Table of Contents

Introduction

A supermarket chain is embarking on a rapid and massive expansion in their business. 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 first part of the contract, we built a client that generates and sends synthetic item purchases to a server in the cloud.

We set up a simple server API that simply accepts and validate requests, then sends a HTTP 200/201 response. Then we designed a multithreaded Java client that we can configure to upload a day of item purchases from multiple stores and exert various loads on the server.

Stores operate for 9 hours per day and are in 3 different time zones; meaning that the opening of stores is staggered.

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

Git repo: https://github.com/hjyou07/Distributed_Systems_Project/tree/A1

Client Design

When running the client some parameters need to be inputted in the command line. The user can either provide only the required parameters which includes the number of stores they want to simulate, the date, and the server address, in that order. The client will set other parameters to default values. If the user wishes to fine tune the parameters, they must supply all of the parameters:

1. number of stores to simulate *(maxStores)*
2. number of customers per store *(numCust)*
3. maximum itemID, which allows numeric values *(maxItemID)*
4. number of purchases per hour *(numPurchases)*
5. number of items for each purchase *(numPurchasItems)*
6. date in YYYYMMDD *(date)*
7. IP/port address of the server *(serverAddress)*

Check Appendix for a complete UML diagram of the classes defined below.

Default package

Client

Client class, as the main thread, takes in the parameters and create the Store objects with the given specifications. The client also creates two BlockingQueue buffers and three CountDownLatch instances to pass into the Store objects. Then store threads are instantiated and saved into an array, ready to run. Before the stores actually launch, two consumer threads, which will take from each buffer that were created before, are instantiated and started. Each consumer thread will be responsible for writing the HTTP response to a csv file and preprocessing the responses for later analysis.

Time stamp is taken before the client starts running the first pool of stores, in East phase. Each East phase stores open and operate for a while and counts down the first latch. It takes only one of the East phase stores to signal the launch of Central phase stores. Then the Central phase stores are started, later signaling the second latch, which prompts the West phase stores to start. As all store threads are up and running, the final latch waits for all of them to finish. Another time stamp is taken when the final latch is counted down.

The client terminates after waiting for the consumer threads to finish, creating a new DataProcessor instance for statistical analysis, and printing out the results to the terminal.

DataProcessor

DataProcessor gets called in the Client class to generate a report. It takes in a custom data storage object, LatencyBucket, generated by one of the consumer threads. It has methods to calculate the mean, medina, and 99th percentile response time, maximum response time, total wall time and the throughput. It also returns the number of successful and unsuccessful requests.

Producer package

Store

Store class is the heart of this client system, and a producer. Store implements Runnable and each store will represent a thread. It takes in two buffers and three latches from the Client class and puts the Response object to the buffers and counts down the latches with its internal logic. It simulates the operating hours by placing *numPurchases* purchases for 9 hours. A POST request is sent to the server for every purchase, and each response that comes back gets put in the two buffers for consumer threads to pull from. When the store has been opened for 3 hours, it counts down the first latch that was passed in. When the store has been opened for 5 hours, it counts down the second latch, and after 9 hours, it counts down the third latch for close signal.

Consumer package

Preprocessor

Preprocessor is one of the consumers in this system. It takes the Response object from the preprocessBuffer, which is concurrently getting put by Store threads. All this class does is taking the Response object until it runs into some termination flag and putting the response into the LatencyBucket instance that it internally creates and own. Termination flag is put by the Client class after all the Store threads are finished.

RecordWriter

RecordWriter is the other consumer in this system. It also takes the Response object from the buffer, but this time, csvBuffer and writes out into a csv file. The Response object is formatted into a String, and written into a BufferedWriter. When the termination flag is received, The BufferedWriter closes and flushes its content out to a csv file.

Model package

Response

Response is an arbitrary POJO that acts as a data container for the HTTP Response received in Store object. It keeps a record of a start and the end timestamp that was taken before and after the API call, the type of the request sent, and the received response code. It simply calculates the latency by capturing the difference between the timestamps. Response object is passed between the producer and consumers through the buffers, and is unfolded for further consumer processing.

