Prototype Trading System

DayTrading Inc.

University of Victoria
SENG 468
Team Kitkat (Group 2)

Allan Liu: V00806981
Parm Johal: V00787710
Mathieu: V00872174

Table of Contents

Architecture

The basic structure of the system includes the workload generator, load balancer, web server, transaction server, trigger server, transaction database, and quote server cache. For this project, the workload generator replaces the web client to simulate multiple transactions. The workload generator takes workload files as input and distributes the formatted data directly into the web server. The web server will ensure that the inputs taken from the workload generator are valid and have correct parameters before transmitting it to the transaction server via sockets. The transaction server will process the requests by accessing the trigger server, transaction database, and quote server cache when needed before sending the response back to the web server.

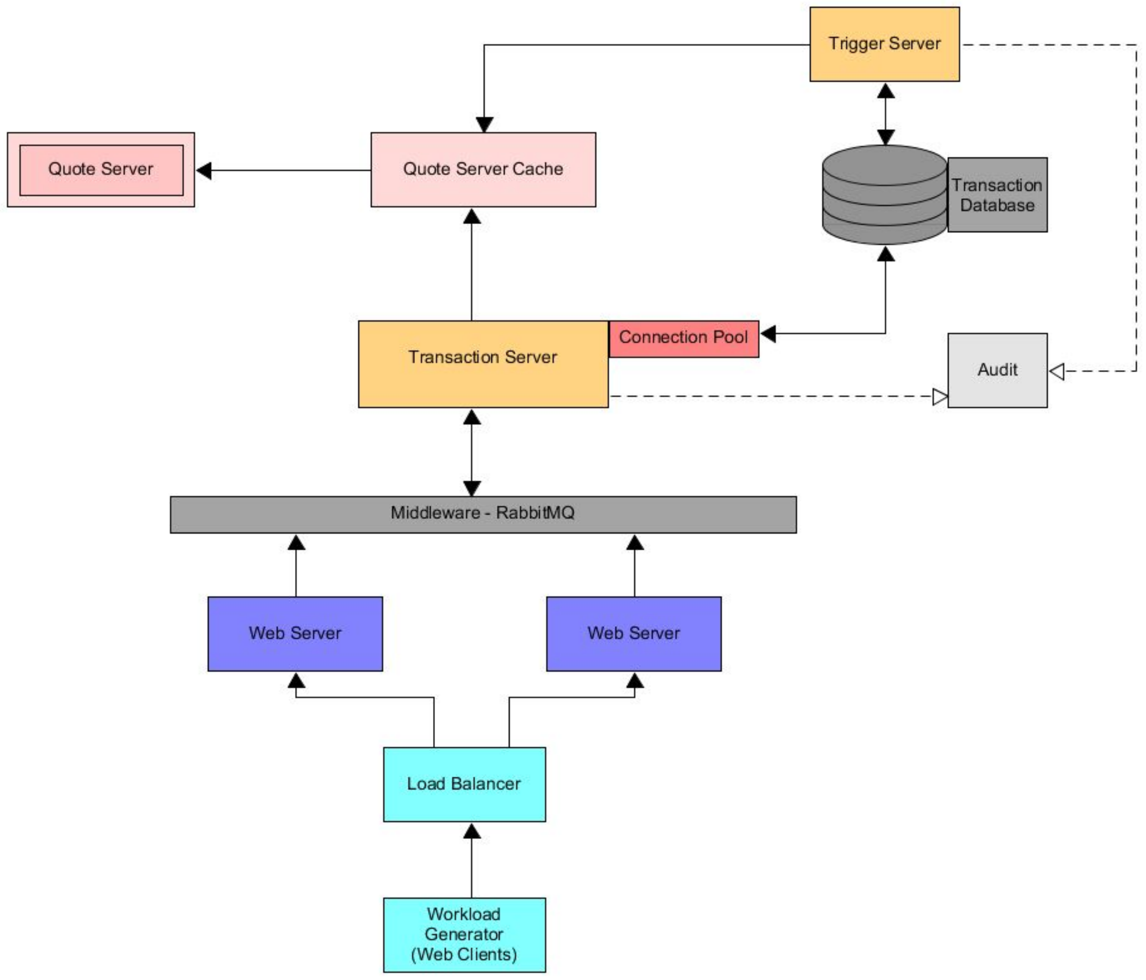


Figure 1. Architecture of the Distributed System

Workload Generator and Load Balancer

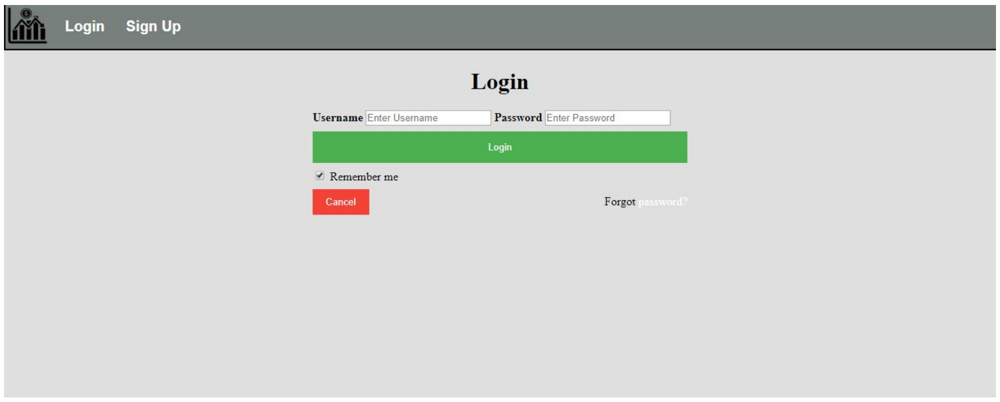
The workload generator is written in python and parses different workload files before sending requests to the web server. Our goal for the workload generator is to parse workload files so that commands are separated by each user and ultimately deliver them to the load balancer. The load balancer would handle sending commands to the web servers using threads.

Web Server

The web server is written in python using persistent TCP that takes input from the workload generator/load balancer. The input is processed and passes it to the transaction socket interface to the transaction server.

Web Client

For the web application framework, Flask is used with the web server to set up the front-end of the web client that is initially used for the demo. The web client is the last component to be added to our distributed system and its sole purpose is to demo the front-end. We choose Flask because it is easy to set up and provides the ability to scale up to complex applications.



