Department of Electrical and Computer Engineering
University of Victoria
SENG 462 — Distributed Systems and the Internet

Project Report

Report submitted on:	11 April, 2017
To:	Prof. S. Neville
Names:	J. Cooper (V00XXXXXX) T. Stephen (V00812021) J. Vlieg (V00XXXXXX)

Architecture and project plan	_____	/5
Security	_____	/5
Test plan	_____	/5
Fault tolerance	_____	/5
Performance analysis	_____	/5
Capacity planning	_____	/5
Total	_____	**/30**

Contents

List of Figures

List of Tables

Overview

The goal of the project was to build a distributed day trading system, with a focus on performance. The client original stated the following actions as business requirements:

- View their account
- Add money to their account
- Get a stock Quote
- Buy a number of shares in a stock
- Sell a number of shares in a stock they own
- Set an automated sell point for a stock
- Set a automated but point for a stock
- Review their complete list of transactions
- Cancel a specified transaction prior to its being committed
- Commit a transaction

Additionally, the overall architectural goals of the system were:

- Minimum transaction processing times.
- Full support for required features.
- Reliability and maintainability of the system.
- High availability and fault recoverability (i.e. proper use of fault tolerance)
- Minimal costs (development, hardware, maintenance, etc.)
- Clean design that is easily understandable and maintainable.
- Appropriate security
- A clean web-client interface.

These requirements form the basis for the distributed daytrading system commissioned by DayTrading Inc.

Chapter 1

Architecture

1.1 Original architecture

Can steal most of this from the first report.

1.2 Technology

1.2.1 Golang

1.2.2 RabbitMQ

1.2.3 Redis

1.2.4 Postgres

1.2.5 Websockets

1.3 Work plan

1.3.1 Timeline

1.4 Final architecture

1.4.1 Worker

1.4.2 Quote manager

1.4.3 Audit logger

1.4.4 AutoTX manager

1.4.5 Load balancer

1.4.6 Frontend

1.4.7 Docker

Chapter 2

Security

lol. just lol.

Chapter 3

Test plan

3.1 User testing

Validate command pre/post conditions. Tested through FE?

Chapter 4

Fault tolerance

Chapter 5

Performance analysis

5.1 Decreasing quote retrieval time

The legacy quote server can delay for up to four seconds before sending a response. This delay is vastly greater than the typical command execution time of dozens of microseconds (see 5.3). The legacy server is the greatest barrier to high command throughput and dozens of hours of research and design was spent mitigating the effects of its delay.

5.1.1 Statistical analysis of legacy quote server

We sent a large number of serial requests to the quote server and recorded the response time with a shell script. Figure 5.1 shows the distribution of response times follow an exponential distribution with 65.75% of responses experiencing only network delay. The expected value of the response time is 563.3 ms.

do a goodness of fit test

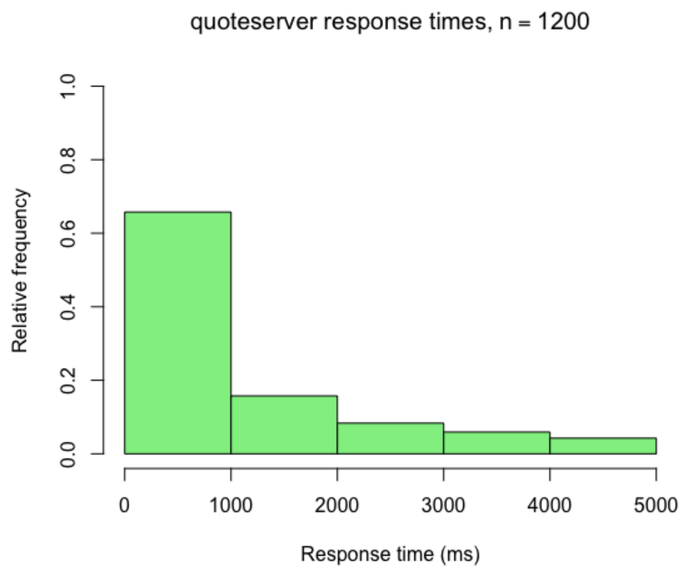

Figure 5.1: Histogram of legacy quote server response times with 1 s buckets

The one second buckets in 5.1 obscures the fact that results are clustered after whole second values. Removing the constant delay portion from each bucket yields the distribution (Table 5.1) of variable network and processing delays.

From this data we can conclude that if the legacy quote server has not sent a response after 30 ms then we will wait at least 1 s for a response.

