Bash project #2 write up

The purpose of this project was to create a web server for our facebook-esque program we developed for project number one. Before I get into how I attempted to solve this problem, I think it is important to review what we know about the functions of web servers and browsers, and their interactions.

In the most basic terms, a web server is connected to your browser – your browser connects to the server and requests a page and the server sends the requested page. The way that happened was that the server broke the URL given into three parts: the http protocol, the server name, and the file name (the website itself). Following the HTTP protocol, the browser sent a GET request to the server, asking for the file, which then, in response, the server send the browser the HTML text for the web page.

In lab, we did all this. Firstly, we opened two terminals – one listening and one sending. The listening terminal had the command 'nc -l 8888 'with 8888 being the port number. The other terminal had the command 'nc localhost 8888' typed in. What we typed in the second browser appeared in the first – we had established a connection. Then we tried to get a web request from a web browser with one terminal listening with the same 'nc -l 8888 'command, and then a web browser opened to the url: http://localhost:8888. Upon connecting, the listening terminal displayed the HTTP request starting with the GET command issued by the web browser. In this case, our open terminal was acting as a server.

We also learned how to analyze web requests. This is where the first script comes in – read.sh. The first line is "read x" which tells the server to read something, in this case, what is coming from the web browser. Then we create the while loop. Inside that while loop I took some of the tips from Takfarinas' read_server.sh script. I first put everything in an if loop where x (the request coming from the web browser) is equal to (GET*) or, in other words, if what's coming from the browser is indeed a request as can be determined by the GET command that browsers issue forth. If whatever comes back does not have the GET command, then it is not an HTTP request and therefore is not valid. The asterisk then means a string of random variables – it can stand for whatever. In other words, if the web browser has issued an HTTP request by using GET, then this read script will listen to it and process the next actions.

Firstly, I wanted the URL. This is because in the previous assignment, we had the server.sh responding to commands made by the client – specifically create, add, post, and display. The server would read whatever command was given and run the connected script – create.sh, add_friend.sh, post_messages.sh, and display_wall.sh respectively. In the case of the web browser, the person needed somewhere to type these commands. I had it appear after the " / " in http://localhost:8888/. So, for example, if they wanted to create a user named Katherine, in theory they could just type "/create/Katherine" after the original URL. In order for the server.sh to know what script to run, it needs to know the command and for that to happen, the command must be essentially parsed from the URL itself. This is done in the read script using the cut command.

The URL is fetched with the line "url= `echo -e $x | cut -d' ' -f2` " where the backticks indicate everything that the URL equals, the whole line is looked at and the second field is cut. This is then set to the variable $url. This gives us everything after the 8888 or whatever port number. This is important because it will be used in our later cuts. Command is the first part of the URL. To find this we, again in backticks because we want the whole equation to equal the value of command, type `echo $url | cut -d'/ ' -f2` then echo command="$command" where f2 refers to the second field – the second part of the URL after the first delimited, which I have set as a /. This is where the command will be typed and thus we want that put

into the command variable. The next line, for $id is the same but with -f3 because it is in the third field. The next part, $friend, can be found in the fourth field so -f4 and then finally the $message in the fifth field so -f5. All of these arguments, the $command, $id, $friend, and $message are then echoed along the server.pipe so that the server can then pick them up and act accordingly. Then the if statement is closed, read x, and it is done.

For the server, I opted to use my own code from the last project and manipulate it according to my needs here. However, at some stages, I added parts of the skeleton provided, of which I shall note accordingly. In this case, the script creates a server pipe if there isn't one with a mkfifo server.pipe command. Then we enter a while loop. While true it reads the arguments that have been sent over the server pipe from read.sh. I then used the html code part of exec_server.sh and tried to decorate it a bit, also having it address the user with the given $id variable. Then using case, I have the server read and execute the commands.

To connect all of these scripts, I created a worker script called run_it.sh where it pipes server.sh with the nc -l 8888 connection command highlighted earlier, then pipes all that with read.sh. I used this all-in a while true loop.

This is all well and good except that we can't repeat it. Once I submit one URL, command, and user name, I cannot refresh and do another without restarting the server. This is where the server_looper.sh comes in. I amended the one given to us because I wanted to change the names to reflect my own script names, as well as changing the $username to $id because that is what I used in my server.sh. I did the same thing with read_master.sh and master_loop.sh.

This is a problem of redirection and for this, we must employ the use of ports. Ports are how a server makes itself and services available to the internet. Each port has a number - one for each service that is available on the server. For example, if a server machine is running a Web server and an FTP server, the Web server could be available on port 80, and the FTP server would be available on port 21. Clients connect to a particular service on a specific port. In localhost, it is the number that follows the ":".

Putting these scripts together proved a challenge, but each in their own right is relatively understandable. The server_looper.sh requires a client port and the id so in that each user gets its own port to the browser (e.g.: localhost:8888, localhost:8889 etc.). While this is true, it pipes this to server.sh but this time with an already given $id, a connection, and then read.sh. The master_loop.sh selects the client port that the server_looper receives. If one port is occupied, it finds another and assigns that. With the exec_master.sh, the web browser actually displays to the user that they are being redirected to another port, which is really cool I think. The read_master.sh plugs the given client port with the URL and the user id and then sends that to the server_looper.sh. I was unsure of how to interconnect all of these things, and ran out of time to figure out how to do so, but it was really cool to have the project finally click. My final thoughts on this project are that I wish it had come at a different and less busy time because I was eager to see my facebook bash project on the local host. It would be nice to see a project through from start to finish.