CS3211 Assignment 2 Report

Tan De Shao (A0218351L)
Alvin Tay Ming Hwee (A0218390E)

Description of data structures used:

***chanMap*:** is a map of string to Pair. The string (key) represents the instrument and the Pair (value) is a pair of channels that communicates with the buy goroutine and sell goroutine respectively. For each instrument, there will be two goroutines, one goroutine for buy and one goroutine for sell. *chanMap* is used to get the channels used to send Orders to the respective buy goroutine or sell goroutine of a particular instrument.

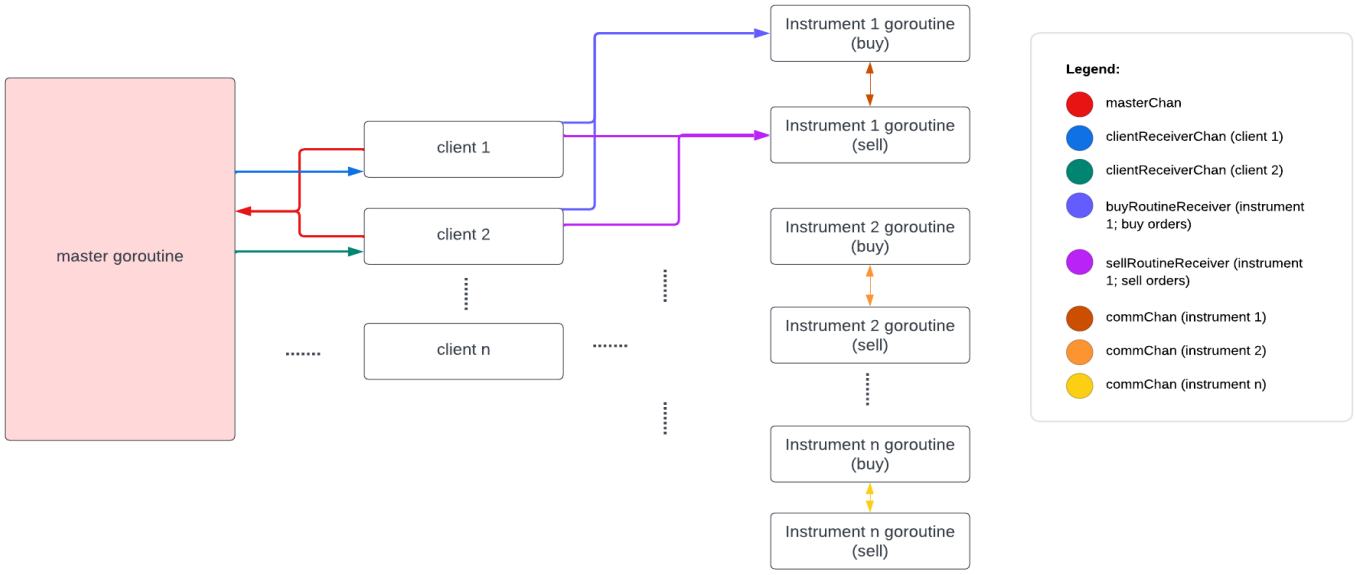
***ChannelMap*:** is a struct that contains *chanMap*. *ChannelMap* has an addInstrument method which is used to add instruments that do not exist in *chanMap* and it also creates one buy goroutine and one sell goroutine for this instrument. This instrument will then be inserted into *chanMap*, with the key being the new instrument and the value is a pair of channels that communicates with the buy goroutine and sell goroutine respectively.

***idToOrderMap*:** For each client, there is an *idToOrderMap* that is used for canceling orders. As cancel order only comes with an orderId, the *idToOrderMap* is used to get the instrument and type of the orderId, which is required to identify which instrument buy or sell goroutine to send the cancel order to.

***book*:** For each instrument, there is one buy goroutine and one sell goroutine. For each of the goroutines, there is a *book*, which is a PriorityQueue containing buy or sell orders. We implemented PriorityQueue, which implements the heap interface. PriorityQueue was chosen as methods such as Pop and Push have an O(logn) time complexity and allows the retrieval of the order with the highest priority in O(logn) time. However, the downside is that for deletion of an order in the PriorityQueue, it is O(n) as it needs to iterate through the PriorityQueue to find and delete the order.

