ECE 568 Homework 4
Scalability: Exchange Matching

Haohong Zhao, Zihao Wu, Yifan Li, Zi Xiong

April 3, 2019

Abstract

In this assignment, we developed an exchange matching engine coded in Go with Redis database as backend. Detailed explanations for our choices of the programming language, synchronization strategy, and database, are given in the first section. Then, how our architecture is developed is illustrated in the second section. Next, experiments were done to test correctness and scalability of our server software. Observations of the benchmark results were analyzed. Finally, we drew a conclusion on the performance and defects of the system and further investigation in the future.

1 Utility Selection

1.1 Golang

Go has an easy-to-use built-in concurrency system -- goroutines which are different from threads. Specifically, a goroutine is able to map multiple threads, and therefore, the overhead of thread creation and context switch is reduced in an elegant fashion. Therefore, Go has an edge over handling high concurrency. In addition, Go is a language which supports fast development, deployment and easier refactoring. Furthermore, third-party packages and dependencies vitalize the community of Go and the ease of importing these packages are simply a “go get” command.

1.2 Read-Write Lock

To reduce impediments to scalability, the resources are locked by global Read-Write Lock rather than by a basic lock. The motivation of Read-Write Lock comes from the fact that race condition is not present in the situation when all access to shared memory is read operation without any write. Considering the frequent operations, order query is a read-only operation to the database; the performance would be tremendously boosted as high concurrency of query commands in the system.

1.3 Redis

Redis is our database used, for the following reasons :
1) Support common-used data structure: String, List, Set, Hash, ZSet (aka. sorted set). The simple key-value database, no relational setup between tables.
2) Up to 100000+ queries per second. Although it is a single thread database, it supports both the in-memory cache database (HashMap in memory) and persistence in the disk. Single thread, in other words, signifies much less overhead out of context switch
3) Multiplexing and non-blocking I/O. This technique is achieved by the combination of select, poll, epoll to make idle threads sleep, in the purpose of reducing operations that does nothing but takes resources.

2 Architecture

The command pattern is employed in this project. Modules are isolated into a business logic (aka. the main services provided by the matching engine), a TCP server, and a request/response XML parser. This design is encouraged to have good maintainability and extensibility. The matching engine system is layered into three: redis-utils (Redis utility), business-utils, business logic. The base layer redis-utils is designed to declare all database manipulations that are invoked in this project, and in the future, when more database manipulations are required, a modification would be made on this layer.

