Fault Tolerance 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

Fault tolerance is the ability for a software system to continue running properly in the event of a fault or failure. This is highly important in systems with many users where a system failure could lose the company a substantial amount of money such as this day trading system. Fault tolerance encompases the systems ability to detect faults, prepare in advance for faults, and recover from faults if they do occur.

Fault tolerance will be looked at in two primary levels. The first level is the overall architecture of the system and how it contributes or detracts from fault tolerance. The second level will be lower level look into the frameworks chosen and how they inherently can provide fault tolerance to the system.

A big part in discovering potential failure points in a system would be to do extensive testing, and catch test cases that could cause faults through bugs and fix them accordingly. Most of the testing will not be covered in this document, but rather the testing documentation instead.

To summarize our experience with fault tolerance, we cut corners in adding extra stability in our project to prioritize working on the performance optimization aspects. Overall our mental model has changed we realized that adding some of these features would have made us more efficient developers in the long run as a frequently failing system made our emulation testing much hard and longer to do.

2.0 Assumptions

The main assumption of the project in terms of fault tolerance is that there is no real consequence if the system you are building has poor fault tolerance in terms of meeting business requirements and verifying log files. This meant that in decision making involving trade offs, performance or ease of development often were prioritized ahead of for example, making sure that the system was safe if one of the computers lost power. Therefore, we assumed for the most part that computers would have a very high reliability in the lab, and if a computer were to fail in the middle of the test, we would be able to conduct another run of a workload generated user file.

One quirk of the project's validity based on the requirements is that, even if the system had major faults, as long as the faults did not take place in the link between the transaction server, audit server, audit queue, or audit database, the system would still be perceived as correct. This is to say that a log file with many error messages and dropped commands would still technically be 'correct' and verifiable. Our group did not take this approach in thinking however, and preferred to have a system where all the transactions completed correctly over one that was incorrect but fault tolerant.

