CS3211 Assignment 2

David Zhu (E0958755), William (E1496974)

AY24/25 Semester 2

1 Go Features Utilized

In this section we discuss the Go features we utilized in our implementation.

1.1 Channels

Channels are used to communicate between the different goroutines. Our orderbook (load balancer) has an `ordersChan` channel in which the clients can send their orders. There are also multiple `queue` channels, one per worker, and one worker per instrument, to send orders to the workers from the load balancer. Each order also has their own `Processed` channel to signal to the client that the order has been processed. This allows the clients' orders to remain in order.

1.2 Gorountines

We use goroutines to implement the balancing functionality of the load balancer, and the worker functionality of the workers. The load balancer has a goroutine that listens to the `ordersChan` channel and sends the orders to the appropriate worker. The workers have a goroutine that listens to the `queue` channel and processes the orders.

1.3 Go Patterns

- **for-select loop** - We use a for-select loop in the load balancer to listen to the `ordersChan` channel and send the orders to the appropriate worker, and to know when to terminate. Within the worker we also use a for-select loop for the same purpose.
- **Fan-in** - We use a fan-in pattern to collect orders from the client goroutines and send them to the load balancer.
- **Fan-out** - We use a fan-out pattern to send orders from the load balancer to the workers.
- **Load Balancer** - We use a load balancer pattern to distribute the orders to the workers. In our case, least load just means the worker with the same instrument
- **Preventing goroutine leaks** We use context with cancel to signal to the goroutines when to terminate. To ensure that the goroutines has finished terminating before the engine shuts down, we use wait groups to wait for goroutines to terminate.

2 Data Structures

- **Order** - a struct that represents an order. Nothing fancy.

- **Worker** - a struct that represents a worker. Only responsible for one instrument. Has two sorted lists for buy and sell orders.
- **OrderBook** - a struct that represents the load balancer. Has a map of workers, one per instrument. Also has a map of instruments, one per order id.
- **SortedList** - encapsulates a slice, adds sorted insertion functionality. Used for priority queues of orders in Worker.

3 Concurrency Level

The concurrency level of our implementation is instrument level. As we have one worker/goroutine per instrument, we can process orders for different instruments concurrently. Instrument-level concurrency is not apparent when only dealing with one client, because we want to prevent reordering of the client's orders. However, when there are multiple clients, we can process orders for different instruments concurrently. While all orders have to pass through the load balancer one-by-one, this isn't a bottleneck because the load balancer is very lightweight and the workers do the heavy lifting. It's like entering The Base salon in Clementi, there's only one door to enter the salon, but there are multiple chairs and hair stylists inside. Customers (Orders) can only enter the salon (load balancer) one at a time, but once they're inside, they can be served concurrently by different hair stylists (workers).

4 Testing Methodology

For testing of the engine, we re-used our python script that generates random input files for the grader and we run it continuously check for any test failures. The generate input files of different sizes, from 4 to 40 clients and up to 40000 orders. It is ran for > 100 iterations for large cases and > 1000 iterations for medium size cases to ensure the engine is likely to be correct. The python script is in `scripts` directory. We also ran our engine the go data race detector to check for data races.