Assignment 2 report

1. Data structures

Data structure	Description	Purpose
Map[string]*Instrument	Maps instrument string to the (reference of the) instrument object	Allows the centralized goroutine to retrieve appropriate instrument objects for clients' requests.
MinHeap, MaxHeap	Heaps of Order objects. Stores resting sell orders and buy orders respectively.	Allows active order to match with the best price.
Map[uint32]*Order	Maps order's id to the Order object in the heap.	Allows the instrument goroutines to quickly find the order in the heap and cancel it (using a flag).
InstrumentRequest	Custom struct that contains an input and an unbuffered reply channel.	Clients send an InstrumentRequest to the centralized goroutine through a channel to query for relevant instrument objects. The central goroutine replies to each client through their respective reply channel.
OrderRequest	Custom struct that contains the input (order) and an unbuffered done channel.	Clients send OrderRequest to respective Instruments through relevant channels. Once an instrument goroutine finishes processing the input, it signals completion to the client by closing the done chan.
Instrument	Custom struct containing a buy channel, a sell channel, and 2 cancel channels.	Client communicates with one of the 2 goroutines managing the instrument through the relevant channel.

2. Testing

We wrote a Python script (`./scripts/make_tests.py`) to generate test cases with configurable parameters. To test for deadlocks under high contention, we use fewer instruments (10) and a narrow price range (100–200) with high client counts (32, 64). This forces client goroutines to compete for the same instrument's priority queues and increases the likelihood of matching opposite orders arriving concurrently. For each client count, we generate 100 test cases to enhance robustness. A test script (`./scripts/run_tests.py`) then runs the grader on all cases, automatically detecting errors or goroutine leaks. No errors, timeouts or leaks strongly indicate correct execution.

3. Goroutine and Channel Usage

Here are the lists of goroutines and channels that we used to enable concurrency:

Goroutines:

- The `coordinator` goroutine is responsible for handling instrument requests from clients via the `coordinatorChan` channel. We use a single centralized Goroutine so that only one goroutine can read and modify the instrument map, thus preventing race conditions. It initializes a new Instrument whenever it sees a new instrument from the client.
- Each new client connection (`Accept` method) spawns two goroutines, one to handle the connection itself (`handleConn`), the other to clean up the connection when the context is done. `handleConn` continuously reads input from the connection, then sends a request to the coordinator via `coordinatorChan` for the relevant instrument, and sends an order request via relevant channels stored in the instrument object.
- Each `Instrument` object spawns two additional goroutines to manage buy and sell orders concurrently (via `buyInstrumentManager` and `sellInstrumentManager`). Each goroutine has its own heap and has exclusive access to it, preventing race conditions. Each goroutine has two channels to read order requests (buy/sell/cancel) from multiple clients and execute them. Two goroutines of the same instrument communicate via one shared unbuffered channel `addChan`, which allows them to send partially matched active orders to be added to respective heaps on the other side.

Channels:

- `coordinatorChan`: This channel is used by each client handler (`handleConn`) to request the central coordinator for the instrument corresponding to their input.
- `buyChan`, `sellChan`, `cancelBuyChan`, `cancelSellChan`: These channels are used by each client to send their order request to the goroutine that processes the buy/sell side of the instrument.
- `doneChan`: Each order request has an associated `doneChan` that ensures the client waits for the order processing to complete before sending a new request. This avoids simultaneous processing of multiple orders from the same client.
- `addChan`: This channel is used between `buyInstrumentManager` and `sellInstrumentManage` of the same instrument to handle adding orders into the resting order heap.

4. Concurrency Explanation

We achieve phase-level concurrency by allowing opposite orders from the same instrument to be processed concurrently while processing orders from different clients at the same time.

Managing Concurrency Between Different Clients:

Each client goroutine sends their order request to the instrument goroutine independently via their respective channels. Hence, orders from different clients may be processed concurrently.

Since each instrument spawns 2 goroutines to manage the resting order heaps, these instrument goroutines can process orders from clients independently. This achieves instrument-level concurrency.

Managing Phase-Level Concurrency (Buy and Sell Orders for the Same Instrument):

The `Instrument` struct spawns 2 separate goroutines for managing buy and sell orders concurrently, using `buyInstrumentManager` and `sellInstrumentManager` for each order type. This enables a buy and sell order for the same instrument to be processed in parallel without waiting for one to finish before the other starts. Cancel orders are also handled separately by each `buyInstrumentManager` and `sellInstrumentManager` depending on whether the order being cancelled is a Buy or Sell order. Essentially, the 2 goroutines of each instrument can read and modify the order heaps independently, allowing for phase level concurrency.

Note that since `buyInstrumentManager` and `sellInstrumentManager` write and read to the same `addChan` channel, there can be deadlock. Hence, we use a **select block** to read or write to `addChan` after the active order is partially matched and ready to be sent to be added. This guarantees **one goroutine sends and one goroutine reads** from `addChan` at any time, preventing deadlock when both read or both write. In addition, when an opposite order is read from `addChan` and added to the heap, the partially matched order has to **perform matching again** in case its price matches with the newly added order.

5. Shutdown Handling

Context-based Cancellation: goroutines are managed with the context package, which ensures proper cancellation of running goroutines when the context is done (`<-ctx.Done()`)). Each goroutine checks for context cancellation within select statements where they read or write to a channel, ensuring that goroutines will never get blocked after the context is cancelled. This prevents goroutine leaks.

Graceful Shutdown: The `Engine` uses a `WaitGroup` to wait for all goroutines to finish before completely shutting down, ensuring that no goroutines are left hanging. Whenever we spawn a new goroutine, we increment the WaitGroup and only decrement when the goroutine terminates.