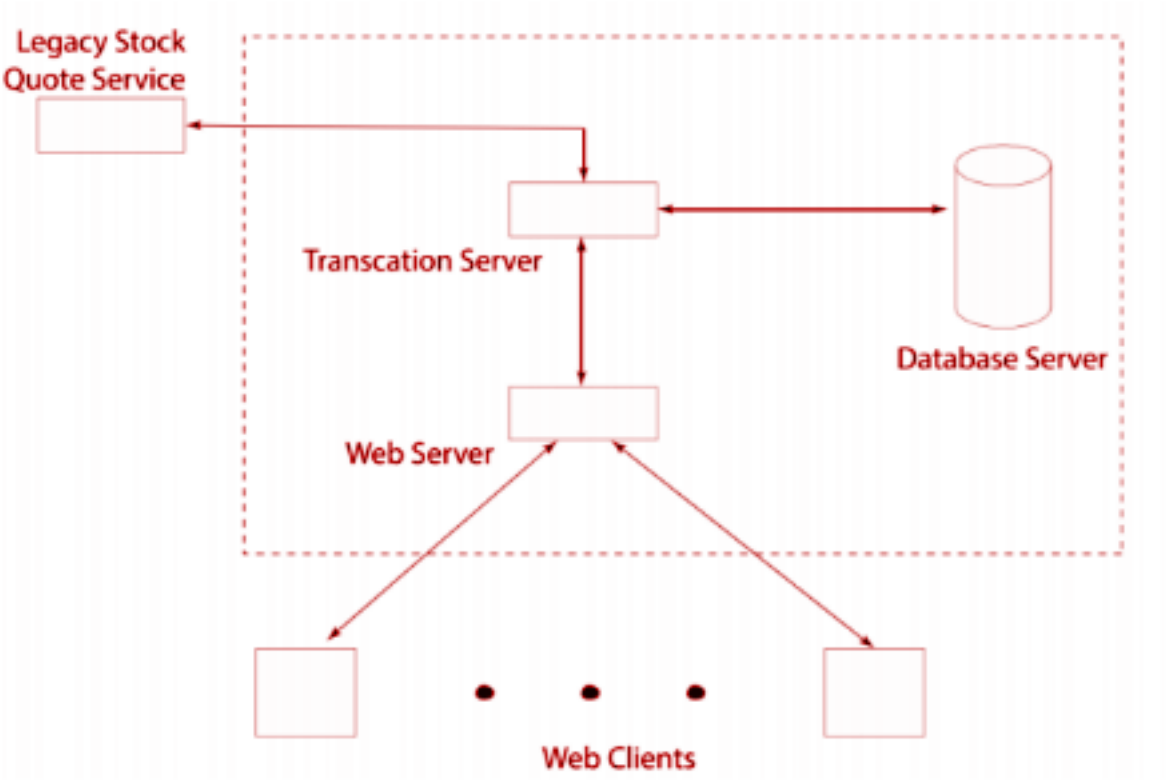
SEng468 Project Report

A simple day trading system



By: Oscar Wu & Zijian Chen

V00876811 & V00867494

Table Of Contents

INTRODUCTION

Our day trading system is mostly coded in python. In steps, we have created a web server, transaction server, audit server, and a database that interacts with a legacy quote server and lists of commands provided by the instructor to support the functionalities of our system. First of all, we have files that consist of lines of commands that simulate multiple users performing multiple commands. The commands that users can input are for example to quote the price of a stock, load funds into their accounts, buy and sell shares of a stock, set a specific amount in dollars and let our system purchase or sell a stock at the specified price, etc…

Our goal of this project is to construct a distributed system that can take a command file and process all of the commands inside the file. To do so, we have to build our initial system and scale it up to be capable of processing many users' commands at the same time within a relatively short amount of time. During this process, we have faced many problems as the number of users we are dealing with goes up and I will address them in this report as well as the final stage of our project.

FINAL ARCHITECTURE

Tools:

1. Python3
2. Docker
3. Docker Swarm
4. Mysql database
5. Redis

Overall Design:

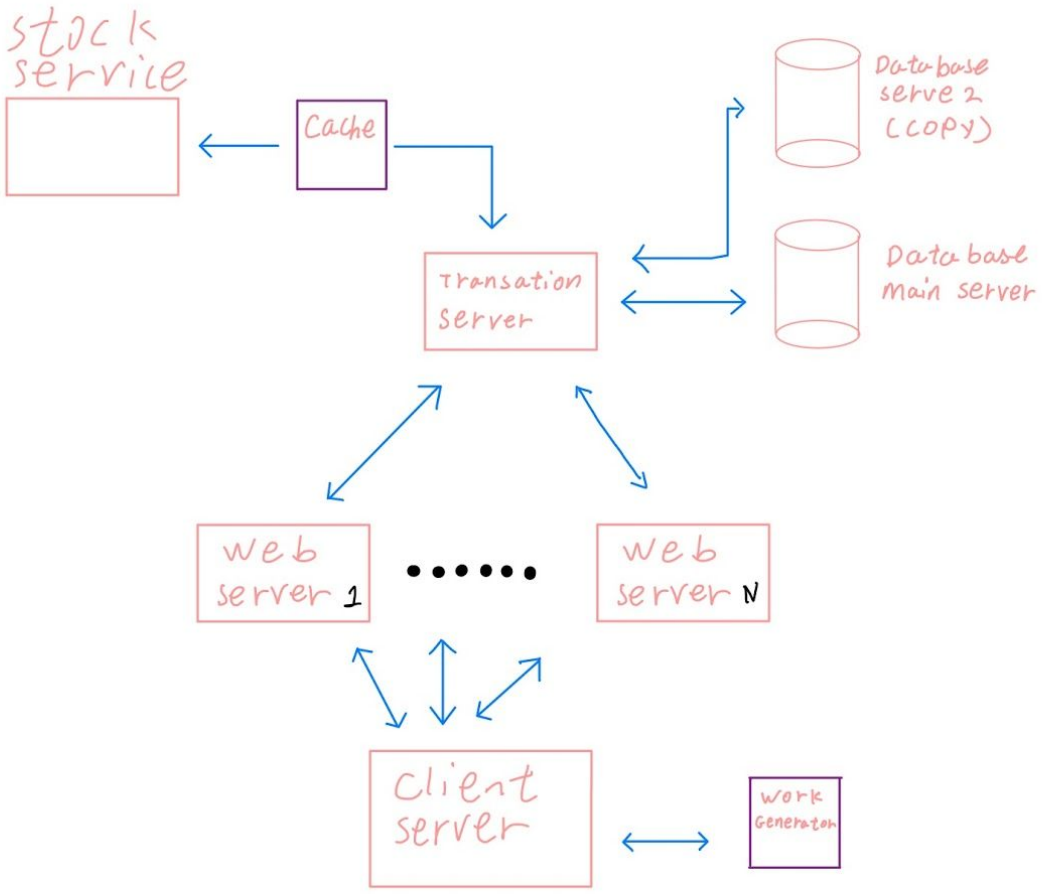


Figure 1. System design blueprint: This is an overview of the system structurel.

Workload:

The workload takes every users' commands in a file and sort them into a separate small file. For example, in the 1000 users, we need to scale the system, so we have multiple transaction servers to deal with those jobs. We must separate the entire command into smaller groups of users. We do manual load balancing because we don't actually have the users to test out our system, and we use this way to simulate the user's input commands.

Design Process:

In our original design, our workload is contained inside the Http Server and will do the work and read the command file while we run the Http Server. Now we separate out the workload because once we move to larger numbers of users, it will not continue to use old design because there will be more users and we will need to scale the system, so we must do the workload first to sort out the users into smaller groups before fetching them to the servers to process.

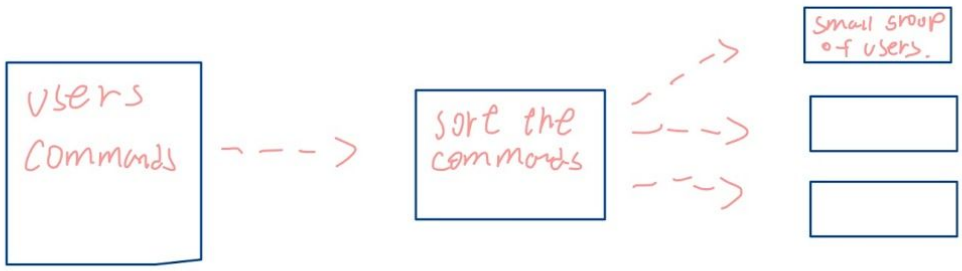


Figure 2. Workload Structure.

Http-Server:

The Http server will accept the user's commands and forward them into the correct transaction server because the transaction server will take the user information saved as a cache, and if we forward our users to the corresponding transaction server, it will increase the transaction speed. Also, the Http server will accept the responses from the transaction server and send to the client.

Design Process:

In the original design, we combined the workload and Http Server together, and it was not good to be used in a scalable system, so we separated them. After running the Http server we realized if the Http has only the one main thread, it is

not enough to handle the system. We needed to have two threads; one is used to listen to the incoming user job tasks, and the second one is used to wait for receiving the data sent back from the transaction server. This design will help us work on sending jobs and receiving data concurrently, hence increase our performance a lot because we no longer need to wait for the responses then send the next job task.

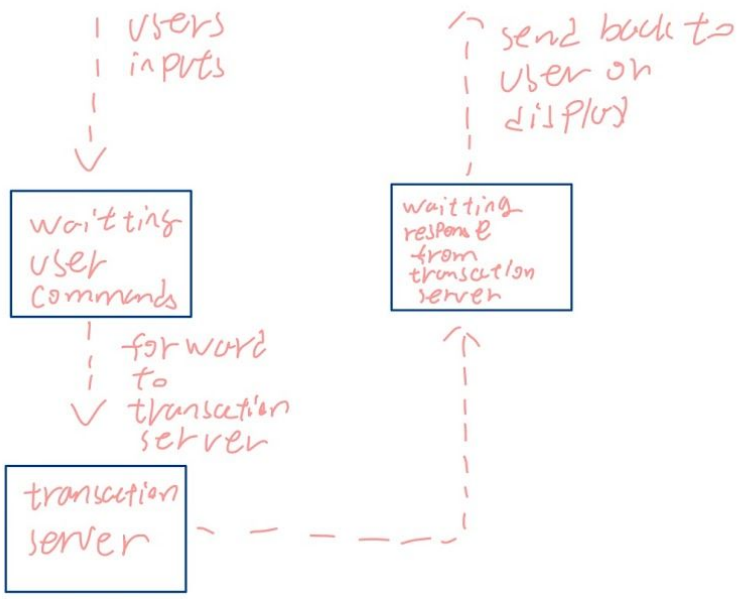


Figure 3. Http Server Structure.