3.0 Implementations

3.1 Architecture Level Fault Tolerance

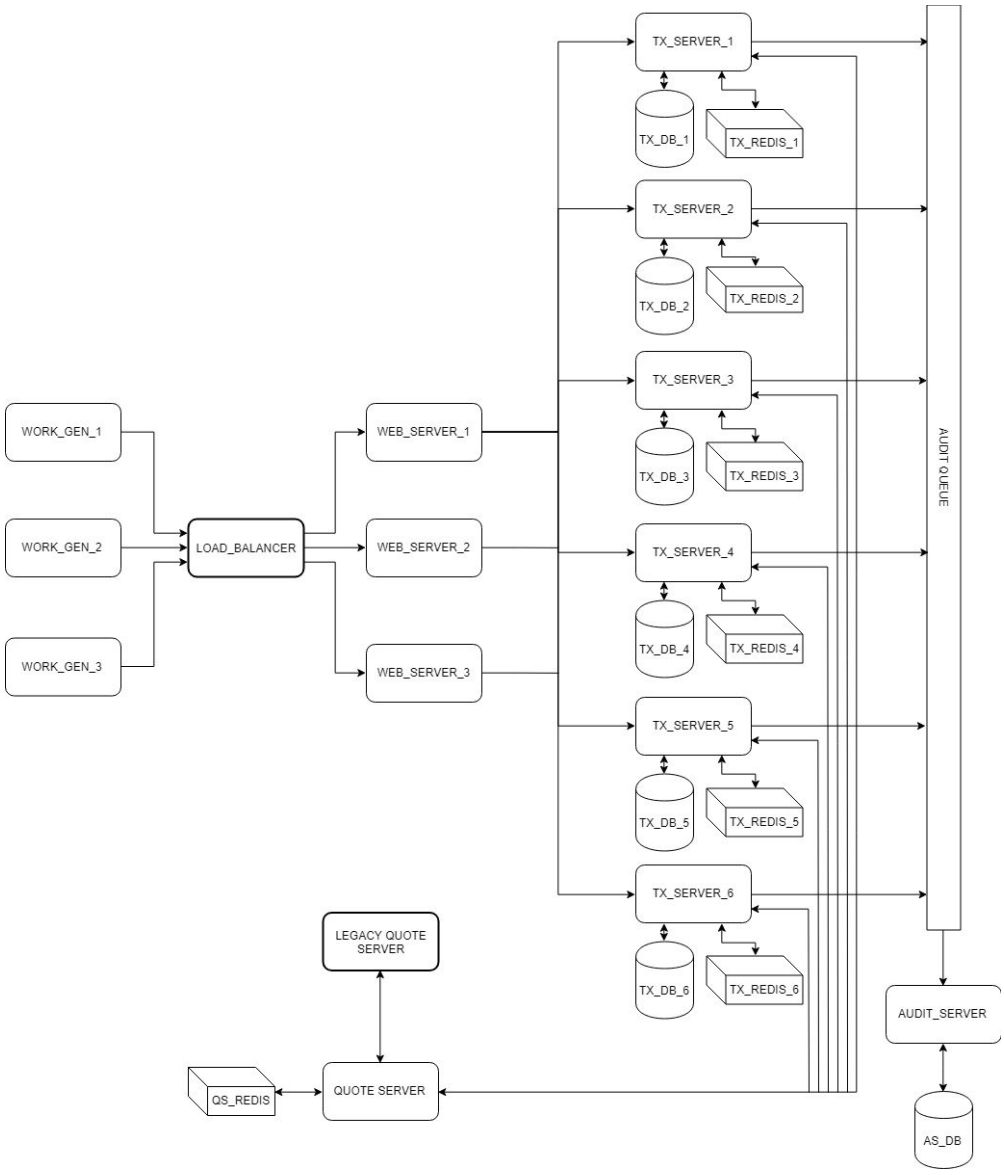

Figure 1: System architecture

Fault Tolerant Benefits

To be blunt, this is not a system developed with fault tolerance in mind, but some things added had the side effect of being fault tolerant.

The multiple web servers were added initially because of the bottleneck around network limitations. Because the web servers are load balanced by least connections, if one or two go down the system is still able to run thanks to the HAProxy redirecting the traffic to the web servers that are available. The tradeoff is that HAProxy becomes a single point of failure in the system.
The multiple transaction servers are load balanced based on username. This means that if one of the 6 transaction servers, redis caches, or databases were to go down then ⅙ of the users cannot use the day trader. However, the failures are independent of the other transaction servers, so that means the other ⅚ users should experience no downtime because of it.

The Redis instance in each transaction server acts as write through cache for the database. This was done to improve the performance on all the read actions from Redis, but also comes with being a pseudo backup for the database. If the database were to fail, backup scripts could quickly write all the Redis information to disk in order to not lose the information. For the other scenario, if the transaction server fails and loses the Redis data, it could be slowly rebuilt from the database either all at once, or slowly by populating the Redis cache back on missed hits reads and writes. Although neither of these failsafes are actually implemented, they have the potential to be based on the current architecture.

The audit queue also gives a layer of fault tolerance in that the messages are kept in memory and in order if something were to happen to either end of the queue. On the other hand, because there are no backups for the queue, if that service were to fail many audit messages would be lost.

Fault Tolerant Critical Analysis

Every single point of failure in the system is a problem, which makes for a lot of problems.
If the quote server fails then users cannot do any Buy, Sell, or Quote commands until it is back up, and the triggers cannot function as well. If the quote sever crashes completely the cache will be lost as well because it is stored in memory in Redis. This is not too catastrophic as the cached quotes are temporary anyways, so only performance would be lost for the first 60 seconds it is back online as it rebuilds the cache.

The audit server, queue, and database are all single points of failure as well. If any of those crash then transactions can still occur for the users (assuming connection errors from the transaction servers are not fatal errors), but all logging information will be lost.

