**Seng468: Final Project Report**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**<REDACTED>– <REDACTED>– James Ryan – <REDACTED>**

Table of Contents

[**1.0 Introduction 3**](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.yhxe316ox8eb)

[**2.0 Project Plan 3**](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.u5mx7p1249ni)

[2.2 Assigned Responsibilities 4](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.b96fiuvggxyf)

[2.3 Work Effort 5](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.b58o9ze6o4s3)

[James Ryan 5](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.6gsxovxo8l1d)

[Shawn Dhaliwal 5](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.20q1bhokl86q)

[Max Gunton 5](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.k1e6kwq0exyd)

[Callum Thomas 6](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.jif4nsq9wg39)

[Group Effort 6](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.rhjg18ej6m0k)

[**3.0 Design 6**](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.u1eezbc4l1ce)

[3.1 Initial Design 6](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.5182jyoo9mdh)

[Design Decisions 7](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.uwk27rrxs07r)

[Transaction Server 7](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.l5i1bay0tx72)

[Load Balancer 7](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.720vr1h8ijh)

[Web Server 8](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.iiwon7auwrwe)

[Database 8](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.ihxo0oi0pim4)

[Client 8](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.jivtyv5etyhh)

[Quote Server 8](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.fffkfv663d7f)

[3.2 Final Design 9](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.yn2zsi41xpkt)

[Design Decisions 10](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.4igbbu7tfjn1)

[Tx Server 10](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.d7kwj21bb7zr)

[Load Balancer 10](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.f9g1c7btk4b1)

[Web Server 11](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.jqwhhfreco38)

[Database 12](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.6w1s5k9syj79)

[Client 12](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.7nra857ca1rv)

[Quote Server 12](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.6ed2vmokqp90)

[**4.0 Analysis of Quality Attributes 13**](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.6o1gr7jlkjx3)

[4.1 Documentation 13](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.8dap3434p3h8)

[4.2 Security Design 13](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.4i0mcdbcxkkd)

[4.3 Testing 14](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.1u76ru4hzeem)

[Future Unit Testing 15](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.hj9hgzf2tqm3)

[Future Integration Testing 15](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.keuge6fmv9tc)

[4.4 Capacity Planning 15](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.hrcgbq2k7ib1)

[4.5 Fault Tolerance 17](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.djqo8dixlwoi)

[4.6 Performance 18](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.j7d46sx46qjz)

[**5.0 Conclusion 20**](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.j0j3abfw6q2b)

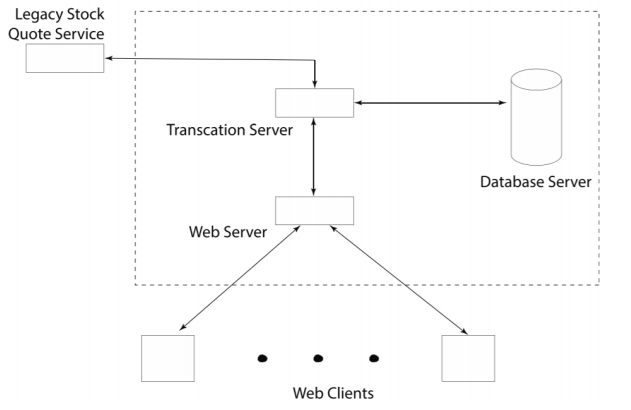
[**6.0 References 21**](https://docs.google.com/document/d/1TiUAF_gCkQjXO4w7KccVRx7te-QtzoIAR_4ilW4jfb0/edit#heading=h.mx1kk3z2v6md)

# **1.0 Introduction**

DayTradingInc. has assigned us to build a stock market trading system, in which users can add money, buy and sell stocks, and get quotes on stocks. This report describes the architecture that our team created to accomplish these requirements, as well as how our system performs, recovers from crashes, and security features. First, the report will go over our project plan, i.e. the assigned responsibilities of each team member, then their contributions throughout the term. Next, our initial, then final architectures are analyzed for the design decisions made during their creation. Finally, the quality attributes (security, performance, fault tolerance, and others) are examined within the system. To conclude, our project had a very high tps of ~27,000, however other quality attributes suffered such as fault tolerance and security.

