CS3211 Assignment 2 Report

Hubertus Adhy Pratama Setiawan - A0200816R
Simon Julian Lauw - A0196678A

Architecture

The engine starts with a connection handler and a distributor goroutine. Upon receiving a new connection, a new goroutine is spawned to handle the communication with the client. When a new order is received, the communication handler first forwards the order to the distributor. Then, the distributor checks its internal map whether the symbol exists or not. If the symbol is not in the map, it spawns a new goroutine to handle all orders of that symbol. The distributor also stores the channel to the newly spawned goroutine in its internal map. If the symbol is in the map, the distributor sends the order to the worker. The architecture can be seen in figure 1.

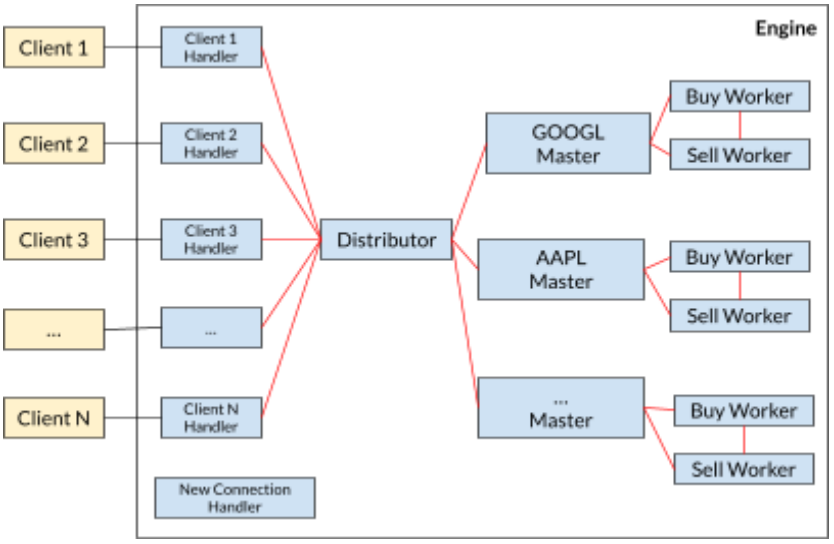

Figure 1. Architecture of the matching engine. Each blue box represents a goroutine and each red line represents one or more channels between the goroutines.

Concurrency

The level of concurrency achieved by the matching engine is **phase-level concurrency**.

1. Orders of different symbols can execute concurrently.
2. Orders of the same side cannot execute concurrently. One order from each side (i.e. one buy and one sell) may execute concurrently. In this regard, cancellation of a resting buy order is treated as a sell order and cancellation of a resting sell order is treated as a buy order.

A. Handling requests from multiple clients

To support the concurrent execution of orders coming from multiple parallel clients, a goroutine is spawned to handle each client. When an order is received from the client, the client handler will send an Order object to the Distributor through a channel. However, since the orders from a client cannot be re-ordered, the orders from a single client are sent one by one. To enforce this, each Order object has a done channel to indicate that the current order has been processed before the next order can be sent by the client handler. Only one goroutine that can guarantee the current order won't be processed further should send a feedback (i.e. using empty struct) to this done channel.

B. Instrument-level concurrency

For each symbol, three goroutines are spawned. The master (worker) determines the type of the order and sends the orders to the respective worker. The workers, one for each side (buy, sell), perform the matching and cancellation of orders. Observe that orders with different symbols will go to different sets of goroutines. Goroutines of different symbols do not communicate with each other and do not share the same data structure. This implies that they are independent of each other and can work in parallel, enabling instrument-level concurrency.

C. Phase-level concurrency

When executing a buy order and a sell order concurrently, the matching engine needs to handle the case where a "flying" order from both sides needs to be written to the order book. Since it is possible for the flying orders to match, these insertions needs to be serialised. The algorithm to handle such case is as follows:

1. If the other side is idle, match the order (i.e. send the order to the buy/sell worker).
2. Else, check if there is a possibility to match the current order with the currently active order being matched on the other side (i.e. sell price <= buy price).
    a. If it is possible for the flying orders to match, wait until the other side is finished (read from buyDone/sellDone channel)
    b. Else, match the current order.

To wait for each worker to finish, there are two channels of size one which acts like a mutex for each worker (called buyDone and sellDone). Upon initialization, an empty struct is pushed to buyDone and sellDone (unlocking mutex). To acquire the "mutex", the goroutines can just read from the channel.

Note that if the current active order is not fully matched (i.e. it becomes a resting order), the worker must send the resting order to the worker of the opposing side. The insertion of a resting Buy Order must be done before the matching of the next Sell order and vice versa. Otherwise, there is a possibility that the resting Buy Order can match with the Sell Order but the engine doesn't match them. To resolve this, the buy/sell workers only indicate that they are done after they have inserted the orders to the opposing side worker. Suppose we have a sell order to be inserted, this means the sell worker needs to send this resting sell order to the buy worker to be inserted. During this process, if the master wants to send a new buy order to be matched, it will need to check if the resting sell order can match with the current buy order since the buy worker is not "done" yet. If the current buy order can match with the resting sell order, the master will wait until the insertion is finished. Otherwise it can immediately send the buy order to be matched to the buy worker.

Go patterns

Confinement: There is no shared data structure between the goroutines; each data structure is local to a goroutine. The goroutines synchronise with each other by communicating through channels.

For Select loop: The distributor and the worker components need to read input from multiple channels and perform an action depending on the channel. The for select loop pattern is used to implement this behaviour.

Fan-Out: As seen from the architecture diagram, the distributor component spawns multiple goroutines and de-multiplexes the orders to different goroutines based on its symbol.

Data Structures

Component	Data Structures	Explanation
Distributor	1. `Map[string]chan<- Order` 2. `Map[uint32]string`	1. Maps symbols to the master (worker) channel. 2. Maps order IDs to its symbol for cancel order
Master Worker	1. `Map[uint32]inputType`	1. Maps order IDs to its type (Buy/Sell) for cancel order
Buy/Sell Worker	1. `Map[uint32]*Order` 2. `PriorityQueue<Order>`	1. Maps order IDs to the actual Order struct 2. Stores the Order structs in sorted order based on best price

Table 1. List of data structures used and its purpose

Testing Methodology

The matching engine was tested using both handwritten and auto-generated test cases used in assignment 1. The engine was tested using 40 clients, 500 symbols, and 25 million random orders. The script to generate the test case can be found in **scripts/test.py**.