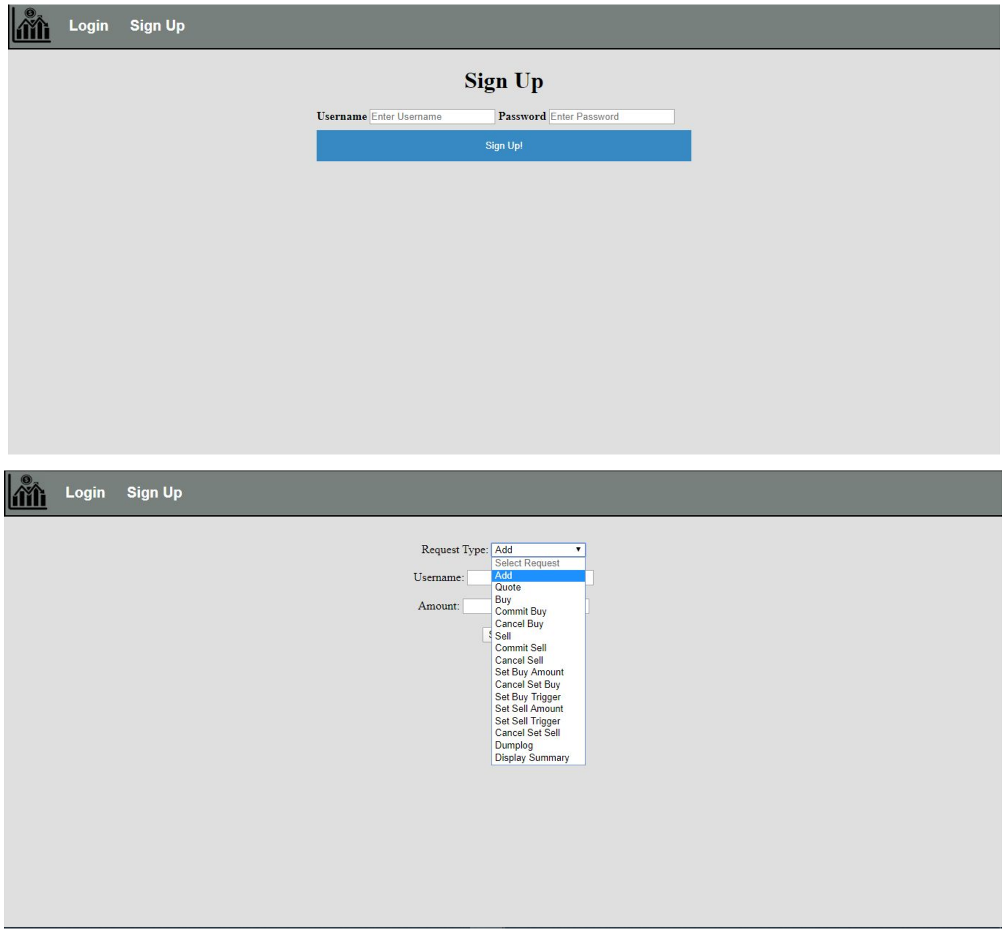


Figure 2. Web Client Interface

Transaction Server

The transaction server is written in python and deals with all of the user transactions, computations, and communications with most of the components. It accepts messages passed from the web server through sockets and validates the message and decides whether it needs to make calls to the quote server cache to get requested quotes, database for accessing or creating user records, or trigger server. All the transactions made in the transaction server are audited.

Trigger Server

The trigger server handles the set triggers by periodically checking the stock prices in the cache and updating it if required. The trigger server will go through all the users with valid set triggers in the database and will systematically check each one with the current stock price in the cache. When stock prices change, the trigger server is notified and checks the new stock values. The trigger server does not communicate with the transaction server, but rather it communicates directly with the database and the quote server cache.

Transaction Database

The transaction database is where all the records of user accounts, transaction histories, and triggers will be stored. Every request will hit the transaction database with at least one write or read. Thus, it is crucial that the database is able to handle multiple connections as efficiently as possible. We use Postgres as our dbms because it provides the best balance of speed, stability, security and features that enables us to handle the amount of traffic through an SQL database. The transaction server also makes use of connection pooling to efficiently pass commands to the database.

Quote Server Cache

The quote server is a legacy software that is both slow and expensive to access. It is the main bottleneck of the distributed system that causes the most performance issue. The cache is implemented to improve performance by keeping track of quotes and allow the transaction server or trigger server to reuse the quotes as long as they are still valid. The cache is placed near the transaction server and the quote server since that is where it is most likely to be accessed by locality.

Project Plan

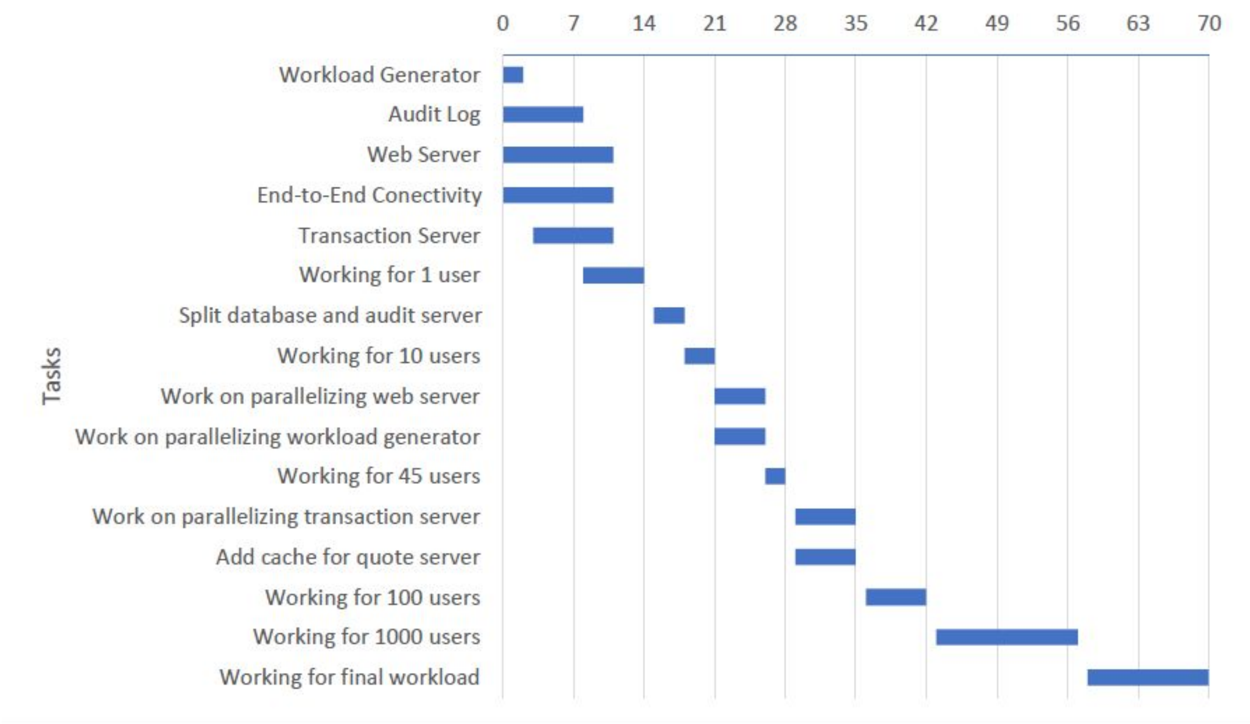


Figure 3. Gantt Chart of Project Plan

Figure 3 describes the initial schedule of tasks for this project. Allan was initially tasked to work on all front-end related components such as web client and web server. Mathieu was responsible for the workload generator, load balancer, and transaction server, and Parm was responsible for all database, audit and log file work. These tasks eventually changed throughout the project as problems occured. Although we were meeting with each other regularly, we had encountered several unfortunate events and issues throughout our project that affected the final outcome of the project. This will be discussed in the work effort portion of this report.