LatencyBucket

LatencyBucket is also an important component in this system. This data structure gets instantiated in each Preprocessor consumer thread. It keeps an integer array of size 10,000, a successCount, and a failureCount. The integer array keeps track of the number of latency values that fall within each time increment, hence a bucket. The index of the array represents the latency value from 0 to 10,000 milliseconds. The maximum latency is at 10,000 as it is the

default time-out value for a http-request. The value at each index represents the number of responses which their latency equals the index.

It takes in a Response object and check its latency, put it in the corresponding “bucket”, and counts the success or failure of the response.

Statistics Analysis

Part 1

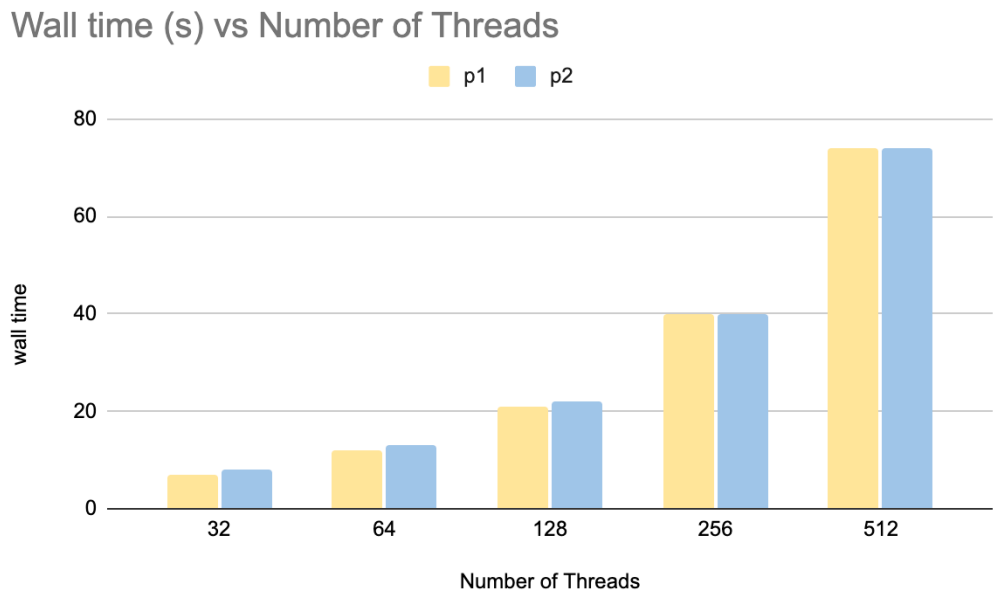

Figure 1. Wall Time vs. Number of Threads

Part 1 calculated for the wall time and throughput only without any instrumentation. Figure 1. shows the wall time(s) with different number of threads. It is seen that the wall time linearly increases with the increasing number of threads. Note the figure may seem exponential at first, but the x-axis increases by two folds each increment. Not shown here but if taken the logarithmic values on both x-axis and y-axis, the linear trend is clearly shown.

For the output of each trial, refer to Appendix.

Part 2

Part 2 involved instrumenting the client for deeper insights into the performance of the system. Ideally, this overhead should be minimal as we're concurrently running the consumer threads and the difference in wall time should be less than 5%.

Table 1. p1 vs p2 Wall Time/Throughput Comparison

	wall time (s)			throughput (request/s)		
	p1	p2	% difference	p1	p2	% difference
32	7	8	14.29%	2468	2160	12.48%
64	12	13	8.33%	2880	2658	7.71%
128	21	22	4.76%	3291	3141	4.56%
256	40	40	0.00%	3456	3456	0.00%
512	74	74	0.00%	3736	3736	0.00%

Table 1. compares the wall time and the throughput of the client run for part 1 and part 2. Notice the wall time difference is relatively big for smaller number of threads, but as the number of threads increase, the difference diminishes. There are two factors to this occurrence. First, the number of threads and the corresponding wall times are too small in the first trial, so the percent difference is amplified. The absolute values are 7 seconds and 8 seconds for p1 and p2 client. I would still consider this a minimal difference. Second, there's a general trend of the response time for the threads that contributes to this. The first ten threads or so have extremely high response time until most of the response time resorts close to the median value. If there's lower number of threads, this initial behavior would greatly affect the wall time.