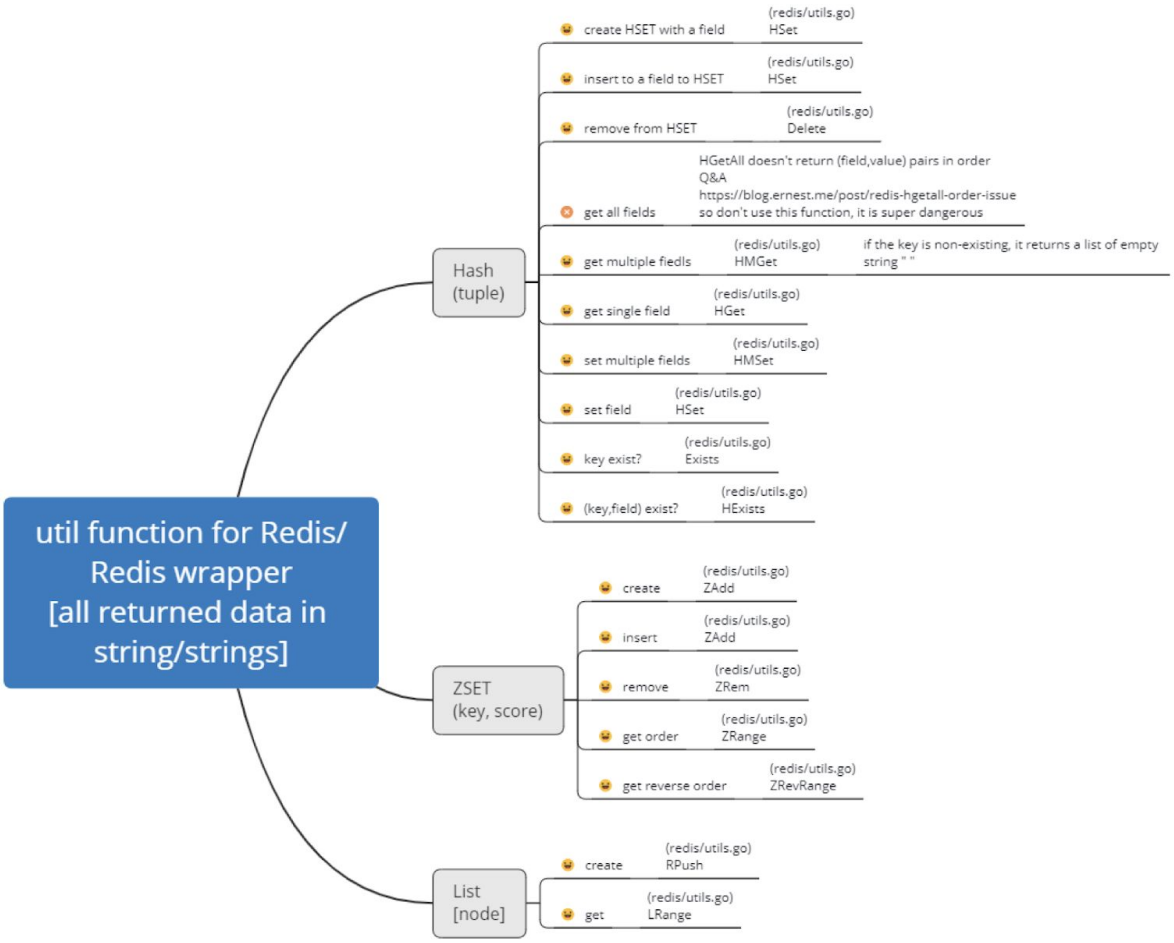

Figure 1

The following layer, business-utils, is defined as the utility functions which are used to integrate the next layer. Objects are classified in this layer. Consider the situation of a client interacting with our system: If the client wants to buy SPY via the matching engine, he first needs to register an account and save money to the account (“Account“). Then, order (“Open Order”) is placed in the market on demand of “SPY”. What’s more, he is able to query the order he made before, which may be canceled, executed, or still opened in the market (“Canceled History” and “Executed History”). In order to coordinate with ZSet in Redis which is actually a sorted set, a queue sorting open order based on limit price is required. Two queues are appended to the system (“Buy Open Order Queue” and “Sell Open Order Queue”).

