Advanced Systems Lab – First Milestone

Name: *Rabeeh Karimi Mahabadi*
Legi number: *13923941*

November 6, 2015

Contents

1 System Description

1.1 Database

The database underlying the project is *Postgresql*, and the version used in this project is *Postgresql 9.3*.

1.1.1 Schema and Indexes

I used two tables for storing the messages and queues. In which *queue_id* in message table references the *id* in queue table to prevent invalid *queue_ids* to be inserted into the message table. Additionally, it would be a good idea to consider the third table for user authentications. However, I did not consider the user table as it was not directly asked in the project specification. The Database schemas are shown below:

Field	Type	Description
id	Serial	Primary key
creation_time	Timestamp	Time of the queue creation
creator_id	Integer	Not null, id of the creator of the queue

Table 1: Queue table

Field	Type	Description
id	Serial	Primary key
sender_id	Integer	Not null, id of the sender
receiver_id	Integer	Id of the receiver
queue_id	Integer	Not null, id of the queue, references *id* in queue table
arrival_time	Timestamp	Time of arrival of the message
message	Text	Message, not null

Table 2: Message table

In order to speed up the queries, I looked at all the possible queries of the system, and used indexes for the ones which need *ORDER BY* operation. The reasoning is that, if one does not use indexes for these type of queries, for getting the only topmost message, all the messages should be read from disk and get sorted which will become very costly, specially if the query is executed several times in a short period of time. Furthermore, for this project in which the system needs to insert several messages per second to the database, the size of database will grow over time, and without indexes the queries would take much more time as it needs to sort huge amount of data. I considered the following three indexes for the project:

- *queue_id*, *arrival_time*
- *receiver_id*, *queue_id*
- *sender_id*, *arrival_time*

The first index is useful for reading the topmost message from the queue, the query is shown below:

Query topmost message in a queue for a particular receiver

```
    SELECT id, sender_id, message FROM message
    WHERE queue_id = _queue_id AND
    (receiver_id IS NULL OR receiver_id = _receiver_id)
    ORDER BY queue_id, arrival_time
    LIMIT 1;
```

The second index is very useful to improving the performance in querying the queues as shown below, this query can be done without *ORDER BY*, but by using *ORDER BY* the query planer would understand to make a use of the indexes:

Query queues

```
SELECT DISTINCT queue_id FROM message
WHERE receiver_id=_receiver_id
ORDER BY queue_id;
```

The third index is used to speed up querying the message of a particular sender:

Query message of a particular sender

```
SELECT  id, sender_id, message FROM message
WHERE  _sender_id = sender_id AND
(receiver_id IS NULL OR receiver_id = _receiver_id)
ORDER BY sender_id, arrival_time
```

1.1.2 Stored Procedures

I used stored procedure for all the queries that systems need to do, also to create tables, indexes, and delete them. The benefit of stored procedure is that they increase the performance of the database because:

- it reduces the amount of information sent to the database server. It becomes more important when the bandwidth of the network is less.
- compilation step is required once when the stored procedure is created, and it does not require recompilation before each time execution.
- it helps in reusability of the SQL code.

1.1.3 Design decisions

I translated send message to *INSERT*, and peek and query the messages to *SELECT* in SQL. For the popping of the messages, in order to avoid race condition, I used *SELECT ... FOR UPDATE* statement, to lock only one row of the table when executing the pop statement. It helps to improve the performance of the database as the number of records locked at any given time for pop operation are kept to the minimum.

I tried to tweak a little bit the parameters in the database configuration file, and I set *shared_buffer* to 3GB, *work_mem* to 4MB, also I increased the number of the *checkpoint_segments* to 32 as initially I got the error message that "checkpoints are occurring too frequently" during my experiments, so I increased the number of *checkpoint_segments* to keep the number of checkpoint operations happening during the experiments as small as possible. I also enabled asynchronous commit which is an option that allows transactions to complete more quickly. In general, I tried to increase the memory limits to allow having more throughput for the experiment. However, these values are far from the optimal values. By careful selection of the parameters, one could be able to get a better results.