Some things are stored in memory that would cause problems in the event of a power outage. The Redis in the transaction servers for one, also some values are stored in maps and queues that would need to be rebuilt from the database, specifically a stack of recent buy or sell commands for the users, and timers since each trigger was last checked. Obviously if any singular database were to be lost completely that would be catastrophic.

Because transaction servers are split based on user data, if one of them goes down then ⅙ of the users will not be able to use the site at all.

Reliability Analysis

In Figure 2 below, the system is laid out pictorially to better represent a reliability analysis. From left to right the system reliability will be discussed. First of all, the workload generators are not really a part of the system to be considered, because in a real scenario each would be represented as an individual client whose reliability is considered in terms of the system reliability. The load balancer is in series with three parallel web servers. Each web server has six connections --one to each transaction server. As the transaction servers are not really parallel, it is easier to consider the system reliability from the perspective of a single user. Each transaction server has a Redis instance and Postgres database that can (with assumptions discussed above) act as parallel backups. Each transaction server is connected in series to the quote server and audit queue.

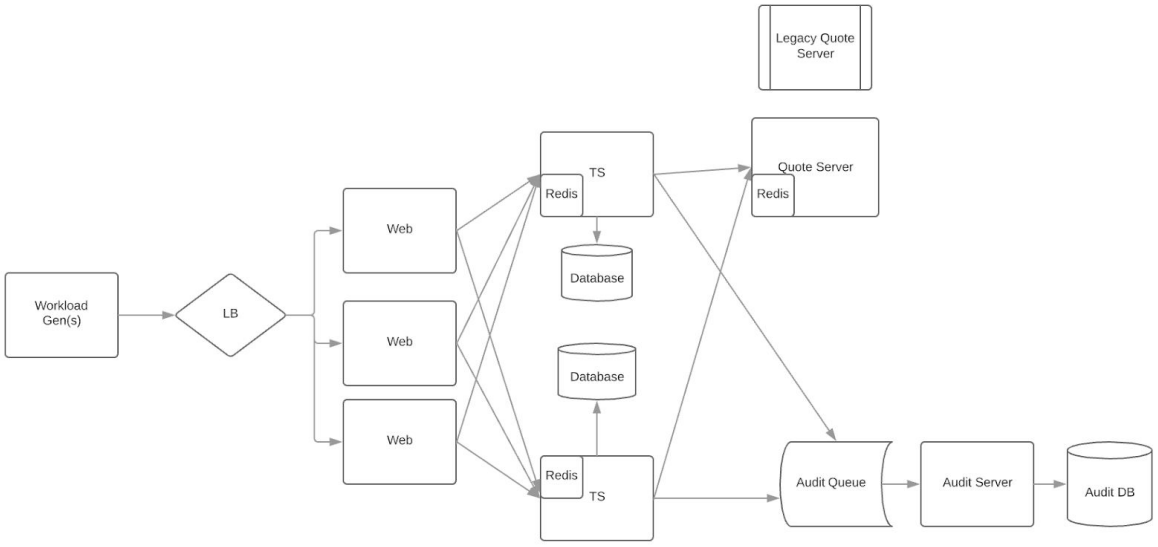

Figure 2: Alternate layout of architecture for reliability analysis

For one user (their username will always be routed to the same transaction server) the reliability would be (with each multiplied):

$R_{\text{Load Balancer}}(1-(1-R_{\text{Web}})^3)\, R_{\text{Transaction}}(1-(1-R_{\text{TDB}})^2)R_{\text{Quote}}\, R_{\text{Quote Redis}}\, R_{\text{Audit Queue}} R_{\text{Audit Server}}\, R_{\text{AuditDb}}$

One assumption is that the quote server and audit server are always both needed for the transaction to compete; in for some cases a command may not require access to the quote server, but this formula is generalized. Another assumption is that the Redis cache in the quote server failing likely will not cause a system failure, but could cause large slowdowns of the systems due to the lengthy waits from the legacy server.

