Lab Project

Project 1: a Facebook Server in Bash

Monday 14th November, 2016

General Instructions.

- You are encouraged to collaborate with your peers on this project, but all written work must be your own. In particular we expect you to be able to explain every aspect of your solution if asked.
- We ask you to hand in an archive (zip or tar.gz) of your solution: code/scripts, README.txt file describing how to run your programs, a 5-10 page pdf report of your work (no need to include code in it).
- The report should include the following sections:
  1. a short introduction
  2. a requirement section that answers the question *what* is the system supposed to do
  3. an architecture/design section that answers the question *how* your solution has been designed to address the requirements described in the previous section
  4. a series of sections that describe the different challenges you faced and your solutions. For instance, take one of the script, describe the difficulty you faced and your solution. These sections can be short – the objective here is to show how you crafted the solutions with the tools you have learned so far.
  5. a short conclusion
- Due date: 26/11/2016

1 The Basic Commands of your Server

The server is composed of a set of users. A user whose ID is $id possesses a directory $id. An example of $id is "anthony" or "Dr A.". Each user has two files in their directory:

- a `wall` file made of a sequence of text lines – each of them being a message posted by a friend. Each message has the following structure: id_of_the_friend: message.
- a `friends` file which contains all the friends of the user (an id per line). The only users who can post messages on a user's wall are the friends of this user.

An example for the user "anthony": in the repository anthony there are two files, a file anthony/wall:

```
$> less anthony/wall
sean: great show last night
niamh: it was ok
sean: just ok?
```

and a file anthony/friends:

```
$> less anthony/friends
john
niamh
sean
max
```

The server needs to implement the following 4 operations.

- **create** a user, i.e., create a directory for a user and initialise the two files for this user.
- **add** a friend to a user
- **post** a message on someone's wall
- **display** the content of a wall

The rest of this section details those 4 operations.

1.1 Creating a User

First write a script `create.sh $id` that requires one and only one parameter ($id) and creates the directory and the files of the user $id. Your script will print the following messages in the terminal to notify what has happened during the execution of the script:

- "`nok: no identifier provided`" if the script didn't get any parameter
- "`nok: user already exists`" if user $id is already on the server
- "`ok: user created!`" if everything went well.

Those 3 messages are the only allowed outputs of the script. This will be important at the end of the project.

1.2 Adding a Friend

Now you need to add the feature that allows to add a friend ($friend) to a user ($id) – this will be done in a script `add_friend.sh $id $friend`. First check that the user exists (i.e., there is a repository called $id). A friend is also a user, so also check that the friend exists as well (i.e., the repository $friend exists). Finally, you need to check that user $friend is not already in the list of friends of the user $id before adding them. To check that, look up for the expression $friend in the file $id/friends. For instance you could check the exit code of the grep command:

```
if grep "^$friend" "$id"/friends > /dev/null; then
...
fi
```

The exit code is 0 (true) if $friend is in the user $id's friend file and 1 (false) otherwise. The redirection to `/dev/null` is important not to spam the standard output of your script. Your script `add_friend.sh` has three possible output messages:

- "`nok: user '$id' does not exist`" if the user $id does not exist (i.e., there is no repository $id).
- "`nok: user '$friend' does not exist`" if the user $id does not exist (i.e., there is no repository $friend).
- "`ok`" otherwise (make sure to write the message on the standard output).

Make sure your script `add_friend.sh` has only those 3 outputs – in particular make sure that the error messages are redirected to `/dev/null`.

[]There is one thing I haven't mentioned yet: is friendship symmetric or not? This is really your call :)]

1.3 Posting Messages

Next, your program has to be able to post messages on to user's walls. Implement a script `post_messages.sh $sender $receiver message(s)` that takes at least three parameters. The first two parameters are users of the server: $sender is posting the message and $receiver receives the message on their wall. Next parameters compose the message. Adding a message on a wall consists in writing the line `$sender: message` on the file `$receiver/wall`. Your script must output the following messages depending on what happened:

- `nok user '$sender' does not exist` if the $sender does not exist

- `nok user '$receiver' does not exist` if the $receiver does not exist
- `nok user '$sender' is not a friend of '$receiver'` if the two users ($sender and $receiver) are not friends.
- `ok` otherwise

Again, make sure to have only those outputs. You can obviously copy a lot of the code used in the previous scripts.

1.4 Displaying Walls

Finally you need to add the possibility to display users' walls. Implement a script `display_wall.sh $id` that gets only one parameter, the user ID. The script needs to output the following depending on the context:

- "`nok: user '$id' does not exist`" if the user $id does not exist
- otherwise, your script prints `start_of_file`, followed by the content of $id's wall, followed by the line `end_of_file`.

```
$> ./display_wall.sh Balthasar
start_of_file
Abraham: Do you bite your thumb at us, sir?
Sampson: I do bite my thumb, sir.
Abraham: Do you bite your thumb at us, sir?
Sampson: Is the law of our side, if I say ay?
Gregory: No.
Sampson: No, sir, I do not bite my thumb at you, sir, but I bite my thumb, sir.
Gregory: Do you quarrel, sir?
Abraham: Quarrel sir! no, sir.
Sampson: If you do, sir, I am for you: I serve as good a man as you.
end_of_file
```

