Table of Contents

Introduction

An old decrepit supermarket chain, Tigle, is embarking on a rapid acquisition strategy to buy up new stores nationwide. exercise. They're in need of a solution to integrate these new stores into their existing business system and hired us to help them build the new system.

For the third part of the contract, we built upon the previous milestones and completed the design of the system. We made modifications to the server and database design in the previous contract to make the application more responsive to client requests and introduced new microservices to delegate tasks.

Previously, we had a mock client exerting a heavy POST requests load to the server and the server concurrently writing the record to the relational database. We now delegate this stress to a microservice called *Purchases* that "eventually" persists the record to the database. The server writes all the new POST request payloads to a queue to deliver the request to the microservice.

An additional microservice called *Store* is introduced. Store also receives every purchase record and create an in-memory data structure that records the quantity of each item purchased at each store. The Store microservice is designed to answer two queries:

1. What are the top 10 most purchased items at Store N
2. Which are the top 5 stores for sales for item N

The implementation of the client can be found in the Github repository below.

Git repo: https://github.com/hjyou07/Distributed_Systems_Project/tree/A3

Design Overview

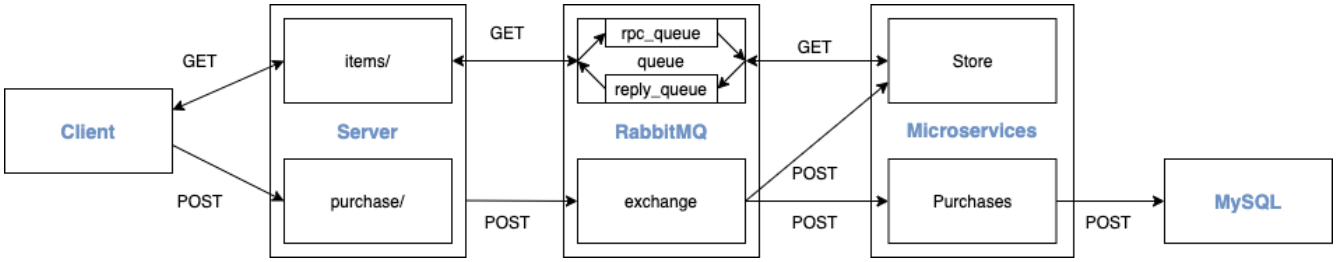

Figure 1. Overview of the System Structure

The overall structure of this Tigle Supermarket system is shown in *Figure 1*.

The client sends concurrent requests to the server at two different endpoints. One takes GET requests to fetch some insights on the sales, and the other takes POST requests to persist the *Purchase* data in a relational database. Server receives the requests and propagates the payload to the RabbitMQ server.

RabbitMQ keeps an exchange to fan out the incoming *Purchase* POST requests to the microservices, **Store** and **Purchases**. **Purchases** writes to the database, and **Store** saves the data in an internal data structure to efficiently respond to GET requests.

It also keeps a separate queue dedicated for GET requests supporting remote method invocation(RMI). The endpoint of this GET request is the **Store** server and **Store** returns a list of top 10 items for a store or top 5 stores for an item back.

Client

The mock multithreaded client imposes stress on the server, send 300 purchases POST request every hour per store. Each store operate 9 hours a day and is in 3 different time zones; meaning that the opening hours are staggered. Refer to A1 report for a detailed implementation of the client.

Server

The server now has two servlets serving different endpoints. The endpoint */purchase/** takes the POST requests with a payload, and another endpoint */items/** takes in two GET requests to answer business insights. A GET request to */items/store/[storeID]* returns the top 10 items of a store of interest, and */items/top5/[itemID]* returns the top 5 stores of an item of interest.

Now that the *Purchase* servlet is passing requests to the queue, it needs to open multiple channels to connect to RabbitMQ for concurrency. A pool of channel is maintained where each POST request borrows a channel from the pool and return it for other threads to use.

Note that the servlet for GET requests is designed to be single-threaded. There was no need for it to be multithreaded as those requests are designed for management, hence the request won't be as frequent.

Server is deployed on an Amazon EC2 Linux instance, and multiple load balanced servers are tested for performance efficiency. As later discussed, a single server suffices to endure the workload.