1.1.4 Performance characteristics

Inorder to evaluate the performance of the database, I measured throughput and response time over different number of database connections for three different experiments:

- Only read operation
- Only write operation
- Combination of read (50 %), write (25 %), send (25 %) operations: Percantage of different workloads have been chosen in a way to keep the database size constant over time.

I ran all the experiments for 3 minutes. Throughput values are averaged for the whole duration of the experiment. Response time values are averaged for every 20 seconds and I computed the average of the all intervals. In all the experiments, I sent as many requests as possible with *think_time* of zero. Also, *15000* messages are initially distributed over *20* queues in database. I used *m3.large* instance for the database machine. The logs can be found in *logs/database*. Response times and throughputs are shown in figures 2, 1. As expected, *Read* experiment has the most throughput, and the throughput of *Write* experiment is less than that. The reasoning is that in the write operation, data is flushed to the disk which is time consuming. *Combination* experiment throughput is bigger than write operation, because it has peek operations which are not expensive Also, one should consider that during the *write*, and *delete* operations database needs to update also the indexes, which also plays a role in decreasing the throughput for *write*, and *combination* experiments.

Furthermore, I can see that I have the best performance for *16* database connections. Increading the database connections up to *16* connections, make a substantial improvement in throughput. However, after *16* connections, adding more connections does not affect the throughput that much and even in some cases decreases it. The behavior can be explained by the fact that the machine I used for database has only two cores, and by adding more database connections, processes needs to fight for avialable resources which decreases the performance. However, in *16-24* connections, because the time when processes are blocked for reading or writing from file system, can be used to answer more requests, the throughput has been increased. In addition, response time also increases substantially by adding more database connections.The reasoning, can again be attributed to the fact that processes needs to fight for the resources with having more database connections.

1.2 Middleware

1.2.1 Design overview

The system overall design is depicted in figure 3. The input data is read and buffered asynchronously by *connectionManager*. *ConnectionManager*, creates a *requestDispatcher* object for each incoming request. The job is added to the frontend pool, where a worker from the frontend pool executes the task. During which,*requestDispatcher* creates the corresponding backend task and add it to the backend pool. A worker from backend pool takes the task and executes it to the database. After that, a *responseDispatcher* is created for the received response, and the task is added to the Frontend pool. Finally, A worker from frontend pool executes the task and delivers the response to the client.

A few points should be mentioned about the design:

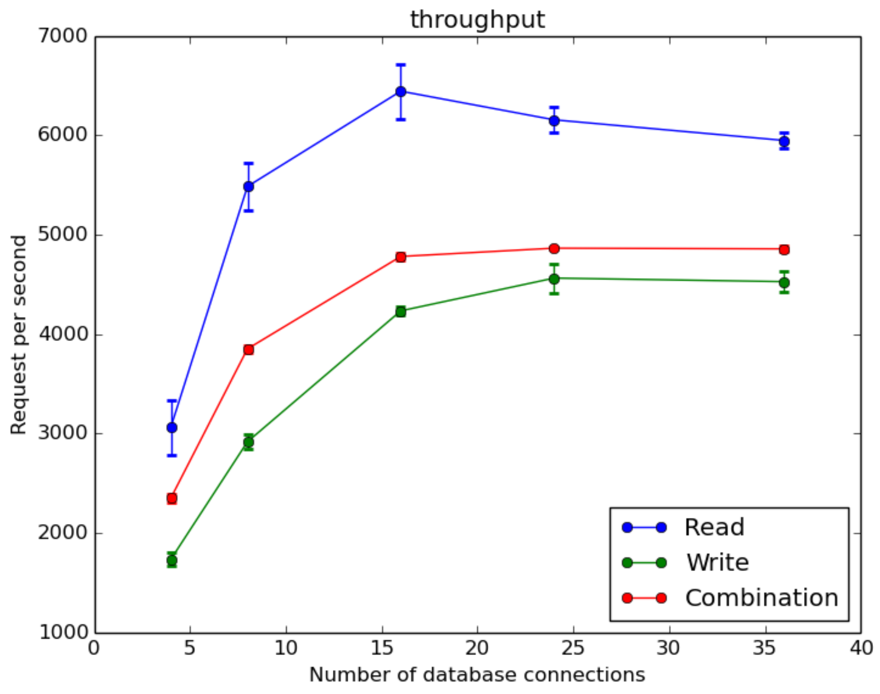

Figure 1: Database throughput over different number of database connections for three different workloads