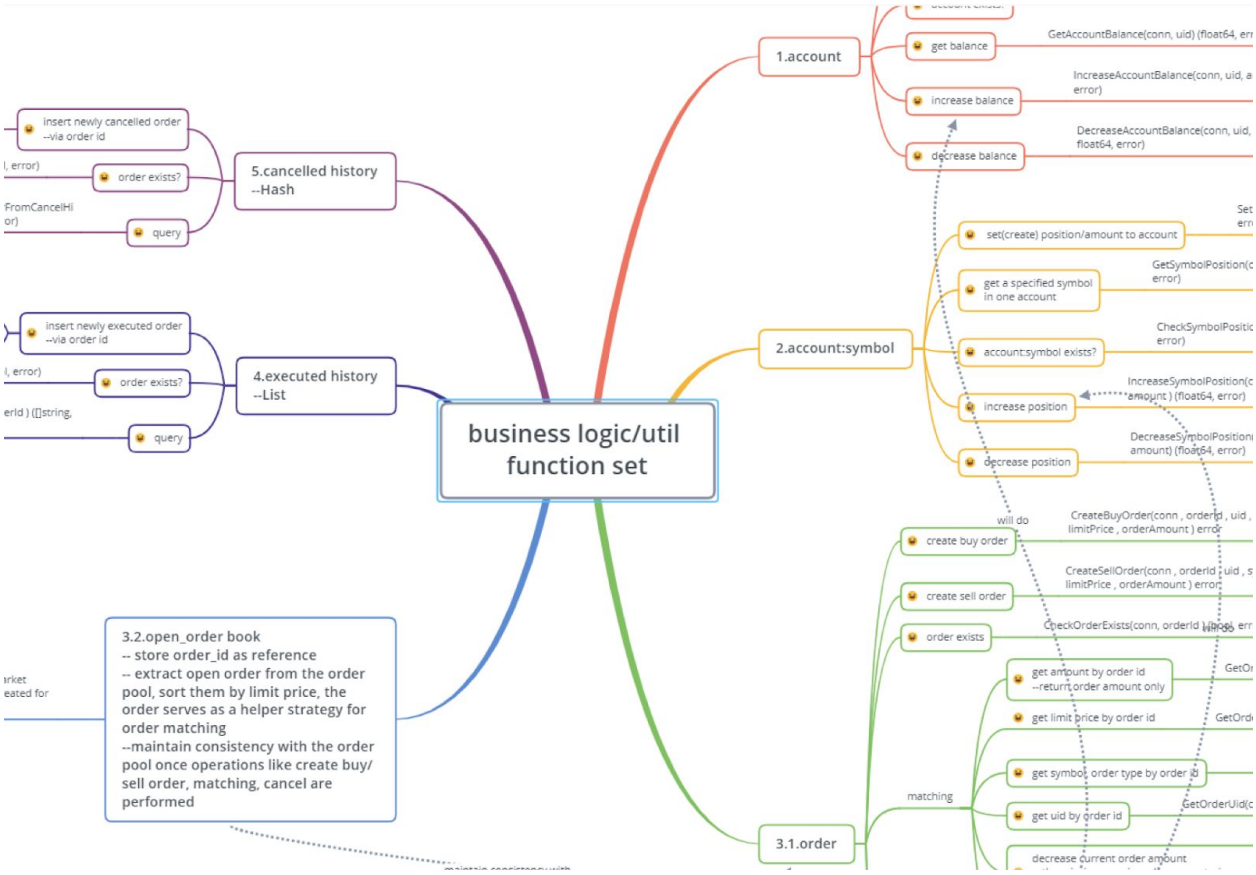

Figure 2

The final layer, business logic, is highly tied to commands a client issues to the system: Create Account, Create Position, Set Buy Order, Set Sell Order, Cancel Order, Query Order.

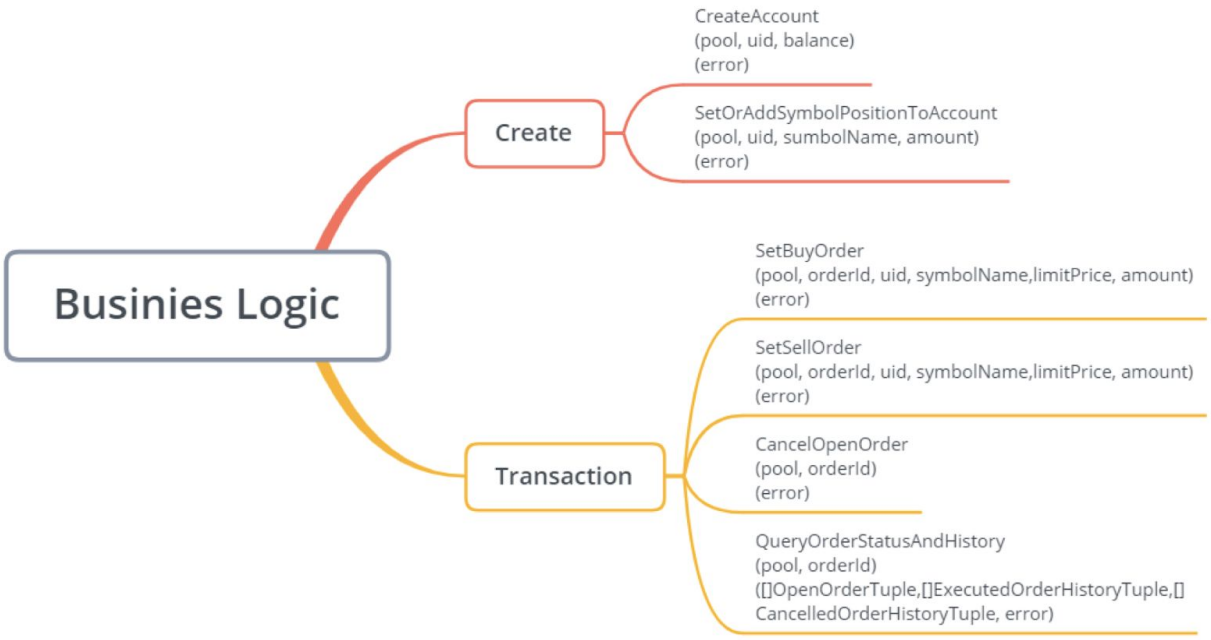

Figure 3

To bridge the communication between the TCP server module and the business logic module, an intermediate layer called command pattern is inserted. The intermediary will store the information required for executing specified commands. Each command has simply two methods: execute a command and get a response. The intermediary is instantiated with information parsed via XML parser which takes inputs from messaged received from clients and return information for specified commands.

