Architecture Report

April 14 2018

SEng 468 : Spring 2018

Group G12

Heather Cape V00795197
Geoff Lorne V00802043
Tanner Zinck V00801442

Table of Contents

1.0 Introduction

We set out to build a stock day trading system to meet a very specific set of business constraints, with a focus on maximizing throughput and minimizing costs. The architecture was designed with performance in mind at every step.

Our final system achieved 12154 transactions per second, with a cost of $433.70 dollars due to quotes requested from the legacy quote server. Our system meets all business requirements as set out by DayTrading Inc., while also providing the capabilities to scale to a much larger number of users in a real world setting.

A tech stack based on Golang, Postgres, RabbitMQ, and Redis was used. In the future, more research could be done on picking different frameworks based on any performance improvements they may provide. They were picked typically for their compatibility and ubiquity making development less painful.

2.0 Technology

2.1 Golang

Golang was chosen as it is a language we have worked with previously, it provides very simple concurrency mechanisms, and is designed to be highly performant. We enjoy working with Golang, and Golang works well with the other technologies we were interested in using for this project.

2.2 Postgres

Postgres is completely ACID compliant, offers more advanced features than its competitors, and is highly tunable for performance. These were all very important to us when choosing our data layer. Postgres has very good support for Golang, and we have used them together in the past with high degrees of success.

2.3 RabbitMQ

RabbitMQ offers a nice mechanism to implement a producer/consumer pattern between components in the system, and integrates very well with Golang. It was chosen as the medium used to move audit messages from the transaction server to the audit server.

2.4 Redis

During our initial architecture planning, we identified some key areas where having a higher level of data locality than Postgres can offer would be highly beneficial in terms of performance. Redis is a simple Key Value database which operated entirely in memory. Since there is no disk IO involved in Reads/Writes from Redis, it is orders of magnitude faster than Postgres. Values can be set to expire based on time, which perfectly matches the business requirement of quotes expiring. Redis is used as a cache for quotes, and as a cache for user portfolios and funds

3.0 Architecture Planning

During the initial stages of planning, we chose to keep things simple for the sake of getting a functional system that met all business requirements as soon as possible. This meant implementing each peace of the system as its own web server, with all communication being over HTTP. This architecture was not ideal, but allowed us to quickly get a functioning system, and then make changes to the architecture as the course progressed.

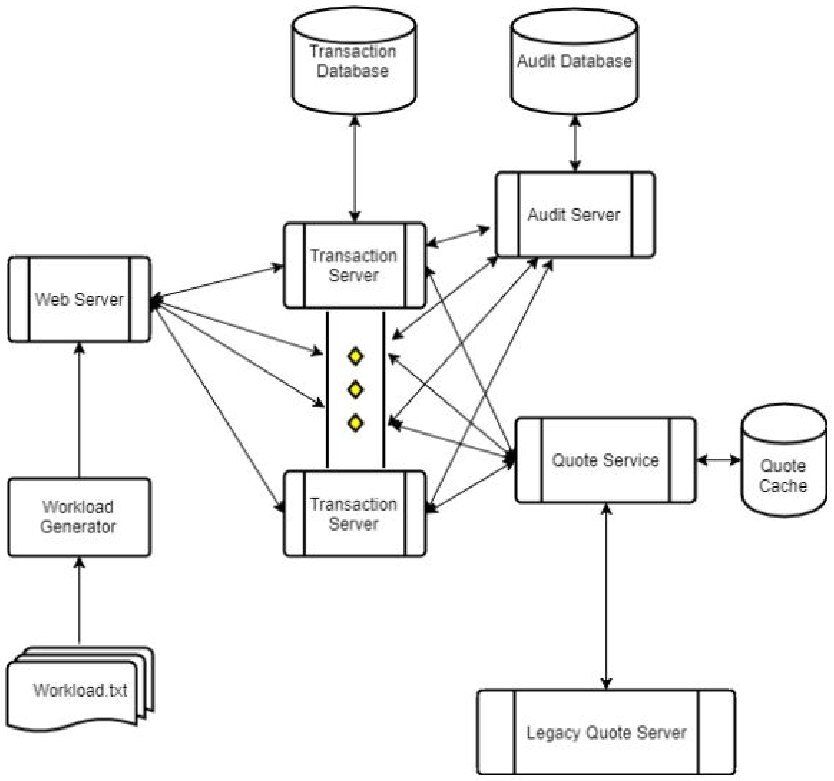

Figure 1: Initial planned architecture from the proposal report

4.0 Final Architecture

This section of the report will examine our final architecture in great detail.