Transaction-Server:

The transaction server is a major part of this project, it connects to the audit server, database server, Redis, and Http server. So, it acts like a CPU. Its main job is to receive the job task and store it in the job queue, then the system will create a thread to deal with this job or the user, so this feature allows the transaction to work concurrently and parallelly to process the works. During the processing of the job, the server will also log the information and use the JSON format to send the data to the audit server for logs and monitor the system event. Also, the transaction server is connected to the MySQL database and it connects to the Redis server which will store the temporary data as cache. When the user needs to access their data, the transaction server will first go to the Redis server to check if the data is in the Redis cache or not. If there is no data stored in the cache the system will then proceed to the database to request the data and send it back to the client which is the Http server.

Design Process:

In the old design, we did not implement a job queue on it, so it needs to finish the current job and in order to take the next job. This way is too slow and not a good approach to handle larger numbers of users, so we chose to use a thread pool to process the jobs concurrently. But there is a limit on the virtual machine so we cannot start as many threads as we want because there are memory and CPU process limits, and the Quote Server as well. To fix this problem we used the thread pool to limit the max thread we allow to be started. Also, the database has its own limitations, there are maximum connections. The Mysql DB only allows 151 active connections at a time, if we go over the maximum connections the database will crash. To fix this problem we spend a lot of time working on it after, if we scale the system it will be much harder to control the thread, and to fix this problem we still don't have a very efficient way to do it because once we scale the transaction server, it will distribute to several machines and then the docker swarm will do the work, keep in mind that it is not easy to share the information between those machines, so the only way we can bypass this problem was to increase the maximum user connections on the Mysql database. We know it is not a good way to do this, but it is the only way to fit in our design. If in the future we get to do more studies about this, we will try to explore a better way to fix this problem.

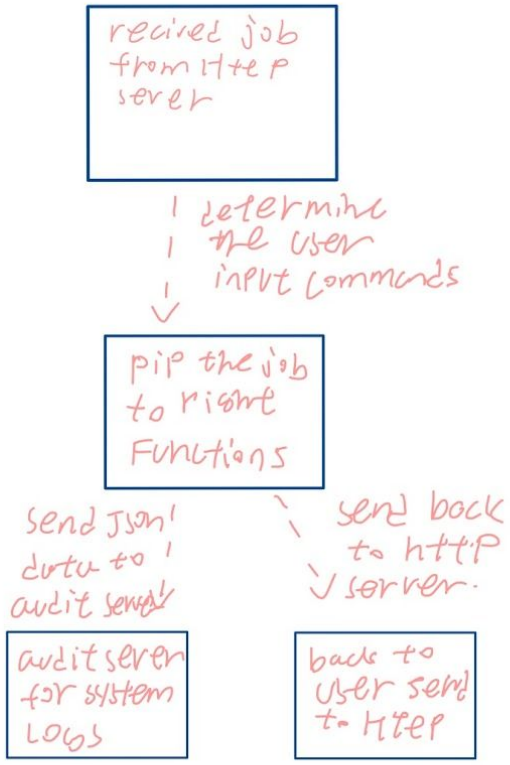


Figure 4. Transaction Server Structure.

Audit-Server:

The audit server will continue to accept the transaction server's log data and it is JSON format. Also, it will write an XML file for users to log the information and monitor the system. The audit server only deals with log files and writes the XML format data.

Design Process:

In the beginning of designing the Audit Server, we used TCP connections for receiving the string data from the transaction server and this is not good for receiving large string data without format, so we decided to use the JSON data format to transmit the data. The second change is to create another thread for writing the data into the XML file, so on the main thread we continue receiving the data from the transaction server and put them into the queue, and the second thread will keep writing the data from the queue. This design will increase the performance because it makes the audit server perform, receive and write concurrently.

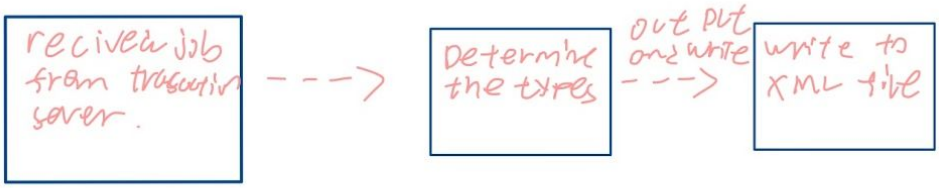


Figure 5. Audit Server Structure.

MySQL Database:

The database we decide to use is a MySQL database and it is stored on the local machine. We try to make two separate databases and separate the data from different machines. The reason we are doing two separate databases is making a copy for that if we have one database server is down and we will use the second one to continue the work of the use command.

