Capacity Plan 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

This document will outline the scalability issues and results of adding more and more users to the system. This document differs from the performance document as while that one is focused on the TPS of the overall system, this document outlines the aspects of our system that were necessary to scale to more users while keeping a consistent performance.

Although this document is called capacity planning, we admittedly did not know the limitations of the lab computers, network, or frameworks well enough before starting to accurately outline a plan of action before diving into the project. In order to plan for scaling issues we worked in several days worth of work into our planned schedule at each milestone to address tech debt and tackling problems that arise with emergent behaviours. The last 3 milestones (100, 1000, and final) were left fairly vague in terms of planning as without the results of testing the system we knew we would not have a solid idea of any bottlenecks or limitations.[1]

Overall this document will examine the lessons learned at each milestone, examine the emergent behaviours and growing pains at each step, evaluate the limits of the current system, and make recommendations on a future system.

2.0 Assumptions

This report assumes that in scaling up to more users, the workloads these additional users would generate would be somewhat similar to the workload files we have been testing against. On a per user basis, this is probably a fairly poor assumption, as certain users will most likely have their own distinct behavioural patterns within the system, but when averaging all users behavioural patterns, this assumption should be somewhat safe.

We are also assuming that in scaling the system up to more users, we will still be using the hardware/software environment available to us in the lab, otherwise all of our analysis would not apply, ie, more powerful hardware would completely change system behaviours.

We are also assuming that the functions the system performs will not be changing. The commands currently supported are the only commands that will be supported, and the business logic behind this set of commands will also not be changing as we bring more users into the system.

[1] This information can be found in our original proposal document - Initial Documentation Plan Winter 2018

The data and testing is based on the lab of 22 lab computers running CentOS 7 with aged hard drives and CPUS. An easy way to improve performance or boost capacity would be to make hardware or software changes.

3.0 Results of Scaling

In this section the data from milestone submissions will be analysed along with the system alterations we had to make in order to accommodate the new users.

3.1 One to Ten Users

The one user test was fairly trivial, as there was only one user with one stock, parallelizing processes in the code and creating a quote cache was not necessary (the single quote value was kept in memory).

In the first test with 10 users, the workload generator only read in the single file and did not prepare parallelized goroutines for each user. The web and transaction servers automatically created goroutines for each received http requests, so at this point the bottleneck of the system was the workload generator. This was fixed in order to improve the TPS/User to 6.7

At this point we made use of a redis cache to cache quote server responses, but the system was processing too slowly to make much use of the hits before they expire.

Users	Quote	TPS	TPS/User	Response Time	Date
1	1	19.436[2]	19.436	0.05	Jan 25th
10	410	17.115	1.7	0.588	Feb 2nd
10	63	67.609	6.7	0.150	Feb 18th

3.2 Ten to Forty-Five Users

In this milestone, note that there was one very slow run of the 45 user before the workload generator parallelization was fixed (10 user was retested at this point).

There were two new capacity-related problems that emerged with this increase of users. The connections to the audit database were beginning to reach its limits and would crash because of it, and the maps that stored user trigger information. The former we attempted to solve and help

[2] This was not calculated by the verification system, but the time it took was 5.145 seconds. 100/5.145 = 19.436

improve our system by adding an asynchronous dumping queue from the transaction server to the audit server. This actually made the connection limit problem worse as our systems speed improved because we did not rate limit the connections in anyway, so the system would max out the PSQL connection limit and crash

The second issue was that the default maps in Golang are not concurrent safe. Concurrent map write errors caused our system to crash and forced us to restart the system and run the file again. When introducing more concurrent processes (from 10 to 45) the likelihood that two processes would be using the same map entry at the same time increased. This problem was avoidable in the short term by restarting the test runs until one was successful enough to verify, but the code was reworked a few days after the 45 user milestone to include the Golang Sync package Map implementation. This is similar to a standard map structure and handles all concurrency issues.

In the graph below, it is apparent that the first 45 user test is not an acceptable level of response time. There is no requirements given for this metric, but as discussed in class over 2 second response time can cause people to leave the website.

Users	Quote	TPS	TPS/User	Response Time	Date
45	967	11.280	0.251	3.98	Feb 15th
45	149	72.373	1.608	0.622	Feb 18th
45	151	69.545	1.545	0.647	Feb 18th

3.3 Forty-Five to One Hundred Users

At this point capacity at a network level was beginning to become a problem. The TCP connections between computers were being maxed out. At this point modifying some low level variables helped keep our system from running out of connections and crashing.