Our plan was to use an agile approach to implement the basic structure of the end system. The goal for the first month was to be able to run 1 user locally on a single machine with all components connected. After 10 users, we planned on focusing more on optimization and performance improvements to be able to run 45 and 100 users workloads. Although we had discussed ideas among the team on possible improvements on security and fault handling throughout the term, we chose to do security and fault implementations last because we wanted to ensure that the core functionality of our distributed system was as efficient as possible.

Security Design

Several security design decisions were made throughout the project. Due to unforeseen circumstances, not all security implementations were able to be integrated in our distributed system. Thus, we will discuss our security implementations done so far as well as what we initially planned on doing.

The web client is implemented in HTML5. In the top toolbar of each page has a “Sign Up” and “Login” link where a user can enter their own username and password. SSL certificate is used to encrypt data that is transmitted over a network to make the data and the network more secure and reliable. We planned on implementing cookies for users that are logged in as well as authentication upon logging in. Since there is a possibility that cookies can be manipulated by criminals to gain access to protected areas, the cookies will not be used to store critical information such as remembering passwords. Moreover the cookies will have an expiry date where it will only be valid for a certain period before it gets deleted as well as encrypted data for user balance in the cookie. Lastly, We also planned on creating a “root” user login that allows the admin to access the dump command for the generation of logs.

The web server takes commands from the workload generator or web client, sends to the transaction server for execution, and forwards the result to the web client. The web server and transaction server uses TCP persistent connection to reduce overhead and improve round-trip time performance. It does this by opening one channel and remains open after a request exchange of data by using TCP keep-alive packets to prevent the connection from timing out. We attempted to increase speeds by switching to a much faster but unreliable UDP, an idea discussed in class. We simulated the mechanisms such as flow control, error control, checksums and packet retransmissions used in TCP to create a reliable UDP.

Ultimately, we settled on reverting back to using TCP for our connections because even though UDP is faster, when a packet is corrupted or lost in the network (lots of noise in the channel) it took longer to retransmit since we were using “stop-and-wait” protocol to wait for a timeout before retransmitting the lost packet. Better mechanisms such as “Go-back-N” or “Selective acknowledgements” should be used in the future for a reliable UDP.

Flask is used in our web server because it allows us to add common security mechanisms in our application. This includes password hashing, basic HTTP authentication, token based authentication, and role management for admins and regular users.

A proxy server (web cache) was also considered in the architecture to reduce web traffic and security. Unlike a browser cache, the proxy server is placed between the web server and client that undertake web transactions on behalf of the client. The proxy should not cache any confidential information by using cache control headers to determine whether it should cache the data or not. In addition, we also considered a combination of HTML, JavaScript, and CSS encodings to prevent output being interpreted as active content.

Lastly, our transaction server performs processing and verification of the commands being passed in from the web server. Our database also uses PostgreSQL and “psycopg2”. This is to prevent any possible Cross-site scripting and SQL injection attacks. We choose PostgreSQL because it has native support for using SSL connections to encrypt client/server communications for increased security but also provides the best balance of speed, and stability to handle the level of traffic through an SQL database.

Test Plan

The goal is to develop a prototype end-to-end solution to support day trading users/clients and services. The system will be used for stock trading such that it must handle several users, be reliable, and be secure. System down-time is not acceptable and system availability is a key concern. Users may request a single command such that if the command is not processed, cancelled, or failed during the run-time of any of the components, the transaction must not go through. The software requires the installation of python3, Flask, and psycopg2 to be able to run.

Test Strategy

Scope:

- Testing transaction server is first priority
- Database and quote server cache is second priority
- Time spent testing web server and trigger server should be equal
- Web client and workload generator/load balancer tests should be low priority

Out of scope:

- Legacy quote server
- Network latency
- Computer specifications and speeds
- Maintenance and upgrade costs
- Resources required in the lab

Agile testing will be used to test the transaction server for performance, fault tolerance, and security. The transaction database will have performance tests with more focus on its security and fault tolerance for transactions. The web server and web client will use unit testing to test for potential vulnerabilities such as cross-site-scripting attacks and SQL injections. Integration tests will be used for workload generator/load balancer and quote server cache.

Risks:

- Not able to test in lab environment may result in noticeable differences in performance or issues that could be caught only in the lab
- Project schedule is too tight, may be hard to complete the project on time
- Misunderstandings in online communication with team members can potentially cause confusion in testing or missing a test case
- Wrong estimate of time or diverging from the projected project plan
- Lack experience in testing

Responsibilities in testing:

- Allan will test web server, web client, and quote cache server
- Parm will test database, transaction server and audit servers
- Mathieu will test generator, and trigger server