Table 5.1: Legacy quote server network delays

Minimum	1st Quartile	Median	Mean	3rd Quartile	Maximum
7 ms	9 ms	9 ms	9.414 ms	10 ms	26 ms

5.1.2 Using timeouts to ensure fast quote retrieval

We decided to use a request timeout strategy to minimize the total time spent waiting for a quote. If there was no response from the quote server after a given timeout we cancel the request by closing the socket connection and issue a new request. We used the tail of the network delay data (i.e. ..., 16, 16, 17, 17, 20, 26 ms) to set an initial timeout at 20 ms and used a 5 ms exponential backoff. This backoff strategy requires six iterations to exceed the expected delay of 563.3 ms. Exceeding 1 s total delay has a likelihood of 0.0299%. If the total timeout exceeds 4 s and there is still no response from the quote server then the service is assumed unreachable and the quote manager raises an error.

5.1.3 Timeout effectiveness

We implemented the timeout strategy in 5.1.2 and gathered response time data directly from the quote manager. Figure 5.2 shows frequency of retry attempts before a quote resolved. Strangely, the distribution does not match that of Figure 5.1 and the legacy quote server has an 89.35% likelihood (instead of 65.75%) of resolving in under 20 ms. Day Trading Inc. has assured us that the behavior of the legacy quote server is stationary so perhaps the change arises from our serial and timeout-retry request methods.

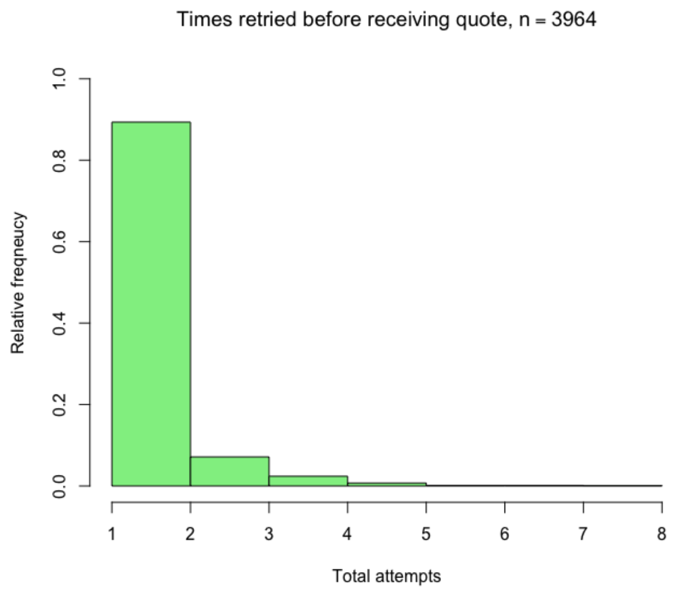

Figure 5.2: Quote retry attempt histogram

The distribution of total waiting times is shown in Figure 5.3. 89.18% of quotes are resolved before 50 ms.

Only three quotes take longer than 500 ms and none longer than 850 ms. This adds confidence to our derivation that waiting longer than 1 s for a quote should be a $\approx \frac{1}{3400}$ event.

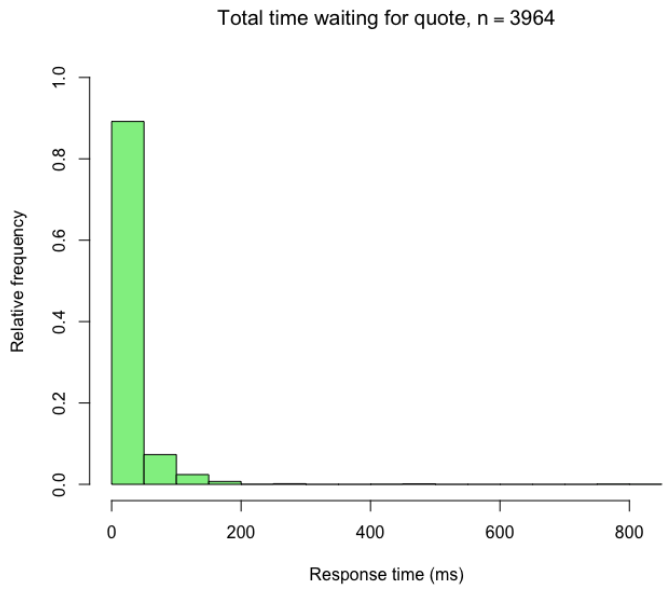

Figure 5.3: Total waiting time to retrieve a quote with 50 ms buckets

5.1.4 Minimizing lingering TCP connections

When a TCP connection is terminated with a FIN command the socket enters the TIME-WAIT state until the termination is acknowledged with a FIN-ACK command. The socket stays in the TIME-WAIT state for twice the connection MSL, typically two minutes. Each open socket occupies approximately 1 Mb of memory and is considered an open file. Thus, the number of open sockets is limited the the system memory and OS limitation on concurrent open files.

