CS3211

Assignment 2: Exchange matching engine in Go

1) Explanation of our usage of channels and goroutines that enable instrument-level concurrency

We have three types of goroutines in our code: the mainRoutine, client goroutine (handleConn routine), and instrumentRoutine.

We have three channels: the main input channel, the instrument channel, and the done channel for each client.

When the engine is first created in main.go, its constructor is called, which starts the mainRoutine. mainRoutine will keep reading from the main input channel to receive a Wrapper (contains the done channel and the input) from all client goroutines, and then depending on the instrument of each input, send that Wrapper to the instrument-specific channel accordingly. To do this, it has a map of instruments to channels. If the instrument channel and goroutine do not exist, mainRoutine will start an instrument goroutine for that instrument, create an input channel for that specific instrument and add that input channel to its map.

The instrumentRoutines will constantly receive Wrappers from their instrument channel, and upon completion of processing input in the Wrapper, print the output and then signal the done channel of the client. When the client channel receives a signal from its done channel, it will then read another line of input, wrap it in a Wrapper and send it to mainRoutine accordingly.

2) Explanation of how we support the concurrent execution of orders coming from parallel clients

Clients send their doneChannel and input orders (in the Wrapper struct) to the mainRoutine, which then routes the Wrapper to the corresponding instrument goroutines. These instrument goroutines can operate independently of each other, and as such, orders of different instruments can be processed at the same time, and we have instrument-level concurrency.

Cases when instrument-level concurrency does not occur:

When the client first receives an order, and until it sends that order to mainRoutine and mainRoutine is routing all the orders from all clients to their respective instrument goroutines, there is no instrument-level concurrency, as mainRoutine is routing orders one by one. However, this only occurs a minority of the time, and the majority of the time orders of different instruments are being concurrently processed by instrument goroutines.

3) Description of Go patterns and data structures used

One of the main patterns we use in our order book is the implementation of goroutines and channels to manage concurrent tasks, which is a common practice in Go for building concurrent applications. This

pattern is shown in our design when each client passes a `Wrapper` object into the `wrapperChannel` so that the main goroutine can retrieve each client's orders and redirect them to the appropriate instrument goroutine using instrument-specific channels. Each client also passes a `doneChannel` to the main goroutine which is then passed to the appropriate instrument goroutine. We use the done channel to indicate to the client when an order has been processed so that it can send the next one. This ensures sequential execution within every client.

Another Go pattern we employed is the worker pattern, where we dedicate one goroutine to each instrument which then processes orders related to that respective instrument. This pattern serves to distribute the load and enable task parallelism. This also ensures that all instruments are handled separately from each other.

For data structures, we first used three main structs: Order, Wrapper, and Engine. `Order` represents a single order, with details such as order ID, type, and price. `Wrapper` acts as a container for passing new inputs along with a doneChannel. This struct facilitates synchronisation between different parts of the system and allows for a signal to indicate to the client that an order has been processed. `Engine` is the core struct that holds the channel of Wrapper objects, wrapperChannel. It also facilitates the creation of the main goroutine and the routing of orders to the respective instrument goroutines through the use of channels.

We also create two heaps, `BuyOrderHeap` and `SellOrderHeap`, as specialised slices of type uint32 that implement the "container/heap" interface from Go. They function as max-heaps and min-heaps respectively, and serve as a way to quickly access the best current price.

`instrumentToChannelMap` is a map that stores the mapping between every instrument name and its respective channel so that the main goroutine knows which channel to send wrappers to. `orderIdToInputMap` is a way of allowing the system to check for the instrument name of an order in case of a cancel order, as cancel orders only include the order ID and not the instrument name. `buyPriceToQueue` and `sellPriceToQueue` are both maps that store the queue of orders (for chronological ordering) at each price level.

4) Explanation of testing methodology

We generated multiple test cases with multiple threads and ran them (Upwards of 100 threads and 4 million orders). We also ran the same test cases multiple times with a bash script. We also ran "go vet" on our engine.go file to find any potential issues.