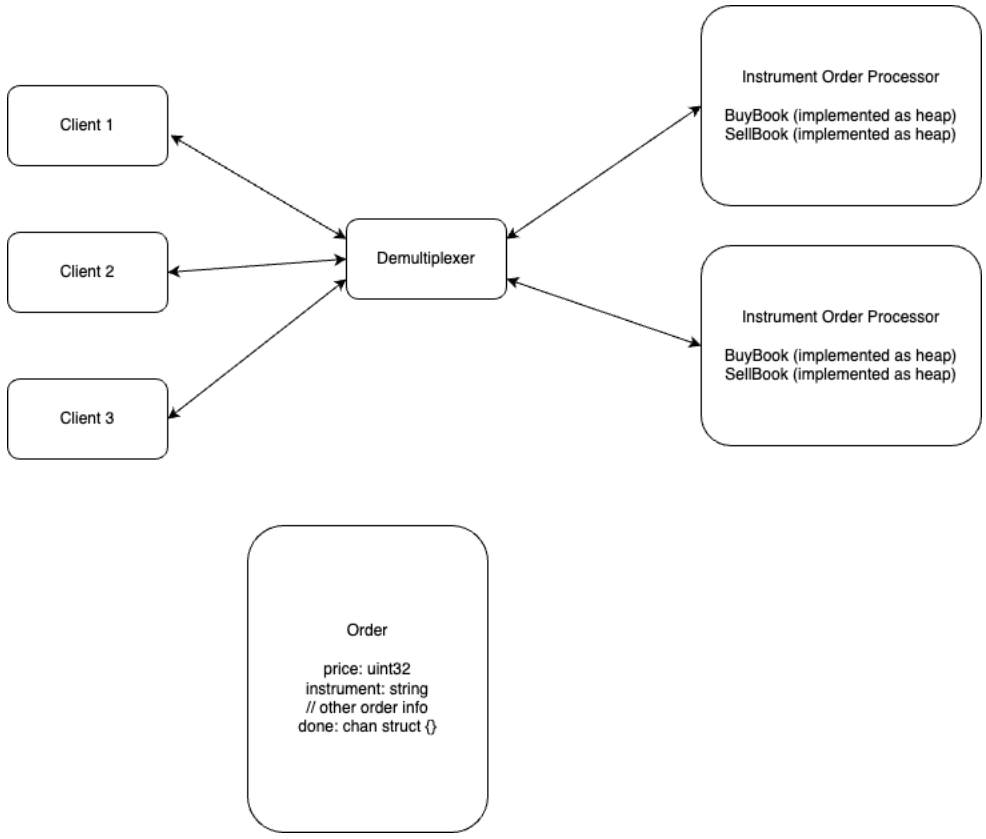
Ng Choon Siong (A0196600H)
CS3211 Assignment 2 Report

High-level architecture

The exchange matching engine consists of 1 order processor (an individual goroutine) per instrument to handle all buy and sell orders related to that instrument, and a demultiplexer that acts as a central order router to route orders to the appropriate order processor.



Data flow

When a client sends a order, it is sent via a single channel shared with all clients to a demultiplexer that acts as a router. The client adds into this a done channel and blocks while waiting afterwards to receive on this channel. The demultiplexer stores a map of dedicated channels to different order processor goroutines for each instrument, and forwards received orders accordingly.

At each order processor, the goroutine will receive the order, and perform matching. It stores two separate order books stored as a heap/priority queue (ordered by price-time priority), one for buy orders, and one for sell orders. When a buy order is received, the order processor tries to match against the sell order book, and vice versa. If there is a partial match, it executes until the order is fully executed, or the order book is empty. An order is added to its order book when the opposing order book is empty. After adding to the order book, the processor signals on the done channel that processing is complete.

If the order is a cancellation order, the demultiplexer first tries to retrieve the relevant instrument name and rejects the cancellation if it cannot find the instrument. It stores a map of order ids to instrument name to do this. After forwarding the cancel order to the relevant processor, the processor then tries to cancel the order. To do this, it stores a local map of each

order id to order struct pointers. If it can find the order id inside this map, it sets the count of the order struct to 0, and deletes the order id from this map. This is because deleting directly from the heap causes problems, and so the cancelled order will live on the heap with a count of 0 until it is removed and discarded during the order matching process.

Concurrency and Synchronization

Level of concurrency: Instrument-level concurrency

- Each client connection is handled in its own goroutine. This allows multiple clients to concurrently submit orders, which are sent to the demultiplexer.
- The demultiplexer runs in its own goroutine, receiving orders from all the client goroutines. It looks up or creates the channel for the order's instrument and routes the order to the appropriate channel.
- Each instrument has its own independent order processor goroutine. This allows orders for different instruments to be processed concurrently. The processor receives orders on the instrument's channel and applies them to the buy and sell books.
- Within the order processor, no explicit locking is needed for the orderMapping or the buy and sell books as there is only a single goroutine to handle orders of a single instrument. When an order is matched or cancelled, the processor sends on the order's done channel, signalling that processing is complete for that order and the client can send more orders.

Data Structures Used

For the demultiplexer

- A map (orderInstrumentMap) from orderID to instrument name.
- A map (instrumentRouters) from instrument name to the channel leading to the relevant order processor
- A channel (clientOrdersChan) to receive orders that need to be routed

For the order processor (instrument go-routine)

- Priority queues (implemented as min/max heaps) for the buy and sell order books. The BuyBook and SellBook types implement the heap interface from the container/heap package with the appropriate price-time comparator
- A map (orderMapping) from order ID to Order pointer, Channels are used extensively for communication between goroutines.

Go Patterns

Fan-in-fan-out: The demultiplexer fans in orders from different client channel, and fans out the orders to the appropriate client channel.

Testing

A python script (gen_test.py) was used to generate test cases with the following parameters: number of threads, total number of orders, total number of instruments. It first opened all threads specified, and synchronized them. Then, it generated a list of random instruments. Afterwards, it selected a command randomly with equal probability from a buy order, a sell order, or cancel order. For buy/sell orders, it chose a thread and instrument, price, and count at random, and outputted the order. For cancel commands, it selected a previously created order at random, and cancelled it.

Test cases up to 40 threads, 100,000 orders and 50 instruments were tested to confirm that the engine works.