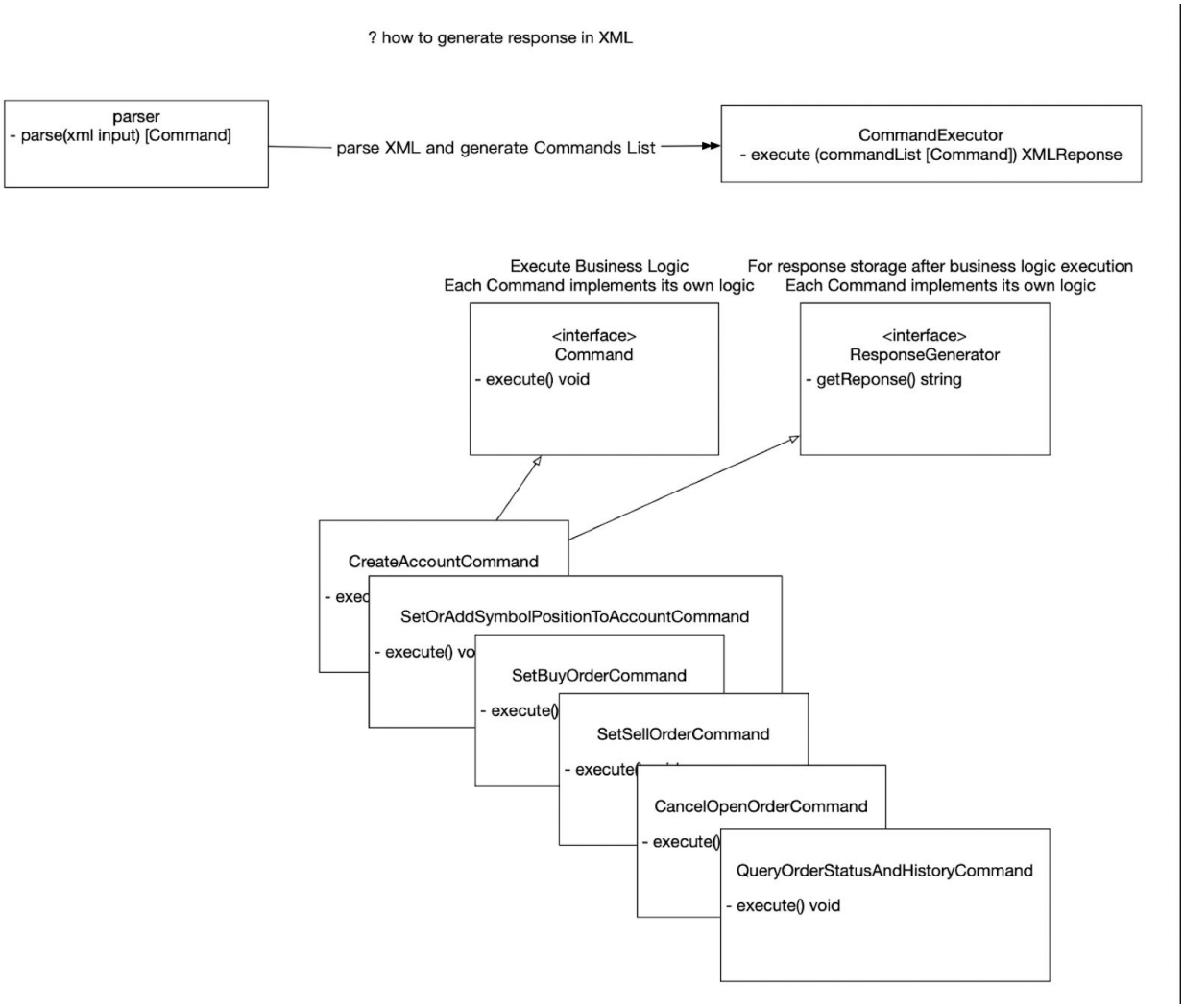

Figure 4

The final architecture is about TCP server, which is essentially basic socket programming. A server is waiting to receive a connection from clients. When a connection is established, it will be passed to instantiate a connection object. The request handling will be done in a Goroutine to achieve concurrency.