# **2.0 Project Plan**

This section outlines how we assigned project responsibilities to group members, then how the work was actually completed. We examined the given initial architecture diagram [Figure 1.] and divided the workload based on group member expertise.



[Figure 1: <https://www.ece.uvic.ca/~seng468/CourseProject.pdf>]

## **2.2 Assigned Responsibilities**

At the beginning of the project, we divided the project workload into two categories: front-end and back-end. The reasoning behind this was to exploit the strengths of our group members. Callum Thomas and Shawn Dhaliwal were assigned to the front-end portion because they both had more experience in that area. James Ryan and Max Gunton were assigned to work on the back-end portion, also due to their experience. The front-end involved the web client and web server, as well as creating a random workload generator. The back-end involved the transaction server and the database.

## **2.3 Work Effort**

Next, given the assigned responsibilities we outline the work each group member did during the project.

### **James Ryan**

James built the initial blueprint for the transaction server, which was then primarily developed by **<REDACTED>**. James also wrote the initial project documentation for the first project deliverable, which is the Project Plan word document included in the documentation folder of the repository. James acted as a consultant for much of the project, advising on design decisions or changes to the architecture. The overall project documentation was created by James, with assistance from each of the other group members.

**<REDACTED>**

Handled the setup of Docker and Docker-Compose, along with briefing the other group members. **<REDACTED>** did the same for the use of HAProxy as the load balancer between clients and the transaction servers. **<REDACTED>**and **<REDACTED>**both codeveloped the front-end system of our platform including the web server and web clients. Shawn also acted as a consultant during much of the project.

**<REDACTED>**

**<REDACTED>**performed continual development on the transaction server throughout the project. The databases were also built by **<REDACTED>**. Most of the transaction server functionality was built by **<REDACTED>**, with some help from all other members of the group. Additionally, a significant amount of our performance improvement was the result of **<REDACTED>** considerable amount of work on the transaction servers and databases.

**<REDACTED>**

**<REDACTED>** built the web server for the front-end web interface of the system, a random workload generator for testing, and co-wrote the final project report using documentation created by James Ryan.

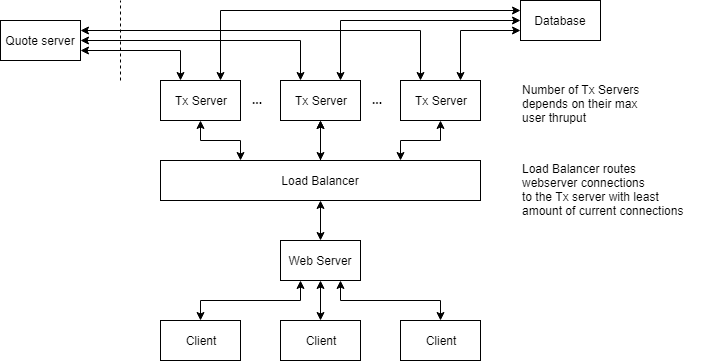
### **Group Effort**

Throughout the project, our group submitted logs to the project logging site to track how we spent our time.

# **3.0 Design**

## **3.1 Initial Design**

This section outlines our initial design, after viewing figure 1. given to us by DayTradingInc. Below [Figure 2.] is the diagram we created, revising the initial architecture shown in figure 1.



[Figure 2.]

### **Design Decisions**

#### Transaction Server

The transaction server (Tx Server). We decided to put all the business logic of the system into containerized transaction servers running in parallel. Each transaction server would be load balanced by a load balancer to increase performance. We decided to use Python to develop this component because our group members were well versed in it, allowing us to start on the project promptly.

#### Load Balancer

We decided to place a load balancer between the web server and the transaction server. Due to a large number of users and commands, the transaction servers would be taking heavy loads, therefore they need to be load balanced to improve performance. We chose HAProxy as the load balancer after researching options online. We found that HAProxy was the easiest to configure and use.