Table 2. p2 Statics Summary

	success	failure	wall time(s)	throughput (request/s)	mean (ms)	median (ms)	p99 (ms)	max (ms)
32	17280	0	8	2160	10	9	37	564
64	34560	0	13	2658	16	10	91	551
128	69120	0	22	3141	28	24	99	623
256	138240	0	40	3456	50	28	322	739
512	276480	0	74	3736	99	29	879	2878
1024	552960	0	187	2957	249	157	1203	2512
2048	1105920	0	437	2530	583	422	2449	5370
4096	Time out							

Table 2. shows the statistical result taken from instrumentation of the client performance. It was only required to run up to 256 threads, but the summary includes the results from higher loads as well for stress testing. The client timed out and failed at 4096 store threads.

Notice the distribution is extremely right skewed, the median is much smaller than the mean and there is a long tail in the positive direction observed by the p99 and max response time value. This means that most of the response are "fast". What's not shown here is the mode response time, but it can easily be measured with the LatencyBucket.

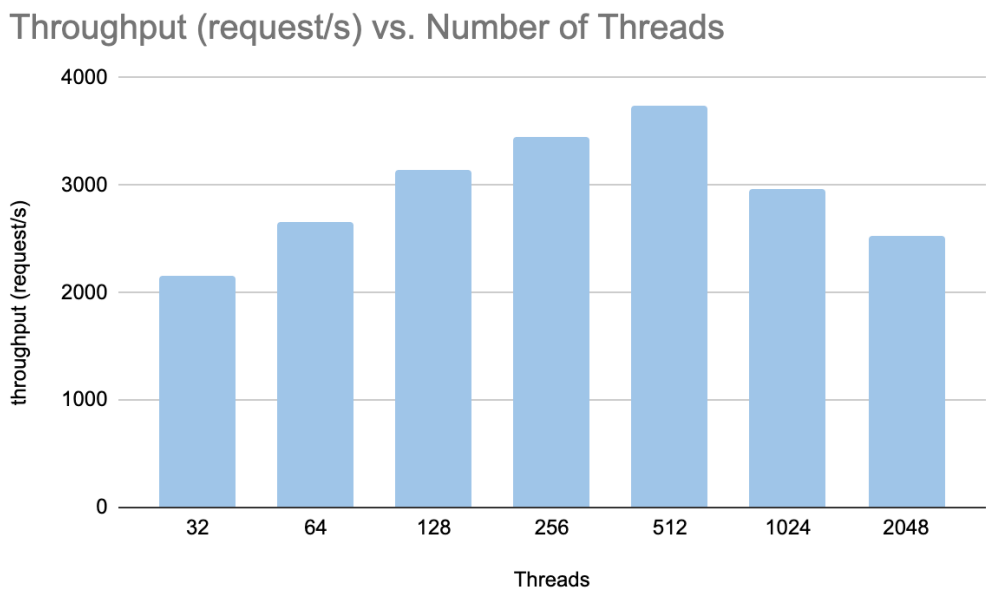

Figure 2. Throughput (requests/s) vs. Number of Threads

Figure 3. plots the throughput against the different number of threads. It is seen that the efficiency of the threads increase to a certain point, then starts to decrease as the stress is overloaded.

