Table of Contents

Introduction

An old decrepit supermarket chain, Tigle, is embarking on a rapid acquisition strategy to buy up new stores nationwide. 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 last part of the contract, we explored different design options of our completed system built upon the previous milestones. We have finalized the prototype of the system by placing an asynchronous messaging broker within and introducing new microservices to delegate tasks.

Refer to the Appendix for our base design description. Now, we consider different implementation to mitigate the bottlenecks we had found in previous design, and further optimize the performance. We have considered making our messaging broker a distributed service, replacing a relational database with a NoSQL database, and lastly, the exploring the effect of vertical scaling – for fun.

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

Git repo: https://github.com/hjyou07/Distributed_Systems_Project/tree/A4

Design Overview

Experiment 1 – Distributed Messaging Broker

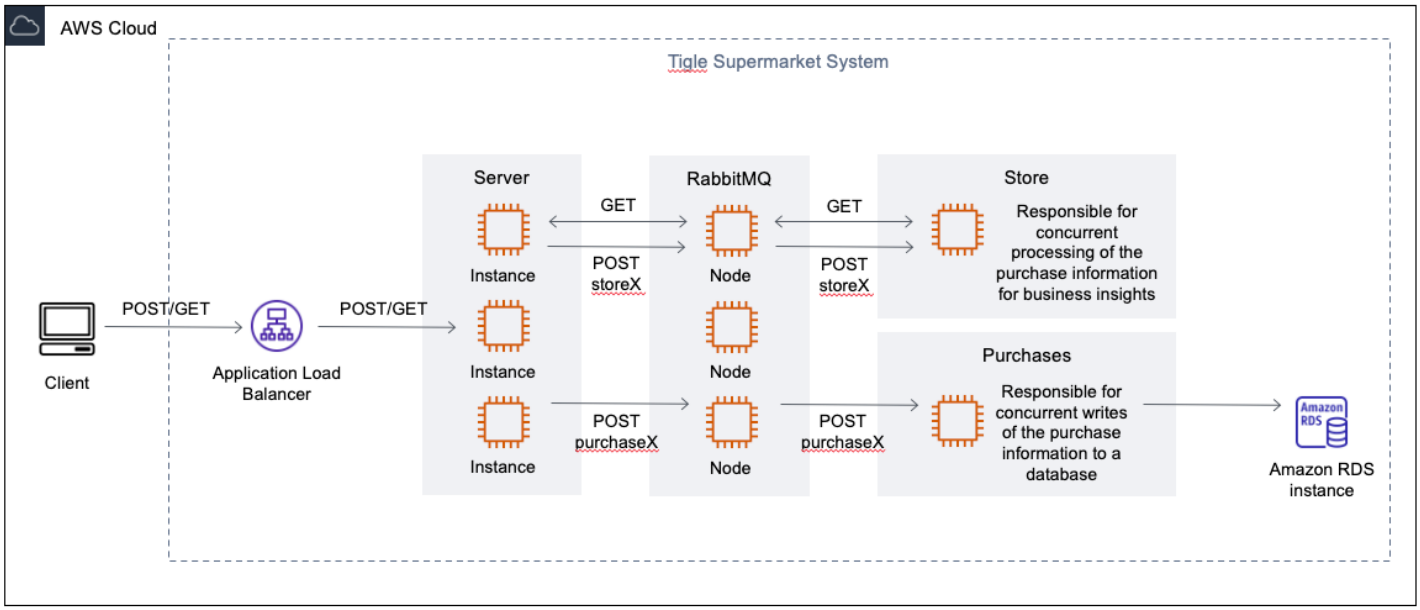

Figure 1. Tigle Architecture with Distributed Messaging Broker

In the previous milestone, we had thought a single node of messaging queue was the main source of bottleneck. The performance with load balanced servers was worse than that of a single server, which indicated that the queue didn't have the capacity for the increased number of incoming requests. Hence, the first experiment was to make our broker part of the distributed system. An option was to implement the broker using Apache Kafka which is innately distributed, but we decided to implement our own version of distributed broker by sharding queues in RabbitMQ node cluster instead of using a framework.

