Lab 1: Stock Lookup and Trading System

Design Document

Authors: Isha Nilesh Gohel, Roshini Sanikop

Status: Final

Version: 1.0

1. Introduction

a. Overview

This document describes the design of a multi-client stock lookup and trading system that allows Gauls to query meme stock prices and trade stocks. The system consists of two implementations:

1. A socket-based client-server model with a custom thread pool (Part 1)
2. A gRPC-based client-server model using a built-in thread pool (Part 2)

We also **evaluate performance** (Part 3) by measuring the latency of different requests under varying client loads.

b. Goals

Our primary goal was to:

1. Support concurrent stock lookup and trades
2. A threadpool implementation from scratch - Part 1
3. Leveraging the built-in thread pool mechanisms for request handling and gRPC - Part 2
4. Measure Performance and Scalability

2. System Architecture

a. Socket-based Client-Server with a Custom Thread Pool:

Components -

i. Client (client.py) - The client initiates a lookup request via a socket connection. This request is sent to the server.

ii. Server (server.py) - The server receives the request through the socket. It passes the request to the middleware, which determines how to process it.

iii. ThreadPool (threadpool.py) - The middleware forwards the request to the thread pool. The thread pool uses semaphores to manage concurrent requests. An idle thread is selected to handle the request.
iv. Catalog (Catalog.py) - The assigned thread calls the lookup function in catalog.py. The catalog retrieves the relevant stock data or if it doesn't find anything relevant, it returns an error message.

Response Flow -

The response is sent back from catalog.py → thread pool → server. The server transmits the response back to the client via the socket.

b. gRPC-based Client-Server with Built-in Thread Pool

Components:

i. gRPC .proto File - it defines requests, responses, and functions (Lookup, Trade, Update). The grpc command generates the required Python files.
ii. Server (server.py) - The server processes gRPC requests using the generated stubs. It also handles Lookup, Trade, and Update requests.
iii. Catalog for Storing Meme Stocks (Catalog.py) - the Catalog stores the stocks and supports Lookup, Trade, and Update operations.
iv. Synchronization using Locks - This is done to protect Lookup, Trade, and Update methods and ensure thread safety when multiple clients modify stock data(Trade and Update).

A separate Update client periodically updates stock prices using time.sleep().

3. Design Choices:

In general, we have separated our business logic - Catalog.py from network logic to ensure modularity and easy maintenance, code reusability and easier debugging & testing. We have also written a simple **Bash Script** to automate **multiple clients execution** to simulate **concurrent requests**.

Part - 1:

I used **semaphores** to control concurrency **instead of locks in my thread pool implementation**, thereby, allowing better **system throughput** by minimizing thread blocking. This is because part 1 required us to perform only lookup requests on the Catalog.

Part - 2 :

I have used locks Instead of semaphores for synchronization in Part-2 to ensure that only one request at a time modifies stock data, thereby preventing race conditions when multiple clients update stock values.

I have used a toss variable to route requests to ensure a balanced workload between Lookup and Trade operations.

I have a separate Client for Update Requests to allow for periodic updates without affecting main client-server interactions.

4. Concurrency & Synchronization

Part 1 -

The socket-based client-server model follows a thread-per-request approach, where each incoming client request is handled by a dedicated thread from the custom thread pool. When a client sends a lookup request, the server places it in a request queue. An idle worker thread from the pool picks up the request, processes it, and sends back the response before returning to an idle state, ready for the next request.

Since multiple clients can issue requests concurrently, semaphores are used to limit the number of active threads, preventing excessive resource consumption. This ensures that the system can handle multiple client connections efficiently without overwhelming the server. The thread-per-request model allows scalability by ensuring that each request is processed independently while keeping resource usage under control.

Part 2 -

In the gRPC-based model, concurrency is handled by gRPC's built-in thread pool, allowing multiple requests to be processed in parallel. I have used Locks in Catalog.py to ensure thread safety when modifying stock data, specifically in Trade() and Update(). I have also used lock for Lookup(). I have written a separate Update client that periodically modifies stock prices using time.sleep(), introducing additional concurrent updates. I have written a bash script to launch multiple clients concurrently, simulating a distributed environment.

5. Performance Evaluation

We have created a document which gives more details about the performance of both part 1 and part 2. It answers all the questions asked in part 3.

6. References

https://docs.python.org/3/library/threading.html
https://docs.python.org/3/library/threading.html#condition-objects
https://www.pyfilesystem.org/
https://github.com/umass-cs677-current/spring25-lablet1-roshinisanikop
https://grpc.io/docs/languages/python/basics/
https://github.com/grpc/grpc/blob/v1.66.0/examples/python/route_guide/route_guide_client.py
https://github.com/grpc/grpc/blob/v1.66.0/examples/python/route_guide/route_guide_server.py
https://grpc.io/docs/what-is-grpc/introduction/
https://grpc.io/docs/languages/python/quickstart/
https://www.w3schools.com/python/module_random.asp
https://docs.python.org/3/library/time.html
https://www.geeksforgeeks.org/python-time-time-method/