
PART-1: SOCKET IMPLEMENTATION WITH MANUAL THREAD IMPLEMENTATION

We have implemented part-1 of socket communication with multi threads created dynamically using user input. We have divided the files into client and server where the client side files are divided again into client and client_runner. The partition goes like this:

—---------

CLIENT

—---------

client.py
client_runner.py

—---------

SERVER

—---------

server.py

server.py has the code two methods lookup and check_task_queue.

We have created a thread list called thread_pool_list which is a list of threads created and binded to a method called check_task_queue and these threads are started after creation which indeed looks for check_task_queue method execution. The server program first establishes the connection to the client and we store all these client sockets in a queue called task_queue. Once a client task is put into task_queue then immediately the idle thread starts executing the check_task_queue method automatically. We get two arguments from the client which is a function name and the stock name as mentioned. We have a global dictionary stock_info which has company names as keys and price, trading volume as sub keys. After getting the function name and stock_name from the client we call the lookup function to see if a stock name is really registered or not. If yes we return the price of the stock name else we return -1. The server always tries to listen to the client socket and puts the task in the task queue once it gets a request from the client.

PSEUDO FLOW OF SERVER:

a) Take optional input from user like number of threads, host and port
b) Create number of static threads and start it and bind it to a function
c) Listen to client for a connection always
d) Once the client request is got, push it to task queue

e) The function binded to the threads is always looking for the size of the task_queue, if it is empty, it doesn’t perform any action. If the queue is not empty, then the function picks up the task from the task_queue.

f) Idle threads take the request from the queue and process it.

g) The price or -1 is returned from the lookup function to the client through the same socket.

client_runner.py

It takes 3 optional arguments from the command line which are host, port and stock name. We are running a loop for 100 times to send 100 requests from the client as multiple client processes where we are calling client.py 100 times from this loop which are sequential requests.

client.py

We take the 3 arguments host, port and stock name. Create a socket object, create a list of function name and stock name. Create a socket connection to server and send the input list to server and then receive the output from server socket. So if we run client_runner.py and server.py then the server receives 100 sequential requests from a client as multiple client processes.

PSEUDO FLOW OF CLIENT:

a) Establish socket connection to server
b) Run 100 times through loop to send sequential requests to server
c) Every time establish a socket connection to server
d) Send inputs to server
e) Servers processes it
f) Receive output from server

PART 2: Implementation of gRPC with python’s inbuilt Threadpool

gRPC is the modern remote procedure call framework, in which we have to define protocol buffers (protobuf). Protocol Buffers is a cross-platform data format.
In gRPC, we have to start with defining the gRPC services and the message requests, responses using protobufs. In our case they are defined in `stock_service.proto` file. After defining the proto file, we need to compile it to make it compatible with our python code. It auto-generates client stub and servicer code to interact with the gRPC methods.

Then we began with setting up the server. We created the gRPC server with a thread pool such that it can handle the client requests in a concurrent manner. We then implemented our three main rpc methods namely `Lookup`, `Trade`, `Update` to handle the logic stated in the problem statement.

Clients interact with the server through these rpc methods via the client stub which is auto-generated earlier.

In server, we also defined a dictionary to store the stock information such as price, trading volume and maximum volume. We are taking the number of threads in a threadpool and a maximum volume that a company can trade through the command line arguments. The maximum volume we got as input is updated in the pre-defined dictionary.

In client side, we have three different client processes each with a unique functionality.
`lookup.py` (calls Lookup rpc method to get a given stock information)
`trade.py` (calls Trade rpc method to increase the trading volume of a given stock)
`price_updater.py` (call Update rpc method to update the price of a random stock with random price). For price_updater, we declared a list of companies along with their price range (with positive min and max prices). A random company is chosen and it updates its price with random value within that range. This operation will take place at random intervals for every 10 seconds (interval is configurable by changing the values inside the code). Extra care is taken to handle the edge cases in the client programs.