CS3211 Assignment 2

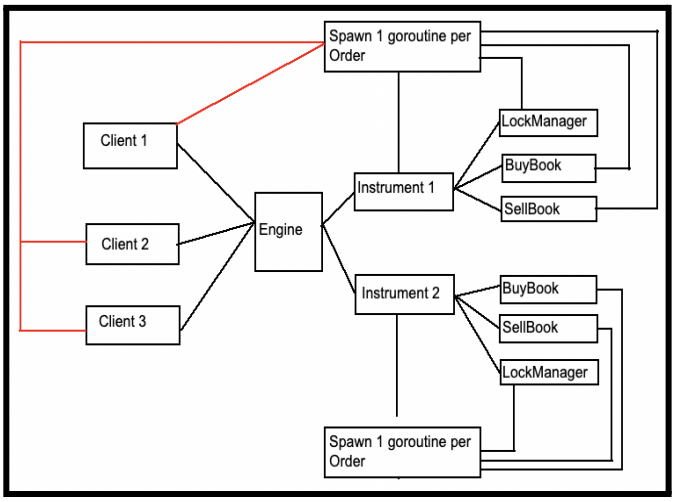
- Albert Sutiono (A0219773U)
- Daffa Fathani Adila (A0219822A)

How Channels and Goroutines enable concurrency

Client Concurrency:

First of all, to support concurrency for clients, **every client will be delegated to a goroutine**. This will allow the matching engine to process multiple clients in parallel.

As data-sharing is not allowed, **every client that has an order will send a request to the Engine** (a goroutine handler worker) through a channel. T**his request will be processed and passed to the appropriate Instrument object**.



Next, in the instrument struct, **each of the received** (by the instrument) **request will spawn an order goroutine** that will communicate with the instrument's lockmanager, buybook, and sellbook handler to manage its own schedule**.**

However, the client needs to wait until the order has been fulfilled completely before it can send the next Request. To support this constraint, **each client has a channel called clientChan**. The client will block until this channel is filled, and **once the order has been fulfilled, the channel will be filled**, thus unblocking the client to send the next order.

Orders-execution Concurrency:

Within every Instrument, we allow **several orders to be executed in parallel**, and **each instrument will have its' own buy and sell book handler**. To provide mutual exclusion when each goroutine accesses the buy/sell order book, **the buy and sell book** (Represented as a linked list or order) **has its' own goroutine handler** that **only allows communication through a channel**. Hence, if a goroutine needs to access/modify the buy/sell book, it will need to send a message through a channel to the handler instead of the usual synchronized read/write through mutexes. This way, **there will be no data races when multiple goroutines try to access/modify the linked list**.

Explanation of how to support the concurrent execution of orders

Phase Order: (Instrument-level + some concurrency among orders of same type)

Instrument level:

we deliberately designed our structs such that **Orderbook holds multiple channels to Instrument handler goroutine** and the logic to process each process is inside the Instrument handler. As explained above, **each order request** (from a client) **will be sent to its Instrument channel** (as an Order message that is passed to the Instrument's channel) to be processed concurrently. **Note that sending orders to the instrument is already synchronized as we use channels for this purpose.**

Note: Cancel Order for each instrument must be run exclusively.
Important: Logic for Buy and Sell orders:

The more complicated part is to **enable concurrency among orders of the same type (some BUYs or some SELLs orders) to be processed concurrently**. WLOG, let's take multiple BUY orders. Notice that an **active order that can be matched** (mainly deals with popping sellSet) **can be processed together with active orders that cannot be matched** (only deals with inserting to buySet). Essentially, every buy order will either wait to be matched, or directly added into the orderbook. Every incoming buy order will peek the best-sell order by sending a message to the Sell Book handler. If it can match the best Sell, then it will call the executeBuy function.

Synchronization and correctness:

To ensure exclusivity when executing the orders, we can synchronize this using a channel that acts like a mutex as such:

1. Create a channel with a buffer size of 1, and initialize it with a message
2. To enter the critical section (i.e. executing the order), it will get an element from the channel. If the channel is empty, it blocks.
3. To exit from a critical section (finish executing the order), it will add an element to the channel, allowing the goroutines that has waited to enter the critical section

Atomicity when concurrently accessing/inserting elements is guaranteed as we use channel to communicate with the handler that serializes the request. However, **most of the processing logic can be run in parallel** (matching orders and inserting orders to the order book).

For correctness, **BUY orders and SELL orders must run exclusively from each other.** However, **multiple BUY orders must be able to run concurrently.** Therefore, before processing the order, we have to check whether it **matches the current order type that is executed/processed. (Heavily use the LockManager)** Otherwise, wait until there is no more order that is processed.

Explanation of Go Patterns used

As seen in the diagram above, we first used the **fan in pattern (multiple clients and 1 engine)**. Then, we will use a **fan out pattern to distribute the order** to its designated instrument channel **(1 engine multiple instrument).**

To **synchronize between the multiple consecutive orders of same type** (multiple consecutive buys / sells / cancels), we use a **lockmanager which api's goal is to manage which order can come in and be processed** and **which order should wait for its turn**. As everything is **done using goroutine and message passing**, we are inspired by the **patterns that microservice systems** use.

For **data structures, buyBook and sellBook is just implemented as a linked list of orders**. To get the bestBuyOrder or bestSellOrder, the linked list must be traversed as it is not sorted in any way.

Testing Methodology

We wrote a script to generate test cases by specifying the number of threads, the number of buy, sell, and cancel orders, and the number of different instruments. We tested the program using various combinations of the test case (1 - 40 threads, 100 - 100,000 buys, sells, and cancels, 1 - 10000 instruments) and used the grader to verify the correctness. The script can be found under the /scripts directory.