COSC349 Assignment 1

Josh Whitney (4442561)

For this assignment, I set about the task of creating an application to manage and run a simple game/chat server across 3 VMs on a Vagrant base. These three VMs are as follows: a database server, a webserver, and a game server. In short, the Database server houses player information is updated by the game server. The game server is the main workhorse of the application, hosting a service that users can connect to via TCP to engage in chat and games of tic-tac-toe in this implementation. Finally the webserver provides a basic website from which users can see a scoreboard from the database server, and also hosts a simple client download that can be used to connect to the game server.

Materials:

- Ubuntu/xenial64 – (All VMs) – A familiar Linux back-end
  - <300MB, First Build only
- MySQLServer – A common database host service
  - <30 MB, Each Build
- Python3 with MySQL Connector – The main language interface of the game server, and an addon to allow communication with the database
  - <30MB, Each Build
- Apache2 with PHP and MYSQL mods – Our webserver's main application for hosting the website, with addons to allow PHP and SQL support for the database.
  - <30MB, Each Build

Usage:

After using the initial "vagrant up", users should be able to simply navigate in their browser of choice to 127.0.0.1:8080, where they will find the webpage being hosted. Here they'll see the scoreboard, notably preloaded with the data from 100 games of tic-tac-toe already played to generate sample data within the game server on boot. From this page the user can download the java client, and start running it. Notably also, users can choose to use their own terminal with the telnet command to interface the game server too, useful for testing. From either method, the user can connect to 127.0.0.1:6969 (or port 6969 on whatever host the servers are running on), and the client will be connected and provided with prompts from there. A good amount of input sanitization should go on from there; And users will be able to set their username with !name, join the chat server with !join chat, and begin messaging all the other clients (this is best demonstrated with two clients open, each connected and in the chat room), or !join ttt to enter the tic-tac-toe lobby. This is the main function of the game server, and players may join a game against an AI (that always picks a random valid square), or against another player if one is available and also queued up to play another player. Upon game completion, users will be able to refresh the browser page to see their results updated accordingly, and compete on the leaderboard. If needed, one can also use the handy search function on the webpage.

Building and Debugging:

While the most straightforward approach to rebuilding is to simply "vagrant destroy" and "vagrant up" everything again, the better approach depends on the modifications:

Database server: "vagrant ssh dbserver" into allows us full access to anything in the database, and a convenient .my.cnf file has been generated to allow users to "mysql" then "use tictac_db" and start working immediately. Scripts that are intended to run on boot are housed in the shared folder, under the db directory. Via the SSH, users can modify and develop scripts while live for the database, but should be careful in removing columns from the players table, as these directly interact with the game and web servers. Adding columns or rows however, should be fine, as all the SQL interactions are specific to column names in the other two servers.

Webserver: This simply runs the index.php page live, and any updates will be observed as soon as a user refreshes their browser. This is shared under the web/www directory, and can be live changed from host computer.

Game server: Being that the game files are all in python, there's no particularly special build process involved, however it's not efficient during debugging to restart the server every time there's a crash. I'd recommend removing the two python3 commands from build-gameserver.sh, and starting the server. From here we enter "vagrant ssh gameserver", navigate to /vagrant/game, and can set about running "python3 game_manager.py". From here, the user can observer their changes from running the client normally; And can kill the game server by simply Control-C'ing out of the game_manager.py. This will disconnect the clients involved, requiring them to reconnect if wanting to test further. Beyond that, all the python files are stored in the shared /game folder, and are easily accessible from the host computer to edit. I'd recommend keeping the ssh terminal open during this process, allowing us to start/stop as needed, however if necessary, one can kill the process manually inside ssh; with "ps aux | grep -I python3" and then "kill ####" where #### is the appropriate ID which is running the game_manager.py.

Java Client: Ultimately there's nothing special tying the servers to the java client, it's very much just a serviceable plaintext TCP client, however if users wish to edit it in any way, they can edit the ticTacClient.java file under /client, compiling and running to debug as necessary. If the user is satisfied with their build, they can compile it into an executable jar file using the command that can be found in /client/makeJar.sh – Doing this will not change the jar file shown on the webserver's page however, and users must copy their own jar file to /web/www/files/ticTacClient.jar, or edit the webpage's download link, if they wish to have users download their new client.

Extensions & Modifications:

I've attempted to make python library in the game server as OO-style as possible, such that a developer can make their own entire game to add to the server (for example, chess), and the developer only needs make sure that he extends all the functions contained in /game/game.py in their own python module, and then adds at least one instance of their new game object to game_manager.py's gameList dictionary (convieniently placed at the top of the file), importing their library as needed. From there, the game_manager.py should pick it all up on it's next start, and begin accepting lobby entries from clients

One could also build a more elaborate user system, that might (for example) not allow multiple users of the same name online at the same time, or even add some layer of authentication to the process, by modifying the game_manager.py’s cmdInterpreter function, particularly around the !name area. Getting true encryption to work on this however, is likely to require a lot more legwork, in editing the game_manager’s main loop, player.py’s sendUpdate function, and probably requires a lot of setting up a specialized client.

Speaking of client, due to the simple nature of the consistently-formatted commands on the game server (which developers are encouraged to follow suit on), a developer could make their own client using any framework that supports a TCP socket connection; I went with Java for the easy multi-platform uses; However if one wanted to make a more specialized or graphical interface (with clickable tic-tac-toe squares) for instance, the need only build the connection and a way of interpreting the commands to/from the server.