RabbitMQ has a plugin that supports a distributed messaging system, and a *cluster* of nodes is required. Three Ubuntu instances hosting RabbitMQ was deployed and registered as a cluster, and the queues on each node were sharded to process incoming POST requests from the servers. The messages are posted to the *exchange,* which allows us to view this as logically one queue. The cluster will appear as an atomic unit from the client's view. There are two exchanges that are responsible for delivering messages to each microservices – *Purchases* and *Store.* The underlying structure of a sharded queue is shown in *Figure 2.*

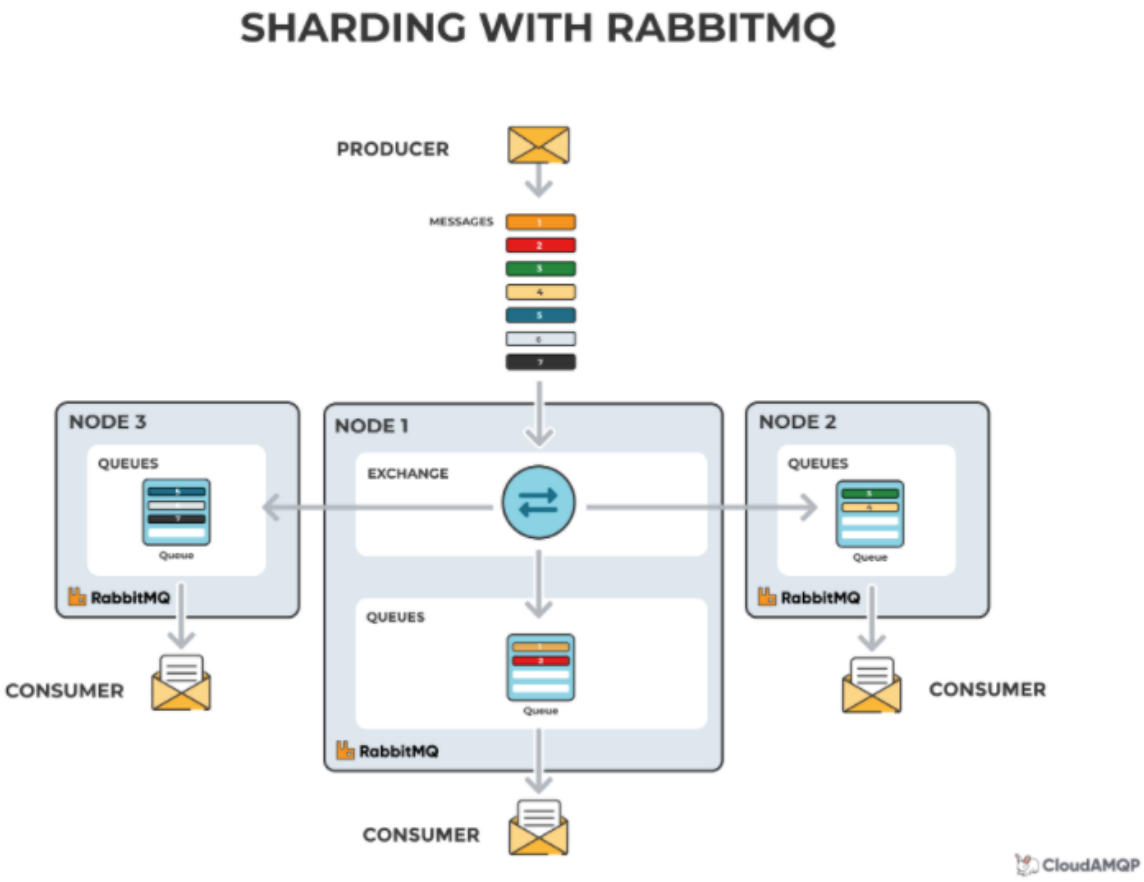

Figure 2. Underlying Structure of Sharded RabbitMQ.