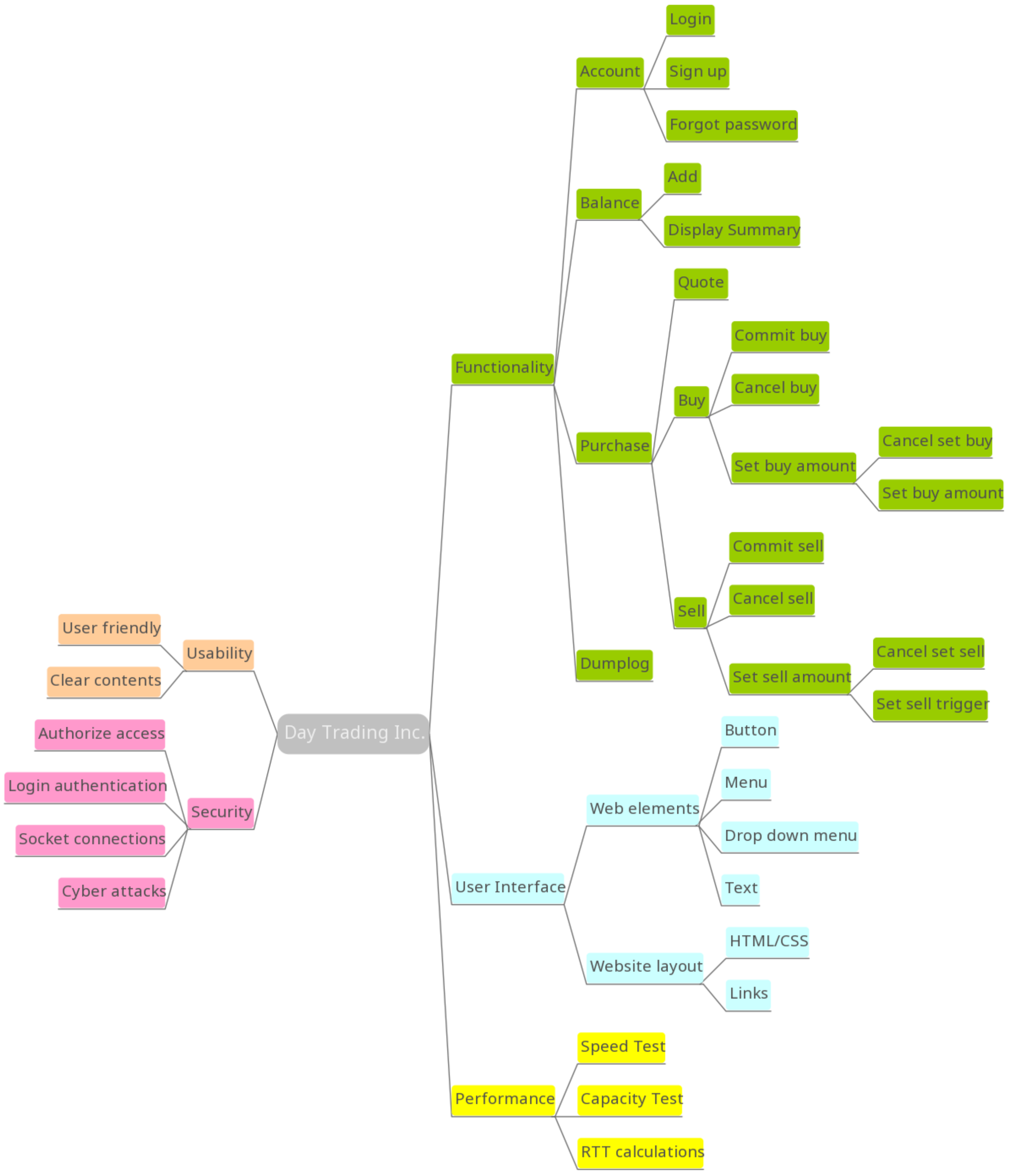


Figure 4. Mind map of test objectives

Schedule and Estimation

Task	Members	Estimate effort
Create the test specification	Allan	3 hours
Performance testing	Allan, Parm, Mathieu	10 hours
Security Testing	Allan, Parm, Mathieu	10 hours
Fault Tolerance Testing	Allan, Parm, Mathieu	5 hours
Functionality Testing	Allan, Parm, Mathieu	2 hours
Usability	Allan	1 hour
User Interface	Allan	1 hour
Total		32 hours

Table 1. Assigned tasks for testing

Fault Tolerance Design

Our system is designed with “ACID” properties in mind to ensure atomicity, consistency, isolation and durability with the transactions. Since every component is interconnected with each other, missing any of these components will prevent the entire system from working. Mechanisms to detect faults and recovering from it is required. Since certain components in our system have a greater risk for failure, we utilize both passive and active replication appropriately.

Passive replication is used for components that do not require data to be saved onto disk. These are the web client, web server, transaction server, and trigger server. Active replication is used for the transaction database and caches. This decision is made because in passive replication (standby spare), data can potentially be lost upon a crash but deals with less consistency issues at the tradeoff of higher down time. Since the web client, web server, transaction server, and trigger server does not write anything to disk, all it needs to do during a crash is to revert the transaction and use the backup component. Active replication is expensive because it utilizes spares by actively processing all transactions in parallel, which causes consistency problems in return for no downtime. Replication also adds redundancy to the system so an adaptive replicas creation algorithm should be used. In the degree of replication, to attain a high level of consistency, a large number of replicas is needed. If the number of replicas is low or less it would affect the scalability, performance and multiple fault tolerance capability. As a result, using both passive and active replication for high risk components provides a balance between system downtime and performance cost.

For detecting faults, we discussed implementing a watchdog timer in python. The watchdog timer ensures that a function does not take longer than “x” amount of seconds to execute and regularly executed functions execute at least every “y” seconds. These values will need to be adjusted based on the average performance of the system. When a function does not satisfy these conditions it will switch to the backup systems and run system recovery.

In active replication we recover from a fault using a check-point based rollback. A check-point stores the current state of the system, which is done after a successful transaction. This is so that when a failure occurs in the middle of a transaction, the transaction will not go through. The check point information is stored in a backup database for easy roll back when there is a component failure allowing for a fluid transition from primary to backup.

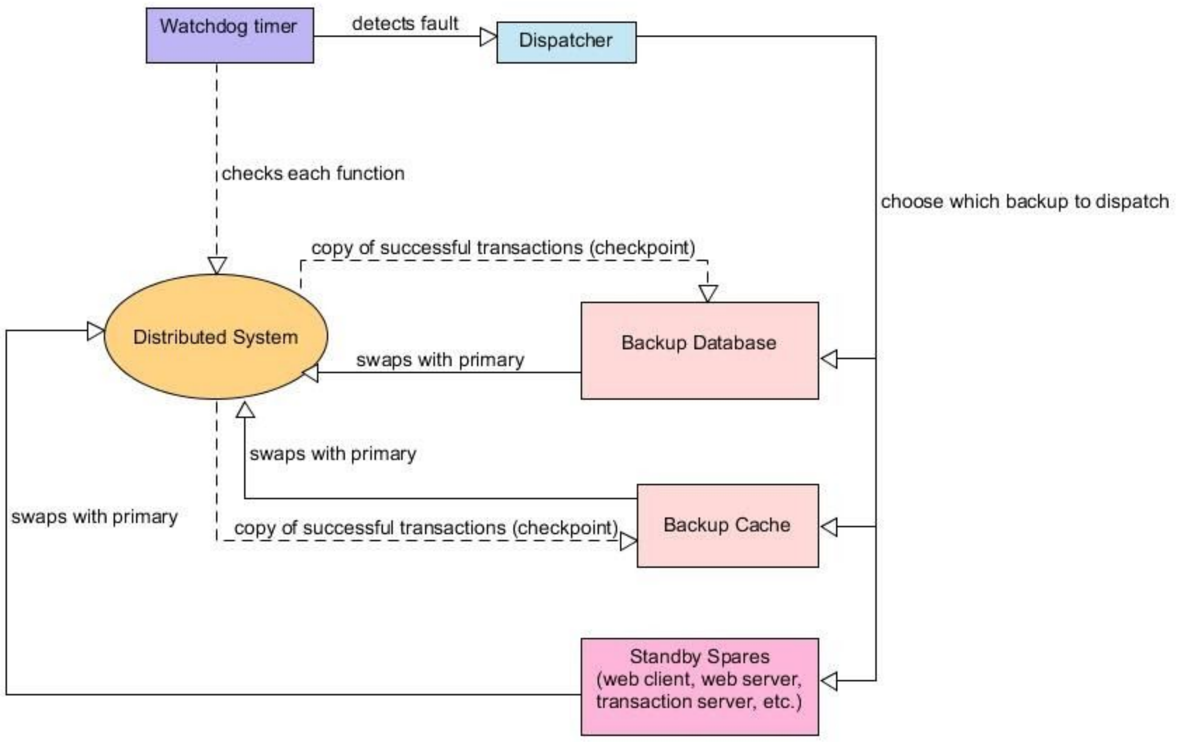

Watchdog timer
detects fault
Dispatcher
checks each function
choose which backup to dispatch
copy of successful transactions (checkpoint)
Distributed System
swaps with primary
Backup Database
swaps with primary
swaps with primary
copy of successful transactions (checkpoint)
Backup Cache
Standby Spares (web client, web server, transaction server, etc.)

