Michael Tan
Computer Networks
Final Design Report
Fall 2017

Computer Networks Final Design Report

ByteRacer — A Console Based Typing Game

The network design behind byte racer is to utilize sockets as a realtime platform for "racing" your peers through a typing test. It is based on sites such as 10fastfingers.com and typeracer.com.

Sockets

ByteRacer uses sockets as its way of communicating over a network. The final implementation is designed based off a traditional client-server model, however, the original plan of the project was to build a decentralized network of nodes (players/clients) that would all share the responsibility of making sure all clients knew the progress of all other clients. However, after discussion with the professor, the benefits of such a network were seen as non-significant — at least not significant enough to warrant the extra time required to properly engineer such a network.

As such, to reiterate, ByteRacer is based off a traditional client/server model. To highlight the requirements of both the server and client scripts, the following characteristics were required:

For the server:

The server needed to support multiple connections. Originally, I had implemented this feature through the threading interface (import threading on python), and had created a new thread for each client connected to the server. In the end, I chose to accomplish this multiple connection ability through the select module (import select). The reasons for doing which will be explained later.

The server needed to be able to receive data from any client at any moment in time, as well as send data to each and every client at every moment in time. The first requirement (receiving data from each client) was achievable through the threading interface, as each thread could respectively run a permanent loop with a recv() call. However, threading did not provide a foundation for being able to send updates to each client. Why is this? Because if each thread is running on a continuous loop that is blocking (due to the recv() call), they are not able to, at any moment in time, send an update to that client. This conflicted with the requirements I had established for the server, and thus I was forced to look for alternate solutions to this problem.

One consideration I had to solving this was to spawn a separate thread (two per connection), where one thread constantly listened for a client message and the other could constantly send to clients. This turned out to be problematic, however, as not only was it difficult to maintain the logic for multiple threads per client, but there were also difficulties with threads talking to other threads (not to mention improper fundamental coding practices). Lastly, this method also did not work if you envision the following scenario: let us say that a client connected to the server is waiting to receive some update, but simultaneously is sending data to the server. When both the server and the client call recv(), the program is essentially stuck, as both nodes will be waiting (blocked) for data from the other connection.

To combat this issue in the end, I relied on the select module to provide me an interface for polling an input list of sockets (and on the client side, stdin as well) to check if any of the input sockets were ready to be recv()'d from. This turned out to be a much more efficient solution for implementing my requirements for the server, and was the chosen method of design for my final demonstration.

For the client:

The requirements for the client were very similar to that of the server in that the client needs to also be able to send and receive data at any moment in time. An additional complication of the client code was that I had imposed an additional requirement of being able to read input based on keypress. By that, I mean that when a user enters a key on the keyboard, the client code will parse it and, if desired, will send an update to the server that a user had entered a correct key. The traditional approach might be to read from stdin based on a nextLine() call, which would read the input up until the enter key was pressed. However, due to the real-time nature of the game I was implementing, I required this character-by-character feature.

In order to implement this, I was directed to the curses module by the professor. This was a fantastic starting place for me. It was the primary method for me to grab character input for this project. Additionally, the curses module provided me the ability to "draw" on my console window. The curses module is similar to that of canvas for HTML, and has characteristics similar to that of a game running on a continuous loop with an update() or tick() function. My client application essentially clears the entire console window on each update/tick, and "draws" text on screen, which allows me to simulate a real time application.

The Game

The game itself had several requirements as well that have network implications.

First of all, I wanted the game to only begin when all clients who are connected to the network choose to "ready up". This requires me tracking user specific metadata such as connection address (IP and port), a username (specified through a command line argument when running the client.py script), a ready status, their current progress, and other parameters. These can all be found within the code.

Since there were several back and forth stages in the initial setup of the game, I decided to implement the logic for the game using states. It is an extremely simple design as of now, but there are essentially three states to the game (pre-game, countdown, started, post-game), which are referred to throughout the code using the integers 1 through 4 respectively. The current state of the game changes the workings of the client and the server.

In pre-game, the server accepts any new connections from clients that would like to play the game. Upon receiving a new connection, the server stores the client using a unique ID generated from their IP address and port, along with some initial metadata initializes the status of that player for the game. The server will then wait for a "ready" message from each client, and upon receiving any ready message, will blast out an update to all connected clients informing them of how many users are currently readied up.

Michael Tan
Computer Networks
Final Design Report
Fall 2017

The method of communication for this pre-game setup process was using TCP sockets. A brief note on using TCP: UDP was considered for this project, but was quickly discarded as the networking requirements were not strenuous enough such that the relative slowness of TCP would add any benefits to the program. I developed my own extremely basic protocol between players and the server within this program for the setup process. There are essentially some commands that the server accepts, and will process from the client. So when the client connects to the server, it sends a “txt pls” request that the server interprets as its initial connection. The server saves the new connection info and sends back a randomly generated text prompt to the client in preparation for the actual game. When the client “readies up”, it sends a “ready” request. This paradigm holds throughout the entirety of the program and, while not strictly defined as a certain set of rules, provides a concrete method for communication between clients and server.

Parts of the Code and Running a Demo

The code for this project is essentially comprised of three files: a server (server.py), client (client.py), and text file which holds the top 300 most common english words for the prompt generation.

In order to run your own version of the demo, simply download these three files (in the FOR_DEMO folder).

Start by running the server (you can choose your own options using —help for argument details, or more simply run it in debug mode with “python3 server.py -d”)

Next, create as many clients as you would like using multiple terminal windows and the following command: python3 client.py

It is suggested to include an optional -u (or —username) parameter when running the client.py script so that you can see who is who (if no user is specified, a random uuid64 ID will be generated.

When you are ready to begin the demo, simply type “ready” in all the client windows, the server will start a countdown and the game will begin. Note that your WPM (words per minute) is also tracked while playing the game. The calculation of which is as follows:
((characters typed / 5) / time_elapsed) / 60

These instructions will also be on GitHub, in the form of a readme.