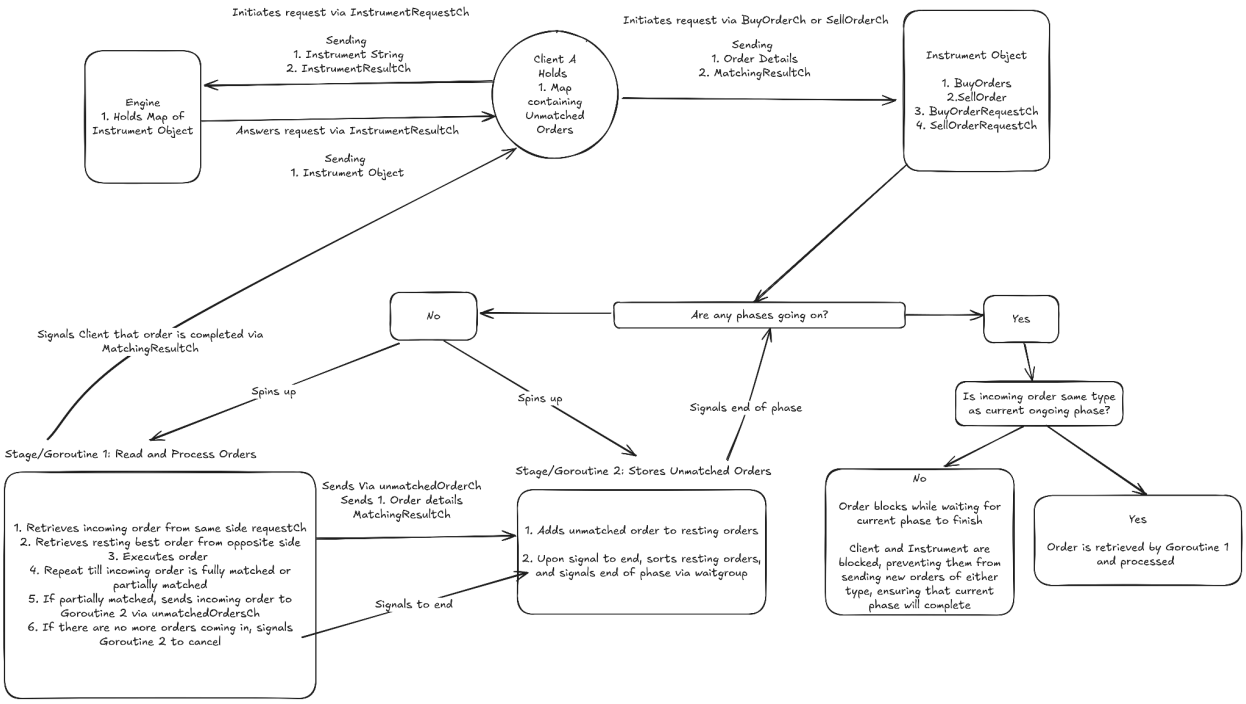

Figure 1: Program Overview

The program consists of 3 main components, the engine, clients and instruments.

Engine

The engine holds onto all instances of instrument objects and runs an instrument-managing goroutine that creates and provides those instrument objects to clients when requested.

Client(s)

The clients are connected to the engine via `handleConn()`.

Each client creates

1. `unmatchedOrders` (map of unmatched orders)
2. `instrumentResultCh`
3. `matchingResultCh`

`unmatchedOrders` is a map of all unmatched orders sent by the client, which helps track data needed for order cancellations. `instrumentResultCh` is used to receive the instrument object from the engine that the client will send their order to. `matchingResultCh` is a done channel used to signal the client that their order has been processed and can continue to the next order.

Instrument(s)

The instrument objects each have two slices for the resting buy orders and sell orders, along with two request channels for buy and sell orders for that instrument. It also runs a main instrument goroutine that will start up order processing goroutines to respond to orders.

The instrument will initiate processing in discrete “phases”. During a phase, all incoming orders of a particular type (buy or sell) are processed in a pipelined manner, until there are none left. Orders of the other type are placed on hold while the current phase is ongoing.

Illustration of Incoming Buy Order, while no phase is active

1. **Order Arrival:** Buy order is received via the `buyOrdersCh` channel.
2. **Phase Select:** Instrument selects the order from the `buyOrdersCh` channel and prepares to enter the buy phase.
3. **Phase Initialization:** A new phase is initiated, during which two goroutines are started.
   - **Goroutine 1:** Reads and matches incoming buy orders from `buyOrdersCh`.
   - **Goroutine 2:** Adds unmatched buy orders to the resting buy orders slice.