The legacy quote server timeout method necessarily leaves many connections in the TIME-WAIT state. When we first implemented the timeout method the quote manager would become unresponsive and crash as connections lingered in the TIME-WAIT state and the host had its memory occupied entirely with open TCP connections.

Changing connection methods in our application from the generic `Dial` method to the specific `DialTCP` method resolved the lingering connection issue. We suspect that `Dial` discards the connection in such a way that the FIN-ACK is not received by the socket and the OS maintains the connection for the entire double MSL period. This difference is behavior is undocumented and should be disseminated to developers who use Go in applications with high TCP socket turnover.

5.2 Worker scaling

5.2.1 The sixty second golden window

5.2.2 Scaling results

5.3 Command execution time analysis

In order to determine the performance of individual commands we created a fork of the worker that logged elapsed wall clock times for *valid* commands. A valid command is properly formatted (e.g. no negative dollar amounts) and meets all account state pre-conditions. This assures a common execution path for results and prevents a multi-modal distribution caused by commands resolving early and avoiding lengthy inter-service communication delays. Results are summarized in Table 5.2.

Table 5.2: Command execution times gathered from 3 runs of the 1000 user workload

Command	n	Execution time percentile (µs)				
		50%	95%	99%	99.9%	99.99%
Add	168 227	24.9	35.7	54.3	176.7	1 977.6
Buy	197 897	217.5	427.8	488.2	65 650.4	230 294.0
Cancel Buy	38 812	25.7	36.0	53.9	189.3	2 197.3
Cancel Sell	41	27.3	31.9	50.4	55.1	55.6
Cancel Set Buy	102 925	21.5	30.8	46.8	169.5	1 979.1
Cancel Set Sell	102 642	21.5	30.8	46.8	177.8	1 998.6
Commit Buy	121 658	27.1	37.9	55.8	188.6	1 996.4
Commit Sell	92	25.7	33.1	39.4	73.0	76.3
Display Summary	225 298	21.3	30.2	45.9	161.9	1 953.0
Dumplog	3	45 754.6	45 926.5	45 941.8	45 945.3	45 945.6
Quote	355 578	213.1	429.7	809.8	218 334.6	293 673.5
Sell	152	220.7	420.8	455.4	481.6	485.2
Set Buy Amount	238 759	22.2	31.6	47.5	160.7	1 986.2
Set Buy Trigger	229 465	22.0	31.5	47.7	171.9	1 972.5
Set Sell Amount	196 997	22.1	31.6	47.9	164.4	1 979.5
Set Sell Trigger	190 648	22.0	31.5	47.6	159.9	1 982.9

Command execution times show two general patterns. Commands that do not use stock prices as pre-conditions (e.g. Add, Commit Buy, Set Sell Amount) resolve at a median time of ≈ 25 µs. Commands that do use stock prices resolve at a median time of ≈ 220 µs. The order of magnitude slow down is the result of storing stock values externally (but locally) in Redis. The retrieval delay is costly but is countered by offloading quote TTL enforcement.

There are several interesting anomalies in Table 5.2. The first is the paucity of data from Sell commands. This is because the set of pre-conditions is the largest for any command: the user must have already purchased stock *and* chosen a sell amount that resolve for pseudo-random stock prices.

Next, the 99.99% values for the non-stock, high-frequency commands increase by an order of magnitude compared to the 99.9% values. Since execution of these commands doesn't involve any external services the delay is likely the result an ill-timed application garbage collection cycle. Since this delay doesn't manifest in the tails of the low-frequency events (e.g. Cancel Sell) the likelihood of being slowed by a garbage collection event is $\approx \frac{1}{30000}$.

Dumplog data is largely uninformative, given the extremely small sample size. The execution time results from a path that necessarily interfaces with external services and is subject to communication delay.

Buy and Quote do exhibit bi-modal behaviors despite the experiment setup. This is a result of quote cache hits and misses having execution times that differ by several orders of magnitude. Histograms for all of the commands are presented in Appendix.

