John Parry  
Dr. Nemo  
Network Fundamentals  
18 April 2019

Project 2 - Multi-Threaded Server

# Code and Architechture

CSERVER

The code for this project is mostly the same code I used for Project 1 but with a few tweaks and added threading functionality. Below in Figure 1 is a list of all the functions used in the cserver program. Though many of them are new functions, they simply exist to house code that was previously in the main part of the program so the main thread could always focus on accepting new meisters. Other functions that previously existed including addQuestion and getQuestion were removed because they were only ever called once or twice and so their code was simply integrate in without making a function call.

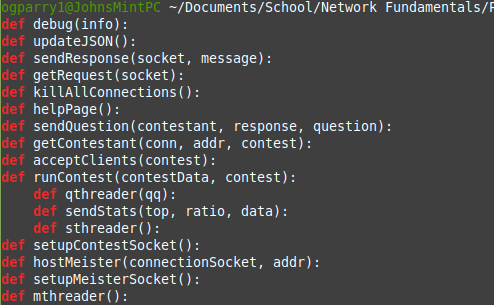


Figure : Server Functions

Some of the new functions include hostMeister, which houses what was originally the main loop of Project 1 so that multiple instances could be called for different meisters. Previously, I had hard-coded a starting port from which the server should attach itself and would loop incrementally until it found an open port. In this project, I used some of the code from the provided pre-made code to directly bind to an open port in functions setupContestSocket and setupMeisterSocket with the only difference between them being that the meister port was saved as a global variable because all meisters would run through the same port and so this would only need to be setup once.

Another important function that now exists is the killAllConnections function which essentially does it’s best to close all connections as gracefully as possible. Whenever a new connection is established, whether that be to a meister or a contestant, (c, addr) calculated from socket.accept() is appended to a list called allcons if a meister ever sends the ‘k’ or ‘kill’ command, the qbank file will be updated with the newest version of questions in the global qbank dictionary and allcons loops through every element and closes all connections that are still alive before exiting the main program. Early in development, I tried to keep track of all the threads as well, but eventually discovered how to mark a thread as a daemon which means that as soon as only daemon threads are left alive, they all exit. My strategy then became to instantiate all threads as daemons so exiting the main thread would cause kill the process without leaving open threads.

CONTESTMEISTER

The Contest Meister is mostly the same as the original client code for Project 1 except that it also incorporates a bit of threading. Specifically, the old qclient serially sent a request and waited for a response which it then printed to the screen. However, because I wanted contestmeister to be able to exit gracefully should another meister decide hit the kill switch. I achieved this by opening a side thread that would continuously be listening on the socket for data being transmitted. Once captured, the message is stored in global variable ‘response’. The message is first checked against the string ‘EXIT’. If it matches, then the socket is closed and os.\_exit(0) is called to end the client, otherwise, the message is printed to stdout.

While a side thread is handling the network traffic from the server, the main loop simply waits for standard input from the user. Once input is provided, the first character of the string provided is tested for ‘p’ which is the only special case where the request must be build. A small loop builds the request string exactly as it is input until the choices are given. The function getValidChoices is then called which parses the remaining input as it comes into a dictionary of choices and checks the validity of the answer letter give compared with the keys of the choices dictionary. It then builds and returns the remainder of the response string once a valid answer has been given and at least one choice has been entered. This string is then return concatenated with the previous input and sent to the server for parsing and handling. Since parsing is handled by splitting the string at each newline character, all other single-line commands are immediately directed to the server where no parsing is necessary aside from the question number or contest number arguments given.

CONTESTANT

The contestant is the simplest of the programs because it simply waits for a message from the server, displays the appropriate information to the user which most times is completely supplied as the message from the server, and returns the user’s input back to the server. Wash, rinse, repeat. I didn’t include much error detection in the contestant client because it was so simple. For instance, the user technically doesn’t have to provide an answer that was displayed or an answer at all. The only thing the user can’t do is enter an empty string. They will be re-prompted to answer in that case, otherwise, if the answer they provide is not the upper or lower case version of the answer, then it is marked wrong. The only slightly sophisticated portion of this code is for getting a valid nickname for the user which requires some threading.

# Message Structure and Meaning

The basic message structure is simply a concatenation of all things input my the user with a few exchanges of keywords such as ‘EXIT’ and ‘FINISHED’. The most complex messages are built in the server such as statistics on each question and are concatenated in such a way that directly printing them will be aesthetic and ordered. As stated before, messages such as put messages are parsed into their elements first by newline characters and then by whitespace.

When it comes to keyword messages, they mainly exist as asynchronous broadcasts to other network processes. The ‘EXIT’ keyword is the universal response to all clients from the server to gracefully disconnect and shutdown. It is broadcast whenever the kill command is given. Another one is the ‘FINISHED’ keyword which is specific to contestant processes and indicates to them that the contest has completed. On completion, the contestants will output a thank you message and exit.

# Local Storage

# On the first project, I used sqlite3 for local storage but for this project, I found it easier to use JSON because it would provide better flexibility with differing numbers of choices in each question and lack of much setup. All meisters share a global dictionary called qbank which contains the number: question\_data of every question entered. The program loads an initial databank from a file called qbank if the file exists; otherwise, qbank defaults to an empty dictionary which is immediately dumped into a new qbank file using the updateJSON function. This function is called after every time the dictionary is updated and before killAllConnections is signaled as well just for safety.