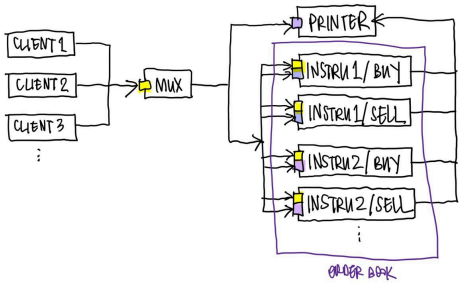
CS3211 Assignment 2 Report

Data Structures

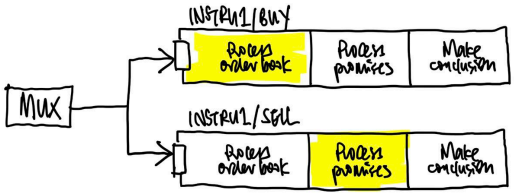


Our order book is composed of individual instrument/side order books, stored locally in multiple worker goroutines and represented each by a slice of Order pointers. Without a singleton order book, the fan-in/fan-out pattern is applied by using a multiplexer to multiplex orders from multiple clients onto multiple workers. Every client feeds its orders into the incomingOrders channel and the multiplexer dispatches the orders to the correct instrument/side worker. There is also a printer that prints out processing results sent by instrument/side workers. Both multiplexer and printer are implemented as goroutines that run from engine initialization to program exit.

To ensure client ordering among instruments and sides, we applied the promise pattern by creating promises along with incoming orders. To achieve a non-blocking consumption of promises, we implemented an unbounded buffered channel - using 2 unbuffered channels and 1 slice. Unbounded buffered channels are indicated in purple in the above diagram, while yellow indicates unbuffered channels.

Phase-Level Concurrency

With the order book separated into goroutines for each instrument/side, orders of different instruments and sides can be processed concurrently. The correctness of concurrently processing orders of different instruments is easily justified, as there is no dependence between different instruments. However, the correctness of processing orders of different sides (same instrument) concurrently is to be considered more carefully to ensure correct client ordering. This is because processing of an order on one side A results in adding of the order on the other side B should there be no match, and this addition may result in a potential match on the other side B. Essentially, since order processing on one side can affect the other side, we ensured both sides communicate to each other through channels to enforce synchronization.



The processing of an order on each instrument/side goroutine is broken down into 3 stages: 1) processing already processed orders, 2) processing orders yet to be processed but are dispatched earlier than the current order, and finally 3) performing the necessary action depending on order type, i.e., cancel order not found, match executed, order added to order book for no match found. Opposing sides' goroutines can process already processed orders concurrently to speed up the overall processing time before waiting for their turn to make the final conclusion to ensure client ordering.

To enforce client ordering among instruments, a work struct promise with an outputCh is sent to the promiseInCh together when the work struct is sent to the inputCh for processing. During printing, the printer goroutine reads promises from promiseOutCh and prints all print_inputs from outputCh until closed. print_inputs are sent by worker goroutines to the corresponding outputCh when processing the orders. Note that as such, the actual timestamp is not required, rather an incremental counter as timestamp is sufficient to represent correct relative time.

To enforce client ordering among sides, a work struct promise with an toAddCh is sent to the promises of both side worker goroutines together when the work struct is sent to the inputCh for processing. toAddCh is a buffered channel of capacity 1, where if it does not contain anything, the work is yet to be processed; if it contains a work struct, it is the order to be added into the order book; if it contains an empty work struct, it means that the work is done processing. Such a check is done during Stage 2 - the worker will wait for its turn until earlier promises are fulfilled, i.e., an empty work struct has been inserted to toAddCh of earlier promises, until it makes its final processing conclusion.

When earlier promises are not fulfilled, the worker will take the promise and process it. Note that any unprocessed promise to be added to the order book on the current side is the result of the opposing side. Equivalently, any unprocessed promise of the current side is the current order, where the goroutine will break out of the promises processing loop and carry out its actual order processing. Such an invariant is guaranteed because consuming orders is blocking.

Also, adding of orders are required to be done in both for-select cases to allow for addition of orders regardless of whether there is an incoming order or not. When an order is sent to the unbuffered toAddThereCh of worker A, it synchronizes with the any two receive, i.e., in each for-select case, of the order at the same underlying channel unbuffered toAddHereCh of worker B.

Testing

Testing effort is similar to A1 where we tested our engine with multiple clients (up to 40 clients).

To stress test the engine's performance with many clients, alternating buy/orders, and many instruments, we used the script from A1 to generate test files with different sequences of commands from 40 different clients and instruments (with 100 instruments as an arbitrary upper limit).