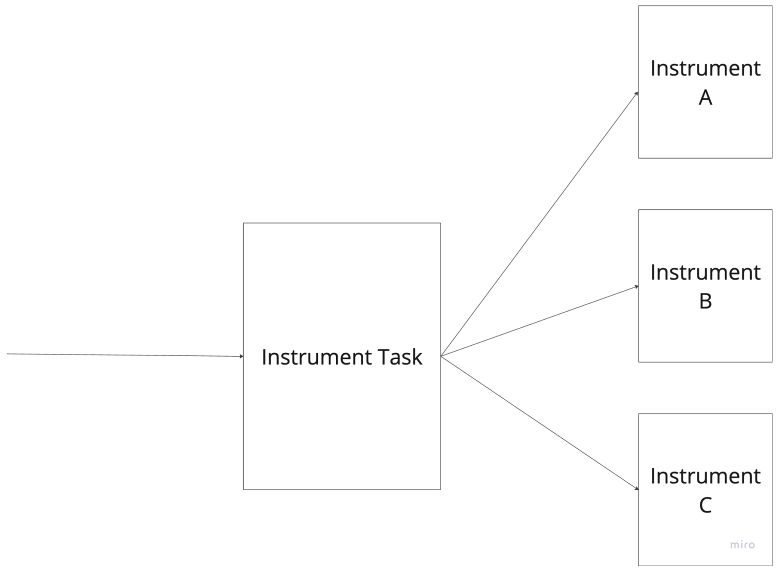
Implementation

In this assignment, we achieved **instrument level** concurrency. The exchange consists of an inputStream channel and instrumentTask goroutine. It can broken down into 3 main phases:

1. **Receiving Task**: orders are received through a connection and sent to the exchange

In the exchange, we read in the input from the I/O and pass it into the inputStream channel with a newly created wait channel to the instrumentTask goroutine. We then wait for the process to be completed when a confirmation is received in the wait channel after the order is processed per connection.

2. **Instrument Task**: orders are received and routed to its respective instrument through an instrument channel



It consists of an instrumentMap of instrument strings to channels and an orderCache, a map of orderId to instrument strings. Each instrument is a separate goroutine.

We utilized a **fan out**, **pipelining** **and** **for-select** pattern to achieve concurrency in this phase.

- Pipelining: As shown in the above diagram, we have one goroutine that handles the incoming task and dispatches it to whichever instrument it belongs to. This 2-staged pipeline allows instruments to be processed within the instrument goroutine and being decided where to go concurrently.
- Fan Out: Whenever we encounter a new instrument, we will **fan out** and generate a new goroutine for that instrument itself, storing the channel into the instrumentMap. Orders will be passed into the goroutine via its corresponding channel. We use a **for-select** loop to ensure that it does not terminate. This allows many different types of instruments to process concurrently.

3. **Order Task**: where the orders are being processed within each instrument

It consists of 2 heaps, a buyHeap and sellHeap, ordered in ascending and descending order. The received order finds a match or cancels in their respective heap and prints the output appropriately.

After we complete the whole process for a single order, we will pass in a value into the wait channel for the order. The reason for having a wait channel is to synchronize the orders for each goroutine and ensure the “happens before” relationship for each client is satisfied. This means that in our input test case, if client A is supposed to process order 1 then 2, we need to ensure that client A finishes 1 before it can process order 2. If we do not have the wait channel, we can read and process concurrently. However, this means that we are not guaranteed that order 1 happens before order 2 and it might be re-ordered (leading to the test case failing) as writing to the channel subsequently will get blocked. Therefore synchronization between clients is important.

The concurrency level works best when we have as many different types of instruments as possible, because different instruments can process concurrently and clients do not have to wait for each other. To put this into perspective, suppose that we have one order in GOOG and one order in APPL, the two separate orders can be processed separately in their goroutines as they do not share the same memory space within each goroutine.

However, the limitation is that if we have multiple orders from the same instrument, the waiting orders need to wait for the processing order to complete before it can begin processing.

Testing Methodology

The basic test cases provided focus more on the correctness of our matching strategy. This ensures the correctness of our code in a single thread, taking into account the different possible scenarios. If the basic test cases fail, there is something wrong with our code and we will not be able to proceed with the intermediate cases.

Next are the written intermediate test cases that help us to check whether we have achieved phase-level concurrency. These tests include multiple threads working on a small number of orders.

Finally, we wrote a python script that generates an input file with random orders with the given number of threads, orders and instruments. We will increment the values slowly and run the input file up to 20 times to ensure that the output passes. A huge number of orders, threads and instruments are given to stress test our implementation and see if it can handle the large scale of orders and threads received.