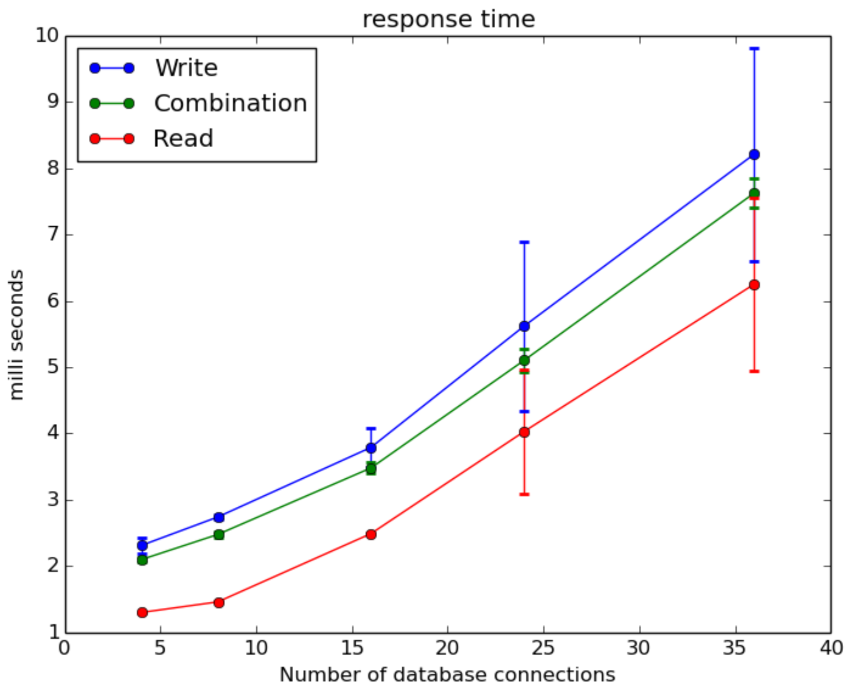

Figure 2: Database response time over different number of database connections for three different workloads

- Inorder to avoid networking overhead, each clients opens the connection once and use this connection to send all of its requests.

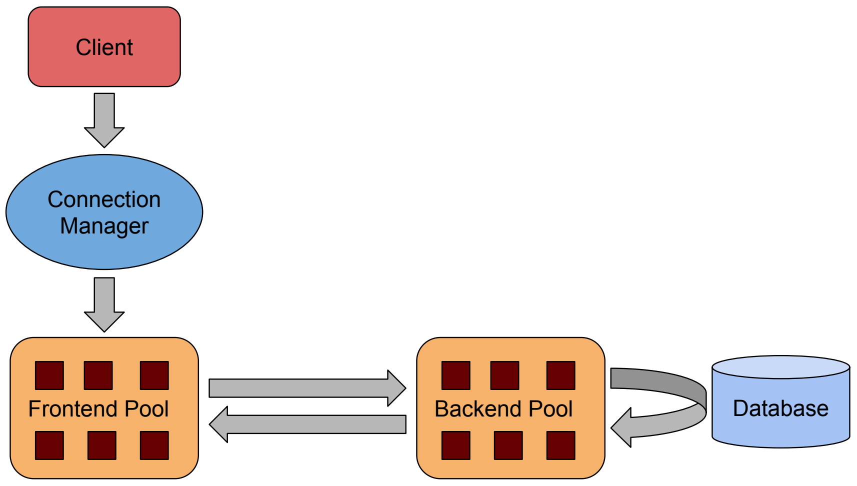

Figure 3: System overall design

- To use resources as efficiently as possible, *Java nio* package is used to handle incoming requests and build a highly scalable system. Hence, I did not used the intuitive approach of *thread-per-connection.*
- Two different backend and fronend tasks are considered in the system, and the goal was to keep the backend tasks as light as possible. Backend tasks are used to execute the query to the database, and frontend tasks are used to deliver the message to the client back, and also to parse a request from the client.

