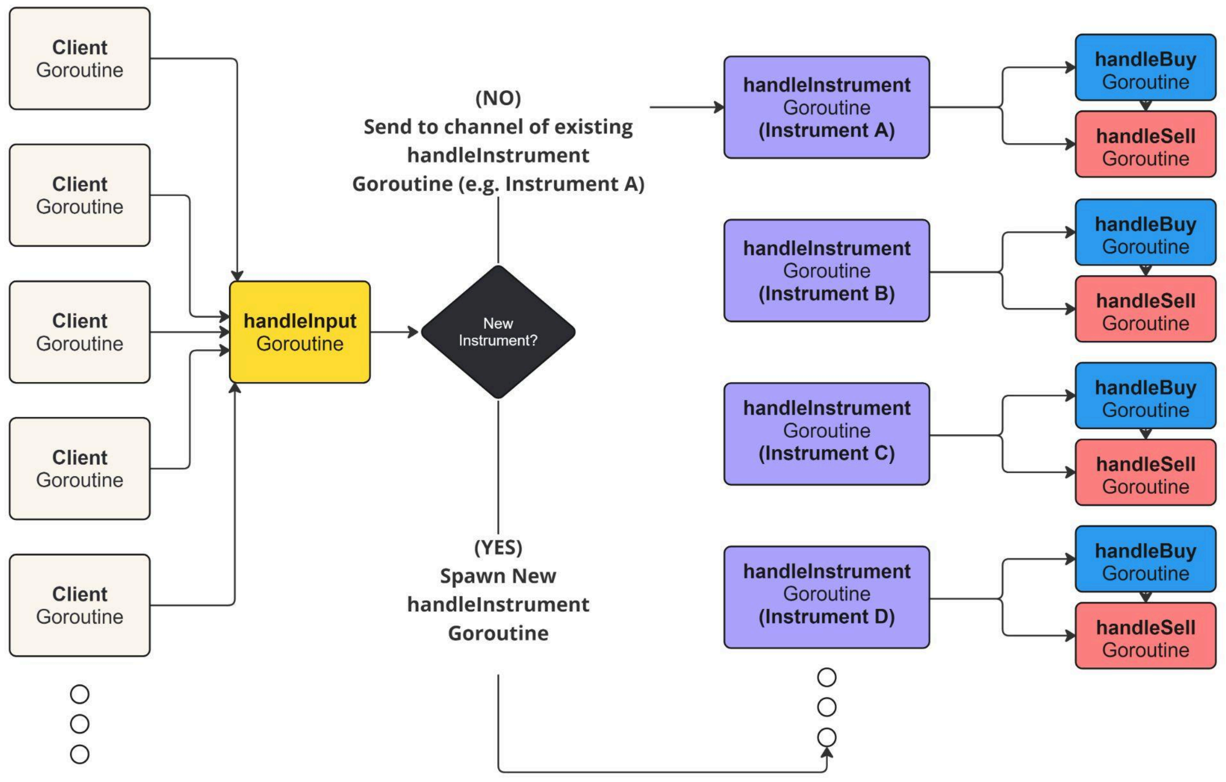
Team: e0774645_e1324860

Description of Program Architecture


Client Goroutine
Client Goroutine
Client Goroutine
Client Goroutine
Client Goroutine
handleInput Goroutine
New Instrument?
(NO) Send to channel of existing handleInstrument Goroutine (e.g. Instrument A)
(YES) Spawn New handleInstrument Goroutine
handleInstrument Goroutine (Instrument A)
handleInstrument Goroutine (Instrument B)
handleInstrument Goroutine (Instrument C)
handleInstrument Goroutine (Instrument D)
handleBuy Goroutine
handleSell Goroutine
handleBuy Goroutine
handleSell Goroutine
handleBuy Goroutine
handleSell Goroutine
handleBuy Goroutine
handleSell Goroutine

<u>handleInput Goroutine</u>:

The **handleInput** goroutine is first created in the main function. When a client reads an input, that input is sent through a channel to the **handleInput** goroutine. The **handleInput** goroutine is responsible for routing that order to its corresponding **handleInstrument** goroutine via its instrumentChannel.

When an instrument is first encountered by **handleInput**, its **handleInstrument** goroutine and its instrumentChannel will be created. A hashmap is used to map each instrument to its corresponding instrumentChannel.

An additional hashmap that maps orderId to its instrument is maintained for cancel orders.

<u>handleInstrument Goroutine</u>:

The **handleInstrument** goroutine receives orders from the instrumentChannel. When the **handleInstrument** goroutine is spawned, it also spawns two other goroutines (**handleSell** and **handleBuy**), as well as channels to communicate with them.

The **handleInstrument** goroutine is responsible for routing sell orders to **handleSell** via its sellChannel and buy orders to **handleBuy** via its buyChannel**.**

An additional hashmap that maps orderId to order is maintained to query for cancel orders.

Team: e0774645_e1324860

handleBuy and handleSell Goroutine:
Each goroutine contains a priority queue that keeps track of resting orders, sorted according to price time. When a new order is received from sellChannel or buyChannel, it will attempt to execute these orders. Priority queue implementation is referenced from golang documentation.

Concurrency

The level of concurrency in this program is **phase-level concurrency**.
Orders for different instruments can be executed concurrently as orders for each instrument are handled by its own instrument goroutine, achieving **instrument-level concurrency**.
Some orders for the same instrument are also able to be executed concurrently. In our implementation, a sell order and a buy order for an instrument is handled by its own goroutine and they can be executed concurrently under the right conditions. When a sell order and a buy order of the same instrument is received by its corresponding goroutine, there is a possibility that these two orders are matching (i.e buy order has price more than or equal to sell order). If they are matching, then they will not be processed concurrently, as this could lead to the possibility of both orders being added to their order books without matching.
However, in the case where the two orders are not matching, they can be processed independently as their execution will not affect each other. In this scenario, our program executes both buy and sell orders concurrently and achieves **phase-level concurrency.**

Patterns Used

For Select: In the **handleInput**, **handleInstrument** and **handleBuy/handleSell** goroutines, for select loop is used to read input from channels and depending on the input type, we perform different behaviours.
Confinement: Between any two goroutines, there is no sharing of any data structure, the communication is done solely via channels.
Fan out: Fan out pattern is used here to distribute work across multiple goroutines, as seen in **handleInput** distributing orders to their corresponding **handleInstrument** goroutine.

Testing Methodology

To test the program, we used a combination of the provided test cases and also generated new test cases with a script. The script generates random test cases with given parameters: number of connections, instruments, orders. Generating tests with different parameters allowed us to test our implementation more comprehensively, testing its correctness with a large and varying number of connections, instruments or orders.

In addition, we made use of the Go Race Detector (-race) to check for race conditions.