Marek Pinto, Jacob Stein
CS3211 Assignment 2

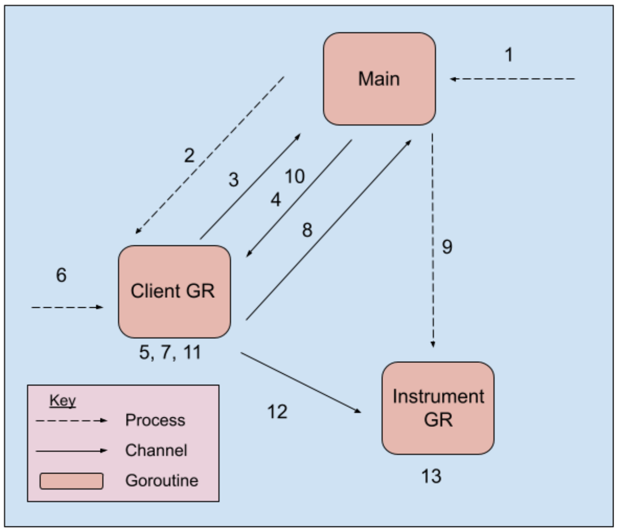
Explanation of Concurrency

We use one goroutine for each client and one goroutine for each instrument to enable concurrency. We also have a main goroutine which coordinates with the clients to spawn instrument goroutines as needed and maintain synchronization of channels between client goroutines and instrument goroutines. The main goroutine is run as an anonymous function in main.go, and it maintains a master source of information that all client goroutines draw from. It owns two channels that it reads from, *clientReadCh* and *newClientCh*.

Data Structures

- *clientWriteChSlice:* Owned by main goroutine. A slice that stores a channel for each active client. New channels are added here every time a new client is initiated. Every time a new instrument is initiated, information about that instrument and its corresponding channel are sent to every client in this slice.
- *instrumentChMap*: **Owned by main goroutine**. A map with instrument strings as keys and instrument channels as values. Whenever a new instrument is read from, a new goroutine is created for that instrument with a specific channel to read on. That channel is then stored in the *instrumentChMap*.
- *instrumentChMap*: **Owned by a client goroutine**. A map from instrument strings to instrument channels. It is equivalent to the map in main, and is updated whenever information about a new instrument is received on that client's *readCh*.
- *idMap*: Owned by a client goroutine. It is a map from Order IDs to instrument strings. Whenever a cancel comes in for an order, the client must first look up the instrument corresponding to that order in this map, and then find the proper instrument channel to send the cancel order to.
- *tickerSlice*: Owned by an instrument goroutine. It is used to store resting orders of that instrument. It stores input type, id, price, count, and execution ID. It is traversed when finding a match.

Concurrent Execution



Initializing Client Goroutines - *newClientCh* is a buffered higher order channel that any client can write to, and is passed to a new client through the *accept* method (2). Whenever that new client is initiated, it makes a new *InstrumentChannel* struct (used to send information about instrument channels) and sends the channel via *newClientCh* to the main goroutine (3). The main goroutine stores that channel in a slice of client channels, so that whenever a new instrument is initiated, it can send information about that instrument to every client (4). Furthermore, it sends information about all current instruments in the master hashmap over that channel to the newly initiated client, so that the client has a fully updated set of data (5).

Initializing Instrument Goroutines - Whenever a client receives a new order (6), it first checks if it already has that instrument stored in its hashmap. If it does, then it will process that order by searching for a matching order in its slice. If not, it will send the new instrument name as a string to the main goroutine via the *clientReadCh* (8). When the main goroutine receives a new instrument name on the *clientReadCh* and that instrument is not currently in its hashmap, it will spawn a new goroutine for the instrument (9). It will also create a channel for that instrument of type inputPackage, which stores all relevant information pertaining to an incoming input (including its timestamp). It then broadcasts a message of type InstrumentChannel struct to every client channel in its *ClientWriteChSlice* (10). Every client goroutine then receives that message and updates its instrument hashmap with the instrument instrument as a key and the instrument channel as the value (11).

Handling Orders - When a client receives an order and does not have the instrument in its hashmap, it runs the above instrument goroutine initiation process, and blocks until the instrument is added to the hashmap. Then, it will process the order by retrieving the proper instrument channel from its hashmap and sending an input package (consisting of the input and current timestamp) to the instrument goroutine (13). The instrument goroutine takes in orders one at a time, and processes them by matching against a slice of resting orders. If a match is found, the corresponding output is printed and the relevant counts are updated (13). Cancels are sent by the client in the same manner, except that the client will first look up the cancel id in a hashmap that maps ids to instrument strings, and then sends the cancel id to the corresponding instrument channel.

This implementation achieves **Instrument-Level Concurrency**. Each client can receive input independently, and has its own hashmap of instrument channels. These channels are kept consistent across all clients via broadcasts from the main channel, which stores a master source of information about the instruments. Different instruments can run concurrently because each instrument has its own goroutine that orders are sent to for that instrument only. However, orders for the same instrument are serialized because each instrument only takes in one order at a time from its input channel.

Go Patterns

Confinement - We implement confinement by creating a separate goroutine for every client and instrument. These goroutines then communicate via several channels to process orders. Each client and main stores their own hashmap of discovered instruments. The data in the hashmaps are only modified internally by that goroutine, and information is broadcasted to other hashmaps via channels upon discovery rather than providing direct access to the “discovering” client’s hashmap.

For-Select Loops - We use for-select loops in a few situations to continuously read and update information. Our main goroutine constantly reads from clientReadCh for new instruments, then makes their corresponding channel and stores it in a hashmap. Simultaneously, it reads for new clients, then makes their goroutine and copies over the discovered instrument hashmap. We also use for-select loops to constantly read channel information sent from the instrument and client goroutines before calling handleOrder.

Testing

We began by solving obvious logical or concurrency issues in our code, and passing the basic test cases. We then moved on to creating complex test cases using a Python script *generate_test_cases.py*. We generated test files to mimic complex testing cases. Our complex cases are created from 1 of 428 possible instruments, with up to 50,000 orders, random assignment of clients, a price of up to 2000, and a count of up to 1000. More specific parameters are available in the script. We also created two more categories of tests: *medium* (up to 4 clients) and *mediumHard* (up to 20 clients).

We ran into an issue with cancellations, where a cancel order would not find its corresponding stored order. After tracing through our concurrency logic, we found that if two orders with the same undiscovered instrument arrived concurrently, they would create separate goroutines for that instrument. This means they would be stored in two different slices. To fix this, we made each client with a new instrument pause and check to see if the instrument had already existed before proceeding, rather than immediately starting a new goroutine. After implementing this, we were able to achieve 100% accuracy across all concurrent test cases (medium, mediumHard, and complex).