1.2.2 Interfacing with clients

Clients use blocking I/O, and the middleware uses non-blocking I/O to handle the requests. I also considered the following serialization scheme:

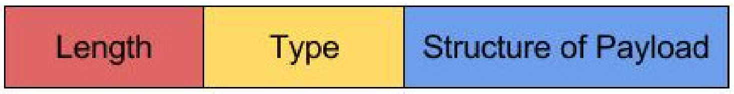

Figure 4: Serialization scheme

So each request starts with 4 bytes length of the payload, then type of the request, and then the actual payload. Different requests have different structure depending on their type. To improve the transmission speed *DataOutputStream* and *DataInputStream* are used because these methods can read and write an integer, a double and a line as a whole at a time instead of byte by byte.

1.2.3 Queuing and Connection pool to database

Before creating the backend pool, one database connection is assigned to each thread. The connection is created once, and is used to execute all queries to the database. The queuing mechanism used in the project is done by *java ThreadPoolExecutor* which construct a threadPool and uses *LinkedBlockingQueue* for queuing the tasks. Making connections once and assigning them to each thread, while avoiding creating connections each time, improves the performance substantially.

1.2.4 Performance characteristics

For testing the middleware scalability, I introduced a new request type called *Echo* in which a client sends a *string* to the middleware, and middleware sends the same message back. This way, the request is isolated from the database, and one could test the performance of the middleware separately. In this experiment, I used different number of clients, and a fixed number of *30 frontendpool workers*, and one middleware instance. Clients send as much request as possible with *think_time* of zero, and I measured throughput and responsetime for each setup the same way I did in database experiment1.1.4. I used *t2.medium* instance both for client and middleware. Throughput and responsetimes are shown in figures 6 and 5. As it is depicted in the figures, one middleware instance can support 64 clients sending *185000* requests per second. Also, the reponse time for *64* clients is around twice of the response time for *32* clients, and still within a very reasonable time. In summary, using *non-blocking I/O* clearly made a very scalable architecture for handling several requests per second. The *logs* for this experiments can be found in *logs/middleware_clients* folder.