It shows that there are quite a few single points of failures in the system that will lead to an overall low reliability once multiplied. Addressing these points would be the first task to increasing the reliability. For reference, if each reliability above were 90%, then the overall reliability of the system would be 47%, so our system would be more likely to not be available than to be working properly.

3.2 Code and Framework Level Fault Tolerance

3.2.1 TCP Connection Limiting

When operating the system with more than three transaction server instances, we would almost always encounter dropped TCP connections due to hitting the TCP connection limit between the web server and the transaction servers. To prevent this from occurring, we used the Go channels to implement a semaphore that only allowed a set number of connections to occur at once. This number of connections is loaded dynamically from environment variables, to allow us to tune it with minimal developer effort.

3.2.2 Use of Sync Map

To manage access to global scope resources within the transaction server from multiple concurrent goroutines, we used the thread safe map implementation provided by the Go ‘sync’ package, Sync.Map. The sync map is a concurrency safe map implementation that is highly optimized to the type of load our shared maps will see (constant keys, no deletes). The alternative to using the sync map would have been to implement our own mutual exclusion using the read/write mutex also provided by the ‘sync’ package, however this opens us up to the potential for having a deadlock occur. By opting for the added transparency of the Sync.Map, we both avoided additional development time, and the potential for deadlocks to occur in the system.

3.2.3 Low Level Error Handling

Idiomatic handling of errors in Go has most error handling done right at the source of the error, which is what he have done throughout all components of the trading system. Error conditions are checked for as soon as they could possibly be generated and the system responds accordingly, whether that be sending an informative response back to the client, or altering program flow to adhere to business logic. We have made sure we catch all errors, rather than ignoring them as is often done it Go code. This allows us to ensure that the client will always receive a meaningful error message.

4.0 Stress Testing

The nature of this project does not lend well to testing for long term reliability or availability. In order to do this, the system would need be on for a long time processing requests over time. In reality, this emulation tests the servers peak and throughput load over a shorter amount of time. In the future, for better testing we would need to write: a rate limited workload generator that can process commands slower to test the long term stability of the system, and a longer generated user file, possibly with more users. The new user file would be required as running the same one over and over can be optimistic due to the cached user information and stock values that will be reused.

Another thing that would have been useful to us would be to write down the number of times we needed to run the system in a specific configuration in order for it to produce a verifiable log file. In some testing periods earlier on in development, the system would crash due to overflowing TCP or PSQL connections almost half the time. Now, anecdotally only, the system is much more stable for these kinds of crashes and should be able to finish a run over 9/10 times. Another issue is that during the middle period of the class (early 1000 user tests), the runs could take over 30 minutes, so collecting a meaningful amount of data would be far too time consuming relative to all the other work that needed to be done.

We did want to push the limits of our system however, to observe which parts would fail first and how. With the finalized setup of 3 workload generators, 3 web servers, and 6 transaction servers, we ran the 1300 final user file 7 times against our system (manually restarting after each one finished). The time value in the chart below comes from the workload generator printing out the time it took to process all the commands in the file, the max time was the longest of the 3 workload generators simultaneously running

Run	Max Time	Problems
1	1min 38s	
2	2min 32sec	
3	2min 54sec	Right on the 3rd round start one of the web servers crashed for TCP connection limit, it was quickly restarted and the tests continued
4	2min 0sec	The amount of funds in the user accounts were surpassing the int limit of PSQL.
5	2min 33sec	The transaction servers outputs are hanging for long periods
6	2min 33sec	
7	2min 57sec	One transaction server failed, audit queues appear to have crashed, audit server slowing under many log.xml writes.

First will be discussion on on the overall observations of the runs. The system started out at around peak performance. The second and third runs took a big dip in performance time. The system started erroring out and not fully completing many transactions, which lead to the apparent speedup. Finally many parts of the system began to fail and it became unrecoverable.