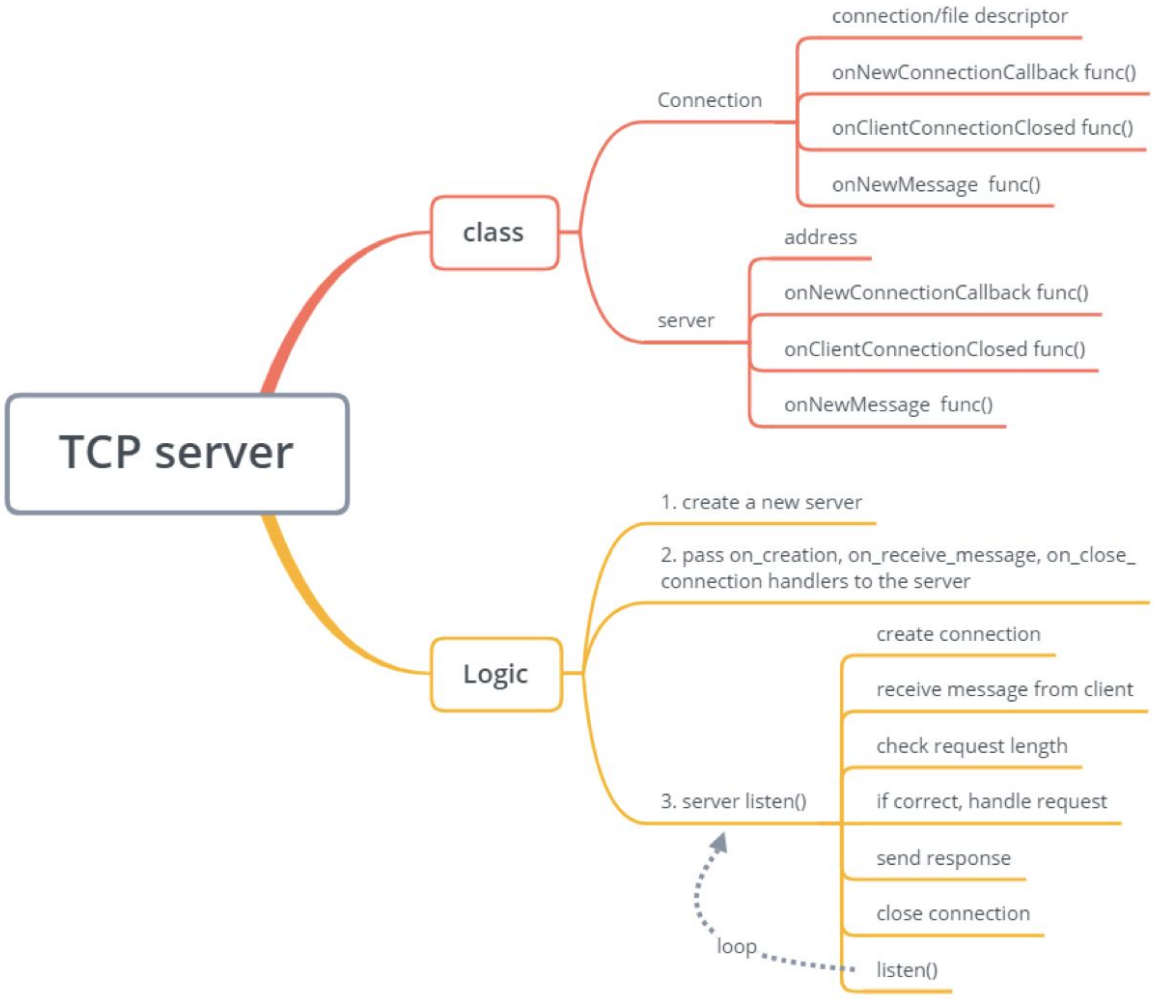

Figure 5

3 Benchmark Results

3.1 Correctness

test1.sh's testcase:
after run test1.sh
enter redis container and run redis-cli, then run
HGET account:12345 balance (get the balance of buyer:12345)
HGET account:34567 balance (get the balance of seller:34567)
HGET account:12345:SPY amount (buyer's SPY amount)
HGET account:34567:SPY amount (seller's SPY amount)

final state should be:
- buyer balance = $9860, SPY = 20
- seller balance = $140, SPY = 80

- buyer-uid: 12345, balance = $10000
- seller-uid: 34567, SPY = 100
- set buy order:
    - orderid = 1, amount = 50, limit = $7
- set sell orders:
    - orderid = 2, amount = -10, limit = $4 (matches with 1)
    - orderid = 3, amount = -10, limit = $5 (matches with 1)
- open order: orderid = 1, amount = 30, limit = $7 (with 2 executed history, buy 10 SPY at 7 dollars)
- cancel open order: orderid = 1, refund $210 to buyer

test2.sh's testcase:
after run test2.sh
enter redis container and run redis-cli, then run
HGET account:12345 balance (get the balance of buyer:12345)
HGET account:34567 balance (get the balance of seller:34567)
HGET account:12345:SPY amount (buyer's SPY amount)
HGET account:34567:SPY amount (seller's SPY amount)

final state should be:
- buyer balance = $0, SPY = 100
- seller balance = $10000, SPY = 0