Design Process:

The new design we use in the database is we create a second database for making a copy on the main database, and this is also is a kind of protection on the data if the main database is down and we are able use the second database to continuously handle the job task and wait for bringing back the main database.

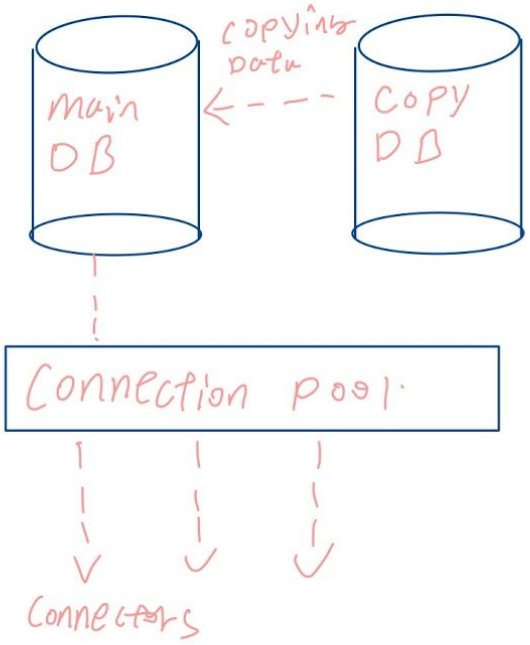


Figure 6. Database Structure.

Redis Cache:

The Redis server is very fast tools use for store data as cache, and we use the Redis server to store the temperate user data, for example, when the user uses the buy command but does not commit the buy and the system will delete the buy history automatically, so if we use the transaction to do checking for that and it will waste a lot of the system resources to do the linear condition checking, so we design to use Redis to do the work because the Redis is able to set an expired time on the data, so it helps to increase the performance at this point.

Design Process:

In the old design, we did not implement the Redis server as the cache, so this is new tools we add into our final architecture, and this will increase our performance by storing the temporary datas. If the user keeps the request the same data and if the data is still valid, it will present in the Redis cache. For example, if the user keeps sending the same Quote command, and the average Quote server response time is 0.5 seconds, this will help us save 0.5 seconds. One second to humen may be fast, but for the computer, it is long enough to process hundreds of job tasks. So, this cache is very important in this project, and it increases 30% of the total process time.

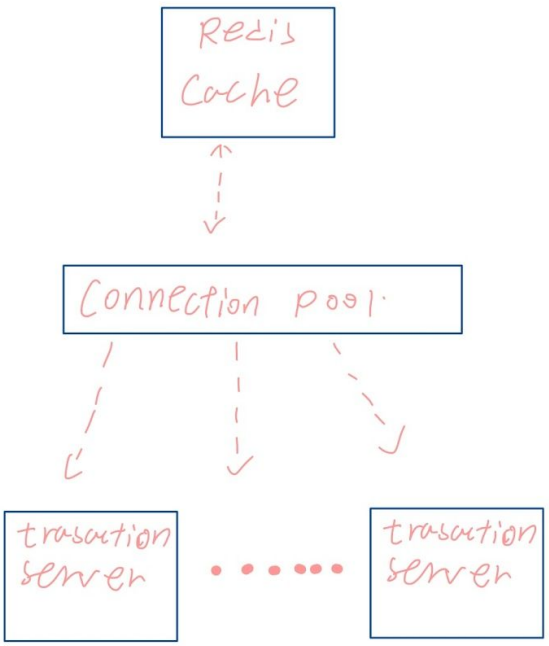


Figure 7. Redis Cache Structure.

Docker Swarm:

This Project requires us to use docker to build the whole system and scale the server. So, we choose to use docker swarm for the scale tool, first, we need a manager machine and several worker machines. We decide minimally we use 5 machines including manager the node. We replicate the transaction server 5 times, so that we will make our power up to 5 times which means 5 transaction servers working for 1000 users at the same time.

Design Process:

In the beginning when we were still building up the system, we did not use any docker tool to build the system. But after we start getting a larger number of users and the virtual machine was no longer enough to handle the system, we then needed the docker container to run the system, and also we need to scale the system so docker swarm will help us to connect the machines and auto load-balancing the transaction server. This will allow us to run multiple transaction servers at the same time, and that makes performance boost up to 1000 TPS or even 10000 TPS, and it depends on how many worker nodes we use to concurrently work on the job tasks.

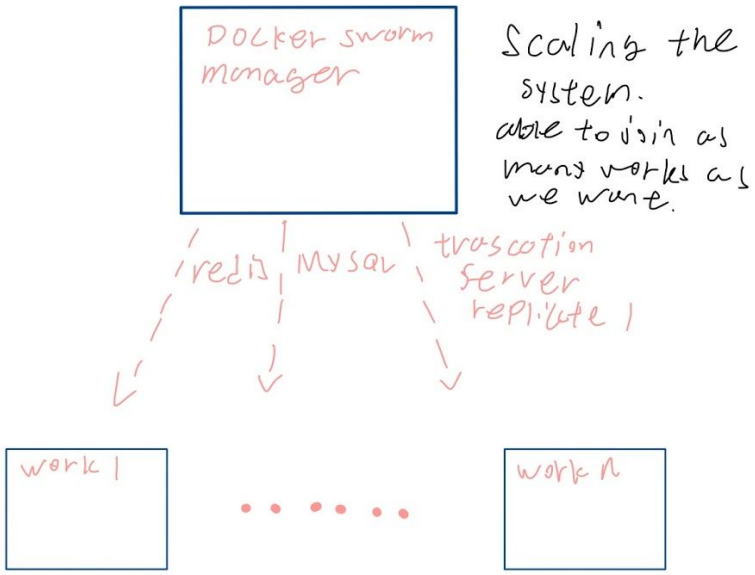


Figure 8. Docker Swarm Structure.