#### Web Server

The web server now should send commands from clients to the load balancer, which distributes the commands to the transaction servers. We left the web server as one containerized application because the final web interface only needed to be used by one user according to the project marking guidelines. Decided to use Node.js to develop the web server because of experience, and suitability for a simple web server.

#### Database

Our database needed to handle both users and logs in our initial design. We chose MySQL to implement this function because of experience and ease of use.

#### Client

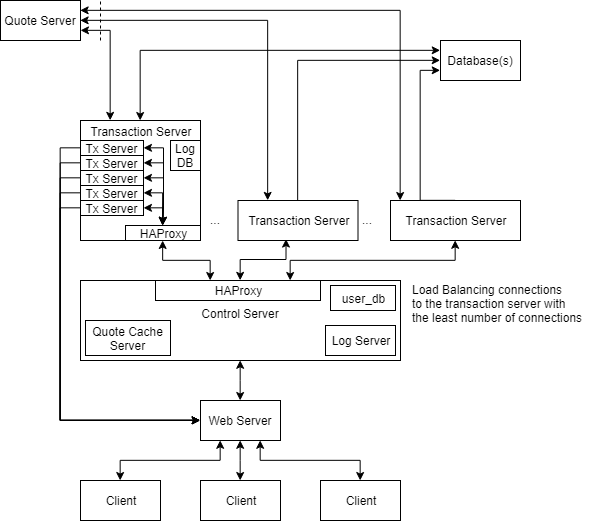
Our initial design for the client was a simple HTML5 webpage that used JavaScript to connect to the web server.

#### Quote Server

To connect to the quote server, we decided to use Python socket API. The socket library in Python is easy to use and familiar to us due to taking other courses such as CSC 361: Computer Networks.

## **3.2 Final Design**

After working on the project for a couple of months, we settled on a final architecture that most accurately reflects the working system that we presented at the Demo on April 5th, 2019. Below is a diagram [Figure 3.] showing the final architecture.



[Figure 3.]

### **Design Decisions**

#### Tx Server

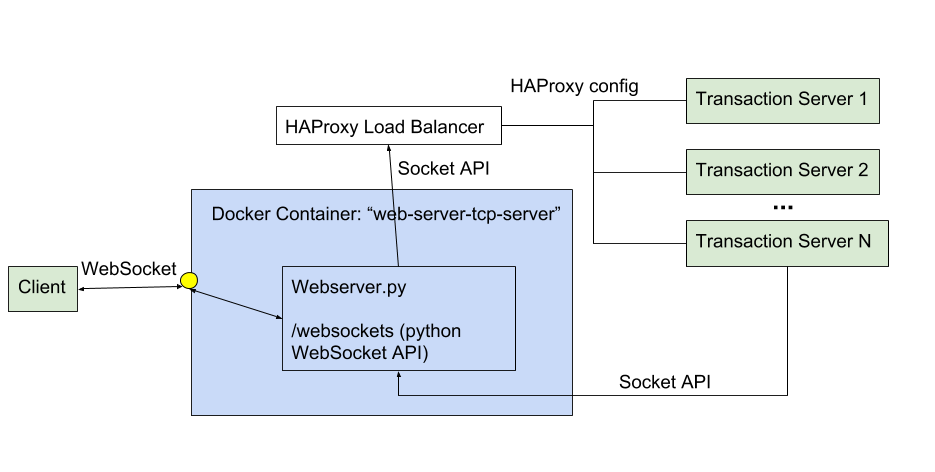
From the initial design, the transaction servers core functionality and architecture design did not change in any drastic way. However, there were some architecture changes made to them which altered their final functionality from our initial plan. One such change was how the transaction servers would cache stock price quotes. We originally did not plan to have cache sharing between each transaction server, as we had thought the overhead was not going to be worth the benefit. However, that was not at all correct and our performance had a significant increase when transaction servers had the ability to pull from their own local cache along with a cache shared among all other transaction servers. The decision to implement this also required a choice on how to handle the correctness of the global cache. When a transaction server was to update their local cache, they sent a message to the global cache server to update the cached quote of a particular stock. We decided that if another transaction server were to request the cached value before a more recent one has fully taken effect, that the old quote would be used. This has the downside that if the old cached quote were expired a new quote from the quote server would need to be made. However, this has the benefit of lower overhead in the global cache server.