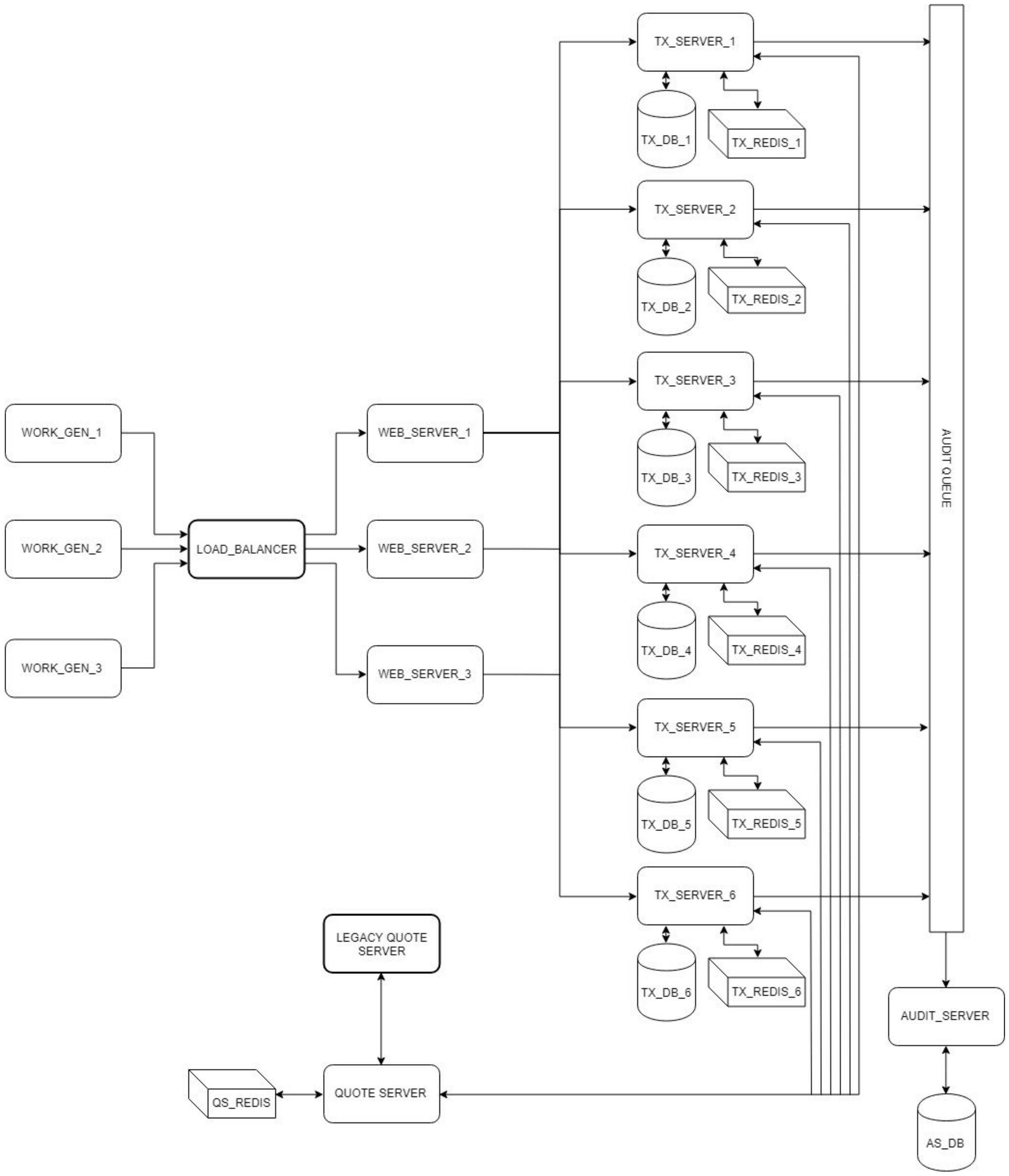

Figure 2. Final Architecture

4.1 Transaction Server

The transaction server is responsible for processing the trades, adjusting account balances, performing error checking and handling, and ensuring that the system conforms to the business requirements. Our architecture evolved to include Redis as a write through cache for users portfolios and funds. This greatly reduced the number of much slower Postgres queries we needed to run. We could also avoid connecting to Postgres only to find that the query would be invalid anyway (eg. trying to sell stock without having any).