Handling Mixed Order Types

Assume that a buy order and a sell order arrive simultaneously during an active “buy phase”, in `buyOrdersCh` and `sellOrdersCh` respectively.

For the Buy Order:

- The buy order is handled by Goroutine 1 as part of the ongoing phase.

For the Sell Order:

- The sell order is not processed by Goroutine 1 (which exclusively handles orders from `buyOrdersCh`), and instead waits in the `sellOrdersCh` until the buy phase concludes.

Phase Transition:

- Once the "buy phase" ends, the instrument initiates a new "sell phase" to process the pending sell order.

Program Concurrency

Overall, the program allows for the first type of phase-level concurrency (most of the time), where orders from the same instrument and type (e.g. multiple buys) can execute concurrently.

Phase-level Concurrency

Phase-level concurrency is achieved through a pipelined approach that utilizes the two goroutines started during a phase, separating the matching logic from the order storage logic.

Goroutine 1 – Order Processing

- Continuously reads incoming orders from the active order channel, attempting to match each order against best rest orders from the opposite side.
- If the order is partially matched or remains unmatched, order is sent to Goroutine 2 for insertion, otherwise signal to client via the done channel that order has been processed.

Goroutine 2 – Unmatched Order Handling

- Listens on an internal channel `unmatchedOrdersCh` for orders that were not fully matched by Goroutine 1.
- Inserts unmatched orders into the correct resting order slice, then signals to the client via the done channel that the order has been processed.
- Sorts resting order at the end of the active phase

Pipeline Dynamics

As goroutine 1 processes incoming orders, it concurrently passes any unmatched orders to goroutine 2. This allows goroutine 1 to process the next incoming order while the previous unmatched order insertion into resting orders is handled by goroutine 2.

Phase termination

Achieved by signaling between the two goroutines using a cancellable context and a wait group.

- Goroutine 1 finishes processing all incoming orders (detected via default select branch), calls the cancel function, signalling that no more orders are arriving.
- Goroutine 2 is notified through cancel function as it is listening on matchingCtx.Done(), sorts the resting orders and then exits, calling done on the wait group
- This signals to the instrument main goroutine that it can move to the next phase.

Synchronization & Avoiding Leaking Goroutines

Channels are used to transfer requests and receive the results of the requests. Contexts and wait groups are used to ensure that certain goroutines wait for others to complete before they terminate. Notably, the engine's wait group is added to the instrument-managing goroutine and all instrument goroutines, which avoids leaking goroutines by preventing the engine from shutting down before all started goroutines.

Order Cancellation

Order cancellation begins by checking the client's `unmatchedOrders` map. If the order ID isn't found, it indicates that the order has either already been fully matched or never existed. If the order is still unmatched, the cancellation request is forwarded to the corresponding instrument as if it were a new incoming order. Notably, since processing a buy order operates on the resting sell orders—and vice versa—a cancellation request for a resting sell order is sent to the `sellOrdersCh` channel, while a cancellation request for a resting buy order is sent to the `buyOrdersCh` channel. This design ensures that cancellations are handled correctly within the existing order matching framework.

Testing Methodology

Besides the basic test cases provided to us, harder test cases were created with the aid of ChatGPT. The test cases were created to test the program in increasing complexity.

1. Fully Matching Orders
2. Partial Matching Orders
3. Partial Fulfillment with Cancellation
4. Multiple Instruments, Independent Concurrency
5. Phase-Level Concurrency (Multiple Buys vs. Single Sell)
6. Multiple Instruments with Phase-Level Concurrency
7. Interleaved Orders with Sleep Delays
8. Multi-Instrument Stress Test (10 Clients)
9. Large Single-Instrument Queue with Multiple Price Levels
10. 40 Clients, All Fully Matching
11. 40 Clients, Mixed Full & Partial Matches with Cancellations

We ran test cases with -race flag to detect race conditions, and used the code's existing `.NumGoroutine()` function to check for goroutine leaks.

Appendix

AI Tools Declaration

ChatGPT was used to aid in generating the test cases

Links to ChatGPT prompts:

- Test cases generation, test case generation queries are near the end of the ChatGPT session

  https://chatgpt.com/share/67cbfc15-ea30-800a-b708-6f053aaf95d4