Figure 5. Handling faults and using checkpoints

Performance Analysis and Testing

(**Note:** Due to unforeseen reasons, performance analysis and results are measured from testing outside of the designated lab.)

Command	Average execution time (seconds)	Number of commands executed	Summed time of each command (seconds)	Percentage of total time (%)
ADD	0.0039046605428059897 //0.004142204920450847	6	0.023427963256835938 //0.024853229522705078	0.09 //2.23
QUOTE	0.7606303691864014 //0.1291499932607015	6	4.563782215118408 //0.774899959564209	18.38 //69.56
BUY	0.7638309121131897 //0.004243385791778564	20	15.276618242263794 //0.08486771583557129	61.53 //7.64
COMMIT_BUY	0.006453707104637509 //0.003545363744099935	21	0.1355278491973877 //0.07445263862609863	0.55 //6.68
CANCEL_BUY	0.003624399503072103 //0.0022968053817749023	6	0.021746397018432617 //0.013780832290649414	0.09 //1.24
SELL	0.7637465794881185 //0.003276268641153971	6	4.582479476928711 //0.019657611846923828	18.46 //1.78
COMMIT_SELL	0.003875553607940674 //0.002782583236694336	8	0.03100442886352539 //0.022260665893554688	0.12 //2.00
CANCEL_SELL	0.003630677858988444 //0.002182563145955404	6	0.021784067153930664 //0.013095378875732422	0.09 //1.18
SET_BUY_AMOUNT	0.004965305328369141 //0.006148099899291992	1	0.004965305328369141 //0.006148099899291992	0.02 //0.57
SET_BUY_TRIGGER	0.00	0	0.00	0.00 //0.00
CANCEL_SET_BUY	0.0031634807586669923 //0.002284669876098633	5	0.015817403793334961 //0.011423349380493164	0.06 //1.03
SET_SELL_AMOUNT	0.003879070281982422 //0.0024292469024658203	1	0.003879070281982422 //0.0024292469024658203	0.02 //0.22
SET_SELL_TRIGGER	0.005682309468587239 //0.002264420191446940	3	0.01704692840576172 //0.00679326057434082	0.07 //0.61

CANCEL_SET_SELL	0.0028162002563476562 //0.0025363922119140624	5	0.014081001281738281 //0.012681961059570312	0.05 //1.14
DISPLAY_SUMMARY	0.019571447372436525 //0.008743381500244141	5	0.09785723686218262 //0.0437169075012207	0.39 //3.92
DUMPLOG	0.019181251525878906 //0.0021848678588867188	1	0.019181251525878906 //0.0021848678588867188	0.08 //0.20
Total elapsed time	-	-	24.829198837280273 //1.114297866821289	100

Table 2. Times and percentages per command for 1 user workload with & without a cache (represented by: without cache //with cache).

1 User

Initially, the architecture is set up so that the transaction server directly requests data from the quote server. This is not ideal since the quote server's timeliness to reply to the transaction server is a major bottleneck within the system. When the initial system is tested and fed a workload generated by 1 user, the elapsed time is approximately 25s. With 100 total transactions, this gives 4 transactions per second (tps), an inefficient tps count.

After multiple runs of the system, the final average system time and average run time of each command are seen in Table 2. It is shown that the 'ADD', 'BUY', and 'SELL' commands take up the most time because they each hit the quote server. These three commands make up about 98% of the total time to execute 100 arbitrary commands from 1 user.

CPU times /Major System Components	User (ms)	System (ms)
Workload Generator	18.65	14.62
Web Server	24.2	16.0
Transaction server	94.2	34.6
Quote Server Cache	8.18	2.09
Audit Server	14.0	3.77

Table 3(a). Average CPU times of the system’s components (1 user workload)

CPU times /Major System Components	User (s)	System (s)
Workload Generator	1.12	3.62
Web Server	1.56	2.17
Transaction server	7.23	7.49
Quote Server Cache	0.623	0.184
Audit Server	0.136	0.027

Table 3(b). Average CPU times of the system’s components (10 user workload)

CPU times /Major System Components	User (s)	System (s)
Workload Generator	1.45	2.33
Web Server	1.96	2.82
Transaction server	8.20	5.46
Quote Server Cache	0.934	0.395
Audit Server	0.139	0.0285

Table 3(c). Average CPU times of the system’s components (45 user workload)

CPU times /Major System Components	User (sec)	System (sec)
Workload Generator	11.3	50.6
Web Server	16.1	58.3
Transaction server	81	103
Quote Server Cache	7.22	5.47
Audit Server	1.28	0.254

Table 3(d). Average CPU times of the system's components (100 user workload)

10 Users and Beyond

For ten users, the bottleneck is even more noticeable as the elapsed time ends up being roughly two hours. At this point, testing a 45 user workload with this architecture would seem pointless. This is where a cache is established into the system.

By introducing a cache between the transaction server and quote server to hold previously accessed quote data, the bottleneck from the quote server diminishes significantly. Referring to Table 2, the total elapsed time of running a 1 user workload is roughly 22 times faster with a cache in place than the initial architecture. This translates to a tps count of about 90. A noticeable effect that this also has is on the 'BUY' and 'SELL' commands. With the cache present, the percentage of time taken up by these two commands decreases as well. Although the 'QUOTE' command's percentage increases, this is a better result from a user standpoint. Having the 'BUY' and 'SELL' commands use up less time lets users buy and sell their stocks at a quicker rate. Similar results can be seen with the 10, 45, and 100 user workloads.

Tables 3(a)-(d) are also shown indicating the average CPU times of each major component of the system. The transaction server can be shown spending the most CPU time inside (system) and outside (user) of the kernel in each workload amount. The significance of CPU time in this server relates to the comparison at each scale. With the exception of the 10 and 45 user workloads being fairly similar in their CPU times throughout the system, the increase in the transaction CPU time from the 1 user workload to the 10/45 user workloads is about 75-90 times. From the 10/45 workloads to 100 workload it is about 10 times of an increase. The significant increase from the 1 user to the 10/45 user workload shows the CPU being more in demand from multiple users. The CPU becomes very demanding in the 100 user workload, as it spends about 3 minutes total executing inside and outside the kernel.