The first is setting the `ulimit -n` to a higher number than the default 1024 (we used 9998). This is the number of open files a process is allowed to have (this includes sockets, pipes, and terminals).

Also there are many systemctl variable to do with TCP limits that can be modified. For example, config values can be changed to lower the amount of time a TCP connection stays alive after closing, increasing the allowable port range, and lower the timeout default time. Freeing up TCP connections faster helped us ensure that our servers would not crash due to the limit.

At this point we also had to vastly improve our biggest bottleneck, the audit server. Also further distribution was done. Using the Docker stats monitoring tool it became apparent that any

database would quickly use over 350% of the CPU resource (out of 400%, 100% for each of the 4 cores). This meant that each database should be on its own computer to maximize the resources available. Golang servers never used more that 150%, but it was suspected that they were not efficiently making use of all 4 cores. Some services, such as the Redis caches, the Quote Server, and the RabbitMQ did not max out their resources at this stage.

Users	Quote	TPS	TPS/User	Response Time	Date
100	3310	278.024	2.780	0.360	March 4th
100	3282	281.783	2.818	0.355	March 4th
100	3242	312.312	3.123	0.320	March 4th
100	2231	1045.446	10.454	0.096	March 6th
100	2226	1066.906	10.669	0.094	March 6th
100	2272	1071.973	10.720	0.093	March 6th
100	2247	1079.098	10.791	0.093	March 6th

3.4 One Hundred Plus

Running the 1000 user file right after submitting the 100 user milestone showed that our system could handle that many user and process the transactions at the same speed. The TPS/User however was slow, and our goal was to get it back to the value it was before the scale up.

Most of the optimization here came from parallelizing resources. Scaling from one of each service to 3 workload generators, 3 web servers and 6 transaction servers. We also added the write-through Redis cache in each transaction database (more information available in the Performance Document).

Users	Quote	TPS	TPS/User	Response Time	Date
1000	18222	967.412	0.967	1.03	March 6th
1000	-	1507.477	1.507	0.664	March 24th
1000	-	2318.362	2.318	0.431	March 26th
1000	-	2394.510	2.395	0.418	March 26th
1000	-	7306.630	7.306	0.137	March 30th
1000	-	10002.200	10.002	0.100	April 3rd
1000	-	10168.595	10.169	0.098	April 3rd
1000	-	10614.472	10.614	0.094	April 3rd
1300	8674	12513.991	9.626	0.104	April 12th