#### Load Balancer

In our final architecture, we still used HAProxy, however, we did add to the configuration some features. First is that our HAProxy server, which we ran in a Docker container, now checked each server in its list of possible servers before connecting. This allowed us to add 50-100 servers in the configuration and still run the server even if only 10 actual transaction servers are running.

#### Web Server

In the final architecture, the web server has gone through some changes. First, the language changed from JavaScript (Node.js) to Python. This was done to simplify build time and speed up development time before the project demo. Due to the constraint of running our system in Lab A321, re-building a Node application every time a computer was taken by another group was wasting time installing instead of developing. Instead, we used Python 3 because Python 3 was already installed on all the lab computers, and the language matches the transaction server, making it easier to connect to the transaction servers via socket. The web server accepts client connections, forwards commands from the client to the transaction server (via the load balancer), receives output from the transaction server, then sends that output back to the original client. A diagram below [Figure 4.] shows this design in detail.



[Figure 4.]

The connection from the client to the web server was implemented using WebSocket, an HTML5 API for connecting from client to back-end server via sockets. The design choice to connect the transaction server directly back to the web server, rather than back through HAProxy, was made to simplify and speed up the connection between the web server and transaction server. The web server has to send its IP address and port to HAProxy, which distributes that information to the transaction servers, selecting the one with the least connections, which establishes a connection from the web server to a transaction server. The web server listens to this connection and forwards output from the transaction servers back to the original client via the established web socket connection. This direct connection allows for fast feedback to the client.

#### Database

Our database needed to handle both users and logs in our initial design. We chose MySQL to implement this function because of past experience and ease of use.

#### Client

Our initial design for the client was a simple HTML5 webpage that used JavaScript to connect to the web server. As we further analyzed the project requirements we realized the utilizing a front-end framework would make its development a lot more efficient.

#### Quote Server

To connect to the quote server, we decided to use the Python socket API. The socket library in Python is easy to use and familiar to us due to taking other courses such as CSC 361: Computer Networks.

# **4.0 Analysis of Quality Attributes**

## **4.1 Documentation**

Our documentation can be found in the “Documentation” folder in our GitHub repository for the project at: <https://github.com/Deniablesummer/seng468-spring2019/tree/demo/Documentation>. Each document contains documentation specific to its title, i.e. security, performance, fault tolerance, etc.

## **4.2 Security Design**

The current system has very little security design implemented. Of course, this would be unacceptable for a real stock trading system. However, here in this section, we identify areas that could be made secure if this were a real-world project.

The first and most obvious security design fix would be to install an SSL certificate within the front-end components in order to upgrade both the WebSocket connection and the HTTP connection. With an SSL certificate, the WebSocket connection would be made WebSocket Secure (wss:// instead of ws://) providing much more security as data going through that socket would be encrypted. The same goes for upgrading HTTP to HTTPS, preventing data from being breached over the connection through a man-in-the-middle attack [1].

Our database, running in MySQL, can also be upgraded for security. We could purchase the enterprise version of MySQL to gain its “industry standard functionality for asymmetric encryption…”[2]. This includes digital signing tools to ensure that the data has not been corrupted in transit (RSA, DSA), and validating data authenticity. All of these would ensure that the database adheres to security standards.

Finally, given that our transaction servers, and the whole system, run in the lab, it is already protected by Uvic’s security. You cannot access the system from outside the lab unless you set up a port forwarding system, which can easily be protected with secret key authentication. In the real world, you would probably do something similar, not allowing access to any part of the system except through trusted channels.

## **4.3 Testing**

Throughout the project, we tested our system using the provided workload files. To verify the results, we manually tested running containers by ssh’ing into them and checking their status. For example, when testing user data consistency, we manually ssh’ed into the Database containers and checked their tables using SQL statements, such as SELECT \* FROM USERS and analyzing the returned output to make sure the user data was correct based on the given workload file. In a real-world system, more in-depth testing would be needed to ensure correct functionality.