2 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 id [args]`. There are four different types of requests, corresponding to the four scripts you've written so far:

- `create $id`: which creates user id
- `add $id $friend`: adds a friend to the user id
- `post $sender $receiver message`: posts a message on receiver's wall
- `display $id`: displays id's wall

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

```
while true; do
case "$1" in
        create)
            # do something
            ;;
        add)
            # do something
            ;;

        post)
```

```
            # do something
            ;;
        display)
            # do something
            ;;
        *)
            echo "Usage: $0 {create|add|post|display}"
            exit 1
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 instance, you could try to run the following sequence of commands (or any similar scenario):

```
$> ./create.sh anthony
$> ./create.sh saorla
$> ./add_friend.sh anthony saorla
$> ./create.sh mary
$> ./add_friend.sh mary saorla
$> less saorla/friends
anthony
mary
$> ./post_messages.sh mary saorla "what's up?"
$> less saorla/wall
mary: what's up?
```

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 risks of inconsistencies; for instance when two commands accessing the same file runs concurrently. You then need to update all your scripts to avoid those potential inconsistencies. An option is to use a lock: $id.lock, whenever your server's scripts try to access user $id's repository or files. Modify the server script `server.sh` to start the commands in the background and update the four scripts to run concurrently. Note that the script `post_messages.sh` writes the messages on the receiver's wall (not the sender's) so you must protect the receiver's files from concurrent accesses, not the sender's.

3 The Clients

The last component of your system is the client application, which will send requests to the server. You will write a script `client.sh id`, which gets only one parameter $id representing a user.

First your script has to check if a parameter has been given and if yes it enters an endless loop. Every time it enters the loop, the script reads a request from the user (requests have the form `req args`). If the request is well formed, then the script prints `req id args` with $id the $id given as parameter of the script `client.sh`.

Now 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. Modify the scripts `server.sh` and `client.sh` with named pipes.

It is likely that so far you've been using control + c to stop the scripts. now we want you to be able to kill the named pipes whenever you want to quit a script – so you need to trap the command control + c. Use the following idea in your programs to do so: trap control + c and then delete the named pipe and then exit with the exit code 0.

```
#!/bin/bash

# trap ctrl-c and call ctrl_c()
trap ctrl_c INT

function ctrl_c() {
        #do something when control c is trapped
}
```

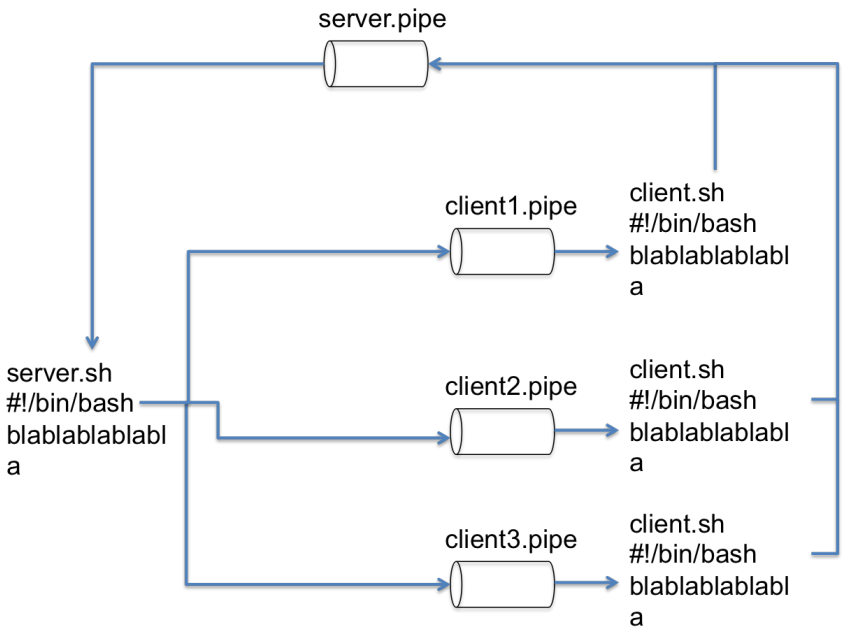

Figure 1: This diagram details the communication architecture of your application.

```
#rest of the script
```

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. You're recommended to open two terminals, one for the client and one for the server.

The server now needs to send the replies to the clients' named pipes and not on the terminal. There should be only a limited (4 ?) number of redirections in all your scripts. Now the client script needs to receive what's been written on the client's named pipe. There are three options here:

- the first word is `start_of_file` then what has to be printed on the terminal is everything until the keyword `end_of_file`
- the first word is `ok`, which means that the command executed correctly and you can just print a message such as `command successfully executed`
- the first word is `nok`, which means that the command did not execute on the server and you can print a message such as `Error: $msg` where $msg contains the end of the received line (remember that $msg gets the end of the line when you use `read ok msg`).

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 1 should be ok so start with the scripts and focus only on them – later integrate small pieces together, they may make more sense then.

You probably notice that the last paragraph of the Server section and most of the Client(s) section introduce (concurrent access, named pipes) concepts that you haven't seen yet – but that you will see in the next lab sessions.

To help you with this project I will organise a couple of (optional) extra sessions to go through some of the difficulties, etc.

Good luck!