Security Design

Database:

In the system, we use the second DB to back up the main BD's data, if the main DB is down and the second DB will take the job and store the data or respond to the transaction server until the main DB server is back to work. Once the main database is ready to work it will recover the data from the second DB, so it will not lose any data after the DB shutdown service. In the real world, two databases both shutdown is a low probability, so we think it is a much safer design than just using one single database.

Also, the database only connects to the transaction server and no other server, so we set the limit on the database only the transaction server is able to read and modify the data from our database. It is not 100% safe the back-end security if the hacker hack into our system and modify our data or copy our dat, but this course is not a security course and our task is not to make the data 100% safe, but this is a good idea for future studies to make the data transfer secure by use data encryption.

Docker Swarm:

In the system design, we use the docker swarm model, and as we know the docker has its own security protection, and this will help us do a lot of work on making the virtual machine or containers how they talk safely. Also, the docker swarm has auto load-balancing and this also helps us to monitor and control the machine, not to memory lack or buffer overflow.

Thread Pool:

We use the thread pool to control the maximum thread we allow to use and because there is some problem on the XML file crypto key and the database maximum allows connections, so we must limit the threads. Using the thread pool will help us to control the threads and also help us save the memory on start and close the threads.

Redis Connections Pool:

This is the same as the thread pool when we need to use the cache data and we will need to request a connector from the connections pool, and this helps users save the time on making a new connection to the Redis cache server.

RISKS

	Problem	Impact	Possible Solution
Data transmission	We transmit data between servers without encryption	Potential loss of security since data are easy to be read	Add more security features when we send the data
MySQL Database	MySQL is not a safe database for storing datas.	Data can be stored wrong and type error, so it may cause our system data to be lost.	Try to use PostgreSQL or use an online Database like MongoDB etc.

TEST PLAN AND DOCUMENTATION

Procedure

The steps to construct the final system are as follows:

- Construct a working web server that can extract all of the user commands from a workload file.
- Construct a working transaction server that can process all of the required commands.
- Construct a working audit server that can record every transaction made by every user into an XML formatted file.
- Store all information in a database for all other servers to access.
- Scale the system.
- Constantly test for errors.
- Speed up the system while keeping correctness.

The system's capability is tested with the following files:

1. 1 User command file
2. 10 User command file
3. 45 User command file
4. 100 User command file

5. 1000 User command file

Executing through each of these test files helps us build and scale the system step by step. Each time more users are added in, there have been more scaling issues to deal with which solving those enhances our system progressively.

Scaling Issues & Solutions

With a table, we will address different issues and solutions as we scale the system to work with more users.

Number of users	Issues	Solutions
1 User	Since this is the very early stage, minor issues happened when we were trying to get the XML output correctly.	In this stage, we did not bother with making any error events for our output XML file. Since there weren't enough test cases, all we needed to do was format the XML correctly. We thought invalid commands shouldn't be recorded but ended up worth keeping track of.
10 Users	Exposed to multiple errors starting in this stage when tried to send long messages between servers.	To fix the long messages error we add an end message at the end of the data string because the TCP connection guarantees the data will be received in order, so if the receiver once got the end message which means this is the end and it will break the loop continue the next task.
45 Users	Less error with our initial approach, more user meaning each transaction sends less. The database may process more time because the number of users is getting larger.	To fix this issue, we did two things; first, we did database turning and try to use the simple SQL query, second, I implement the Redis as the cache and it will help to store the temporary data, once the user requests the same data and it will go to the cache first then go to the database.
100 Users	More users and more transactions per user lead to more burden on the transaction server. The Quote commands getting more, and it spends 50% of the total run time.	In this case, we add a thread to process the job tasks, in the beginning, we did implement the threads into the system, so we use the thread pool and it makes our system allow process the job tasks concurrently.
1000 Users	In this test case, we need to scale the system to several machines, and once we scale the system we find there is a limit on the Quote Server it does not allow us to concurrently hit the quote server.	In this test case, we did not have time to go through and fix the problem because the COVID-19 lab is closed. We consider there may be a problem with the java string encryption, and in our system, we are using the Python 3 language, so that may cause that problem. To temporarily fix this problem we decide not to log the crypto key because the tester depends

		on our XML file, but we do finish and run the whole test case.

CAPACITY PLANNING MEASUREMENTS & ANALYSIS

1 User:

The testing result on the 1 user if we use one transaction server and it will finish in 3 seconds because in the 1 user test case we did not see that much Quote commands, so it will finish very quickly, and the database response also very fast. In this test case only is used for testing the system basic functionality, and the result data is not useful to analyze the performance.