The transaction server was also converted to produce audit messages onto RabbitMQ queues rather than send them over HTTP. This is much more efficient, and allows the generation of audit events to be completely asynchronous from the transaction server.

The architecture was scaled to have 6 transaction servers, each with their own database. They are split based on username. This number was chosen frankly because we ran out of time to test higher configurations, and we ran out of computers to run the services on.

In the original plan we assumed there would be one single database for all of the transaction servers; this is before the realization that a major bottleneck was writing to databases, and that having more databases than servers would lead to higher performance.

4.2 Web Server

Our Web server evolved to include load balancing capabilities. A simple, high efficiency load balancing scheme was devised which consists of simply splitting up the workload based on the username of the command issuer. This lets us ensure that each users commands always go to the same transaction server instance, which ensures each users commands run in the correct order.

When we scaled up to 5 transaction servers, the new bottleneck was the network limit coming from one web server. In order to accomodate more network connections, we scaled to 3 web servers that were load balanced by an HAProxy service with the least connection balancing algorithm.

4.3 Audit Server

The audit server evolved to batch consume audit events from the RabbitMQ queues. This has much less overhead than HTTP requests for each audit event, and created a very highly performant audit server. Configuration tweaks within the audit server database vastly increased write performance. We increased the shared buffer limit which increases the amount of system memory available to postgres. We also increased the connection limit and increased the size of our connection pool which means less time is spent waiting for an open database connection when querying the database. Synchronous commits was disabled which allowed for asynchronous transactions to be sent to the database.

When adding the queues to speedup the transaction servers, we ran into new engineering problems. The first issue with this was that there were multiple queues, one for each message type. This was because it was easier to change between a REST model to a queue based one if each REST endpoint was converted to an individual queue. With multiple queues, we needed to make sure the queues were cleared out before the dumplog command was executed, otherwise the resulting logfile would be missing some of the last few transactions.

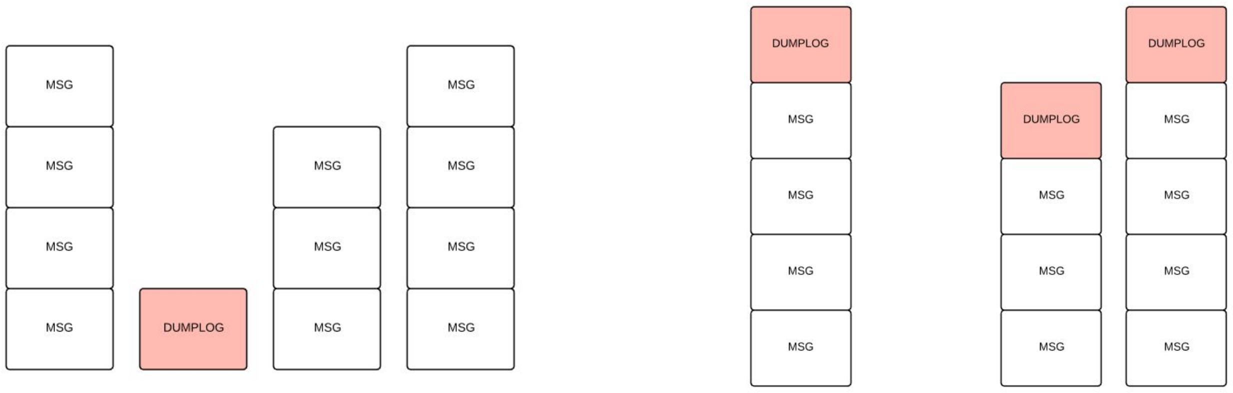

Figure 3: Left is when the dumplog command arrives, right is broadcast to other queues

The solution to this was to broadcast the dumplog command to the other queues to act as markers. Once N dumplog commands were pulled from the queues (where N is the number of unique queues), all the necessary transactions will be logged in the database. It would be mistaken to think that the server could repeatedly check if the queues were empty, as the trigger commands will send residual logs even after all the transactions from the workload generator has been processed.

