# CS3211 Assignment 2 Report

By: Liew Zhao Wei (A0234579R) and Dexter Kwan (A0234688N)

## Concurrency with Channels and Goroutines

For this assignment, we implemented a phase-level concurrency matching algorithm. Specifically, we allow 1 buy and 1 sell order to match concurrently. To support this, we made use of channels and goroutines. There is 1 main goroutine, 1 goroutine per client, and 4 goroutines per instrument. We call the instrument goroutines the daemon goroutine, the buy-side goroutine, the sell-side goroutine, and the reconciliation goroutine. The buy-side goroutine holds resting sell orders for active buy orders to match against, and vice-versa. The reconciliation goroutine is used for synchronization.

The client goroutine reads input and sends it to a main goroutine. The main goroutine then forwards the input to the daemon goroutine of the relevant instrument. The daemon goroutine uses instrument-specific channels to forward the order to the relevant worker goroutines. It use 2 channels to send and receive orders from the order book to the instrument's main goroutine. Within the instrument, we further divide the tasks. We have 8 channels per instrument:

- 2 for sending active orders to a side for matching (1 channel for each side)
- 2 for sending orders to a side for storing in the resting order book (1 channel for each side)
- 2 for sending requests to a side for cancelling an order (1 channel for each side)
- 1 to signal to the reconciliation goroutine to cancel an order. The reconciliation goroutine triages the request and forwards the message to the correct side.
- 1 to signal to the reconciliation goroutine to add an order to a resting order book. This is used to synchronise both sell-side and buy-side goroutines when an order is to be added to the resting order books.

By having 1 goroutine for each side, we allow for active buy and sell orders to match concurrently with one another, and only send to the reconciliation goroutine when synchronisation is necessary.

## Explanation and Level of Concurrency of Our Implementation

We observed that when matching with resting orders, the buy and sell orders can execute concurrently without interference. However, there are a few situations that we need to synchronise, namely when one side is trying to insert or cancel an order.

Here's our solution. Suppose we are executing a buy order (sell-side follows the same logic, but with the sides flipped). First note that only 1 buy and 1 sell order can be executed at any point in time. We first send the buy order to the reconciliation goroutine, indicating that we want to insert it into the resting order located in the sell-side goroutine. The reconciliation goroutine contains a flag indicating the last received side order. Suppose this flag is set to buy-side. The buy order will then be forwarded to the sell-side goroutine. If a sell-side is received next, the flag will be on the opposite side, in which case we send the order back to where it came from, and update the flag to sell-side. The sell order will then be matched with the newly inserted buy order on the sell-side goroutine. If however, it is not able to be matched because of price or quantity differences, the order will be sent back to the reconciliation goroutine and forwarded to the buy side as the flag is now on the sell-side.

To handle cancel orders, we keep track of all added order IDs along with their order types (buy or sell). This is maintained in the instrument's reconciliation goroutine, which tells us which side to forward the cancel order to. We send the cancel order to the reconciliation goroutine, which directs the cancel order to the appropriate side.

In the sell/buy-side goroutines, we use for-select to randomly process orders from one of the available channels: the channel for matching an active order against resting orders (the "match" channel), the channel for adding resting orders (the "add" channel), and the channel for cancelling orders (the "cancel" channel). In the reconciliation goroutine, we also use for-select to randomly process requests from one of the available channels: the channel for signalling to add resting orders (the "add" channel) and the channel for signalling to cancel orders (the "cancel" channel)

## Go Patterns and Data Structures We Used

We have 2 main structs: OrderBook and Instrument. An OrderBook contains multiple Instruments, and an Instrument contains a buy, sell, and reconciliation goroutine. We also maintain a map in the instrument's main goroutine to keep track of the IDs of resting order and their order type to allow cancellation of orders. In each buy and sell goroutine, we have a priority queue to store the opposing side's resting orders. For synchronisation, we have a flag in the reconciliation goroutine that tracks which side last went through, which helps synchronisation of active orders from opposite sides, as explained above.

## Testing Methodology

We ran the program with the grader, using the sample test cases and some other generated multi-client test cases in the `scripts/` folder. We also checked for various scenarios to ensure no deadlock occurs since that is a common issue with channels. Lastly, we verified the correctness of some locking scenarios. Some scenarios include: when one side inserts while the other matches, and when one side cancels while the other matches that same order. Our test cases cover these situations.