RabbitMQ

The RabbitMQ works as a broker of the message between the server and each microservices which are hosted separately. It asynchronously passes on the request payload as a String object that can later be parsed in each microservices to adequately fit their purpose. Even though mainly asynchronous, our queue guarantees some degree of data safety with consumer acknowledgements. Each microservices, will confirm the delivery of the message once the received message is properly processed.

The queue can be configured to be persistent, which means the messages can survive a queue server failure. Persistent queues introduce a bottleneck and a trade-off between performance and durability which will later be discussed.

RabbitMQ is deployed on an Amazon EC2 Ubuntu instance and remains one of the two main bottlenecks. As it has to maintain double the volume of original requests by fanning out, the memory can be a limiting factor. RabbitMQ also starts writing to a disk when there's too many messages in the queue affecting the performance, so it was important to maintain the queue length reasonably short.

The memory threshold of RabbitMQ service is set to 40% of installed RAM by default, but we increased the threshold to 70% to avoid memory shortage as it is the only service running on an instance. Five-hundred threads were used to pull messages off the queue per microservice. This kept the queue size as low as 19 messages.

There is a dedicated queue for a remote procedure call (RPC/RMI), which actually consists of two queues; *rpc_queue* propagates the GET request from the server and *reply_queue* returns the result of the GET request back to the server.

There's a *fan-out* exchange for the POST requests, which delivers the same message to both microservices that subscribes to the exchange. The messages will be parsed differently in each service for further adequate processing.

Microservices

There are two microservices serving out needs which are independently deployable and have their own life cycle.

Purchases, lifts off the pervious stress of synchronous database write from the server and make the operation asynchronous. This uses a producer-consumer based solution to write *Purchase* data to the database "eventually". So the general structure looks very similar to what we have seen in the previous milestone's server, except that now its executing thread is configured to pull messages off the queue, and the child threads writing to the database are managed by the *ExecutorService* as a fixed thread pool of size 500.

Store, as mentioned before, keeps an internal data structure to organize the incoming *Purchase* data such that when GET requests are received, the results are returned efficiently. This

internal data structure is implemented as a 2d primitive int array, synchronized for each row, where each row represents each item and each column represent each store. For our system, it results in a 100,000 by 512 array. Each cell in an array represent the number of items for a particular itemID and storeID.

When GET requests arrives at *Store,* the microservice runs an algorithm to fetch the top 5 stores or top 10 items. Depending on what's requested, it either iterates a specific row for a particular itemID or a column for a particular storeID and adds the element to the min heap of specified size – 5 or 10. If the current element is bigger than the top of the heap, the heap pops out the top element and adds the current element to itself. At the end of the iteration, the heap will be populated with 5 "biggest" stores or 10 "biggest" items where it means they have the largest number of sales.

As mentioned before, GET request for *Store* are single-threaded, but the *Purchase* POST requests are multi-threaded. To allow concurrent writes to the 2d array, each row is protected with a synchronized block. This implies that the data structure allows as many as 512, or the number of stores, concurrent requests, which played a factor in deciding the pool size of the service. A thread pool of size 500 is managed by the *ExecutorService* as well.

The executing thread is responsible for the life cycle of this fixed pool, and it also starts a single thread that processes the GET requests. When a GET request arrives, we can either iterate through a specific row for a particular itemID, or a specific column for a particular storeID to return the appropriate result from the data structure.

MySQL

The *Purchase* data is eventually persisted in a relational database. The Purchase microservice connects to a MySQL instance, and writes the data. What to note is that with limited resources accessible by free-tier Amazon RDS instance, only 66 concurrent connections are allowed to the database. Hence there are 64 threads in database connection pooling. However, there are 500 threads pulling messages off of RabbitMQ. This discrepancy in the number of threads between the threads that perform writing and the threads that pull from RabbitMQ results in a piling of messages waiting to be serviced – the second bottleneck.

RDS instance has a capacity to handle the number of I/O requests that have been submitted by the application but have not been sent to the device because the device is busy servicing other I/O requests. The queue depth was monitored in the Amazon RDS management console to ensure that it's kept in a sensible range.

Statistical Analysis

A few factors were considered in the implementation of the design and each factor was tested in their performance. Discussed here is the system performance depending on queue persistency and number of server instances.

Persistent vs. Non-persistent

We had to consider the possibility of losing messages due to a queue node crash and experimented with persistent versus non-persistent queue configuration to explore the inherent performance trade-offs.