Source: https://www.cloudamqp.com/blog/part1-rabbitmq-best-practice.html

This structure assumes that there are consumers for each node, which indeed is practical. In our system, the producer is the load balanced server exerting POST requests and the consumer is each microservices. It would have been ideal if we could launch corresponding number of microservices to the number of RabbitMQ nodes, but we kept the microservices on one instance each as there were limited resources available to us.

Figure 3. shows a screenshot of RabbitMQ admin GUI during one of the test runs. All incoming messages are directed to each exchange, and the messages get equally distributed among the sharded queues that are created in each node (not shown here). As will be discussed later, we can see that the throughput captured here is at 6k messages/s, which is significantly higher than what we had observed before with a single node of queue.

Exchanges

All exchanges (9)

Pagination

Page 1 of 1 - Filter: Regex ?

Name	Type	Features	Message rate in	Message rate out	+/-
(AMQP default)	direct	D			
amq.direct	direct	D			
amq.fanout	fanout	D			
amq.headers	headers	D			
amq.match	headers	D			
amq.rabbitmq.trace	topic	D I			
amq.topic	topic	D			
purchaseX	x-modulus-hash	**purchaseX**	6,401/s	6,395/s	
storeX	x-modulus-hash	**storeX**	6,401/s	6,396/s	

Figure 3. Incoming Message rate with Distributed RabbitMQ

Experiment 2 – NoSQL Database

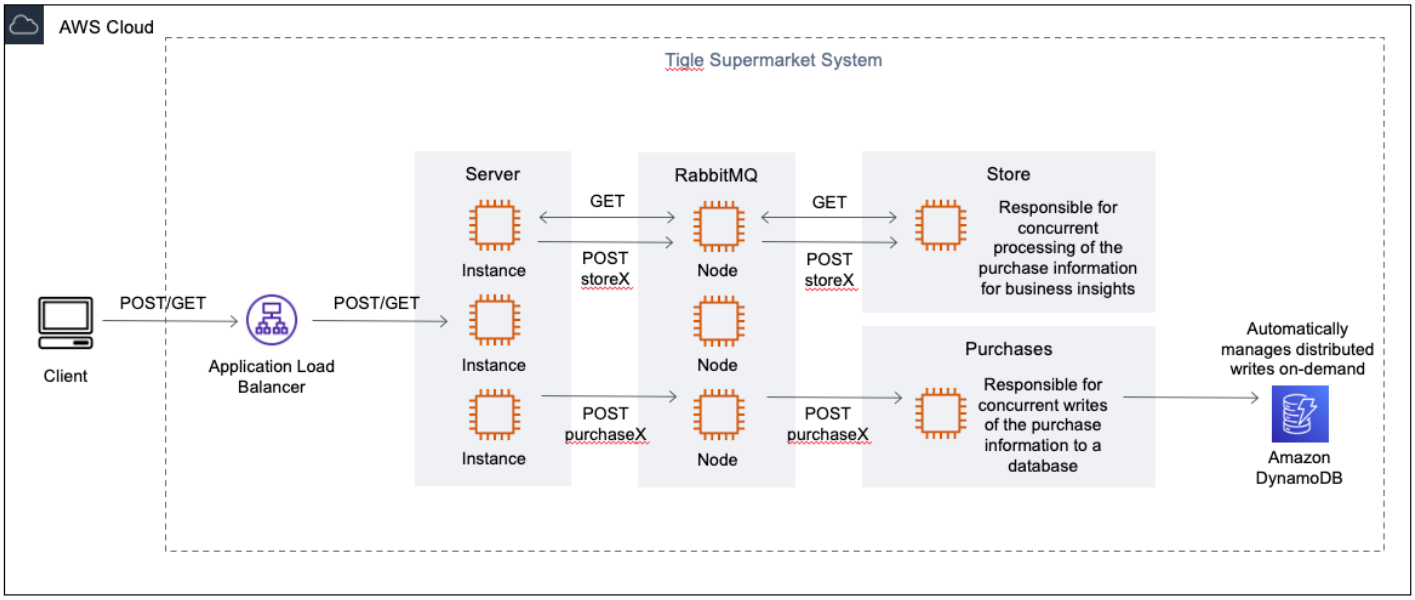

Figure 4. Tigle Architecture with DynamoDB

The next experiment involved replacing the RDS instance that ran on MySQL engine with Amazon DynamoDB. Unlike the *Store* microservice where all information is processed within,

the *Purchases* microservice writes to the database and such I/O operation was seen as the second bottleneck. An option was to implement a distributed database ourselves. However, not only it was unpractical to make use of consistent hashing in sharding the database, but also the data being stored wasn't ideal for a relational database. We also wanted to explore this relatively new field of NoSQL databases – DynamoDB was a great candidate as it is innately distributed and can be configured to scale on-demand.

Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. DynamoDB lets you offload the administrative burdens of operating and scaling a distributed database.

Source:
https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.CoreComponents.html

For these reasons, building off from experiment 1, the second experiment was with DynamoDB. MySQL has a well-established compatibility with Java via JDBC (Java Database Connectivity) API, and such needed to be swapped out to accommodate the new connection to DynamoDB. The available AWS SDK for Java provided thread-safe clients for working with DynamoDB and allowed for an easy transition.

Experiment 3 – Vertical Scaling

Vertical scaling was explored at the end of the project to spend all the remaining credit we have left with AWS Educate account. As much as it was for fun, we did obtain some insights from vertical scaling. All instances were running on a t2.micro (1 vCPU, 1GiB RAM) before, and we upgraded all to t2.xlarge (4 vCPU, 16GiB RAM) instances including the client.

Statistical Analysis

Assumptions

In exploring alternative designs, there are some baseline factors that remain unchanged in all experiments which are the following:

1. Servers are load-balanced
2. Messaging brokers are configured non-persistent
3. Microservices remain undistributed

These were to keep the effects of each experiments as isolated as possible and make the results easier to compare to those of previous milestones. Single servers are not explored as we were focusing on building a scalable distributed system, and the messaging brokers are configured non-persistent for consistency among all milestones. Each microservices remain on a multithreaded single machine to isolate the effects of configuration changes and also due to the limited number of instances available for a free AWS Educate account.