10 Users:

In this test case, the used commands are up to 10,000, and it is useful for testing if the system is able to handle multiple users are not. In our design system with one transaction server, the average result is about 100 TPC. We test it several times and get a similar result.

45 Users:

In this test case, the user commands still 10,000, and the database we did a tuning and also we implement the Redis server as cache, so it makes our system's performance keep 100 TPS.

100 Users:

In this case, the user commands are up to 100,000, so the database is getting slower. In this test case, we add two version code because in the next level 1000 users may contain the wrong user commands, so we add some more checking conditions into the code. The result without the error command checking the performance is about 80 TPS, and the result with error command checking is dropped down to 40 TPS. This result is tested in one transaction server, so if we scale the system it will be increased, and it depends on how much machine we can use, and also how many memory and CPU processes we allow to use. If we make 10 replicates of the transaction server and distribute it on 10 machines and the result will at least 10 increase 10 times, which is 40 * 10 = 400 TPS.

In this case, we also find out there is a crypto key error once we submit our logs

file, so we must set the limit on the threads on each transaction server. If there isn't a limit on the threads we believe this result will be much higher than what we have right now.

1000 Users:

In this case, the total user commands are up to 1,000,000, and this is very large testing data, so we must scale our system and make replicates of our transaction server into several worker machines by using the docker swarm. Unfortunately, we did not have enough time to finish the testing of 1000 users because the COVID-19 the Lab is closed, and we are no longer able to access the lab and run our test. But we did run once the test case, but it is not enough for analysis of the result, and also there is a crypto key error problem, and we must remove the crypto key from XML logs file, and that makes our result not useful for analysis.

In this case, we can not give a precise result, so we calculate out an estimated result if we use 10 machines and with the thread limit and the performance will be 35 TPS per transaction server, and if take off the limit and then it will increase to at least 60 TPS which the total is 60 * 10 = 600 TPS. To complete the whole task will take about 1667 seconds which is about 27 minutes. If we keep increasing the transaction server replicates to 20 or 30 machines, and the time will only need 14 minutes for 20 replicates, and 10 minutes for 30 replicates. Those in the estimation of our system, and in order to get that performance need to solve many technical problems, but it is possible to get that performance on our system.

Upper Bound of Users:

In our lab we only have about 30 computers, and our Quote Sever is fixed, so we are not allowed to modify it and its maximum limit is handled 1000 hits per second. If we need to keep a good performance, the maximum bound on our system is about 2000 users with 1,500,000 users. We did an estimation on the performance if we want to keep 60 TPS per transaction server and the maximum computer we allow to use in the lab is about 30 machines. The result will be 60 * 30 = 1800 TPS, 1,500,000 commands will take about 14 minutes and this is the best case estimation. The worst-case is we are only able to run 10 machines and the processing time will be 42 minutes, which is too long. So, if we want to get the average time we combine the best-cast and worst-cast together and the result will be 28 minutes, which still needs half-hour to wait for the result.

FAULT TOLERANCE ANALYSIS AND DOCUMENTATION

Our system is designed to handle a number of errors without crashing. Here we will discuss some potential problems and our solution to them.

1. Input errors: We never expect users to always type their input in the correct format accepted by the system. That is why we make error handling to prevent our server from shutting down due to these simple problems. The issues we have planned ahead consist of user commands that are incomplete or with wrong value, users making a purchase or sale with insufficient inventory such as funds or number of shares on the account. For any errors that happen at the user end mentioned above, we have created functions to output the “error event” XML messages when errors are spotted. In the original XML output, we exclude whatever is incorrect or missing, this way our final output XML will not trigger an error in validation.
2. System errors: To satisfy security and availability, we have to reduce the amount of downtime that can possibly happen to our system. Maintaining all of our servers’ functionality requires alternative servers in case the main servers fail. To back up our system, we decided to use docker swarm’s unique feature that enables running multiple servers in parallel. Distributing the workloads to multiple transaction servers puts less burden onto each server which greatly reduces the chance of overloading a server with threads running concurrently. By duplicating the database, security is improved by having working storage for data even when one of the databases crashes, this is also applied to the audit server and web server for security and availability purposes.
3. Connection error: TCP is known to be reliable, that is why we used it to transport data between servers. In this project, we encountered connection problems that we did not anticipate when we were using TCP. Because there is an end to end connection, we sacrificed efficiency for correctness. Because of this feature, our system is vulnerable when one of the connection endpoints is blocked off, an example would be if the quote server is down, our entire system goes down and will not function until the quote server starts working again. Also, when there are too many connections made with the quote server at the same time, the quote server will return false crypto keys back to its client. There is no useful solution for when the quote server malfunctions, which means the nominal TPS of our system becomes much less than the potential TPS with the quote server

functioning as it should.

PERFORMANCE EVALUATION & DISCUSSION

Performance of Each Commands

Trasaction	AVG Time (ms)
ADD	11.56
Quote	274.78
BUY	23.54
COMMIT_BUY	212.89
CANCEL_BUY	32.98
SELL	37.84
COMMIT_SELL	142.63
CANCEL_SELL	45.56
SET_BUY_AMOUNT	37.9
CANCEL_SET_BUY	64.67
SET_BUY_TRIGGER	13.34
SET_SELL_AMOUNT	44.37
SET_SELL_TRIGGER	15.21
CANCEL_SET_SELL	71.82
DUMPLOG(USER)	24.53
DUMPLOG	52.97
DISPLAY_SUMMARY	48.93

Figure 9. The Performance of Each Commands.