Manually insert the appendix number

Chapter 6

Capacity planning

Progressing through the series of workload files stresses different parts of the trading system architecture. Workload files differ by the number of unique users, quotes and total transactions. Early workload files have few unique users and quotes but rapidly increase the number of transactions. The primary design goal becomes minimizing individual transaction times. Later workloads rapidly increase the number of unique users and quotes, exposing inefficiencies in different parts of the system that were often the result of optimizing a design for high transaction throughput for early workloads.

This chapter outlines the often surprising manner in which our system failed as we progressed through the workloads, and the design changes that followed. Most often, the manner of failure was the result of an assumption that would be true at smaller scales but became invalid at a later point.

6.1 Logging throughput

The 1000 user workload was the first occasion for our system to operate at nominal TPS for a significant portion of its runtime. Since each transaction needed to write at least one entry to the audit log the volume of log messages was unprecedented. Controlling the "firehose" of log messages was the most significant architectural redesign. It included several false starts and, ultimately, reached a workable but flawed solution.

link to image with 100 v 1000 tps

6.1.1 Limits of logging to a flat file

From the initial prototype through the 100 user workload, the audit service wrote directly to an `.xml` file that could be submitted for validation. Log messages were removed from RMQ and stored in memory for writing by separate threads. However, running the 1000 user workload exceeded the rate that the audit service could clear messages from RMQ, causing a significant backlog of messages to develop. As the total message backlog size approached 700k the rate that messages could be exchanged slowed, causing a slowdown in the rest of the system as execution was blocked on message exchange. Soon after, services would fail as they lost their connection to the RMQ server.

As noted in the "Production Checklist" section of the RMQ user guide[1], performance is heavily tied to available RAM. As the backlog increases RMQ will begin swapping RAM to disk to ensure persistence. The IO penalty for writing to disk causes an intense slowdown. Since the worker services generating the logs are not capable of throttling they eventually push RMQ into resource exhaustion and failure.

Direct to file logging was never intended for production use. Creating per-user dumplogs would be onerous since there was no direct method for searching or sorting the log file. Leaving

[1]https://www.rabbitmq.com/production-checklist.html

the log file implementation in place for most of the project allowed us to focus development efforts on optimizing the quote manager and implementing the auto transaction service. Letting RMQ fail illuminated the "danger zone" for RMQ on the lab machines. Different audit logger refactors could be compared for effectiveness by monitoring the RMQ backlog.

6.1.2 Logging directly to an RDBMS

The first refactor involved inserting logs into Postgres and writing to a file on an as-requested basis. This solved the problem of creating per-user log files but throughput was significantly worse than writing direct to a file. With direct to file, the 100 user workload with 100k transactions generated 2k backlogged messages on RMQ. With the Postgres refactor, the 100 user workload resulted in a 20k message backlog. No attempts at larger workloads were made after this poor result.

This performance slowdown is not surprising. The flat file and RDBMS both store the preformatted `.xml` entry for the event. In addition, the RDBMS stores extra data about the user name, transaction type and creation time to enable queries. The RDBMS is storing more data than the flat file. In addition, the RDBMS suffers a performance penalty from indexing data on insertion. While there are methods to mitigate these problems, such as connection pooling, the performance degredation was extreme enough to justify larger service refactors.

6.1.3 Processing logs with ELK

An RDBMS enables rich querying and enforces data integrity — useful features that are not relevant for storing an append-only log. Moreover, the indexing that provides those useful features introduces a performance penalty that limits throughput. Specialized log storage solutions forgo rich indexing in order to maximize write throughput.

The Elasticsearch - Logstash - Kibana (ELK) suite of applications from elastic.co[2] is a popular distributed log storage method. Logstash consumes and transforms data for indexing and storage in Elasticsearch. Kibana is a graphical monitoring suite that provides information about the logging rate and health of the logstash and elasticsearch services.

A prototype was created that deployed the ELK stack in separate Docker containers collocated with the audit logger. Logstash consumed messages directly from RMQ and sent them to Elasticsearch for indexing and storage. The prototype had abysmal performance. With the 45 user workload there was a 7k (out of 10k total) backlog. Development was abandoned at this point.

The poor performance of the ELK stack was directly related to its resource limitations. Each part of the ELK stack performs better with more available RAM. Collocating all services severely limited the available RAM. Also, ELK requires a non-trivial amount of JVM and OS tuning to provide optimal resource availability. Although there are guides for this process it was unclear how to apply their recommendations on the tower of abstractions in the production environment: a docker OS on a VM OS on a host OS, each needing their own tuning.