Comparison: A2 vs A3 vs A4 implementation

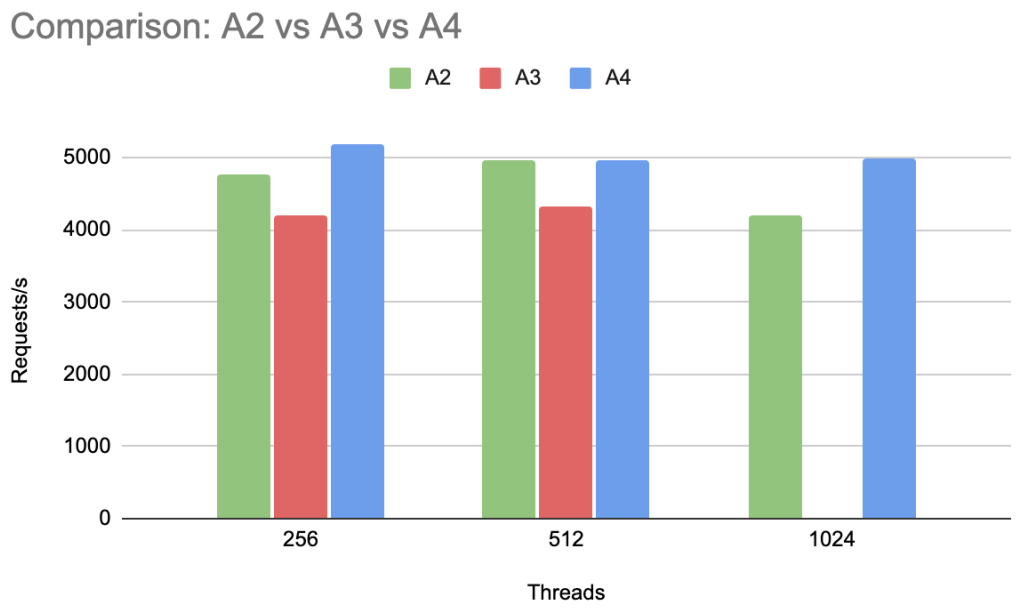

Figure 5. Throughput Comparison of Each Milestone

Figure 5. shows the general throughput comparison between each milestone. In A2, the load balanced servers directly wrote to the database. A3 had an asynchronous message broker that propagates the requests to the microservices. A4 throughput is that obtained by sharding the queues, after experiment 1 but before experiment 2.

Notice the throughput of asynchronous requests (A3) is worse than that of synchronous requests to database write (A2). As discussed in the previous report, this is why the queue was thought to be the bottleneck of the system. The throughput with a single server had increased with the asynchronous messaging system, but not with load-balanced servers. This indicated that the queue didn't have the capacity for the increased number of incoming requests and provided worse performance even though it was asynchronous.

The solution was to try sharding the queue, and we see that the performance improved significantly from a non-sharded queue (A3). It was better even compared to A2 results and gave more stable throughputs with different number of threads. This indicates a potential for even further scalability. The average throughput seem to cap around 5k due to some noise, but an instant throughput was usually higher as shown in *Figure 3. Table 1.* shows the percent difference in throughput between A3 and A4.

	Throughput (requests/s)		
Threads	**RMQ**	**Sharded RMQ**	**% difference**
256	4189	5196	24.04
512	4333	4972	14.75
1024		4999	

Table 1. Effect of Distributed RabbitMQ

Effect of Distributed Database

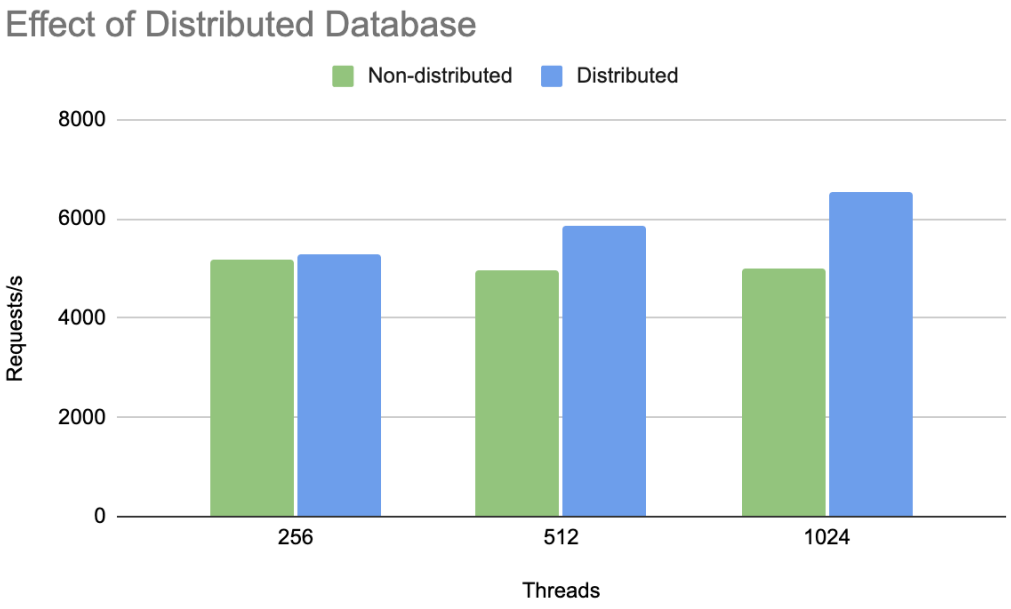

Figure 6. Throughput Comparison of MySQL vs DynamoDB

Sharding of the message broker has provided a consistent throughput around 5k requests/s for all number of stores. Will making the database distributed give even bigger difference?

