System Design

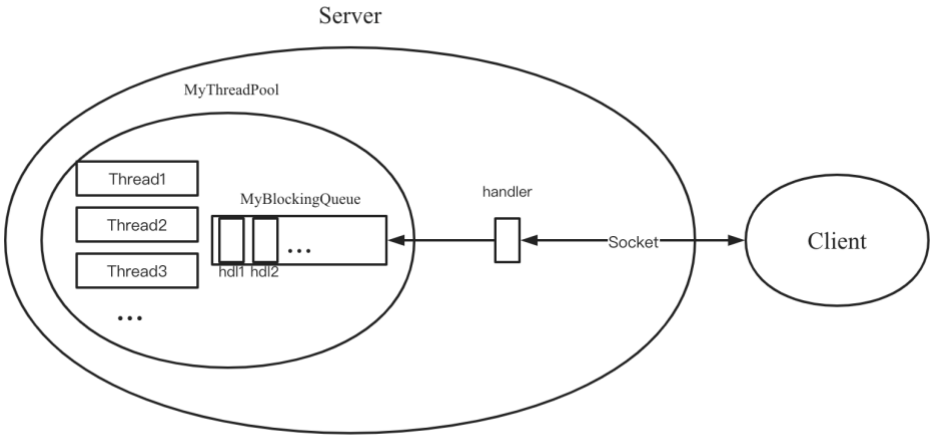
Part 1.

For part 1, we have developed the system based on handwritten thread pool. The system consists of two ends: Server end and Client end. Within the server end, there are thread pool, request handler, blocking queue for quests and the stock structure stored in memory as our main structure. We adopt this structure to implement our system simply because this is how it works in Java built in thread pool model.

The Stock is constructed in a concurrent HashMap and a HashMap which are each for storage of stock number and toy price. The two part are separated because the action of query and buy can be done independently. Also note that the price of a stuffed toy is remaining constant, so it is not considered to be stored at a concurrent one. The stock supplies the query and buy method inside the class which can be called thread safely, though in part 1 there are no write operation that would cause a thread unsafe status.

The handler implements the runnable interface, which contains the core processing function of the server, which is to query the stock and return the price or 0 or -1 as required. This handler is created when new socket comes in, storing information for that socket and getting queued.

The MyBlockingQueue class is what queues them. The MyThreadPool has a blocking queue that waits execution, and sends it to each thread when initialization, which means different threads are going to operate on the same blocking queue to retrieve the newest task when the work on its hands is finished. As a result, the blocking queue must be thread safe. Here we applied a classical producer-consumer problem solution to it: when there's no item in the queue, add lock on take(); when the queue is full, add lock on put(); otherwise, you can both take and put. The operator is also required to hold a lock to get access to the queue!



The MyThreadPool is easy to explain now. Since it is a static number thread pool, it will start all threads in a row upon initialization, with each thread activated for retrieving new tasks to work on till the whole server to shut down.

The overall structure is depicted on the graph above.

Part 2.

Part 2 is built on the gRPC, so most multithreading work are done by it. Here we defined two services in the protocol buffer, which are the query and buy. They are implemented in a similar way of operating on the stock discussed in part 1. To make sure the stock invoked by the gRPC method is the same each time, here we applied a double check singleton design pattern to it.

Test.

We applied JUnit and tempus-fugit to concurrently test both parts. The test code is also included in our GitHub repo.