The Elasticsearch scaling guide[3] recommends adding more shards (i.e. independent instances which maintain a partition of the data) to increase write throughput. This throughput solution — a distributed system within a distributed system — directly links scaling to resource demand. Scaling Elasticsearch would likely reduce the number of systems available to host workers and limit the maximum TPS. The problem of high log throughput would be solved by removing the ability to create logs at a high rate.

[2]https://www.elastic.co

[3]https://www.elastic.co/guide/en/elasticsearch/guide/current/scale.html

6.1.4 Buffered logging

The problem with the RDBMS solution in 6.1.2 is fundamentally a mismatch between the production and consumption rate of log messages. The solution to this problem is to place a buffer between the producer and consumer.

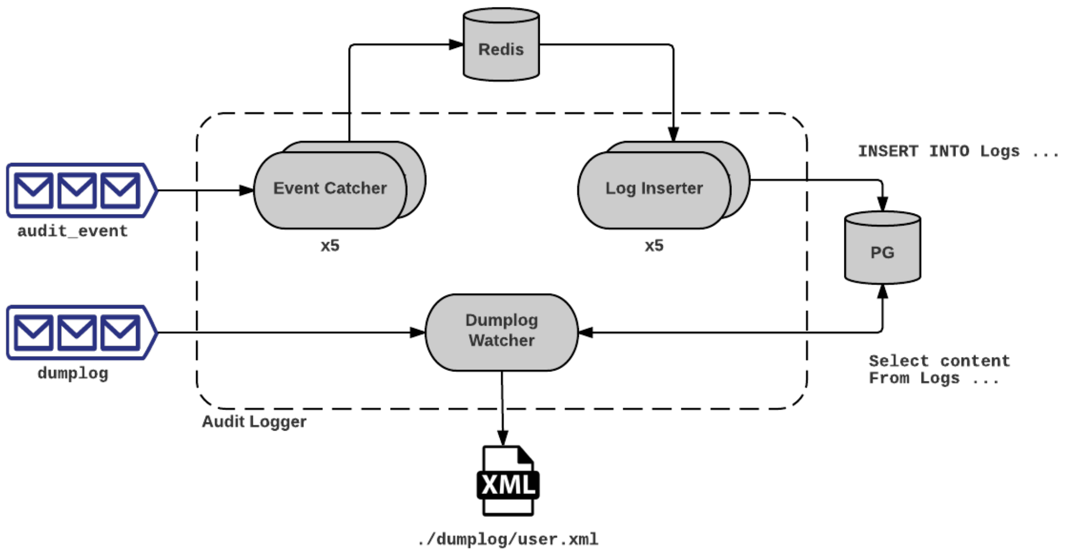

Figure 6.1: Buffered audit logger

Figure 6.1 shows how multiple `Event Catcher` workers remove incoming log messages from RMQ and place them in Redis. `Log Inserter` workers remove items from Redis and store them in Postgres. With this design, the 1000 user workload only reached a 20k backlog of messages and had no observed performance degradation. Although a run would finish in around 45 s it would be upwards of 6 min to finish insertion into Postgres. When a message for a dumplog was processed it would display messages had successfully migrated to Postgres but was unaware of those still in Redis. This is a soft violation of the business requirement that a dumplog should show all transactions proceeding itself. We believe this is acceptable since, in actual use, there is no clear "end of work" to capture. All log messages will make it into Postgres for querying so the requirement is *eventually* satisfied.

This method has an upper limit to its effectiveness since the Redis buffer could run out of storage space under periods of sustained high TPS. The boundary of the buffer memory was not encountered during any testing and we cannot speculate about its value. In order to determine the limits we would need a workload file larger than the final workload. Alternatively, we could induce a period of sustained high TPS by removing the requirement to re-fetch expired quotes and concatenate existing workloads.

6.1.5 Alternate solutions

When the buffer method in 6.1.4 reaches its limit there are several possible development paths for proceeding forward:

1. *Agglomerate messages*: Message traffic can be reduced by combining multiple log events into one message. Workers would only emit messages for logging at fixed intervals or after a certain number of events (whichever comes first) and reduce the overhead associated with creating, sending and processing RMQ messages. The optimal message size would have to be determined through experiment.

2. *RMQ scaling*: RMQ is capable of its own distributed deployment. Increasing resources available to the message bus would allow more messages to be stored before removal into the buffer.

3. *Robust message passing*: Apache's Kafka[4] provides functionality similar to RMQ but is optimized for message storage and large backlogs. This allows consumers to operate at different rates and removes the need to process log messages faster.