Figure 6. compares the throughput of experiment 1 (distributed RabbitMQ + MySQL) and experiment 2 (distributed RabbitMQ + DynamoDB). We observe that there isn't a significant difference, if any, with 256 stores. The throughput takes off with increasing number of stores, showing that the complete distributed system with DynamoDB not only just handles the increasing load, but actually benefits from it – remember, just adding one store exerts 2700 more POST requests.

Table 2. shows the percent difference in throughput between a non-distributed MySQL database on a AWS RDS instance and a distributed Amazon DynamoDB instance.

	Throughput (requests/s)		
Threads	**RDS**	**DynamoDB**	**% difference**
256	5196	5276	1.54
512	4972	5857	17.80
1024	4999	6542	30.87

Table 2. Effect of Distributed Database

Effect of Vertical Scaling

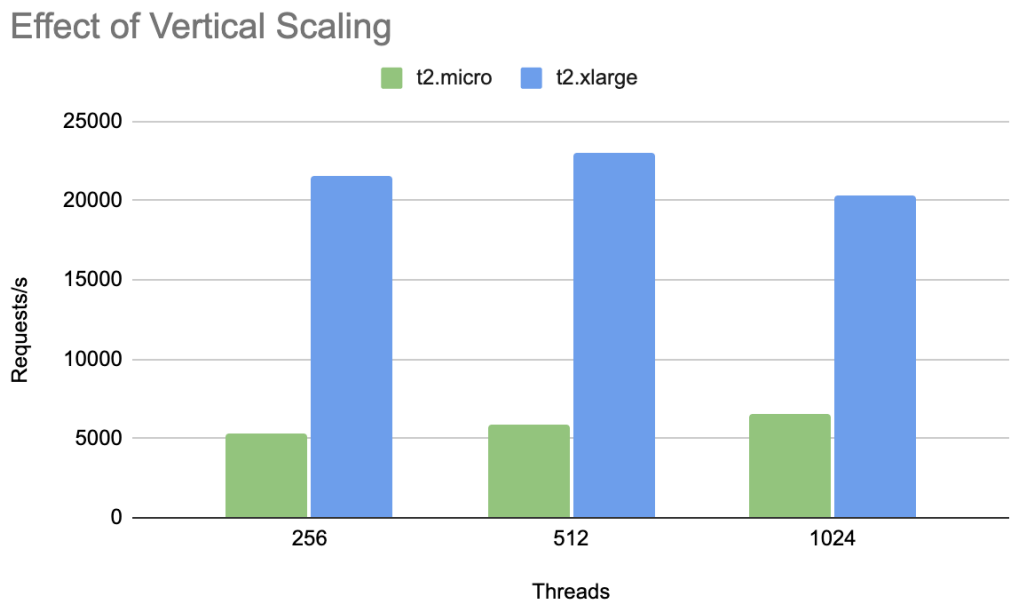

Figure 7. Throughput Comparison of t2.micro vs t2.xlarge

Although done for fun, this experiment lets us ponder a fundamental difference between multiprocessing and multithreading. Both are used to increase the computing power of a system, but in multiprocessing, CPUs are added for increased computing power whereas in multithreading, threads are created within a single process to efficiently exploit the computing power.

As mentioned before, the upgraded t2.xlarge has 4 vCPU whereas t2.micro has 1 vCPU. We actually observe > 4x difference in the throughput with 4x increase in number of vCPUs the instance runs with. It will be naive to conclude that the performance scaled linearly with the number of CPUs as there are many other factors affecting the performance of a machine. What we can conclude is the effect the vertical scaling had on our mock client.