The queue persistency is configured with a boolean argument when declaring an exchange or a queue, and the message that is delivered also has to be marked persistent. *Table 1.* compares the performance of persistent and non-persistent queue on a single server instance. Non-persistent queue has a great performance at throughput of around 4300 requests per second, but once the queue is configured persistent, the throughput decreases significantly to around 2800 requests per second.

Also worth briefly noting is that the throughput slightly increased with 512 threads than 256 threads, indicating we're still benefiting from concurrent requests at this point.

Table 1. Non-persistent vs. Persistent queue on a Single Server

		Single Server	
	Threads	wall time (s)	throughput
Non-persistent	256	161	4293
	512	308	4488
Persistent	256	247	2798
	512	493	2804
	Threads		% difference
Non-persistent	256		-34.82
Persistent	512		-37.52

This decrease in performance is also captured in the real-time monitoring of RabbitMQ node shown in *Figure 2.* and *Figure 3.* Note that Consumer ack rate is nearly double the Publish rate, as the exchange fans out the incoming messages to two microservices.

Not shown in *Figure 2.* but shown in *Figure 3.* is the Disk write rate. To maintain the persistency, RabbitMQ writes the messages out in a disk so that it is resistant to queue node failure. I/O operation introduces a significant overhead which is slowing down the throughput of messages.

Whether to keep the queue persistent or not would depend on the business logic and up to the client to decide based on the trade-offs we explored.