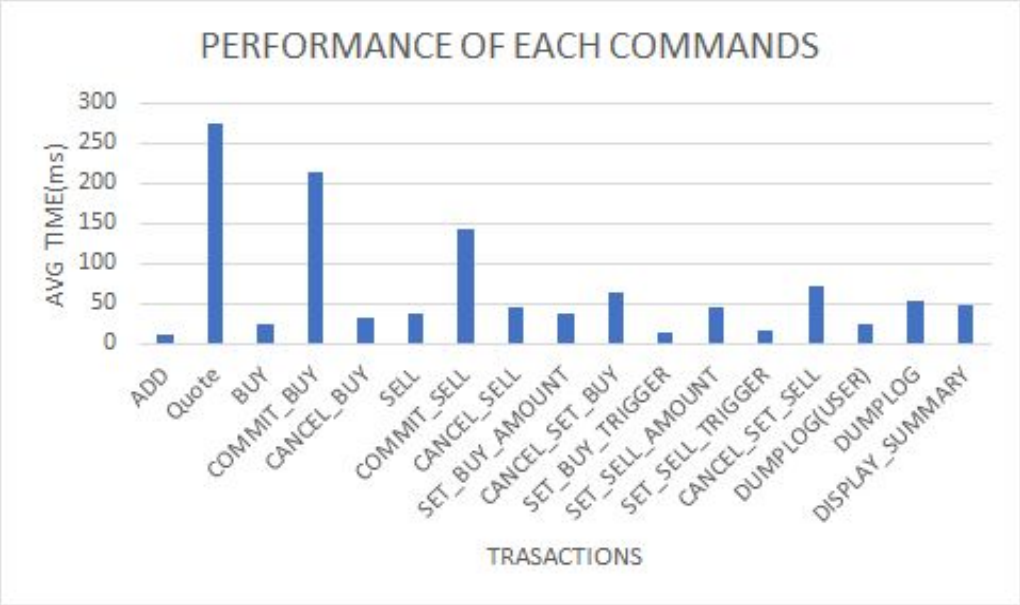


Figure 10. Bar Graph: The Performance of Each Commands.

There were many changes on the code in our system to improve the performance; knowing which section of code is taking the most time to run is crucial. We used APM on our code to find parts that are taking a long time and digged in further to enhance the quality of the code. Overall, time spent on the commands that depend on the quote server such as commit buys or sells, and quotes have a high instability due to the wait time for the quote server to respond back. Commands that constantly check the quote of a stock with a trigger already set up will run in the background which does not impact performance by a lot, although it does put a burden on the limited resources given to the system. Commands such as cancel set buy or sell that require searching from the database to find exactly what to remove takes slightly longer time than just storing data. Tuning the database and using Redis helped to cut off some of the time spent searching the database but more issues are about optimizing the code that deals with the quote server. Taking into account that commands must be executed in sequential order limits the system's ability to process more commands in a shorter time, if we did not wait for commands that take a longer time to run, that would be really efficient but would no longer guarantee correctness. We did not find a solution to keep correctness while letting slower commands run in the background. We could also have distributed the users to more servers on more than 10 machines to speed up the whole system, but we did not bother with that as we can get a rough estimate of the speed-up of doing so, having that there will not be any unexpected errors as we distribute the system to more machines.

45 Users - Performance of Threads vs Times

Threads	Time (s)
1	1250
10	714
20	425
30	248
40	115

Figure 11. The Performance of threads vs time in 45 users.

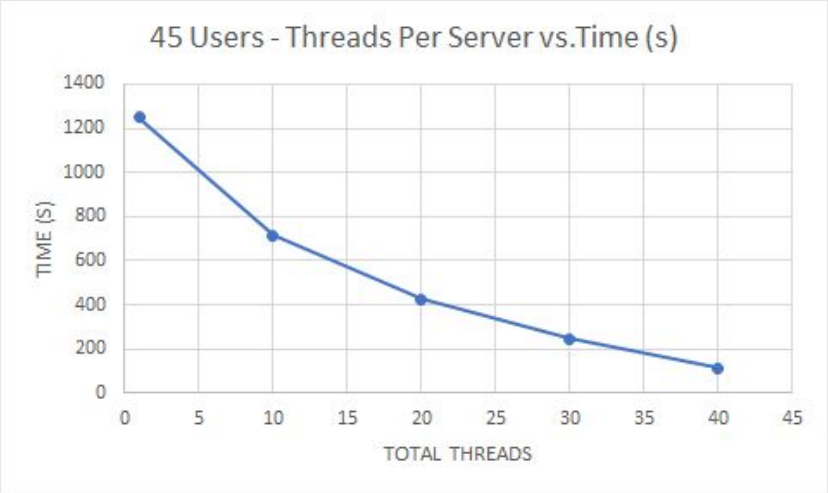


Figure 12. Line Graph: The Performance of threads vs time in 45 users.