The best instant throughput we were getting with the final version of distributed system was ~8k, average being ~5k. The increase to ~20k requests/s is fundamentally different. This indicates that no matter how “well” our distributed system was working, the client was capped at sending <10k requests per second due to its limited computing power available and vertically scaling lifted that cap off.

Conclusion

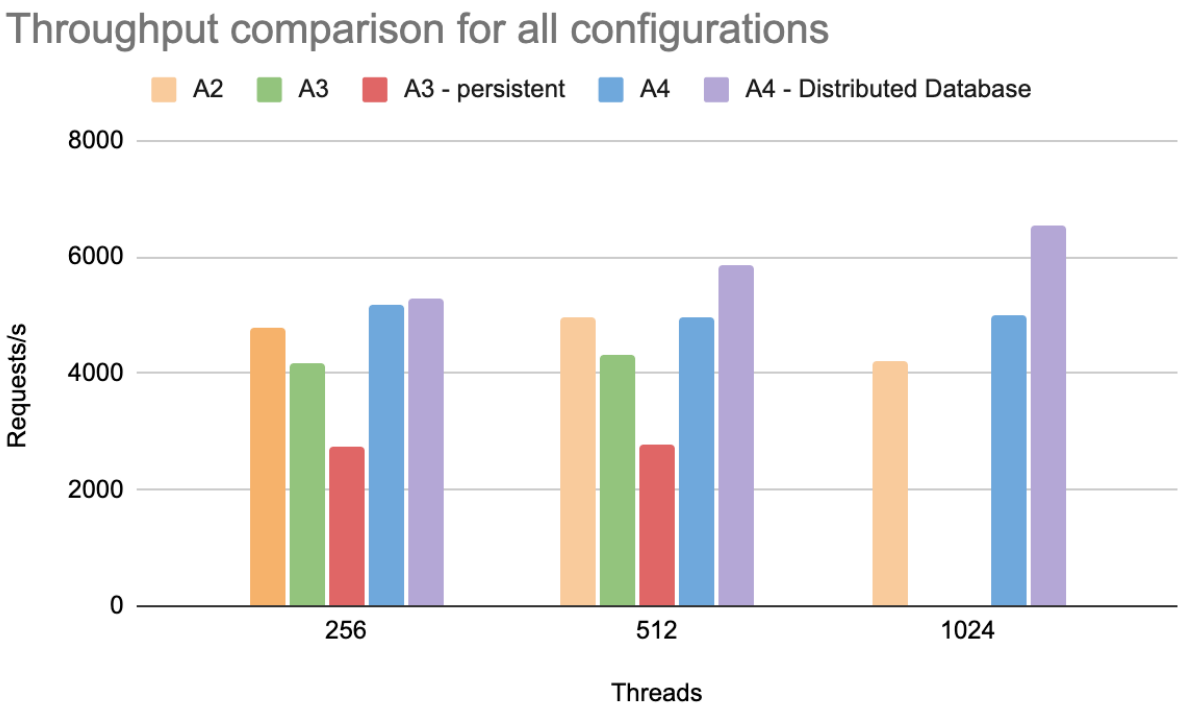

Figure 8. Throughput Comparison for All Configurations

Figure 8. includes all configurations – but the vertical scaling experiment – tested throughout the development of the system. It shows a decrease in efficiency of the system as it becomes complex, but an improvement with optimization. Distributed message broker seemed to promise further scalability, and adding a distributed database took off from that basis. A3 data is missing for 1024 stores as the microservices were configured to only handle up to 512 stores.

In retrospective, choosing to implement the message broker with Apache Kafka instead of distributed RabbitMQ might have resulted in a more reliable and available system. The sharded RabbitMQ instance were not replicated, which can compromise the availability of our system.

Anyhow, we hold the best wishes to our client, Tigle Supermarket chain.