- create buyer-uid: 12345, balance = $10000 for 100 times
- create seller-uid: 34567, with SPY = 1 each time (add 100 SPY in total) for 100 times
- create buy order 200 times : buy 1 SPY, $100/ each (100 success, 100 insufficient fund)
- create sell order 200 times: sell 1 SPY, $100/ each (100 success, 100 insufficient symbol) (all 100 sell orders match successfully)

The benchmark results could be obtained in the Redis container by running
$docker ps
Get container’s id
Then,

$docker exec -it <container> bash
Enter the container,
#redis-cli
Enter Redis database
>HGET account:12345 balance
Get buyer's balance
>HGET account:12345:SPY
Get an account's symbol amount
Similar to verify seller

3.2 Scalability

We included some test cases for parallelization in our testing structures. Specifically, a sample test command and the corresponding input XML might be similar to the following:

```
time seq 100 | parallel -n0 "cat create1.txt | nc localhost 12345"


179
<?xml version="1.0" encoding="UTF-8"?>
<create>
    <account id="12345" balance="10000"/>
    <symbol sym="SPY">
        <account id="12345">0</account>
    </symbol>
</create>
```

The following plots show the measurement of scalability of our exchange engine when various numbers of requests were executed. As we can see from the plots, as the amount of input requests increases, the average runtime scales linearly and does not suffer much from increased lock contention. Specifically, we tried creating a new buyer/seller, creating selling orders and buying orders, and matching open orders.