Another issue is that you cannot send the dumplog command to only one transaction server if there are multiple, as if the fastest server receives the dumplog command it may also generate the log file too early. If the dumplog message is sent to each transaction server, the audit server then needs to wait for MxN dumplog commands in order to execute (where M is the number of transaction servers).

4.4 Quote Server

The quote server was added as a way to put a cache between the transaction server and the legacy quote server. Its primary function is to store all recieved quotes to both reduce overall quote cost, and increase system performance. All quotes needed in the transaction server are received by making requests to the quote service. The Redis cache uses auto expiry to remove the old quote values after 60 seconds. The quote server makes connections with a tunable time out to the legacy server. This time out, and the response distribution modelled is discussed further in the Performance document.

4.5 Workload Generator

The workload generator was modified to attain the highest level of concurrency possible, while still ensuring that the user commands are kept in order. A goroutine is spawned for each user present in the workload file, to ensure that as many transactions as possible are in progress at any given time. To address connection limits, buffered Go channels were used as semaphores to control the number of HTTP requests being sent at once. A Go Wait Group was used to ensure that the dumplog command is sent after all the other transactions have completed. The Wait Group has an integer value that is set to the number of users, and decremented each time a user file is finished processing; only when the value is 0 will the dumplog proceed.

5.0 Future Work

5.1 Refactor Transaction Server for Better Parallelization

For performance sake, we chose to store our users “Buy” and “Sell” stacks as well as triggers as in memory data structures within the transaction server, leveraging the higher level of locality of system memory versus hard disk. This allowed us to operate on our users pending buys, sells, and triggers very quickly, but at the cost of slightly more difficult parallelization. As development progressed, it became quite clear that perhaps the tradeoff of ease of parallelization for added locality was not worth it, as having multiple stateless transaction server instances would make the architecture much more scalable to higher volume workloads, and higher capacity workloads. We would like to move the triggers into a separate service, with its own data layer, and move the buy and sell stacks into either Redis, or Postgres to allow for stateless transaction server instances.

5.2 Database Sharding

The data layer of our system has proven to be the largest bottleneck, requiring the most parallelization to alleviate. In order to decrease the load on each of our Postgres instances, a more rigorous sharding strategy would be something we would look to implement as soon as possible. Postgres variants such as PGPool for database sharding, or Postgres XL for large scaling are possible solutions. Another route would be to convert the auditing database to an InfluxDB instance which is focused on time series data --something that fits well with logging. It was also recommended that we consider CockroachDB which boasts being highly recoverable and scalable. One issue however, is that they only became production ready in June 2017; sometimes it is easier for developers to use tools that have a longer history of support and integration with other languages.

5.3 Less Use of REST

All components were initially developed as REST APIs for simplicity and ease of development. As we continued to chase performance, it became clear to us that alternative, lightweight protocols will be worth using rather than HTTP, as HTTP has quite a bit of overhead when compared to some alternatives such as the use of asynchronous queues or RPC calls between servers.

5.4 Connect to Audit Queues From all Components

Our current architecture has only the transaction server producing onto the audit server queues. This greatly hampers our ability to perform error handling in the web server instances, and log errors from the peripheral services (quote server wrapper, trigger service). Allowing the web server to perform basic error checking would help transfer some of the workload from the heavier loaded transaction servers, and decrease the overall system time spent on each erroneous command. As the system scales up, with errors and emergent behaviours becoming more and more frequent, having the ability to audit system events and errors from every single component in the system will become vital.

5.5 Triggers as a Service (Taas)

Moving the triggers into a separate service makes sense for multiple reasons. Handling the periodic checking of triggers is so fundamentally different than processing all other transactions, that putting the triggers in their own service would be positive in terms of separation of concerns. Currently if user A and B on different transaction servers both have a trigger set on the same stock, the transaction servers will each query the quote server. If all triggers were stored and checked centrally, many fewer connections to the quote server would need to be made. This also helps with transitioning to a stateless transaction server

As discussed previously, if we created a publish subscribe message on quote expiry, there may be a way to have the quote server alert the trigger service of any new values, rather than the other way around.

6.0 Conclusion

Given that our focus for this project was attaining the maximum amount of transactions per second possible, our architecture proved successful. We attained one of the highest TPS out of all groups, and were orders of magnitude ahead of all other groups but one. Given the experience we now have, our mental model of the problem has changed, and if we were to build the system over again, we may make some changes that we otherwise would not have thought to make.