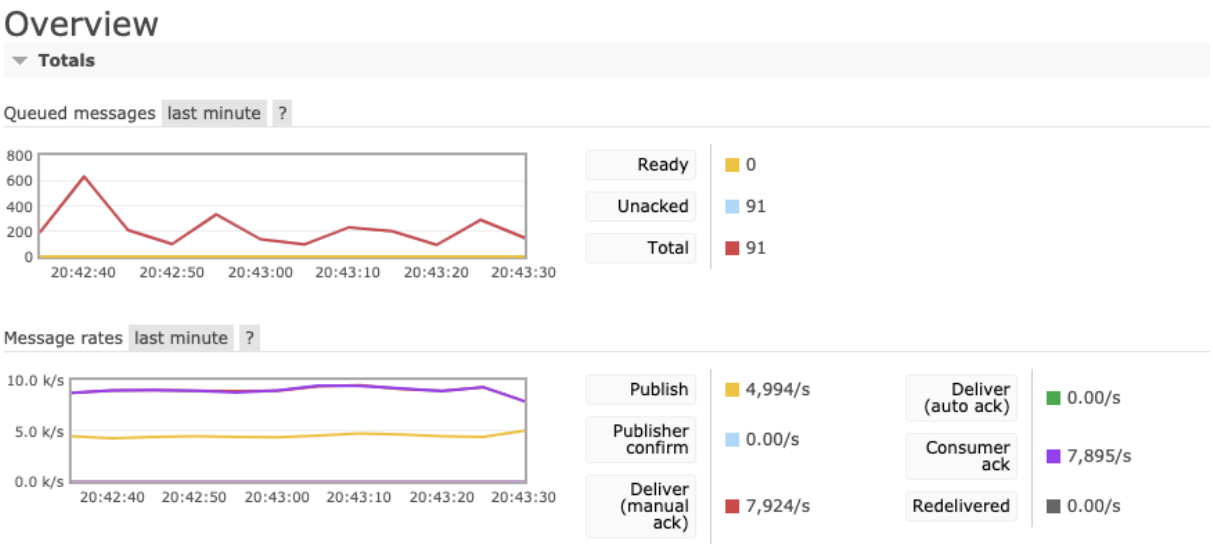

Figure 2. Monitoring of non-persistent queue

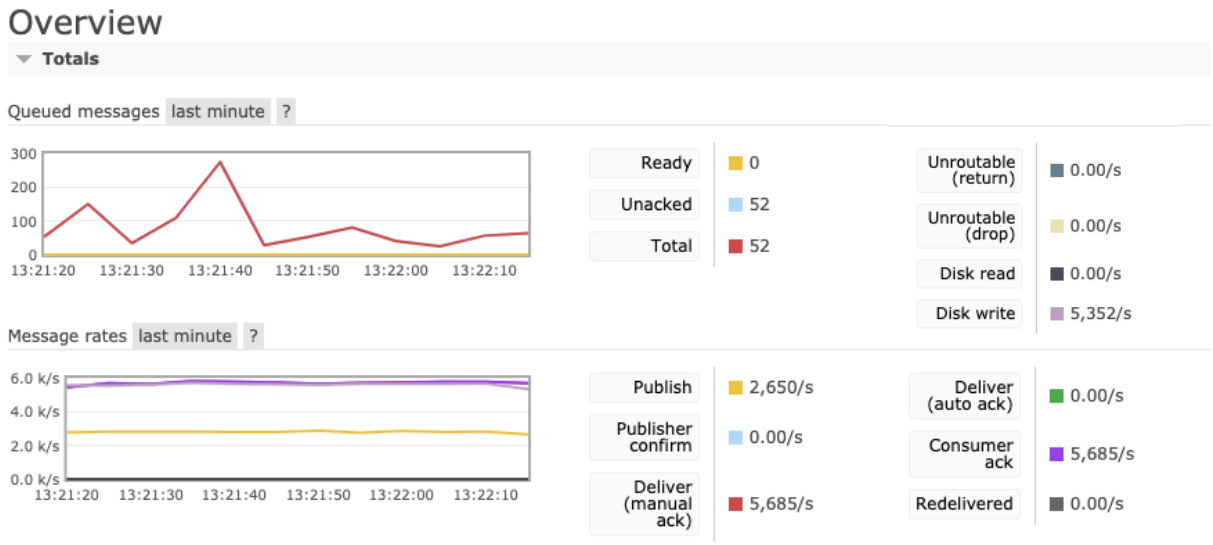

Figure 3. Monitoring of persistent-queue

Single Server vs. Load Balanced

In our previous milestone, we had examined the performance difference between a single server and load balanced servers and observed a great increase in request throughput when the servers were load balanced. This is because the server itself was performing a heavy workload of synchronous writes to the database. However, we have lifted off the task to the *Purchases* microservices and now our server is asynchronously sending requests to the queue.

We expected that a single server would suffice to meet our client's need, as there is no significant overhead imposed on the server anymore. The result supported the hypothesis, there's merely any change in throughputs between two cases; in fact, the load balanced servers even resulted in slightly less throughputs. The percent difference is only -2.5% and -3.4% for

256 threads and 512 threads respectively, but we can conclude that a single server not only suffices, but the load balanced servers becomes inefficient in this case with the extra network hop for routing.

Table 2. Single Server vs Load Balanced Servers on a Non-persistent queue

		Single Server		Load Balanced	
	Threads	wall time (s)	throughput	wall time (s)	throughput
Non-persistent	256	161	4293	165	4189
	512	308	4488	319	4333

Figure 4. shows a snapshot of the queue status during a single server run. The *dataProcessor* queue is the queue that passes to *Store* microservice, and the *dbWriter* queue is the queue that passes to *Purchases* microservice. Both are subscribed to the same fan-out exchange. We can observe that the message rate is very high at 4,637/s at the captured moment, and there are a very few messages left in the queue waiting to be delivered.