Usage of channels and goroutines enables the concurrency (Phase-level concurrency second kind):

Explanation about how your usage of channels and goroutines enables phase-level concurrency:



Goroutines:

- master goroutine: We created a “master goroutine” that ensures that *chanMap* is only read and written by a single goroutine.

- client goroutines: Goroutines that are created to handle concurrent client connections to the engine. Each client has their own respective client goroutine.

- Instrument buy and sell goroutines: Whenever a new instrument is encountered, i.e., not in *chanMap*, the master goroutine calls the addInstrument method of *ChannelMap*. For this instrument, the addInstrument method will create one buy goroutine and one sell goroutine, each processing buy orders and sell orders for the same instrument respectively.

Channels:

- *masterChan*: Whenever a client wants to send an order to the buy/sell goroutine of an instrument, it sends a Pair of order and *clientReciverChan* to the master goroutine through the *masterChan*.

- *clientReciverChan*: After which, the master goroutine would access *chanMap* and send the *buyRoutineReceiver* and *sellRoutineReceiver* channels back to the client based on the order’s instrument via the *clientReciverChan*. *clientReciverChan* is a higher order channel of a pair of *buyRoutineReceiver* and *sellRoutineReceiver* channels. Only one channel from the pair will be used by the client based on the order’s type (i.e buy or sell).
- *buyRoutineReceiver* and *sellRoutineReceiver*: When the instrument buy and sell goroutines are created, the *buyRoutineReceiver* and *sellRoutineReceiver* are created as well. This allows clients to send orders to an instrument buy goroutine or sell goroutine via *buyRoutineReceiver* or *sellRoutineReceiver* respectively.

- *commChan*: one *commChan* is created along with the buy goroutine and sell goroutine of each instrument. For context, the buy goroutine contains the *book* of sell orders for an instrument, and the sell goroutine contains the *book* of buy orders for the same instrument. The *commChan* facilitates communication between the instrument buy and sell goroutine and enables them to exchange orders that will be added into the respective *book*. In the select statements, the instrument buy and sell goroutine will try to read from *commChan* to add the order into the *book*.

Note that all the channels are unbuffered channels.
Note that for each order, there is a *clientDone* channel to make execution sequential for each client. This is because the phase-level concurrency also applies to each client, which means that the commands for a client might be reordered as for each instrument, one buy and

one sell can execute concurrently. Within each client, it will wait until the order is finished processing by reading the *clientDone* channel, before processing the next order. When an order is completed, it will close its *clientDone* channel.

Explanation of how you support the concurrent execution of orders coming from multiple parallel clients. specify which goroutine it is and write the name of the channels (Phase-level concurrency second kind):
Our implementation allows orders of different instruments of opposing sides to execute concurrently, i.e. one buy and one sell, that are coming from multiple parallel clients, as for each instrument, there are two goroutines, one for buy and one for sell, achieving the phase level concurrency.

When a client processes an order, the client calls the getRoutineAndSubmit method to get the *buyRoutineReceiver*/*sellRoutineReceiver* channel from the master goroutine. The client will then send the order to the respective instrument buy/sell goroutine via the channel for processing (match/cancel). This implementation allows the handling of multiple clients, even when there is a situation where orders for the same instrument of the same side are being submitted by multiple clients concurrently as each instrument buy/sell goroutine will only process one order at a time, while the other clients will wait to write into the unbuffered *buyRoutineReceiver*/*sellRoutineReceiver* channel.

Since both buy and sell goroutines of the same instrument can run concurrently, there will be a situation where both of them will add their orders into their respective book concurrently. This would give an invalid result as one of the orders will be added earlier than the other based on their timestamp. The order that is added later should try to match with the earlier added order, instead of adding directly into the book as it could potentially match with the earlier added order.