Except for the TCP connection limit problem, all the problems encountered from this test were ones never before seen in the project, so this was an enlightening experiment. The amount of funds in a user account surpassed the integer limit for many users; from that point on any Add Funds, Sell Stock, or Sell Trigger would fail for those users. At the end one of the 6 transaction servers failed due to an Golang error with the stack library. The stack data type is used to hold the number of recent buys and sells for each user so that a commit command will apply to the most recent one. The error was a segfault, so it is suspected to either have overrun the memory, or run into some sort of concurrency problem where the stack was empty when it should not have been.

One problem that became apparent after adding multiple transaction servers was ensuring that the dumlog command only executed after all the other commands had finished executing and been logged. The solution for this was to send the dumplog command to all N of the transaction servers from the web server once all other transactions had been finished. Then the audit server waits for N of dumplog commands to arrive in order to know that the dumplog is really the last command, and writes a log.xml file upon the Nth arrived dumplog command. Due to this quirk in the system, on any consecutive run of the user file, the audit server would write N log.xml files instead of one. This is an easy bug to fix, but lead to the audit server becoming very backlogged with the write commands. The audit queues failed partially on the 7th run; it may be because the audit server was not pulling commands out of the queue fast enough and the queue overflowed.

Overall however, it seemed that the frameworks themselves such as Redis, RabbitMQ, and PSQL held up relatively well, and that the errors came mostly Go code and server related problems. In some cases it is easier to tweek code than it is to optimize third party tools, so this was encouraging.

5.0 Future Considerations

Moving forward with the development of the trading system, fault tolerance is an area we would begin to focus more of our attention on. This section of the report outlines some of the key areas we have already identified regarding fault tolerance.

5.1 Single Points of Failure

Currently, multiple single points of failure are severely limiting our systems ability to be fault tolerant. Removing these single points of failure will be part of our future fault tolerance strategy. It is important to note however, that because we base our time on the single instance of the saudit server, we have no issues with clock drift or unsynchronised time reference. If multiple audit servers are put into production, extra effort will need to be made to ensure the times are synchronized (for example the NTP server that some other groups used in the lab).

5.1.1 Audit Server

Our current architecture uses only one audit server, and one audit database. While this audit database is completely separate of our other Postgres instances, it is still a single point of failure. If for some reason either the audit server hard crashes, or the audit database goes down, the system will still function, but there will be no way to audit system activity until the audit service is brought back online, which could result in many unaudited account transactions. This would be breaking business requirements if it occurred. The simplest way to remedy this would be to first implement active replication on the audit database, which would allow us to switch operation to the backup in the case that the first database goes down. The next step to making the audit service more fault tolerant would be parallelizing the entire service, with separate audit servers being load balanced, and each audit server having its own set of databases with active replication. This would also increase the maximum potential throughput of the audit service.

5.1.2 Rabbit MQ

In our current architecture, if RabbitMQ goes down, then the transaction servers will no longer be able to send audit events to the audit server, effectively rendering the audit server useless even though it could still be fully functioning. RabbitMQ provides a clustering mechanism, which would allow us to put RabitMQ onto multiple nodes, ensuring better fault tolerance[1]. In the clustered approach, RabbitMQ can handle complete failure of nodes while still staying up overall.

[1] https://www.rabbitmq.com/clustering.html

5.1.3 Transaction Servers (Per User)

Since our current load balancing and database sharding scheme is to split based on username, we are not currently gaining any amount of fault tolerance and reliability by adding more transaction server instances. This will be resolved eventually by transitioning to a stateless transaction server, but in the short term, having an active transaction server and postgres instance for each group of user names would provide us which a much higher level of fault tolerance. The move to a stateless transaction server will also prevent the loss of pending buys and sells, as well as triggers, which are currently stored in memory in the transaction server instances. These would be moved to databases in the stateless version.