3.5 Discussion

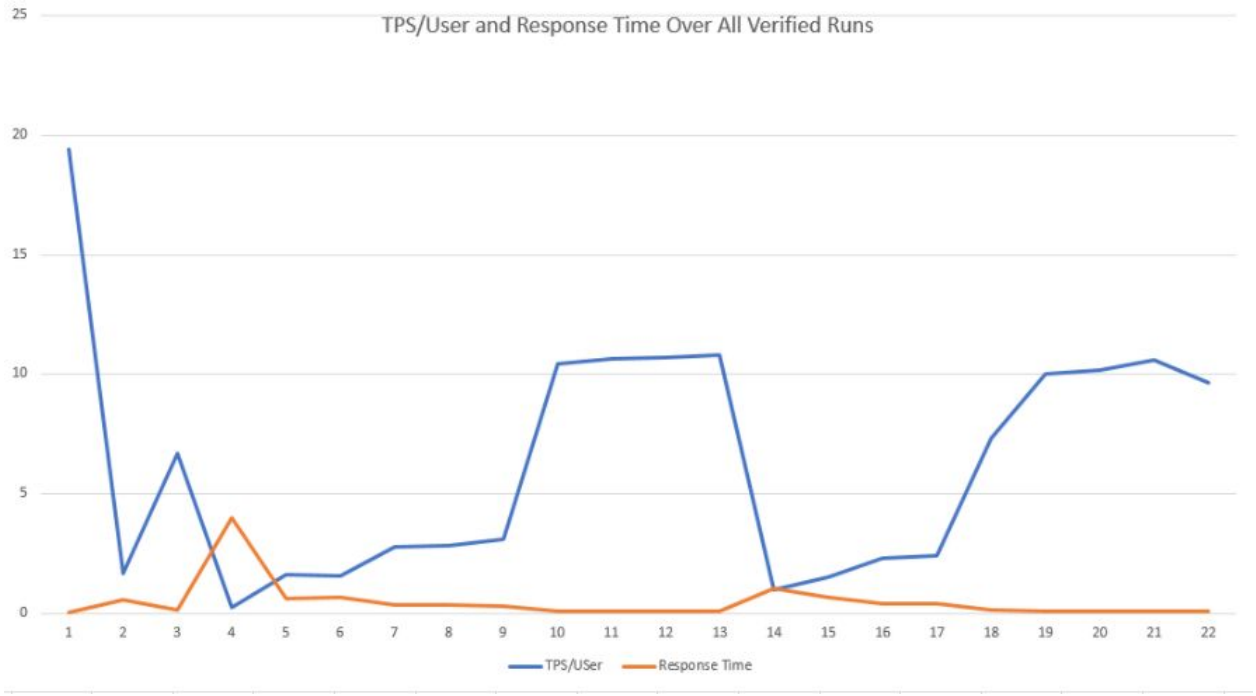

Figure 1: TPS/User and Response Times

Ignoring the anomalous one user run, a pattern can be seen from the figure above. Every time the number of user was increased the system experiences a decrease in TPS/User and increase in response time. Generally, this is because the system processed the requests at the same rate (overall TPS was the same), and improvements needed to be made in order to achieve the same TPS/User as before the scale. In the graph above, points 4, 14 and 22 were the decreases experienced by the user scale. Also not pictured is at point 6 when scaling from 45 to 100 users our system was unable to complete at all until performance improvements were made. From this data, it seems like an achievable TPS/User would be around 10, as we were able to get that rate at the 100 user, scale back up to that at 1000 users, and are close to it with our 1300 user file.

Something interesting we can pull from this is the expected amount of users that the current system could handle. The difference between the (mostly) unchanged architecture of the 1000 and 1300 user files lead to a drop of approximately ~1 TPS per user.

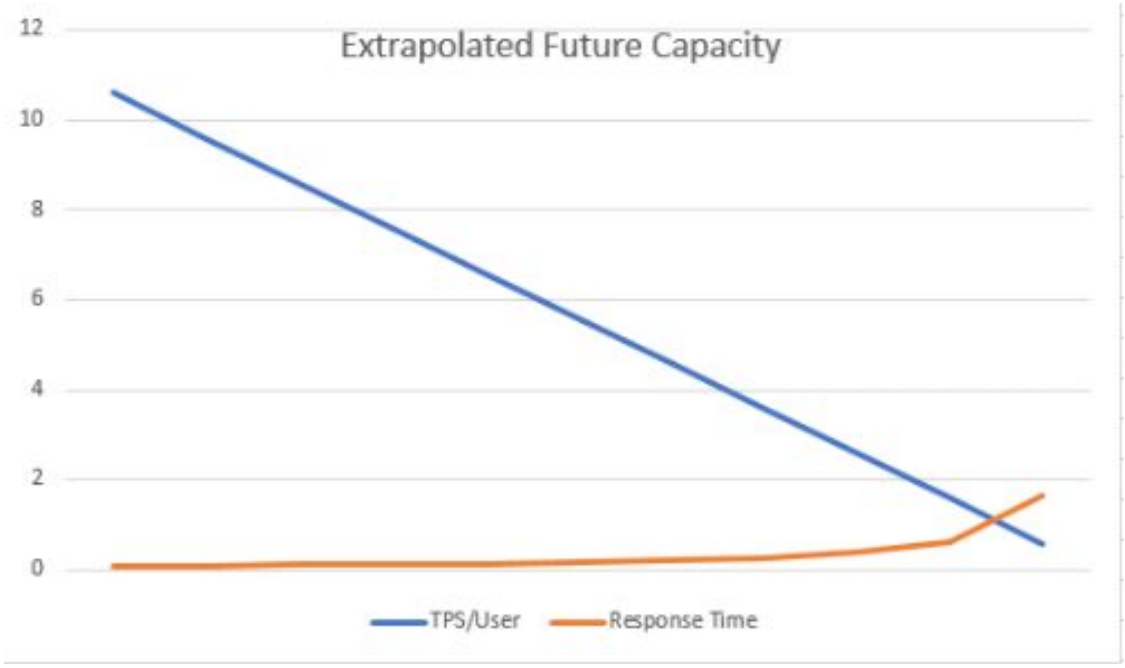

Figure 2: User limit estimation from extrapolation of 1000 and 1300 runs

Extrapolating this data gives us a limit at which the response time crosses over 1 second – what could be seen as an unacceptable response time. At 3200 users the response time is 0.667 and 3500 users brings the response time over the limit to 1.667. Note that this is based on many assumptions such as there are no new bottlenecks, emergent behaviour, and that the utilizations of the resources are constant.

