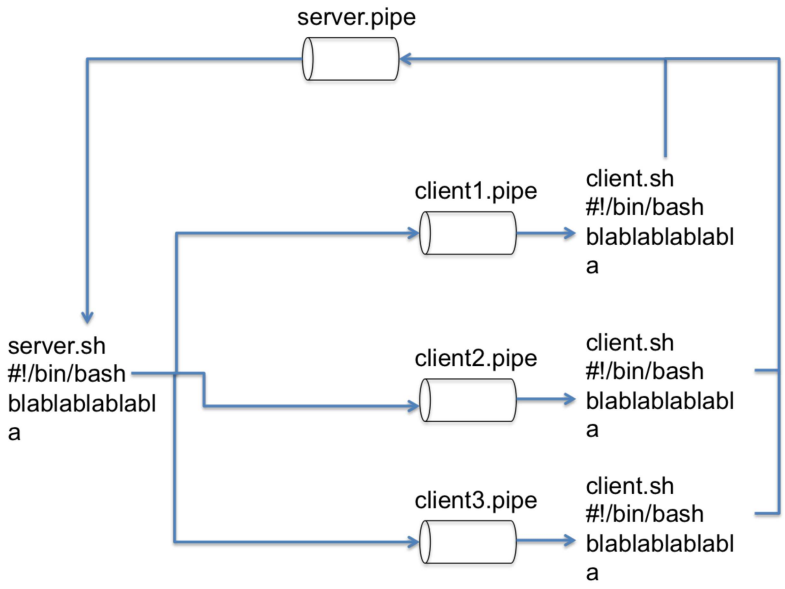

Figure 1: This diagram details the communication architecture of your application.