The quote server cache consists of using threads and queues to be able to concurrently keep any recently requested quotes up to date by interacting with the quote server and servicing the transaction server's requests. We achieve temporal locality by keeping the most recently requested quotes within the cache.

However, given that the 45 and 100 user workloads request quotes from many different companies, the quote server cache is more transparent in its ability to increase the system’s performance at these workloads.

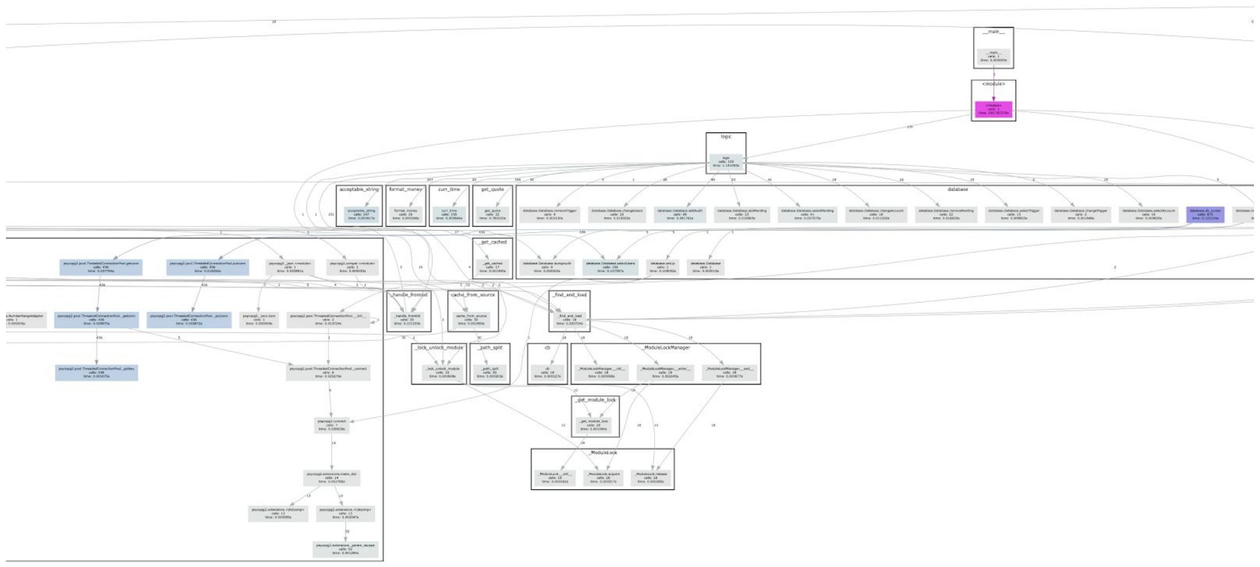


Figure 6(a). Number of executions per function in the transaction server (1 user)

Another bottleneck in the system was connecting to the database. The database connection is an expensive operation and can reduce performance for every record that needs to be stored away into the database. This is solved using a connection pool.

With the connection pool, the system is able to cache many threads containing reusable connections to the database instead of having to repeatedly recreate them. With the transaction server dealing with higher workloads and repeatedly accessing previously accessed records to alter them, reusing an open connection diminishes the potential accumulation of creating a database connection.

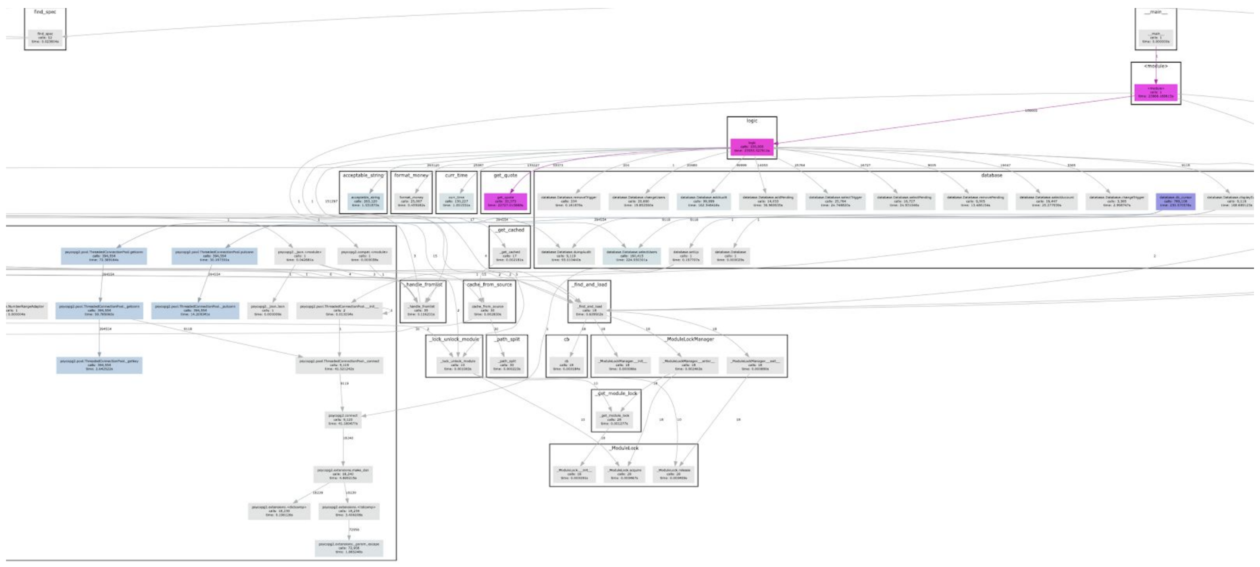
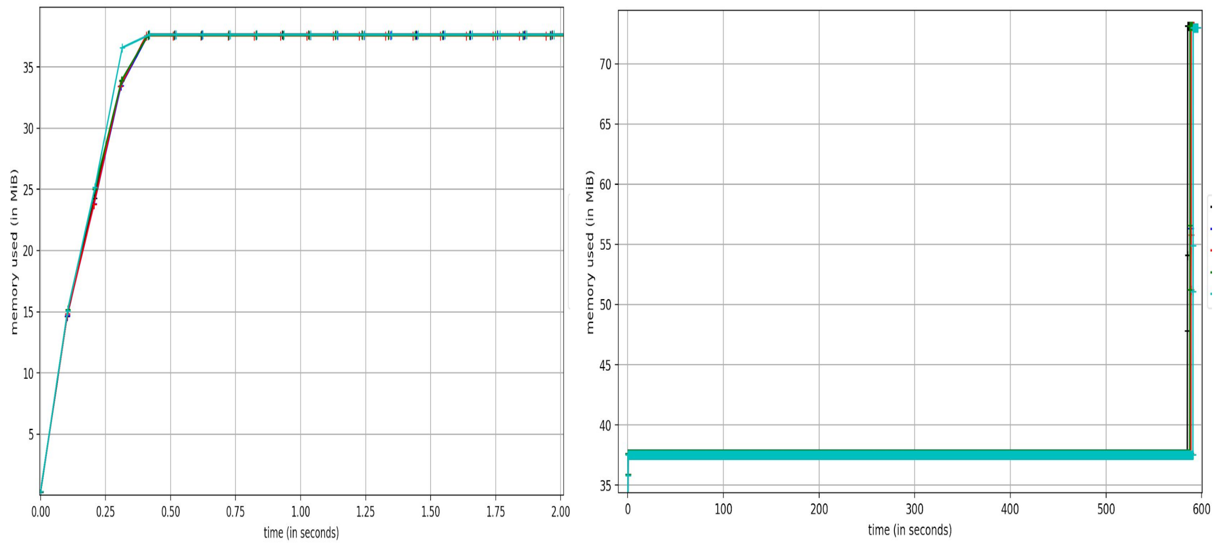