Once the web interface was developed, it was easier to test the system as a whole as you could do a series of commands and observe the state by displaying a summary of the user. While developing the web interface and transaction server in parallel, a simple “smoke test” could be performed after every change to ensure the system was still working. For example, a smoke test would be to sign in as a new user, add money to their account, then buy and sell a stock. Throughout this test, the values in “display summary” would be checked to ensure the system was functioning correctly.

### **Future Unit Testing**

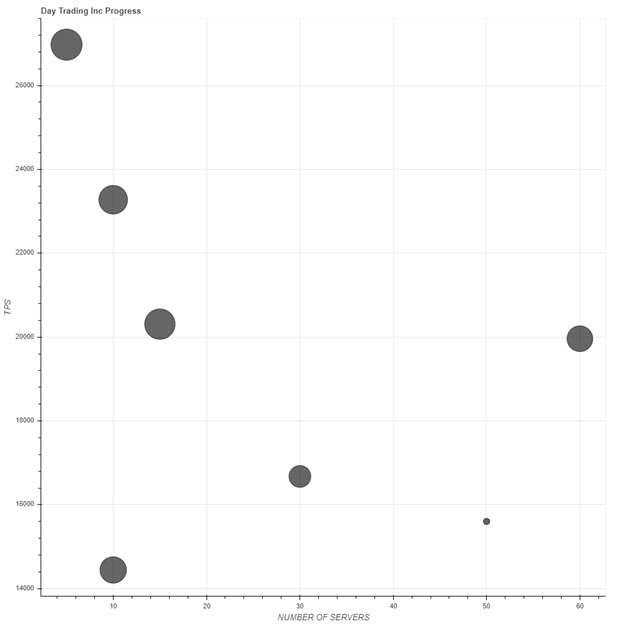
Each component of the application would need a set of unit tests to ensure proper functionality of it. Ideally, these unit tests would be part of a larger automated test suite which would be run throughout future development to ensure that changes to the component do not produce any unexpected changes to the functionality and outcome of the component. Each individual unit test must have the ability to be run independently to the other tests in the test suite for the component.

### **Future Integration Testing**

A testing suite which ensures that each component properly integrates with the other components which it interacts with, and to ensure proper functionality of the entire system.

## **4.4 Capacity Planning**

For analyzing the capacity of our system, we have been testing using workloads with different amounts of users. We found that the hardware our servers were deployed on affected the performance of the system, allowing us to extrapolate how the system will perform with different numbers of users. Our transaction servers are very multithreading heavy, therefore if we deploy on machines with more CPU cores, and more threads, we see better performance. Below is a graph comparing number of servers per machine to tps to understand how our system performs with different loads of users.



[Figure 5.]

The best performance we gained was when we ran 1200 users using one transaction server on one machine. One reason for this is that because of the multi-threading intensive design of our transaction server, when introducing an addition server on a machine in the lab environment the CPU scheduling becomes too much, and bottle necks the performance of both transaction servers.

We also found that too many servers spread over too many physical machines caused too much network overhead when used for even the 1200 user workloads. Future development and testing would be necessary to find the sweet spot of servers to machines to user count of our system.

Further testing would need to be done to narrow down the specifics of how many transaction servers and physical machines would be necessary for a given workload throughout the day. Using that information, we could then configure docker-compose to run specific numbers of transaction servers for a given period where we predict higher or lower usage, such as at market open or close.

Using this information, we can extrapolate that our system would perform reasonably well under higher user loads. We are not sure what the upper bound is for the number of users, but we can see that more testing is necessary to find it.

## **4.5 Fault Tolerance**

Our system, put gently, focuses more on performance than fault tolerance. However, we have implemented a couple of features to increase the ability of our system to sustain crashes and failures.

Through the design of our system, the web server can survive the crash of its connected transaction server. To accomplish this, the web server, in the event that the transaction server it is connected to crashes, will automatically reconnect to HAProxy, which will then return the address of the least connected to transaction server that is available. To do this, the HAProxy configuration is set up to check each server in its list of possible transaction servers for whether the transaction server is available or not. Then it returns an available server to the web server, which re-establishes a new connection - without the client knowing or being affected by this.