Appendix

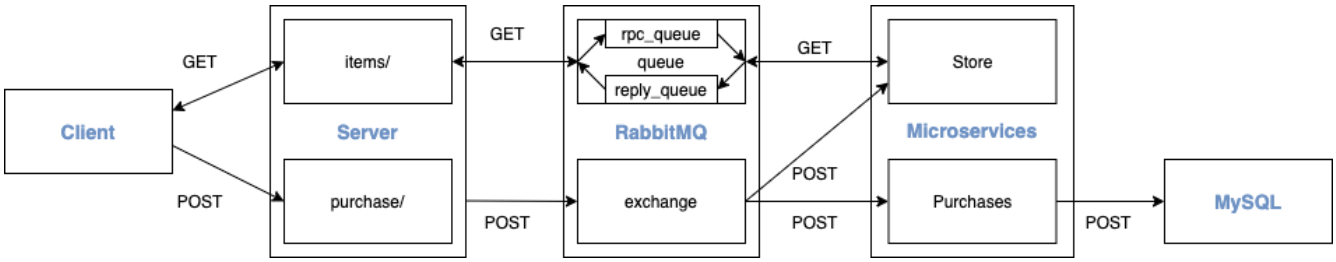

Figure 9. Base Architecture of Tigle Supermarket System

The overall structure of this Tigle Supermarket system is shown in *Figure 9.*

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.

```
number of stores: 256
total number of successful requests: 691200
total number of unsuccessful requests: 0
total wall time (sec): 135
throughput (requests/sec): 5120.0
mean response time for POSTs (millisec): 35.0
median response time for POSTS (millisec): 24
p99 response time for POSTS (millisec): 155
maximum response time for POSTS (millisec): 1044
```

```
number of stores: 512
total number of successful requests: 1382400
total number of unsuccessful requests: 0
total wall time (sec): 278
throughput (requests/sec): 4972.0
mean response time for POSTs (millisec): 78.0
median response time for POSTS (millisec): 78
p99 response time for POSTS (millisec): 159
maximum response time for POSTS (millisec): 1167
```

```
number of stores: 1024
total number of successful requests: 2764800
total number of unsuccessful requests: 0
total wall time (sec): 553
throughput (requests/sec): 4999.0
mean response time for POSTs (millisec): 157.0
median response time for POSTS (millisec): 118
p99 response time for POSTS (millisec): 550
maximum response time for POSTS (millisec): 1827
```

```
number of stores: 256
total number of successful requests: 691200
total number of unsuccessful requests: 0
total wall time (sec): 131
throughput (requests/sec): 5276.0
mean response time for POSTs (millisec): 35.0
median response time for POSTS (millisec): 27
p99 response time for POSTS (millisec): 141
maximum response time for POSTS (millisec): 995
```

```
number of stores: 512
total number of successful requests: 1382400
total number of unsuccessful requests: 0
total wall time (sec): 236
throughput (requests/sec): 5857.0
mean response time for POSTs (millisec): 63.0
median response time for POSTS (millisec): 29
p99 response time for POSTS (millisec): 333
maximum response time for POSTS (millisec): 1212
```

```
number of stores: 256
total number of successful requests: 691200
total number of unsuccessful requests: 0
total wall time (sec): 32
throughput (requests/sec): 21600.0
mean response time for POSTs (millisec): 8.0
median response time for POSTS (millisec): 6
p99 response time for POSTS (millisec): 35
maximum response time for POSTS (millisec): 593
```

```
number of stores: 512
total number of successful requests: 1382400
total number of unsuccessful requests: 0
total wall time (sec): 60
throughput (requests/sec): 23040.0
mean response time for POSTs (millisec): 16.0
median response time for POSTS (millisec): 10
p99 response time for POSTS (millisec): 130
maximum response time for POSTS (millisec): 1061
```

```
number of stores: 1024
total number of successful requests: 2764800
total number of unsuccessful requests: 0
total wall time (sec): 136
throughput (requests/sec): 20329.0
mean response time for POSTs (millisec): 36.0
median response time for POSTS (millisec): 24
p99 response time for POSTS (millisec): 304
maximum response time for POSTS (millisec): 1943
```