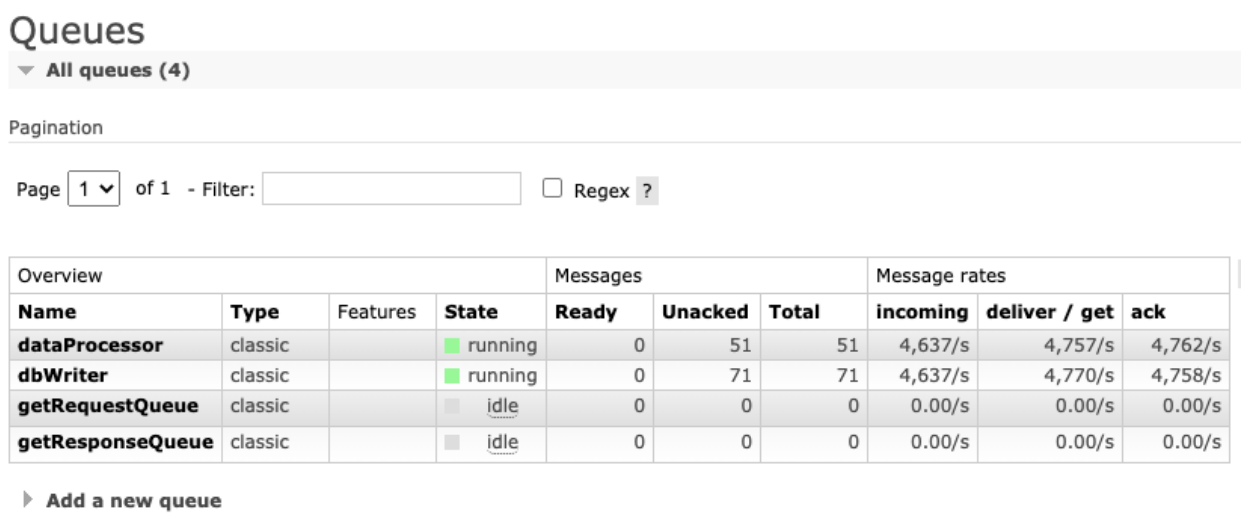

Figure 4. Queue Snapshot on a Single Server

From what we have discussed so far, it is seen that a non-persistent single server results in the best performance. A full comparison table with all cases is provided in *Table 3.*

Table 3. Full Comparison of Different Queue and Server Configuration

		Single Server		Load Balanced	
	Threads	wall time (s)	throughput	wall time (s)	throughput
Non-persistent	256	161	4293	165	4189
	512	308	4488	319	4333
Persistent	256	247	2798	251	2753
	512	493	2804	497	2781

Comparison: A2 vs A3 implementation

It will also be meaningful if we compare the server performance from the previous milestone and see how much we have improved. We had lift off a heavy concurrent write load from the server to a microservice, so the server is now asynchronous.

In A2, implementing a load balanced server has greatly improved the throughput because the server itself was under heavy load. Distributing the workload helped remove the bottleneck. However we have already discussed such is not the case in A3, and load balancing even introduces unnecessary overhead.

Hence it is shown in *Table 4.* that the throughput through a single server has increased by 45%. Note that the improvement by making a synchronous server asynchronous is not as drastic as making a synchronous server load balanced with multiple instances. This further suggests that RabbitMQ can be a new bottleneck. Upgrading the RabbitMQ server instance can be one solution.

Also note that the performance has decreased by 12% from A2 with load balanced servers.

Table 4. Synchronous vs. Asynchronous Server Performance

		wall time (s)		throughput (request/s)		
Threads		A2	A3	A2	A3	% difference
256	Single	234	161	2953	4293	45.38
	Load Balanced	145	165	4766	4189	-12.11

Result of the GET queries - Business Insights

We had set up a Store microservice to respond to GET queries. It uses a remote method invocation(RMI) pattern to return the result to the client, as the request to the exposed server gets redirected to the microservice residing in another server. These GET queries give meaningful business insights. The two queries were:

1. What are the top 10 most purchased items at Store N
2. Which are the top 5 stores for sales for item N

Below is the result tested with POSTMAN.

URL: http://34.199.151.181/superMarketServer/items/store/55

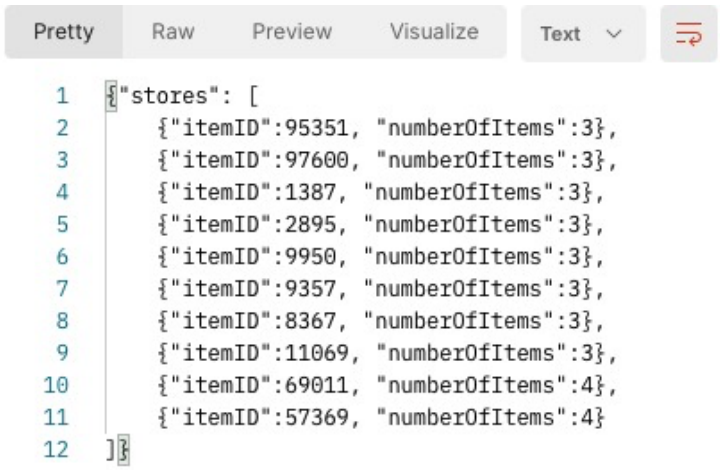

Figure 5. A GET Request for Top 10 Items

This GET request asks for the top 10 most purchase items at store 55. Here we see a very evenly distributed number of items for the itemID and it is very likely that the items that are not included in this list also have three “numberOfItems”. This is because our mock client generates a pseudo-random number for storeID and itemID. True randomness gives equal probability to each pair of storeID and itemID, so that is what we observe.