4.0 Future Considerations and Limits

This section of the report discusses the noteworthy points in the system regarding greatly increasing the system capacity.

4.1 Redis and Postgres Capacity

Our write through cache implementation using Redis will allow us to maintain very high speed reads, even when the number of system users increases drastically. With our relatively small required user capacity for the testing purposes of this course (~1300 users), one could make the argument that the Redis in-memory cache is not required, as Postgres will most likely grab most (if not all) of the user records on every “SELECT” statement, since it is quite likely that all of the records will fit in one block on disk. However, this does not scale with real world required capacities, which could potentially require support for millions of users. Redis provides us with a way to store millions of cache entries, and perform reads on the data in O(N) time. In our

testing, even in the 1000 user case where Postgres is most likely to cache all entries in memory, Redis outperforms Postgres in single row lookup queries, due to the fact that it is optimized for in memory operation, and utilizes a hash table for constant time lookup of keys.

4.2 Transaction Server

Our current architecture has the Transaction server storing each users “Buy” and “Sell” stacks, as well as their triggers, in memory. During the 1000 user workloads, this has proven to be no problem whatsoever, as the memory used by the Transaction server instances was absolutely negligible. However, when scaling up to support millions/tens of millions/hundreds of millions of users, the memory usage of this architecture will most likely start causing problems. Capacity planning for a move to support more than 1000 users would include moving the Buy/Sell stacks into Redis or Postgres, and moving the entire triggers system into its own service, which would also most likely use a combination of Redis and Postgres. This would alleviate any memory usage issues which could arise from an influx of users, while also making the Transaction server much more parallelizable, at the cost of data locality.

4.3 Audit Server Capacity

During all of our testing thus far, our architecture with just one audit server and one audit database instance has proven to be more than capable of handling the load of auditing events that come in on the RabbitMQ queues, and also performing the dumplog commands. However, moving forward and looking to increase capacity, we would like to find a way to make parallelizing the audit server feasible, to ensure that our auditing capabilities are always ahead of our transaction capabilities. Ending up in a situation where the system can process 50K stock transactions per second, but only audit the events to keep up with 35K transactions per second is a very bad place to be in, from a business point of view. Increasing our auditing capacity before it is explicitly needed will both ensure we never end up in such a position, and that we will be able to cope with capacities at much larger scales.

4.4 Data Layer

When moving to numbers of users in the order of millions/tens of millions/hundreds of millions, the portion of the system that will see the greatest relative increase in load is the data layer, specifically our Postgres instances. Moving from having thousands of rows to millions of rows can significantly increase resource utilization, and potentially cause queries to run much slower if indexes are not being exploited in an efficient manner. Additionally, the need to maintain a high efficiency sharding scheme will be even more important with millions of users, and we will look into existing solutions for this, such as CockroachDB. Horizontal database sharding will allow us to support millions of users, while also ensuring that we maintain our high level of performance.

4.5 TCP Connection Limits

Our system runs as many commands concurrently as possible to achieve maximum performance. In the worst case scenario, this means that there could be a command running for every single user in the system. To handle running potentially millions of commands concurrently, careful planning in the architecture will be required to ensure no transactions error out due to tcp connection limits being hit. To help with this problem, we will ideally utilize a hardware/software (different OS) environment with a much higher TCP connection limit than the testing lab.

5.0 Conclusion

Though our focus in implementing this trading system was on optimizing it for high throughput in the given environment, our final system proved to be quite capable of operating on much higher user capacities based on the data we gathered during performance testing. Of course, this conclusion relies on a set of assumptions we have made, meaning this is most likely a best case scenario evaluation. With a relatively small amount of changes, we believe we can greatly increase the overall capacity of the system.