6.2 Quote manager scaling

The system's TPS grows almost 10× once a full set of quotes has been retrieved. further increases to average TPS could be gained by decreasing the time spent fetching quotes. The methods in 5.1 brought the quote server response time to its lower limit so further efficiency could only come from horizontally scaling the quote managers. Intuitively, doubling the number of quote managers should decrease the total time to retrieve all quotes by half, provided the workload is split evenly. If only it were so simple.

6.2.1 Building a "snoopy" quote manager

With one week until the final deadline we decided to refactor the quote manager to participate in a multi-quote manager environment. We ported the "snoopy caching" functionality from the worker and audit logger into the quote manager. The "snoopy" quote server could listen to quote broadcasts and update its local cache accordingly. With this functionality, multiple quote managers could act as workers servicing requests from a single RMQ queue.

A message header with the ID of the quote manager that serviced the request was added to all quote broadcasts. The quote manager cache updaters would discard messages that originated from its own quote manager. This is inefficient but necessary because RMQ does not allow an "anti-match" for message routing keys. That is, one cannot specify, "Capture all messages *except* ones that follow this pattern."

Total development effort was approximately one hour.

6.2.2 Performance analysis

Figure 6.2 shows that TPS correlates *negatively* with the number of snoopy quote managers. This is very counter-intuitive and deserves reflection.

Comparing the single quote manager deployment in multi and single architectures gives a sense of the overhead associated with discarding quote broadcasts. The relation becomes less clear when two and three quote managers are used: the system benefits from having to retrieve fewer quotes per quote manager but adding quotes to the cache also has a delay. The benefits from horizontal scaling start to manifest when three quote managers are used, but it takes the form of, "things stopped getting worse," instead of "performed better than a single quote manager."

The single architecture quote manager is very fast because it already functions like a multi machine quote manager. Each new request spawns a thread that handles communication with the legacy service. Most of the threads are blocking on a response from the legacy service making it very likely that a new thread will find the application in an idle state. Since workers block on the completion of a quote command the maximum number of simultaneous quote requests is equal to the number of workers. Hence, the number of threads requesting quotes in the quote manager is capped, thus preventing thread creation runaways and CPU starvation in the quote manager. Adding more quote managers doesn't increase the number of simultaneous quotes that can be requested by workers.

[4]https://kafka.apache.org

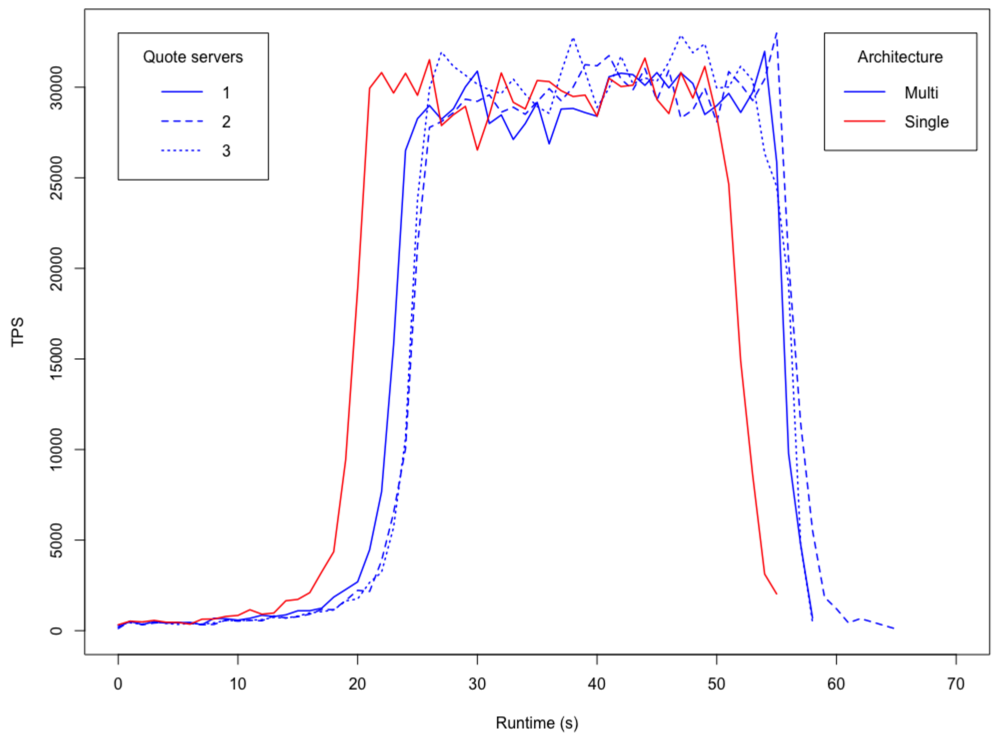

Figure 6.2: Performance of multi and single quote manager designs on 1000 user workload

6.2.3 Alternate solutions

The snoopy quote manager was implemented because it was a small amount of development effort for a large potential payoff. There is another way to implement multiple quote managers, although it is more complicated: quote symbols could be hashed to associate with unique quote managers. This prevents the need for quote managers to do snoopy caching. However, it introduces problems with scaling since the hash will depend on the number of available quote managers.