However, in a realistic scenario where this randomness is not present, the *Store* microservice will be able to return the top 10 items for a particular store that can further be analyzed for current business statistics and a future business plan.

This GET request asks for the top 5 stores for sales for item 158. It is seen that storeID 1 has the most sales of the item, with quantity of 28. The store that follow are rather evenly distributed, which is also derived from the randomness of the client.

URL: http://34.199.151.181/superMarketServer/items/top5/158

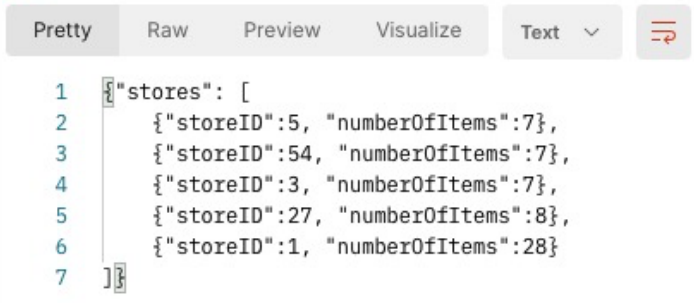

Figure 6. A GET Request for Top 5 Stores

Appendix

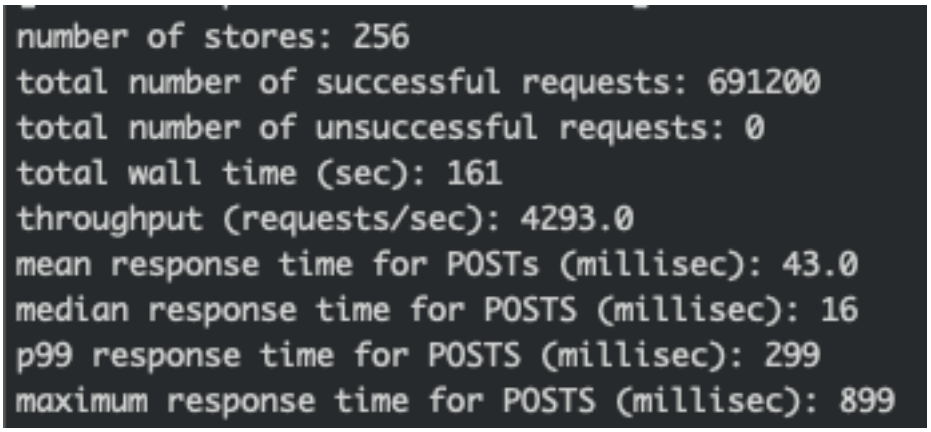

Figure 7. Non-persistent Single Server 256 Threads Run

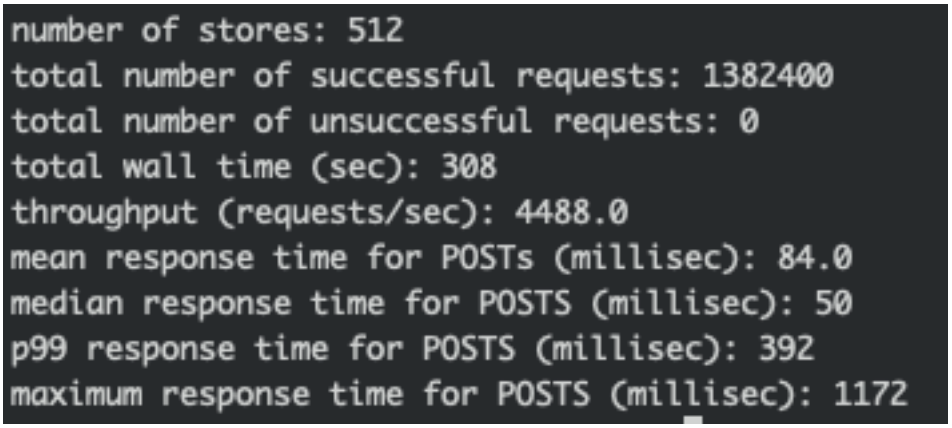

Figure 8. Non-persistent Single Server 512 Threads Run

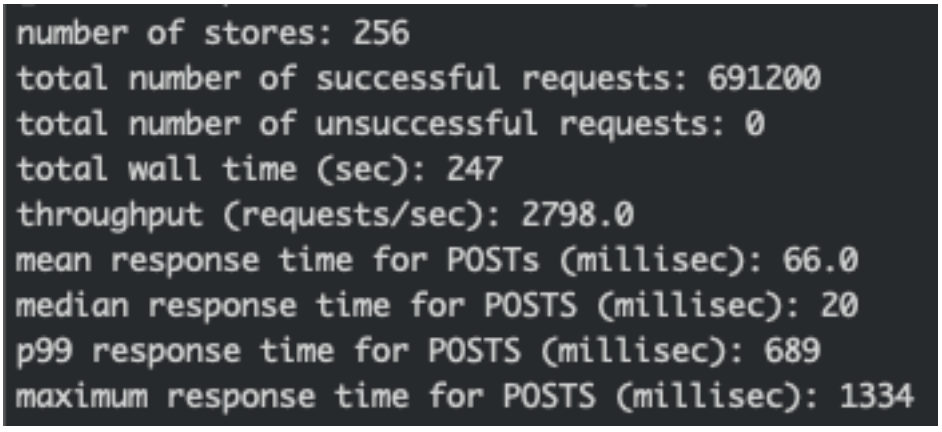

Figure 9. Persistent Single Server 256 Threads Run

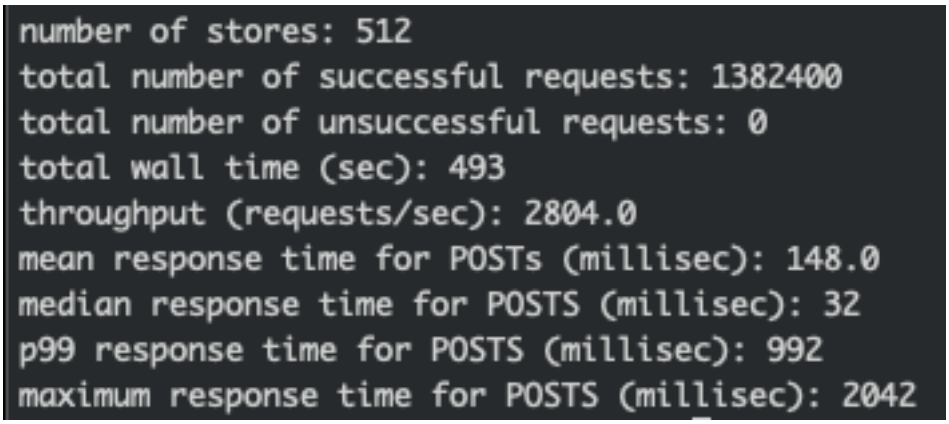