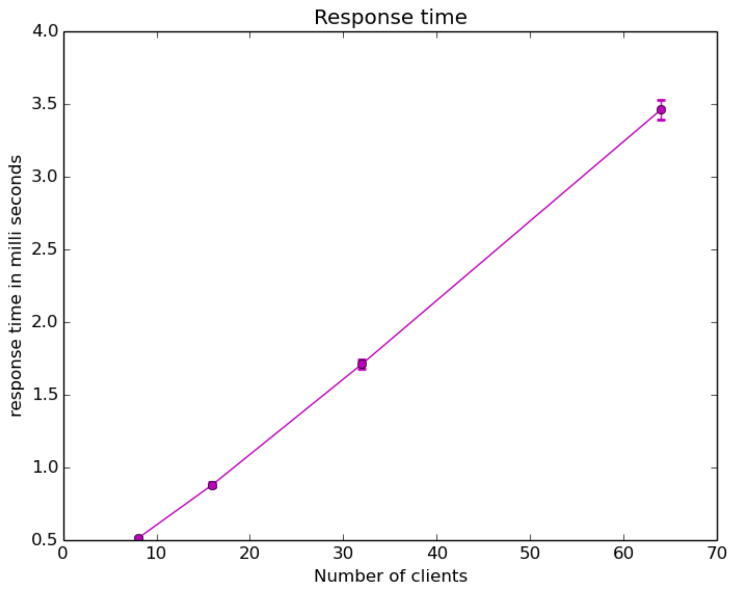

Figure 5: Response-time with one middleware and different number of clients

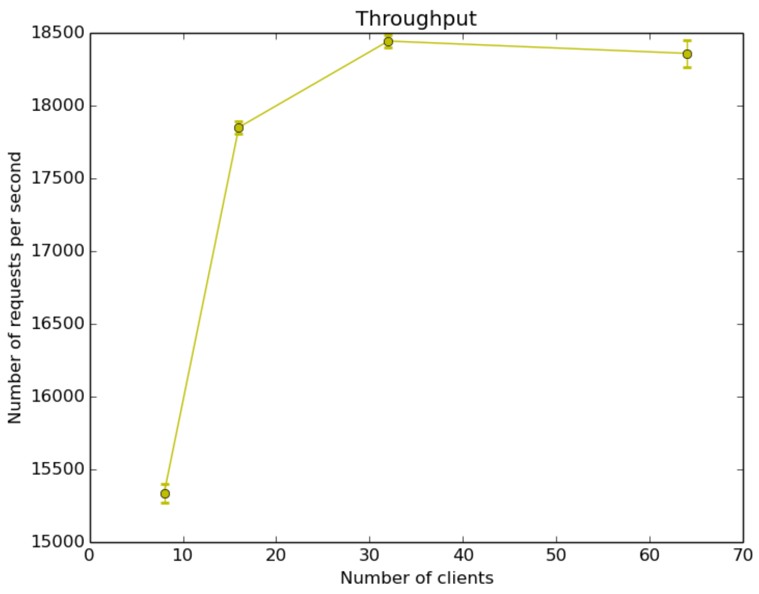

Figure 6: Throughput with one middleware and different number of clients

1.3 Clients

1.3.1 Design and interface

Clients use non-blocking I/O. I implemented two main strategies for generating loads for the experiments:

- Sending as many request as possible with a parameter *think_time* that could be set for each experiment [1]
- Sending the request in a regular intervals for instance sending requests every *1ms*[2]

Each client has assigned a *log file* in which it writes its response time values Also, I considered printing "*One Block*" after every second in log files, such that I can divide the response times in one second intervals and use them in my computations when I need to average the response times over a specific intervals.

1.3.2 Instrumentation

1.3.3 Workloads and deployment

Each client can send one type of the request: query, peek, pop, and send. In order to generate the workload of different combinations for instance query and pop, one could use different clients for this purpose, and assign one operation to each client. Number of clients and their corresponding operations can be set by the argument to the program. All possible parameters of the program such as number of users, length of the message, etc can be set in the code [3]. I considered generating random string of specific length for the message the client wants to send. I also considered generating the *queue_id* and *receiver_id* randomly to simulate more realistic workload generation.

[1] *FlowRequestRunner*
[2] *RegularRateRunner*
[3] *Parameters*

1.3.4 Sanity checks

For ensuring correct load generation and validity of responses, each client will check if it has received a valid response, and throws an exception otherwise in the log file. Then, one could check all the log files and repeat the experiments in case of any error occurance during the experiment.

2 Experimental Setup

For database I used *m3.large* because it has *7.5 GB* of memory, and the memory was large enough to be suitable for my experiments. For server and clients, I used *t2.medium* instance, because I do not need that much memory on client and server, but I need more CPU, and this instance has *2 CPU* and *3.75* GB memory.

2.1 System Configurations

In general, the system consists of one database, a few number of middlewares and several clients which are connected to the middleware instances. Clients send the requests to the middleware, middleware parse the request and send the task to the database, when the results are prepared, middleware send back the response to the clients.

2.2 Configuration and Deployment mechanisms

Inorder to run the experiments, I wrote a *bash* script for each experiment, the scripts are in the folder of *bash_scripts*. All the parameters that one need to set to run the experiments such as *server inet address*, *message length*, etc are passed as arguments to the *jar files* in the scripts. I used up to 2 middleware instances in my experiments. During development of the system, I wrote several *Junit* tests to make sure about the functionality of the each part of the system.

For computing different statistics from the results of the experiments such as computing the *throughput* and *response_time*, I wrote two *python* scripts which are in the folder of *analyze*. The first one is called *analyze_measurements.py* for computing the mean and standard deviation for throughput and response-time of each experiment. The second script is called *generate_time_series.py*. This script is only used for the *stability* experiment for generation the response time and throughput values for the longer runs.

