Design document

1. Part 1

1.1. Problem Statement

To implement online stock trading as a socket-based client-server application.
To implement a thread pool to handle requests concurrently.

1.2. Languages used

Python

1.3. Architecture Overview

The system consists of a client and server communicating through a socket connection. The server is responsible for implementing its own thread pool to handle Lookup requests. The client sends the Lookup request to the server either sequentially or concurrently.

1.4. Client Architecture

The client is implemented in python. The client reads the hostname and port from the config file and establishes a connection with the server through the socket. Once the connection is established the client then sends lookup requests to the server, one request per connection. The client can send requests sequentially or concurrently using threads.

1.5. Server Architecture

The server is implemented in python. The server binds a socket to a host and port and starts listening for requests from the client. The server is responsible for implementing *LookUp*() function and uses the custom handwritten thread pool implementation to handle the requests concurrently. The server also maintains an in-memory database containing data for two stocks: *GameStart*, and *FishCo*. If the requested stock is present in the database, the server returns the price of the stock (with 2 decimal precision) or returns -1 if the stock name is invalid and 0 if the stock is present but suspended. The maximum number of threads is configurable and is read from the *config.py* file at start-up.

1.6. ThreadPool Design

1.6.1. The constructor ***__init__*** initializes the thread pool by creating a list of threads equal to the specified pool_size and starting each thread. Each thread runs the _threadedFunction function in a continuous loop.

1.6.2. The ***getSocketFromQueue*** function retrieves a request from the request queue and returns it to the calling thread. Mutex lock is used here in order to avoid race conditions and to access requests concurrently.

1.6.3. The ***_threadedFunction*** function: Each thread runs the _threadedFunction function. The function continuously checks for pending requests on the request queue. If there is any pending request, the thread removes that request from the queue handles the request, and sends the response on the socket.

1.6.4. The ***addReq*** function is used by the server to add incoming requests to the request queue. It takes a socket connection, client address, thread function, and arguments as input and adds them to the request queue.

1.6.5. The ***endThreads*** function is called to terminate all threads in the thread pool by joining each thread to the main thread.

1.7. In-memory database

The server uses an in-memory dictionary with the following structure: *{stockName: stockObject}* to store the stock data. The stock database is initialized on the server start-up.

The initial stock prices and maximum volume that can be traded are read from the config file- *config.py.*

1.8. Error/Exception handling

Since the in-memory stocks database will be accessed by multiple clients simultaneously, locks should be used to maintain synchronization and ensure that the data is consistent and accurate. try-except blocks are used to handle unknown errors and prevent the application from failing.

1.9. Data model

The Stock data model is used to store the details of the stock. The Stock data model has the following attributes: *{name, price, maxVolume (maximum trading volume), volume (current trading volume), isTradable (a flag indicating if the stock is currently tradable)}*. The constructor takes in the name and price of the stock to initialize the stock object. The other attributes *maxVolume*, *volume*, and *isTradable* are set to default values of 100, 0, and false respectively.

2. Part 2

2.1. Problem Statement

To implement an online stock trading client and server distributed applications using gRPC for communication.

2.2. Implementation Details

2.2.1. **Language Used**: Python

2.2.2. **Framework**: gRPC

2.3. Architecture Overview

The system consists of a client and a server communicating via gRPC. The server is responsible for handling client requests and maintaining the stock database. The client interacts with the server to perform trading activities.

2.4. Proto Design

2.4.1. The ***Lookup***() call will take a StockName message that includes a string field for the stock name. The response will be a StockInfo message that includes two fields: a float field for price and an int field for the trading volume.

2.4.2. The ***Trade***() call will take a TradeInput message that includes three fields: a string field for stock name, an int field for quantity (the number of items to buy or sell), and an enum for trade type (*BUY* for buy and *SELL* for sell). The response will be an int field for the trade status (1 for success, 0 for suspended trading, and -1 otherwise).

2.4.3. The ***Update***() call will take a UpdateInput message that includes a string field for the stock name and a float field for price. The response will be an int field for the status of the update (1 for success, -2 for invalid price, and -1 otherwise).

2.5. Server Architecture

The server is implemented in python using Pythons' built-in thread pool and gRPC. The server is responsible for implementing the three gRPC methods: *Lookup*(), *Trade* (), and *Update*(). The server also maintains an in-memory database containing data for four stocks: *GameStart*, *FishCo*, *BoarCo*, and *MenhirCo*. The server uses an in-built thread pool to handle concurrent requests. The maximum number of threads is configurable and is read from the *config.py* file at start-up.

2.6. Client Architecture

The client is implemented in python. The client reads the hostname and port from the config file and establishes a connection with the server. The client is responsible for sending *Lookup*, *Trade*, and *Update* requests sequentially and concurrently using multiple threads.

2.7. In-memory database

The server uses an in-memory dictionary with the following structure: *{stockName: stockObject}* to store the stock data. The stock database is initialized on the server start-up. The initial stock prices and maximum volume that can be traded are read from the config file- *config.py*.

2.8. Error/Exception handling

Since the in-memory stocks database will be accessed by multiple clients simultaneously, locks should be used to maintain synchronization and ensure that the data is consistent and accurate. try-except blocks are used to handle unknown errors and prevent the application from failing. In addition, scenarios such as invalid stock names, negative prices, negative quantity, and exceeding the maximum allowed trading volume are handled at the server.

2.9. Data model

The Stock data model is used to store the details of the stock. The Stock data model has the following attributes: *{name, price, maxVolume (maximum trading volume), volume (current trading volume), isTradable (a flag indicating if the stock is currently tradable)}*. The constructor takes in the name and price of the stock to initialize the stock object. The other attributes maxVolume, volume, and isTradable are set to default values of 100, 0, and false respectively. Pythons' Decimal module is used to handle the precision of the stock price with 2 decimal places to avoid floating point arithmetic issues. Error handling is also implemented to ensure that the inputs for price and quantity are valid and within the expected range.

The Stock model provides the following methods:

2.9.1. The ***getPrice***() method returns the price of the stock.

2.9.2. The ***updatePrice***() method updates the price of the stock with a new value.

2.9.3. The ***trade***() method allows trading of the stock by taking in the quantity and type of trade (either BUY or SELL) and updating the volume attribute. If the maximum trading volume for the stock is reached, the isTradable flag is set to False.

3. References

3.1. Python concurrent.futures src code

3.2. GRPC Python quickstart guide

3.3. Intro to Python threading

3.4. Threading Mutex Lock in Python