Figure 10. Persistent Single Server 512 Threads Run

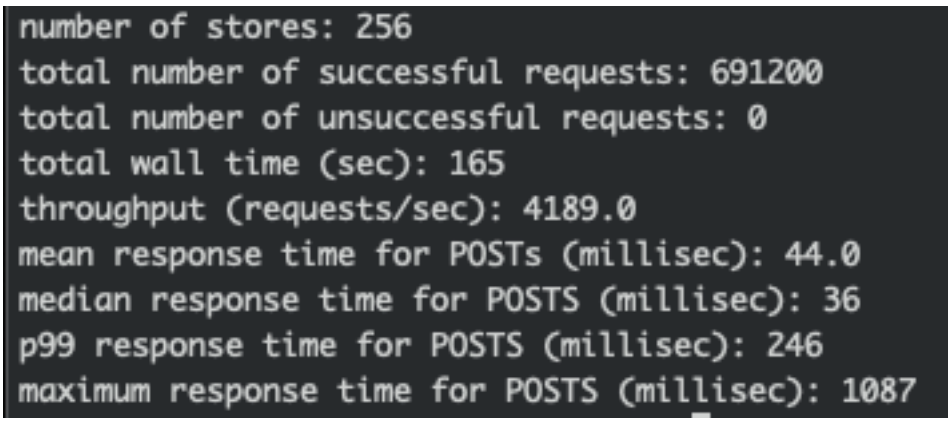

Figure 11. Non-persistent Load Balanced Server 256 Threads Run

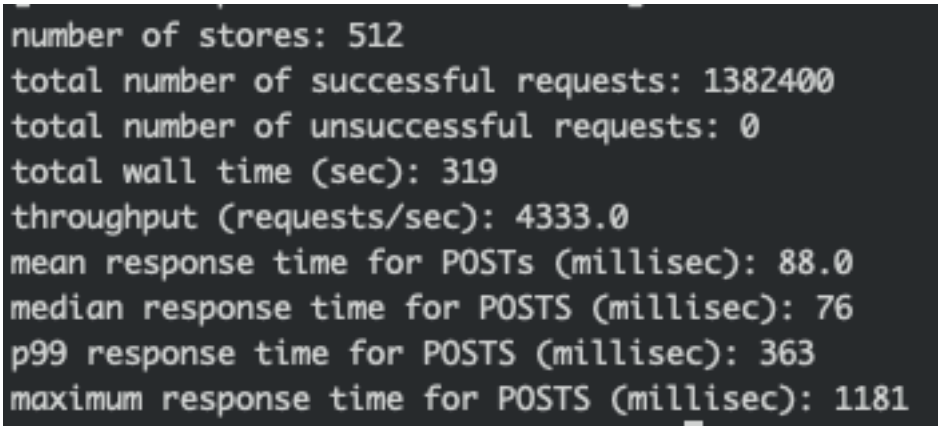

Figure 12. Non-persistent Load Balanced Server 512 Threads Run

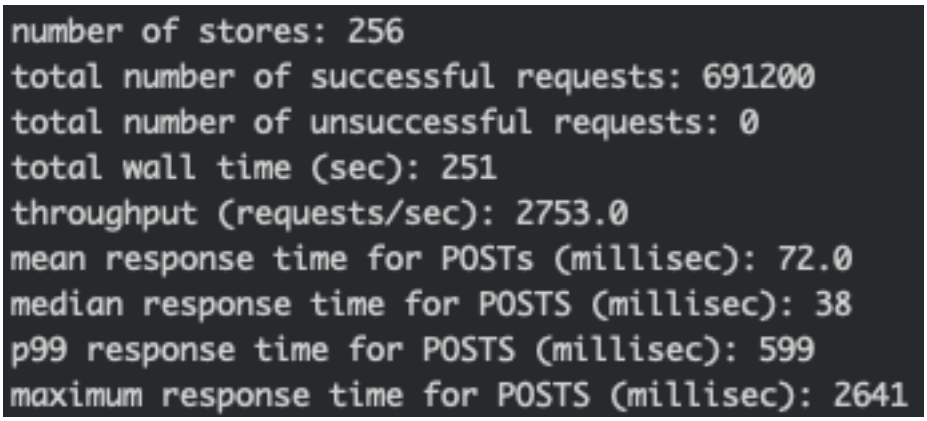

Figure 13. Persistent Load Balanced Server 256 Threads Run

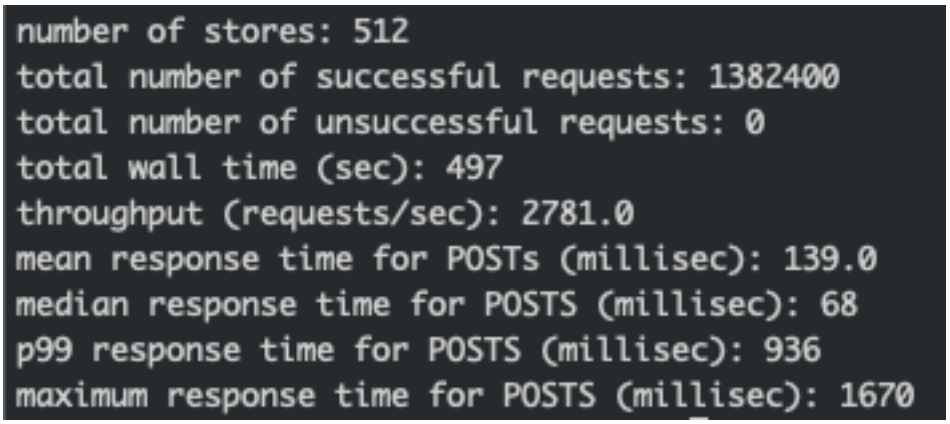

Figure 14. Non-persistent Load Balanced Server 512 Threads Run