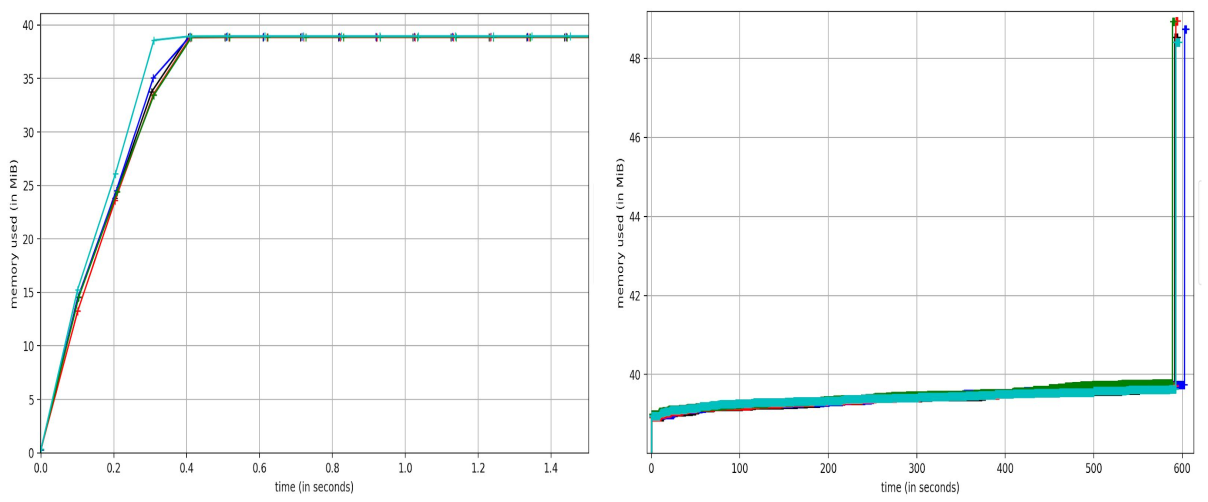
Figure 6(b). Number of executions per function in the transaction server (100 user)

After implementing the connection pool, we see the results in figures 5(a) and 5(b). Between the 1 user and 100 user workloads, the call to the database (purple) and it's set of functions (light blue) don't seem to change relative to the workloads they are handling. From the 1 user workload function graph, we see that the *database.dbcursor()* function has an execution rate of about 7000 calls per second. Looking at the same function in the 100 user workload graph, we have an execution rate of about 3400. The decrease in rate comes from the fact that the connection pool is limited, and as the number of users increases so does the wait time to connect to the database.



(a) (b)

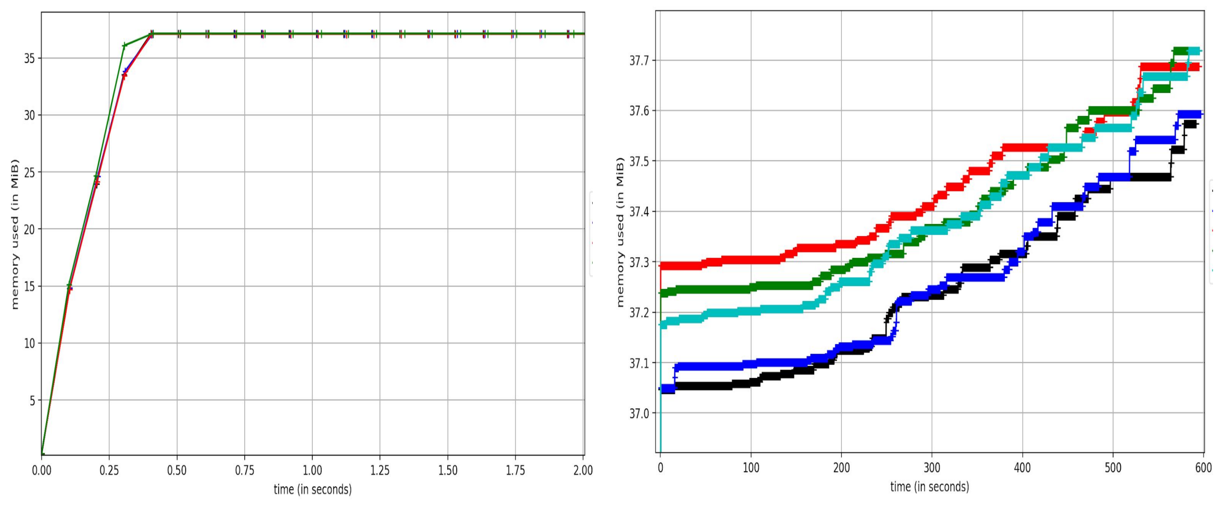
Figure 7. Memory usage in the audit server for (a) 1 user and (b) 45 users



(a) (b)

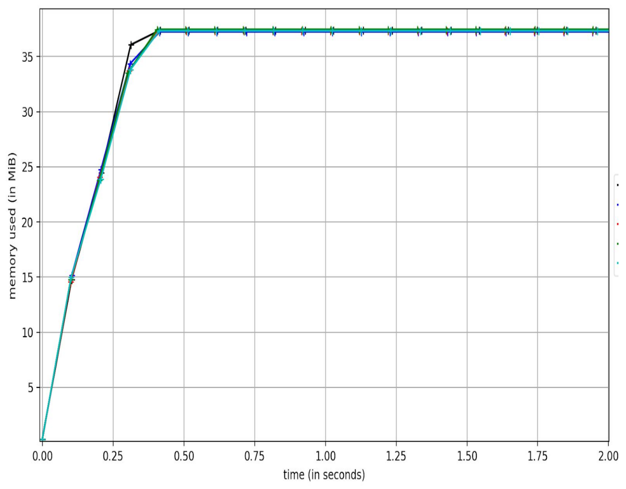
Figure 8. Memory usage in the transaction server for (a) 1 users and (b) 45 users

Taking a look at the memory usage, we see that for a 1 user workload, the 3 servers and cache are consistently the same (Figures 7-10 (a)). Not a lot of variation is given in the 1 user workload since the elapsed time is close to 1 second. However, analyzing the memory used during a 45 user workload run adheres to similar conditions. The memory usage from the system maintains an upper bound of 40 MiB which ensures that it is not a major bottleneck up to 45 users.

