CS3211 Assignment 1

David Zhu (E0958755), William (E1496974)

AY24/25 Semester 2

1 Data Structures

In this section we discuss the data structures used in the implementation of the exchange.

1.1 Concurrent Hashmap

The idea of this concurrent hashmap is to use separate-chaining with a linked list to handle hash collisions. We call each linked list a bucket. Accessing different buckets concurrently is safe as they are different memory locations. Each bucket has its own `shared_mutex` to allow concurrent access to it. The hashmap also has a `rehash_mutex` to lock the entire hashmap when resizing it.

```
template <typename K, typename V> class HashMap {
  using Bucket = std::list<std::pair<K, V>>;
  std::vector<Bucket> buckets;
  std::vector<std::shared_mutex> bucket_mutexes;
  std::atomic<size_t> num_elements;
  std::shared_mutex rehash_mutex;
  float max_load_factor = DEFAULT_LOAD_FACTOR;
};
```

Figure 1: HashMap Fields

1.1.1 Bucket-level Locking

Each bucket employs a dedicated `shared_mutex` for synchronization. Read operations (`contains`, `get`) use shared locks (`shared_lock`) allowing concurrent access, while write operations (`insert`, `remove`) utilize exclusive locks (`unique_lock`) for mutual exclusion. This fine-grained approach enables simultaneous access to different buckets, with contention limited to concurrent accesses within the same bucket.

1.1.2 Rehashing

Table resizing is guarded by a global `shared_mutex` (`rehash_mutex`). Normal operations acquire it in shared mode, while rehashing requires exclusive access. During rehashing, this mutex is locked exclusively after releasing all bucket locks, temporarily blocking new operations but ensuring safe bucket redistribution. The two-phase locking (release bucket locks before acquiring rehash lock) prevents deadlocks during capacity expansion.

1.1.3 Size Tracking

The element count is maintained through an `atomic<size_t>` counter (`num_elements`), enabling thread-safe size queries without explicit locking.

1.2 Priority Queue

This priority queue is implemented using `std::set` with two custom comparator that compares the price of the buy orders and sell orders respectively. Buy orders are sorted from highest to lowest price, while sell orders are sorted from lowest to highest price. We chose `std::set` instead of `std::priority_queue` because we need to be able to remove specific orders from the priority queue to support cancel operations. The priority queue is not thread safe and is protected by a mutex at the instrument level.

2 Exchange Implementation

To store instrument data, we utilize a custom concurrent `HashMap` implementation. The structure of the `HashMap` is as follows:

- **Key**: The name of the instrument.
- **Value**: The `Instrument` class, which contains:
  - Two priority queues: one for buy orders and one for sell orders.
  - A mutex to ensure thread-safe access to the order data.

Each instrument's data is protected by its own mutex. This ensures that only one thread can access the priority queues (buy and sell orders) for a specific instrument at any given time.

2.1 Buy/Sell Order Handling

For brevity, we will go through how we handle buy orders, the handling of sell orders is the same but against the opposite order type.

When a buy order is received, we first check if the instrument exists in the `HashMap`. If it does not, we create a new `Instrument` object and insert it into the `HashMap`. These operations are thread-safe thanks to our concurrent `HashMap` implementation.

We then acquire the instrument's mutex to ensure exclusive access to the order data. Next, check if the buy order is price compatible with the best sell order in the instrument's sell queue.

Case 1: Price compatible
Then we execute the maximum possible quantity between the two orders. While the buy order is not filled and the sell queue is not empty, we continue to match it against the best sell orders in the queue. After this order matching process, we check if the buy order is fully filled. If it is not, we add the remaining quantity to the buy queue.

Case 2: Not price compatible
Then we can conclude that the buy order cannot be executed. We add the buy order to the buy queue.

2.2 Cancel Order Handling

For cancel order handling, we store a shared pointer to the order in a thread local hashmap in the client for all the buy/sell orders we receive. This allows us to quickly access the order to be cancelled without having to search through the priority queue. When a cancel order is received, we first check if the order exists in the thread local hashmap. If it does, then we acquire the instrument lock and check if the order is still available (i.e. not yet fully filled or cancelled). If the order is still available, we remove it from the priority queue and reply with accepted. If the order is not available, we reply with rejected.

3 Concurrency Level

The concurrency level of our implementation is instrument level. As the `HashMap` can be accessed concurrently by multiple threads, allowing multiple instruments to be processed concurrently. However, within an instrument, only one thread can access the order data at a time due to the mutex at the instrument level.

4 Testing Methodology

For testing of our concurrent hashmap, we wrote a test program that asserts the correctness of the hashmap under both single-threaded and multi-threaded scenarios.

For testing of the engine, we wrote a python script that generates random input files for the grader and we run it continuously check for any test failures. The generate input files of different sizes, from 4 to 40 clients and up to 40000 orders. It is ran for > 100 iterations for large cases and > 1000 iterations for medium size cases to ensure the engine is likely to be correct. The python script is in `scripts` directory. We also ran our engine with `tsan`, `asan`, `msan` and `valgrind` to check for data races and memory related bugs.