Description of Data Structures

Our implementation of the matching engine uses a "pipeline" approach in which information is passed from one stage of the pipeline to the next. Each stage of the pipeline has one or more goRoutines performing operations on this information. In order to prevent data races, we have used channels as the main method of allowing different stages of the pipeline to communicate.

There is also some element of Fan-In / Fan-Out, which can be seen in the picture under **Explanation for Concurrency Support**. Client requests are fanned-in into *ImInputChan* for processing, and then subsequently fanned-out into *IChannelManager* to keep track of outstanding Orders, before finally being fanned-in into *PrintChan* for printing messages (when the Order is completed).

donechannel is an unbuffered channel of type struct{} which is closed when the connection is closed, to signal to all goroutines to stop execution. Each goroutine used implements a for-select loop in order to check whether donechannel is unblocking, and returns if it is.

IMinputchan is an unbuffered channel which feeds inputs from individual connection goroutines into a central manager goroutine *InstrumentManager*.

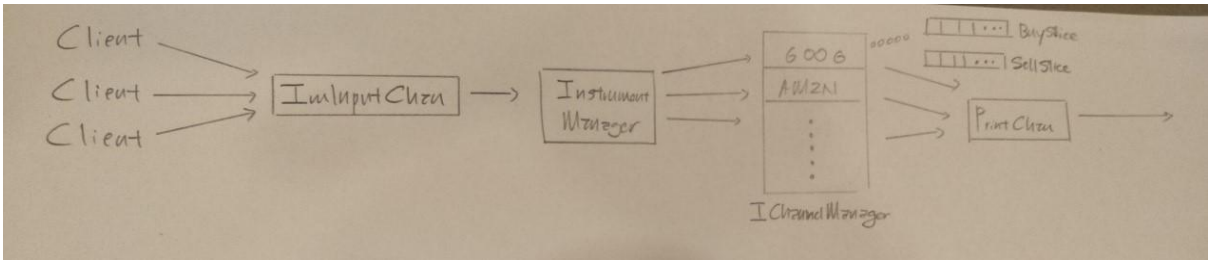
InstrumentManager filters buy and sell orders according to their instrument, then sends each of them to a channel within the channel slice *IChannelSlice* for processing. If a channel doesn't exist for a particular instrument yet, *InstrumentManager* creates it and appends it to both *IChannelSlice* and *IChannelCancelSlice*.

For cancel orders, *InstrumentManager* sends the order to every channel within *IChannelSlice*, then receives results from *IChannelCancelSlice* before processing and printing outputs.

Each individual channel within *IChannelSlice* has one goroutine in charge of it, namely an *IChannelManager* goroutine. Each *IChannelManager* has one slice for buy inputs and one slice for sell inputs, and it updates these slices accordingly when new inputs are received.

For printing to standard output, goroutines send *OutputStructs* to the channel *printchan*, which is managed by a printing goroutine *PrintManager*, which reads each individual output from the channel and prints it to standard output.

Explanation of Concurrency Support



- Clients are able to make orders concurrently, however their orders will sequentially be transmitted through *ImInputChan*.
- *InstrumentManager* sequentially receives and processes the Client's orders from *ImInputChan*, starting a *goRoutine* for each distinct Instrument that it receives.
- Hence, concurrency is achieved between different Instruments.
    - For example, **GOOG** and **AMZN** orders may process in parallel. The *goRoutine* for each of the Instruments is able to concurrently handle the matching of BUY/SELL/CANCEL orders independently.
    - However, BUY/SELL/CANCEL orders within each Instrument will proceed sequentially, according to the sequence that the Client's made their orders. This is to ensure greater accuracy of order processing within instruments.

- For cancel orders, a cancel order with the ID of the order to be cancelled is sent to all *IChannelManagers* by the *InstrumentManager*. The *InstrumentManager* then waits for each *IChannelManager* to finish looking for the order to be cancelled by reading from each available channel within *IChannelCancelSlice*.
    - By processing cancel requests in parallel for all instruments, we are able to cancel orders faster than performing cancellations sequentially.
    - As *IChannelManager* has to wait for all channels to finish finding the ID of the cancel order before proceeding, the processing of individual orders within *IChannelManager* remains sequential.

Description of Testing

1. Test cases (simulating threads) were created and placed into different folders.
2. Scripts were created for each test case to emulate different clients sending out commands on their own threads.

All test cases can be found within the files on GitHub, with test1 referencing files within test1_files and so on. Note that the script files have to be in the correct directory as the engine and client output to work (for submission purposes, we have also consolidated all of them in a *scripts* folder).