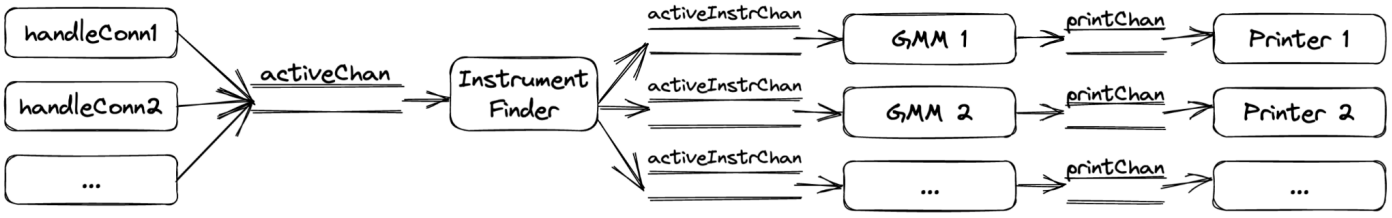
Goroutines and Channels

In order to visualize our explanation better, here is a simple illustration of our Goroutines and Channels.



1. `handleConn`: given Goroutine that handles each client. It sends the input it receives to `InstrumentFinder` through one channel named `activeChan`.
2. `InstrumentFinder`: a daemon Goroutine. Receives the input, constructs an `Order` struct for it and sends it to the corresponding `GenericMatchMaker(GMM)` that handles all orders for the same instrument. To facilitate this, it maintains 2 data structures:
    a. `instrChanMap`: map of `string` to `chan Order`
        i. `string`: input's instrument name
        ii. `chan Order`: the channel for the `GenericMatchMaker` dedicated for this instrument. The channel is named `activeInstrChan` in the diagram and code.
    b. `orderInstrMap`: map of `uint32` to `string`
        i. `uint32`: orderID of the input
        ii. `string`: instrument of the input
        iii. This is to facilitate canceling an order as we need to know its instrument to find which `GenericMatchMaker` to pass it to
3. `GenericMatchMaker`: a Goroutine dedicated for each instrument and is in charge of executing an order (buy, sell and cancel). To facilitate this, it maintains 2 data structures:
    a. `BuyPriorityQueue`: priority queue for buy side `Order`s
    b. `SellPriorityQueue`: priority queue for sell side `Order`s
    c. Both of the above stores resting `Order`s of the same instrument since this Goroutine is dedicated for all orders of 1 instrument. To understand this better, you can imagine `GenericMatchMaker 1` in the diagram above to handle all orders for `GOOG` and `GenericMatchMaker 2` will handle all orders for `AMZN`.
4. `Printer`: a Goroutine just to print things concurrently. Each `GenericMatchMaker` has a dedicated `Printer`. This adds a stage to the pipeline for more concurrency.

Our Goroutines enable concurrency as we have a `handleConn` Goroutine to handle each client concurrently from other clients, a `GenericMatchMaker` Goroutine to handle all orders of each instrument concurrently with orders of other instruments and a `Printer` Goroutine to print all outputs from its respective `GenericMatchMaker` concurrently with other outputs. The dedicated channels we have between these Goroutines allow us to communicate between the Goroutines concurrently.
More explanation about how our Goroutine and channels support the concurrency can be found under the 'Level of Concurrency: Order Level' section.

Go Patterns

Fan-out, fan-in

As shown in our illustration above, we fan-in the inputs from multiple `handleConn` Goroutines to the `InstrumentFinder`. `InstrumentFinder` then finds the channel for the `GenericMatchMaker` that handles orders of this input's instrument using its `instrChanMap` and sends the `Order` to it for it to process. Here, we fan-out the `Order`s from `InstrumentFinder` to the respective `GenericMatchMaker` based on this `Order`'s instrument .

Pipeline

Also shown above, we form a pipeline of 4 stages here.
`handleConn` → `InstrumentFinder` → `GenericMatchMaker` → `Printer`.
Since `InstrumentFinder` stores the 2 maps described earlier and we do not want this to be accessed by anyone else, we cannot have more than 1 `InstrumentFinder`. `InstrumentFinder` mostly does map lookups, and occasionally adds to the map and spins up a new `GMM` if a new instrument is found. This is likely to run much faster than the heap operations in `GMM`. To further improve on this, we made the channels to the `GMM`'s buffer size 100 so that `InstrumentFinder` will likely not be blocked on sending to a `GMM` and can continue processing the next orders.

Level of Concurrency: Order Level

Our implementation achieves order level concurrency through the use of our Goroutines and Go Patterns explained earlier.
We have multiple `handleConn` Goroutines that each handle one of the multiple clients. This achieves concurrency between clients sending orders.
We have multiple `GenericMatchMaker` Goroutines that each handle all orders for one of the multiple instruments that our clients have submitted orders for. This achieves Instrument Level Concurrency as orders of different instruments can be processed at the same time.
Finally, our pipeline achieves Order Level Concurrency because orders of the same instrument can also execute concurrently on the different stages. Example:

1. Order 1 is being printed in `Printer`
2. Order 2 is being executed in `GenericMatchMaker`
3. Order 3 is being handled in `InstrumentFinder`
4. Order 4 is being received in `handleConn`

Explanation Of Testing Methodology

We ran all the provided basic tests as a start.
We built custom_runner.cpp to simulate concurrent clients (uses `std::barrier` to wait for all threads to be created before running tests). Build by running `clang++ -g -O3 -Wall -Wextra -pedantic -Werror -std=c++20 -pthread custom_runner.cpp` in scripts/custom_runner folder.
Run via `./a.out ../../socket <paths to input files>`
Large Testcase (scripts/custom_runner/random_buys_sells_cancels)
We used order_gen_random_buy_sell_cancel.py to generate random orders for 8 clients. 2 instruments, 4 clients for each instrument. The Python script lets us vary the proportion of buy to sell to cancel orders, and lets us vary the price within a range.
We ran the grader format version of the test case (random_bsc_combined.in) and passed the correctness test.
We ran them using custom_runner (random_c0_0.in … random_c7_7000.in).
Data Race
To ensure that no data race occurs, we rebuilt the engine binary with the -race flag and reran all tests to ensure that there were no data races.