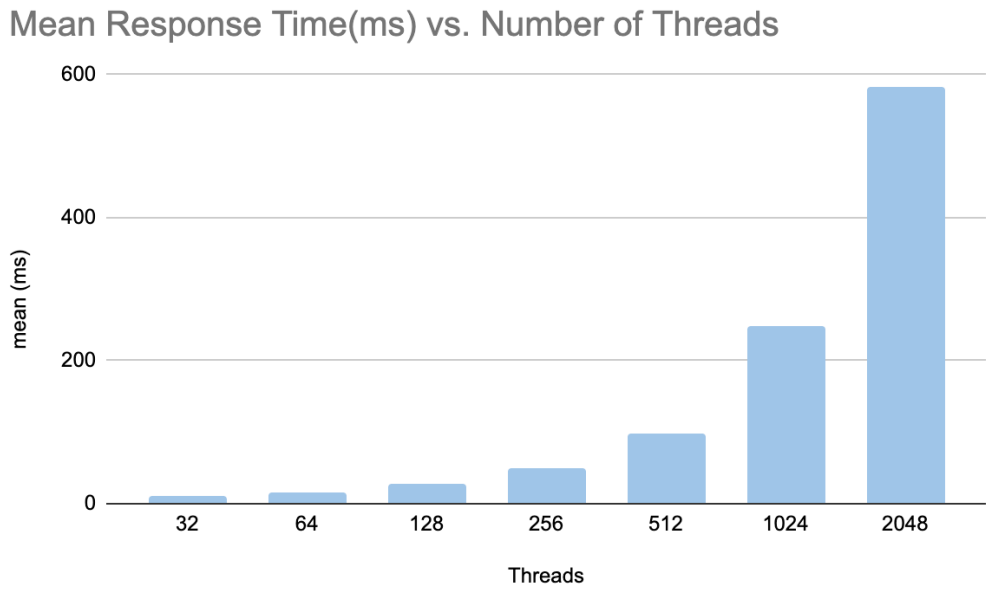

Figure 3. Mean Response Time(ms) vs. Number of Threads

Figure 4. plots the mean response time against the number of threads. Also can be easily mistaken as an exponential increase, this mean response time is linear to the increasing number of threads.