It was unfortunate that COVID-19 stopped our team from being able to continue testing out other users' cases. In this stage we only had time to use the 45 users file to analyze the data. We only used one server to test out the 45 users, based on our result we found that once we increase the number of threads used to process the file, the amount of time the system needs to finish the process is reduced. However the speed up per thread seems to drop down once we hit a certain number of users. For example, 1 user the the performance of per threads is about 8 TPS, and 10 users per thread is dropped to 1.4 TPS, but the total is 14 TPS, so the overall performance is still increased to 14 TPC which is faster than 1 thread. The reason for the TPS drop may be caused by the memory of the machine or CPU being shared by more threads, so if we increase the memory or CPU the speed will be different. Another reason is we did not use docker containers to test this data, so the memory may be spent on other places in the computer environment. In the line graph, we also can see the slope is getting flat which means increasing the number of threads will eventually stop helping to speed up the system. In other words, the TPS for our system has an upper bound and it will not increase anymore even if we keep increasing the threads. To further increase the performance, it depends on how much resources we can utilize in the machine.

Results

Users	AVG TPS (Single Server)	AVG TPS (2 Servers)	AVG TPS (5 Servers)
1 User	88	88	N/A
10 Users	96	154	N/A
45 Users	87	115	367
100 Users	45	64	274
1000 Users	35	N/A	N/A

Figure 13. The Result of scaling the system.

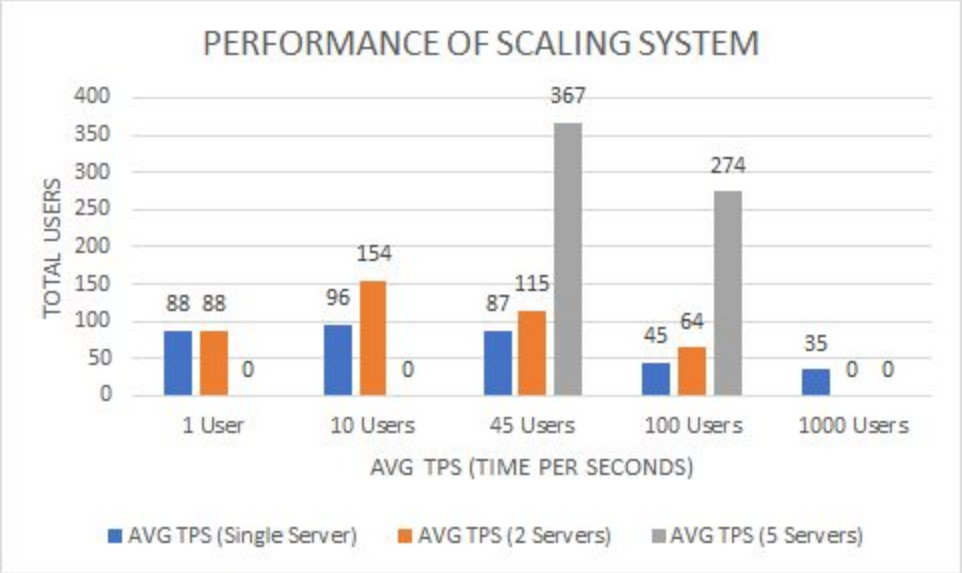


Figure 14. Bar Graph: The Result of scaling the system.

We tried to run the 1000 user workload file with more servers but there seem to be too many connections to the quote server and database at the same time that goes out of control and causes errors that we did not have enough time to fix. If we could fix the problems instead of forcing one transaction server to handle all of the commands, the TPS we expect for the 1000 users workload file will be in the 200 to 400 range, as well as some increase in the numbers for the other workload files.

CONCLUSION

Overall, working on this project helped us understand the interaction in an application and the potential problem that could occur in any scalable system. Scaling a system was not an easy job, it took a lot of time testing and reworking parts of the system to find all the possible ways to get our system to work in the way we hope it does. As technology continues to advance, more and more software that involves software scalability are going to appear into our day to day life. Nowadays, games, online communities, and applications that involve a massive number of users being online at the same time are becoming common things to see, as we have moved into the information age. It is definitely a valuable skill to know how distributed systems and scalability works for being a software engineer, unless people stop using the internet which is highly unlikely for a very long time. As we carefully worked to deliver this Day Trading System, there were a lot of interesting and annoying moments that were remarkable experiences to gain. In the end, Covid-19 was a major factor that has affected our outcome to be incomplete. However, it has also been a time that the use of the internet grew tremendously world wide due to social distancing and quarentein. Come to think of that, even when we did not have time to do all of the testing to satisfy our regrets, knowing this skill's usefulness is going to push us to improve after completing this course. If we get a chance in the future, there is definitely going to be a harder problem to be solved with better results.

REFERENCES

[1] Project Web Site URL: https://www.ece.uvic.ca/~seng468/Body.shtml

[2] User Commands URL:
https://www.ece.uvic.ca/~seng468/ProjectWebSite/Commands.html

[3] Docker Documentation URL: https://docs.docker.com/

[4] MySQL Documentation URL: https://dev.mysql.com/doc/

[5] Redis Documentation URL: https://redis.io/documentation

[6] Redis Commands URL: https://redis.io/commands

[7] Python3 Documentation URL: https://docs.python.org/3/

APPENDICES

Project GitHub Link: https://github.com/zijian-chen96/seng468