5.2 General Future Optimizations

Aside from the single points of failure in the system which are the number one priority in terms of bettering the fault tolerance of our trading system, these are some other key areas which we have identified as having potential to increase our fault tolerance.

5.2.1 Better Audit Service Connectivity

Currently, our utilization of the auditing service is not very fault tolerant. Only the transaction server instances are connected to the audit message queues, which means that if an error occurs at any other point in the system, it is only logged if it propagates back to the transaction server. This means that there is a chance that errors could slip out of the system and never make it into the audit log.

5.2.2 Redis Periodic Write to Disk

As an added measure of fault tolerance, periodically writing the Redis cache to disk provides a small amount of extra fault tolerance by providing a way to potentially restore data to the Postgres instances in the event they go down while Redis stays up. The performance cost to implementing this is almost non existent, so it certainly worth it to set up as a potential data layer fallback.

5.2.3 Handle TCP Connection Timeouts

Currently, if TCP connections in certain portions of the system fail, these can cause the entire system to crash. Implementing comprehensive handling of potential TCP connection errors is high on our list of fixes regarding fault tolerance

5.2.4 Fault Tolerance in the Web Interface

Though currently the system will provide meaningful HTTP responses back to the client, including meaningful error messages, the current web client itself is not very fault tolerant, and may make for a bit of a confusing experience for some users, especially if they encounter an error condition. By ensuring that a “Success” or “Failure” message of some sort is relayed back to the client in the interface would help the users understand what is going on a little better, and allow them to understand what is going on in the event of an error.

5.2.5 Docker Swarm

Docker swarm is a container orchestration mechanism that, among other things, implements automatic restarting of containers in the event of a failure. Currently our containers would need to be restarted manually in the event of a failure, and other container don’t have a way of knowing that a different container has gone down. When running all containers in a swarm, each container will know when another goes down, and can be setup to stop sending requests to it until it comes back up. We could implement all of this manually, but since swarm already provides this functionality, we will be looking to transition to docker swarm in the future.

5.2.6 Transactions

By utilizing Postgres transactions, we can ensure that in the event of an unexpected error during a database operation, any changes made to user accounts can be sufficiently rolled back to ensure that no commands end up running only halfway to completion.

5.2.7 Postgres

Some settings in the audit database were tweaked to increase write performance at the expense of some durability. Specifically, synchronous commit was disabled which allows for asynchronous changes to the database. With synchronous commit enabled, postgres will wait for the current transaction to be safely written to disk and indicate that the transaction has been committed successfully before processing the next transaction. Disabling synchronous commit allows postgres to begin processing subsequent transactions before ensuring the current transaction is safely committed. This substantially increased write speed but comes with a cost; In the event the database crashes or loses power at the same time as a transaction is being committed, the transaction may be lost even though postgres has indicated that it has been committed. This loss in durability can be mitigated by using an uninterruptible power supply or, more realistically, setting up one or more additional postgres servers as replication.

5.2.8 Stretch Goals

While certainly not required, something we would like to look into is setting up a full monitoring suite using an open source tool like Grafana and another service built on a data layer optimized for time series data, such as InfluxDB. Additional stats such as number of current authenticated users, average TPS, and number of goroutines currently running could also be stored and viewed through an admin panel. Ideally this additional insight into the system behaviour would allow us to track potential error sources, and resolve them as soon as possible.

6.0 Conclusion

Fault tolerance is obviously crucial in real world systems, especially from a business perspective in a system such as ours which is handling large volumes of currency and stocks. Throughout the development of our trading system, came to appreciate the value of fault tolerance a little more every time a component of our system failed, and we had to restart every single component to attempt another run. With the system sitting on ~20 computers, this became a huge time sink in testing. Had we implemented a system that was more fault tolerant right from the start, we would have saved a significant amount of time in the testing phase.