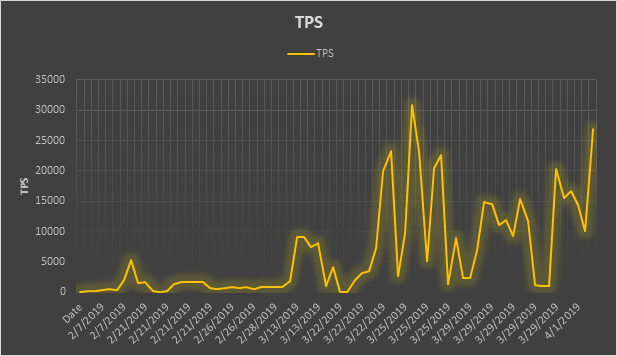
While running the system, having more transaction servers allows for better fault tolerance as there is less chance of the whole system collapsing. By load balancing with HAProxy, this is accomplished.

Another fault-tolerant design choice would be to set the “restart” flag within our Dockerfiles to restart services that crash automatically. This would ensure that components are not down for long periods of time.

To improve the system, we do have to address the fact that the logs are being stored on each individual server’s local hardware, rather than a distributed cloud database. In the event of a hardware failure, the logs stored on that device would be lost, unless we had backups of each server’s hard drive, which would be very costly and not distributed. To improve this, we need to put the logs into a separate database that is distributed in a Docker container, however, this will impact performance. We found during development that logging every transaction was our most significant bottleneck for achieving faster transactions per second.

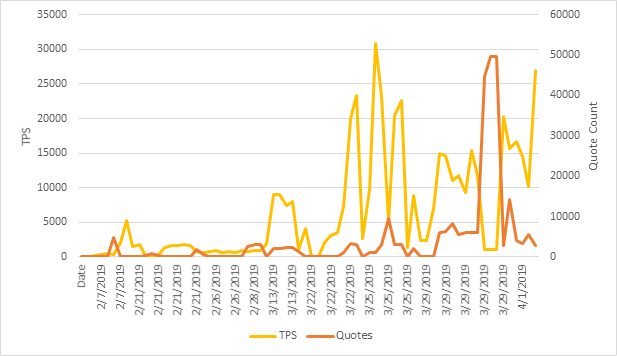
## **4.6 Performance**

Analysis of our project’s performance was gained using the provided user workload files, along with randomly generated workloads which we created using a workload generation script. These workload files were run using our system by a workload tester, and the resulting log file was submitted to the course website which provided verification of correct functionality and a Transactions Per Second (TPS) metric. This metric is what we used to assess our systems overall performance. The following chart shows the TPS of our system over time. The highest TPS time we attained was around 27,000.



[Figure 6.]

As seen above, the performance of our system improved over time with drastic variations. The variations over time typically resulted from some sort of configuration change, good or bad. As seen in the next chart, several the dips in TPS were of a result of some change to our configuration or transaction server which caused an increase in quotes to the quote server. From our understanding of computer networking, this is because of the overhead and delay caused by sending information of a network. Because of this, a recommendation we might pose if this were a product is to have our transaction servers co-located with the quote server(s), or as physically close to the quote server(s) as we could get them. This would be to reduce as much of the network delay as we can, by eliminating the need to travel over a network or by reducing the number of nodes the quote information needs travel through.



[Figure 7.]

# **5.0 Conclusion**

To conclude, our system performs well but needs some more work to implement better security, fault tolerance, and testing. We identified many concrete examples on how to improve in our analysis, such as using Enterprise MySQL for better security and developing unit/integration testing. Each group member was assigned to work on either the front-end or the back-end of the system, and those responsibilities did not change much during the project. In total, this project has allowed us to learn about how distributed systems work, the benefits and difficulties of such a system, and how to turn a list of requirements into a working system.

# **6.0 References**

[1] <https://www.computerworld.com/article/3180690/its-time-to-turn-on-https-the-benefits-are-well-worth-the-effort.html>

[2] <https://www.mysql.com/products/enterprise/encryption.html>