The snoopy quote manager can scale easier since failures would be independent. As the number of workers continues to grow, Figure 6.2 should be re-run to determine the break point between the communication overhead and the increased capacity for simultaneous quote requests.

6.3 Worker loading

To run the workload files we loaded sections of 3300 transactions into workers in a round-robin manner. By measuring the total number of transaction in a worker's backlog at each loading cycle we could determine the limits of our round-robin loading method.

6.3.1 Worker backlog analysis

The curves in Figure 6.3 have two distinct parts: an upward slope that becomes steeper as the number of workers before coming to a maximum and either plateauing, as with eight workers; decreasing at a fixed rate, as with nine to eleven workers, or; decreasing and leveling out, as with twelve workers. It's important to note that the x-axis represents loading cycles and not a uniform time scale. Round robin loading cycle times increase as the number of workers increases.

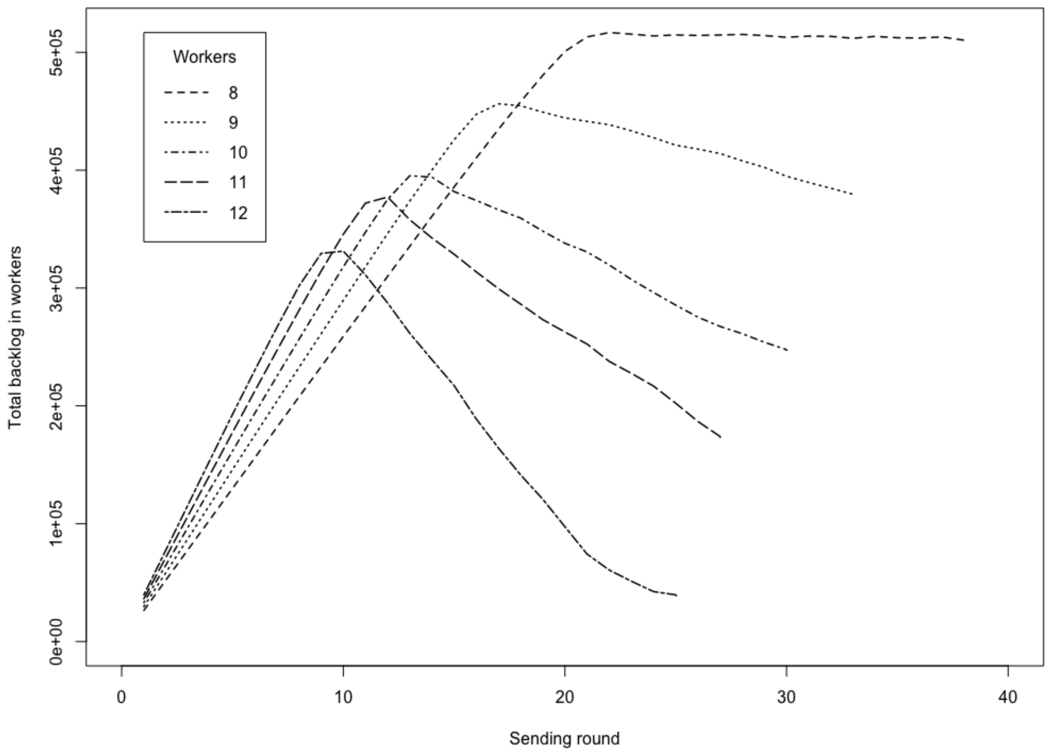

Figure 6.3: Total worker backlog for 1000 user workload

The slope of the curve represents the ratio of transactions coming in to a worker over transactions completed between successive cycles. Essentially, this is

$$\frac{3300\,\text{transactions per cycle} \times \text{cycles per second}}{\text{transactions per second}}$$

The peak in the curves occurs when the system has retrieved a full load of quotes the transactions per second increases drastically. For eight workers, the slope is approximately 1, indicating that the input and output rates are equal. The work in 5.2.2 indicates that the transaction input rate must be approximately 3600 transactions per second for each worker. As the number of workers increases, the number of transactions per second for a worker stays (mostly) constant but the cycles per second decreases. The leveling off with twelve workers indicates that the workers are starved for transactions near the end of the run.