Appendix

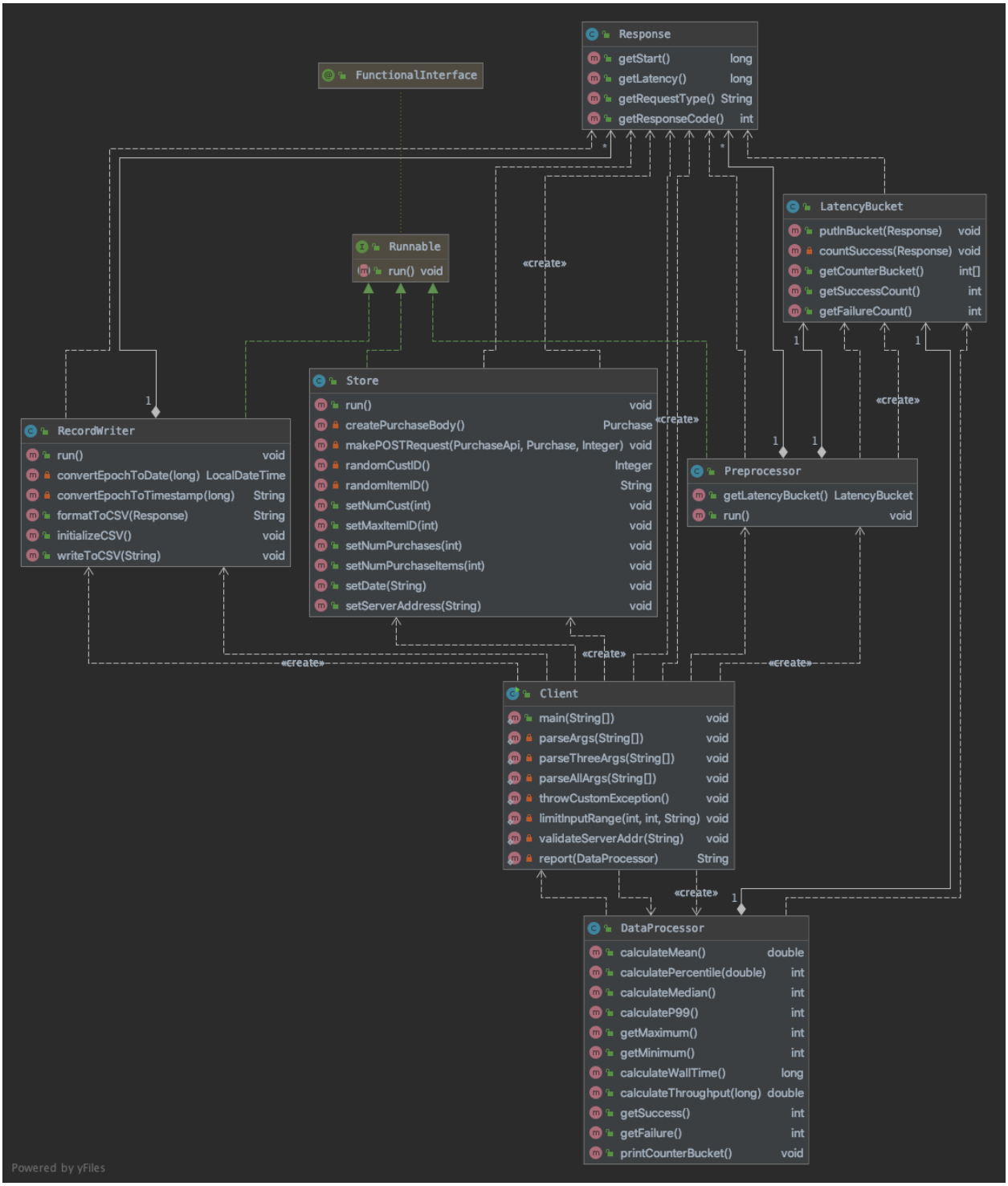

Figure 4. Client UML Diagram

```
number of stores: 32
total number of successful requests: 17280
total number of unsuccessful requests: 0
total wall time (sec): 7
throughput (requests/sec): 2468.0
```

Figure 5. Client1 - 32 threads

```
number of stores: 64
total number of successful requests: 34560
total number of unsuccessful requests: 0
total wall time (sec): 12
throughput (requests/sec): 2880.0
```

Figure 6. Client1 - 64 threads

```
number of stores: 128
total number of successful requests: 69120
total number of unsuccessful requests: 0
total wall time (sec): 21
throughput (requests/sec): 3291.0
```

Figure 7. Client1 - 128 threads

```
number of stores: 256
total number of successful requests: 138240
total number of unsuccessful requests: 0
total wall time (sec): 40
throughput (requests/sec): 3456.0
```

Figure 8. Client1 - 256 threads

```
number of stores: 512
total number of successful requests: 276480
total number of unsuccessful requests: 0
total wall time (sec): 74
throughput (requests/sec): 3736.0
```

Figure 9. Client1 - 512 threads

```
number of stores: 32
total number of successful requests: 17280
total number of unsuccessful requests: 0
total wall time (sec): 8
throughput (requests/sec): 2160.0
mean response time for POSTs (millisec): 10.0
median response time for POSTS (millisec): 9
p99 response time for POSTS (millisec): 37
maximum response time for POSTS (millisec): 564
```

Figure 10. Client2 - 32 threads

```
number of stores: 64
total number of successful requests: 34560
total number of unsuccessful requests: 0
total wall time (sec): 13
throughput (requests/sec): 2658.0
mean response time for POSTs (millisec): 16.0
median response time for POSTS (millisec): 10
p99 response time for POSTS (millisec): 91
maximum response time for POSTS (millisec): 551
```

Figure 11. Client2 - 64 threads

```
number of stores: 128
total number of successful requests: 69120
total number of unsuccessful requests: 0
total wall time (sec): 22
throughput (requests/sec): 3141.0
mean response time for POSTs (millisec): 28.0
median response time for POSTS (millisec): 24
p99 response time for POSTS (millisec): 99
maximum response time for POSTS (millisec): 623
```

Figure 12. Client2 - 128 threads

```
number of stores: 256
total number of successful requests: 138240
total number of unsuccessful requests: 0
total wall time (sec): 40
throughput (requests/sec): 3456.0
mean response time for POSTs (millisec): 52.0
median response time for POSTS (millisec): 26
p99 response time for POSTS (millisec): 318
maximum response time for POSTS (millisec): 842
```

Figure 13. Client2 - 256 threads

```
number of stores: 512
total number of successful requests: 276480
total number of unsuccessful requests: 0
total wall time (sec): 74
throughput (requests/sec): 3736.0
mean response time for POSTs (millisec): 99.0
median response time for POSTS (millisec): 29
p99 response time for POSTS (millisec): 879
maximum response time for POSTS (millisec): 2878
```

Figure 14. Client2 - 512 threads

```
number of stores: 1024
total number of successful requests: 552960
total number of unsuccessful requests: 0
total wall time (sec): 187
throughput (requests/sec): 2957.0
mean response time for POSTs (millisec): 249.0
median response time for POSTS (millisec): 157
p99 response time for POSTS (millisec): 1203
maximum response time for POSTS (millisec): 2512
```

Figure 15. Client2 - 1024 threads

```
number of stores: 2048
total number of successful requests: 1105920
total number of unsuccessful requests: 0
total wall time (sec): 437
throughput (requests/sec): 2530.0
mean response time for POSTs (millisec): 583.0
median response time for POSTS (millisec): 422
p99 response time for POSTS (millisec): 2449
maximum response time for POSTS (millisec): 5370
```

Figure 16. Client2 - 2048 threads