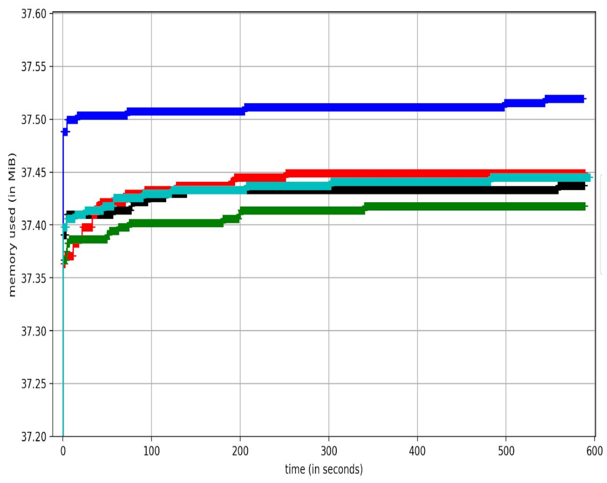


(a) (b)

Figure 9. Memory usage in the web server for (a) 1 users and (b) 45 users



(a)



(b)

Figure 10. Memory usage in the quote server cache for (a) 1 users and (b) 45 users

Capacity Planning

The span to finish this system to serve 1000 users was approximately 3 months. Within the 3 months, weekly to bi-weekly milestones were created (with the exception of the 1 user workload milestone) to maintain a maximum capacity at a certain rate. With limited testing restrictions occurring late in the time period, the current system is tested to service up to an upper bound of 100 users.

With 1 and 10 users, the quote server cache is able to quickly finish the workload within an interval of seconds to minutes. The problem lies in scaling further because the quote server is still a bottleneck given that the number of different company quotes being assessed increases. This gives an inefficient performance when running the 45 and 100 user workloads. Reducing this bottleneck will require having L1 and L2/L3 caches in place. With this implementation, we will be able to access more user quote requests at a reduced time interval. If a certain company's stock fluctuates, then it is highly probable that the number of quote server hits will increase to request that company's stock value, whether it is to buy, sell, or just ask for the price. The L1 cache will be able service quote requests from these companies being accessed frequently. The main reason to implement these caches is to service requests of varying rates. For the L1 cache, this will help maintain system performance when company stocks have a high fluctuation rate. The same applies to the L2 and, if needed, L3 caches. These will help with incoming requests for companies whose stocks are fluctuating at a lower rate.

Looking at the connection pooling implemented between the transaction server and the database, we see that when going from a 1 user workload to a 100 user workload, the rate of execution reduces by about a half. Connecting to a database produces a lot of overhead with a larger workload, and the number of reusable connections needs to be limited or else it may overwhelm the transaction server. Scaling further will require a cache between the transaction server and the database. With an application-layer cache in place, the only time a database connection will be used is when the record to be accessed is not in the database cache. Using a cache-aside strategy, the transaction server will be able to access recently accessed records without the expense of fetching them from the database.

Within the transaction server, there is a lot of redirection that needs to occur for multiple requests at once. Scaling this system without this feature in the transaction server will either break the system or have it running for an excess amount of time. With this bottleneck, we will need to introduce multithreading into the transaction server.

Scaling up to 1000 users and more, the system needs to service the requests of these users at the same time when the requests are redirected to different parts of the system. Commands such as the ‘BUY’, ‘SELL’, and ‘QUOTE’ commands will hit the quote server first. With multiple users running this command, there will also need to be a queue of stock value requests between the transaction server and the quote server. If there are requests for the same company quote, then those requests will be serviced as a group and then sent back into the transaction server for further redirection.

Work Efforts

Our initial progress on the distributed system was slow. Instead of creating the barebones of the system quickly as explained in our project plan, each of us worked on each component individually with scalability of multiple users in mind instead of one user at first. Therefore, it was much more difficult to connect these systems which ultimately took longer for us to successfully run workload files. This was majorly caused by a lack of communication between team members and understanding of other team member’s code that they have worked on. However, we found it easier to scale more users towards the end because we all already had a high-level understanding of our system. As a result, we spent most of the time debugging and connectivity with other components.

Due to classes being cancelled and having to move entirely online without access to the lab machines, it made it difficult for our team to meet and discuss design changes to the system in order to improve performance and capacity for 1000 users and final workload. Thus, we were off track in our schedule again towards the end of the project.

Our team regularly met up each week to discuss our current progress, issues we encountered, potential problems that may occur, and what each member is going to work on the following week.

Logbook Hours

Name	Activity	Time Spent (hrs)	Submit Time
Parm Johal	created db, log server, and started transaction server	12	27/1/2020
Allan Liu	web server, web client, workload generator	15	28/1/2020
Mathieu Trottier	Workload Generator, Documents	7	28/1/2020
Parm Johal	audit server	3	31/1/2020
Allan Liu	Transaction server - ADD, QUOTE, formating money (dollars and cents)	12	1/2/2020
Parm Johal	worked on database/main.py and database/audit.py	3	7/2/2020
Mathieu Trottier	Database	6	7/2/2020
Allan Liu	worked on buy, commit buy, cancel buy, cancel sell	5	7/2/2020
Allan Liu	commit buy, commit sell, cancel buy, cancel sell, set buy amount, cancel set buy amount, display summary	15	9/2/2020
Parm Johal	worked on Transaction Server, Audit server	5	13/2/2020
Mathieu Trottier	Transaction, Database	5	13/2/2020
Allan Liu	Worked on error handling, display summary, audits, bug fixes with Parm, started triggerServer logic	6	21/2/2020
Mathieu Trottier	Transaction, Database: On Vacation	2	21/2/2020
Allan Liu	worked on trigger server. Had midterms	2	28/2/2020
Parm Johal	updated audit.py and TransactionServer.py/midterms	3	1/3/2020
Allan Liu	Finished cache for quoteserver. Started working on persistent connection for webserver	12	5/3/2020
Parm Johal	fixed bugs in audit server; started work on rabbitmq middleware	3	6/3/2020
Mathieu Trottier	Database, Trigger	10	10/3/2020
Allan Liu	Reliable UDP, web client, user login, XML debugging	5	13/3/2020
Parm Johal	Debugged TransactionServer	5	25/3/2020
Allan Liu		72	
Parm Johal		34	
Mathieu Trottier		30	
Total		**136**	

Table 4. Hours logged each week

Table 4 shows the total logged hours for the project and what was worked on each week. As seen in the table, the tasks and milestones we projected to hit in the project plan were off. Allan eventually worked on the web server, web client, transaction server, quote server cache, and the trigger server. Mathieu worked on the workload generator, and part of the database. Parm was responsible for the database connectivity, audit server, log files and middleware.