We have solved this issue using the unbuffered *commChan* (refer to *commChan* explanation above for the context of it). For each instrument, to add to the book in the opposite type goroutine, the goroutine has to write to the *commChan* first, to send the order over. This is a short code snippet (engine.go line 195-210 and 216-230):

```
select {
case commChan <- o:
    return
case orderToAdd := <- commChan:
    timestamp := GetCurrentTimestamp()
    orderToAdd.timestamp = timestamp
    heap.Push(book, &Item{value: *orderToAdd})
    in := input{orderType: orderToAdd.orderType, orderId: orderToAdd.id,
        price: orderToAdd.price, count: orderToAdd.count, instrument: orderToAdd.instrument}
    outputOrderAdded(in, timestamp)
    close(orderToAdd.clientDone)
    match(o, book, commChan)
    return
}
```

Since *commChan* is an unbuffered channel, only **one** case will be possible, the goroutine can either write the order to the *commChan*, or read the orderToAdd from the *commChan*. Note that *commChan* is also read from other places in the code (engine.go line 159 and 177), to read the orderToAdd and process the adding of order to its respective *book*, this prevents deadlock scenarios.

For the first case, the goroutine will try to write the order into *commChan*, to add the order into its respective *book*. If it is able to write, it means that the opposite type go routine is reading from *commChan*, which means that the opposite type go routine is currently not adding an order.

For the second case, if the goroutine can read an orderToAdd from the *commChan*, it will then add the orderToAdd to its *book* and **match again** (similar to CAS pattern), as the goroutine's current order could potentially match with the newly added order.

This implementation solves the situation where both buy goroutine and sell goroutine of the same instrument are going to add their orders into their respective book concurrently.

Description of any Go patterns used in your implementation.
Lexical Confinement: Lexical confinement was used for every goroutine that is initialized with a data structure. For example, The priority queue was initialized within each buy/sell goroutine and is only read and written within that goroutine. As such, if there are other goroutines that would like to mutate the priority queue, they would have to communicate with the buy/sell goroutine through a channel to accomplish that. This approach ensured that the priority queue was confined to the scope of the goroutine and prevented any race conditions that could occur from concurrent access by multiple goroutines. **For-Select Loop:** This loop allows the program to listen to multiple channels simultaneously and execute a block of code when a value is received from any of them. **Higher Order Channels:** Higher order channels were used to enable the dynamic creation and initialization of the goroutines responsible for matching the buy/sell orders for each instrument. The Orderbook struct was initialized with a map of instrument names to a pair of channels, each of which is listened by a buy or sell channel respectively. This approach allows for the creation of channels and go routines at runtime, based on the instrument being traded, thereby enabling the concurrent execution of orders from multiple instruments. **Preventing Goroutine Leaks with Done Channel:** The done channel was used to prevent goroutine leaks. When a go routine is created, it is given a context which has a done channel that is used to signal to the go routine to exit gracefully when the program is terminated. Moreover, the done channel is used to block the processing of orders at the client level, ensuring that the orders processed within each client are sequential.

Explanation of testing methodology:
We created various test cases with different scenarios to test our implementation, including adding orders to the Orderbook, matching orders, deleting non-existent orders, and concurrently matching orders between multiple threads. We utilized scripts and test case files that can be found in the submission's scripts folder. To access the testing/debugging scripts, please navigate to the scripts folder first. We reused tests from the previous assignment, which can handle up to 40 threads, and around 30 of these tests were repurposed from assignment 1 (located in ~/scripts/customTests). To simplify the testing process, we created a bash script called customTest.sh that concurrently executes all the test cases in the customTests folder. To ensure that we tested different scenarios, we created a bash script named ~/scripts/create.sh that randomizes the commands for each test case within the customTests folder. For each test case, we generated 10 randomized variants that comply with the project requirements and stored them in the randomizedTests folder. We then used the randomTest.sh script to concurrently run all the tests in the randomizedTests folder. Each test file has a comment on the first line that describes the scenario being tested, making it easier for us to track tested scenarios and identify new ones. For debugging purposes, we used the log library to help us understand the program state at specific program lines, which proved to be an efficient way to eliminate bugs in our program. However, having too many output statements on the console can be overwhelming, so during the debugging process, we saved the output to ~/scripts/test.out and used the grep command in the shell to filter out useful lines that provided us with meaningful information. We have also compiled our code with address and thread sanitizers with the "-race" and "-asan" flag to ensure that there are no data race and memory leaks