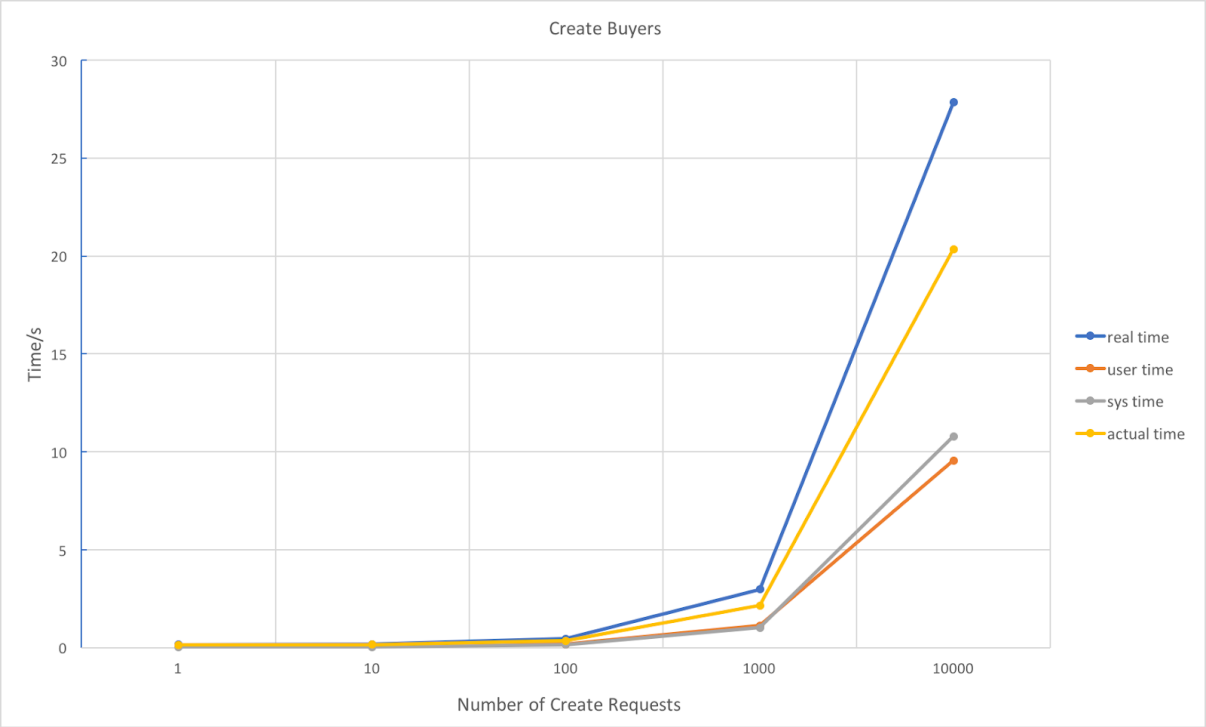

Figure 6. Scalability test for creating buyers

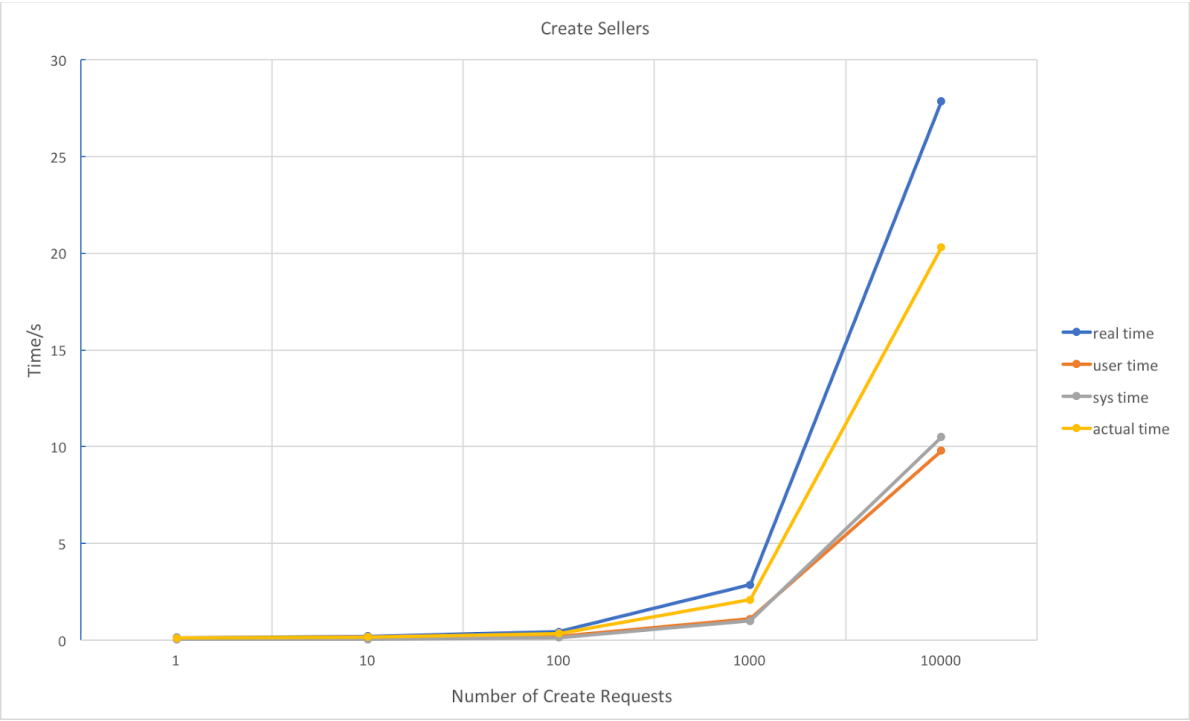

Figure 7. Scalability test for creating sellers

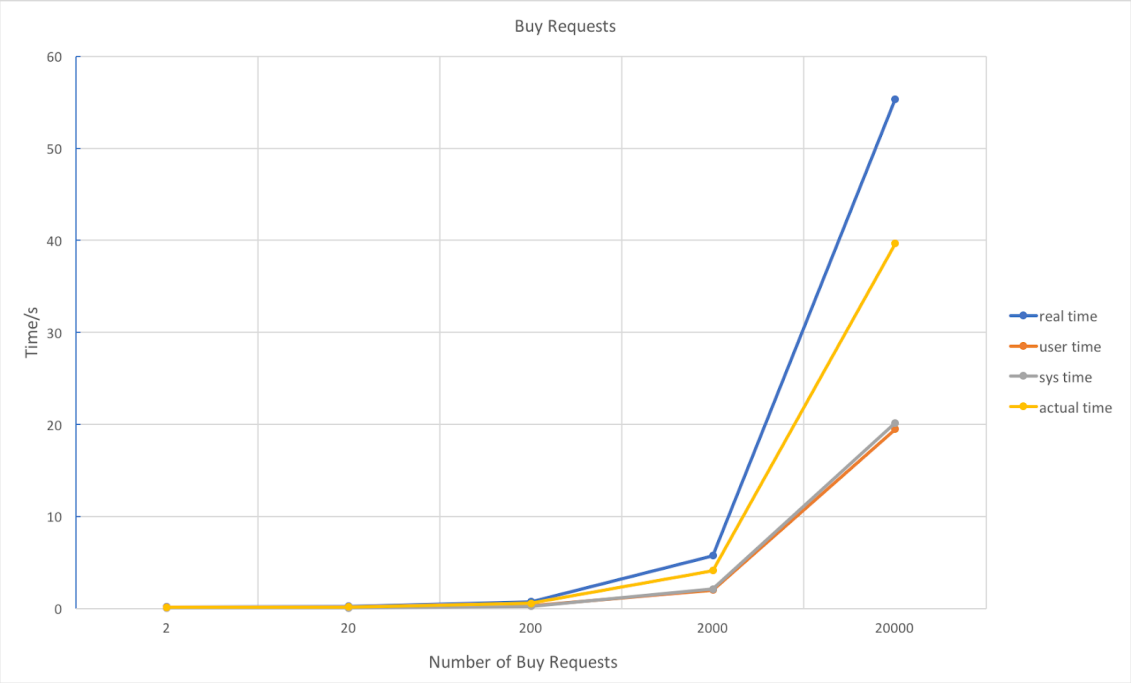

Figure 8. Scalability test for creating buying orders

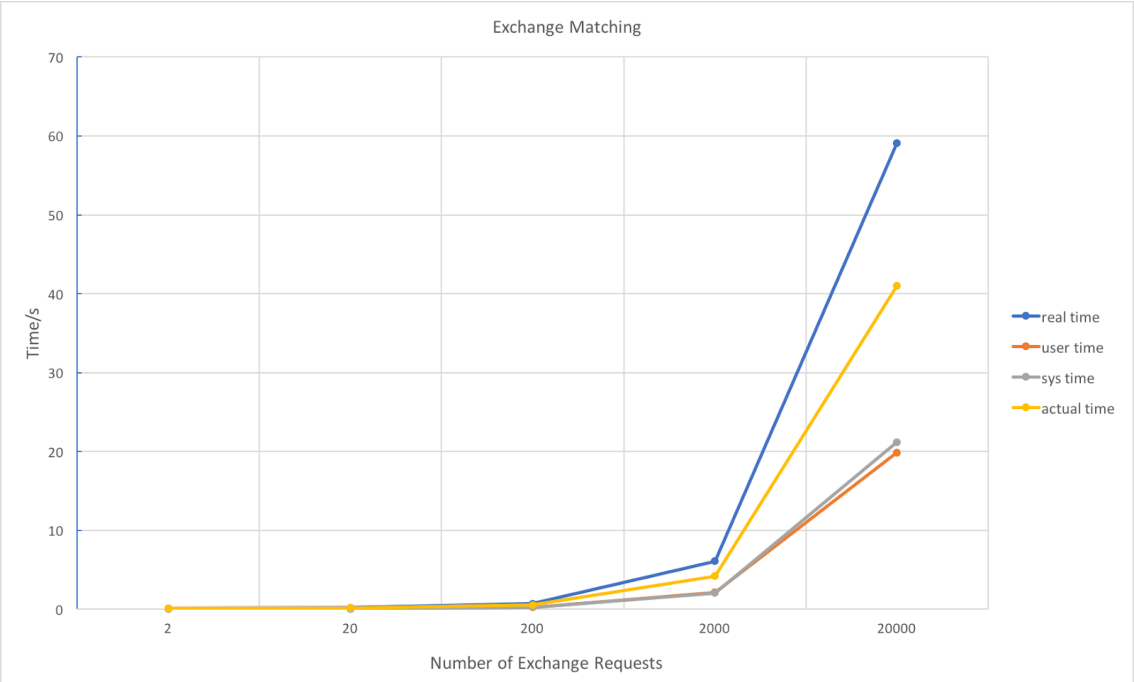

Figure 9. Scalability test for creating buyers

4. Conclusion

The performance of the system is as good as expected since it takes advantages of both concurrency system of GO and fast in-memory cache database Redis. The correctness of the system is proved by test cases covers a wide range of situations. The ability to handle concurrency is tested by sending a huge amount of request simultaneously. The architecture is well-isolated into modules which have good de-couplings; consequently, it is easy to read and extend in the future. In the further investigation, we may consider improving the system with Kafka, a message queue which allows applications to communicate by sending messages to each other, commonly used in subscribe and publish.