The change in the rising slopes is also affected by the increased cycle time but the correlation is less direct. During loading, the transactions per second is significantly lower than 3300 and constant regardless of the number of workers. The slope increases with the number of workers because the capacity for transactions in the system increases (i.e. each worker has its own cache). As the cycle times become longer this decreases the slope.

6.3.2 Worker starvation analysis

Figure 6.4 shows how workers entering starvation at different times. All workers have identical backlogs up until the peak where higher-numbered workers operate at nominal TPS earlier. This is because the low-numbered workers early in the round-robin cycle are “alone” for longer with the transaction list and are more likely to encounter uncached quotes. High-numbered

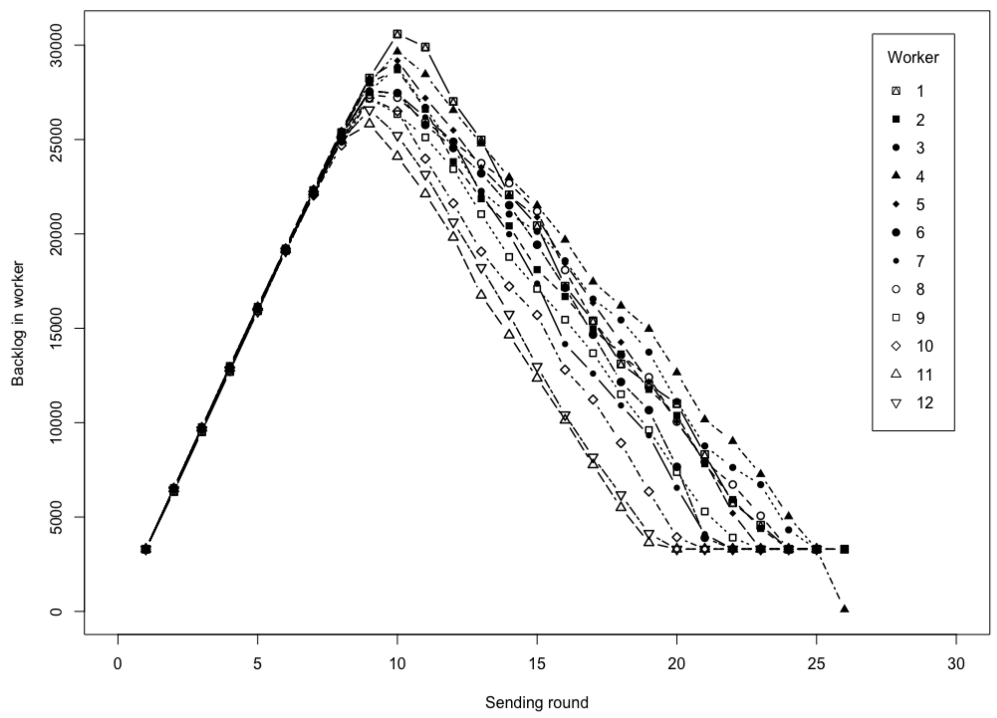

Figure 6.4: Workers entering starvation during a 1000 user workload

workers benefit from the pre-caching. Unfortunately, they enter starvation approximately ten cycles before loading finishes and represent an inefficient use of resources.

The rates of descent are mostly uniform, with the exception of worker 4 (machine B133). Consistently, this machine performed worse than its peers. This could be the result of hardware aging and general, spooky “cruft” on the machine or a non-uniformity in the workload distribution. The latter is unlikely since worker 4 was slow regardless of the total number of workers.

We did not undertake any tests with thirteen workers because of these results with twelve — adding more workers would cause the system to enter starvation earlier and would be a poor use of resources.

6.3.3 Alternate solutions

The ideal operating state is when the incoming and outgoing transaction rates are equal. In order to achieve this state we would have to implement a feedback system with the transaction loader. The number of backlogged transactions in a worker could change the number of transactions sent in a loading cycle.

Though not trivial, this is a very tractable solution. However, we feel it would be overly specific to the testing environment. Could an actually existing trading system exert backpressure on user loads to throttle demand? This seems unlikely, or at least one that would lead to frustrated users. Further testing should involve a dynamic *stream* of transactions that could exhibit richer behavior like cyclic demand cycles and surges. Though the same risk of overfitting the prototype software to the test environment is present, the fidelity has increased and the solutions should be more generally applicable.

Appendix A My appendix

Here is some text for my appendix.