3 Evaluation

3.1 System Stability

For this experiment, I considered 40 clients which are sending 25 requests per second and I distribute their requests into two middleware instances. So, in total system receive the load of 1000 requests per second. 20 clients send the request to the first middleware instance and 20 other clients send their requests to the second middleware instance. I used *m3.large* instance for the database, and two *t2.medium* instances for two middlewares, and another *t2.medium* instance for the clients. I initialize the database with *2000* messages. Number of backend pool workers is set to *10* so having two middleware instances, in total 20 database connections are used. I set this number because in the database experiments, I observed that system achieve the maximum performance for around 15-24 database connections. Number of FrontendPool workers does not really matter, as shown in the middleware experiments, database is the bottleneck and with 15 frontend workers middleware should be able to handle a large number of requests. I used a workload with combination of pop and send with equal portion of each operation such that the size of the database remains the same over time. The parameters I used for this experiments are shown in table 3. The trace of system throughput and response time is depicted in figures 7, and 8.

Message Length	200
Number of clients	40
Number of middlewares	2
Experiment time	30 minutes
Frontend pool size	15
Database pool size	10
Rate of requests	25 requests per second
Workload	Combination of pop, and write (50 % each)

Table 3: Configuration used in the stability experiment

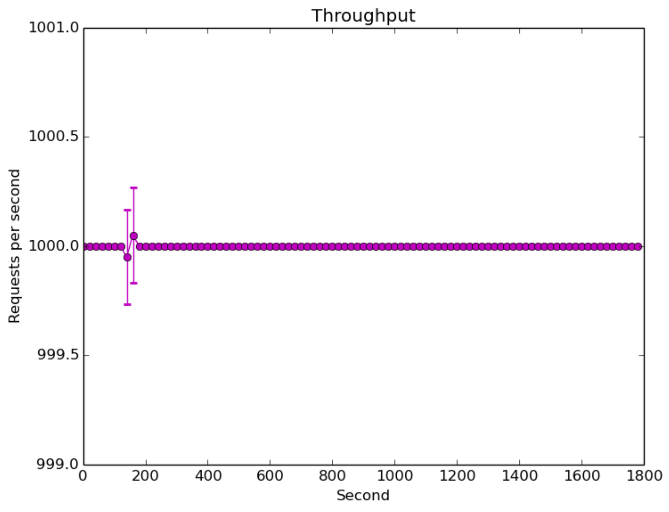

Figure 7: Throughput of the system over 30 minutes run for 1000 requests per second

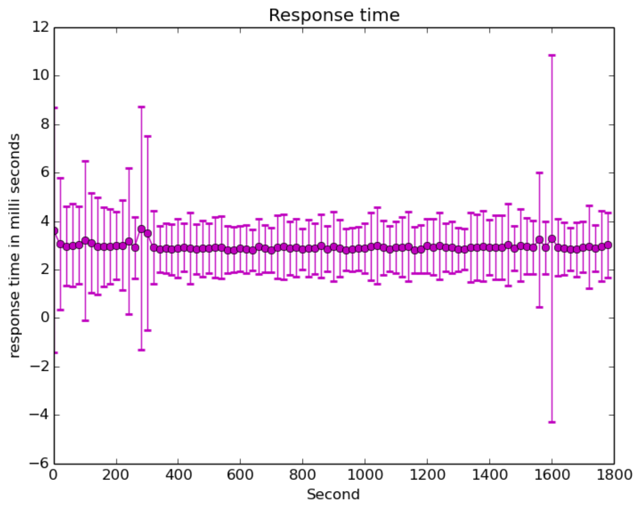

Figure 8: Response time of the system over 30 minutes run for 1000 requests per second

For sanity check, each client, checks the response it receives, and if the response is invalid it logs an error in the log files. The logs for this experiment can be found in the folder *logs/stability*. Response time and throughput values are averaged over *20* seconds intervals. The response times and throughput values are quite stable. Small variances in the response times can be attributed to the shared nature of *AWS EC2* environment.

In the second part of this experiment, I applied 4 times more load, 100 request per second, with the same configuration as the first experiment. In total, the load on the system will be 4000 request per second. I chose these values because in the experiment shown in section 1.1.4, the system is able to support up to 4000 write operations per second and pop and write operations have roughly the same throughput and response times, so the system should be able to handle 4000 requests per second. The logs for this experiment can be found in *logs/stability-2*, and the results are depicted in figures 10 and 9. The results show that the throughput and response times remains stable. The response time and throughput values are averaged over 20 seconds intervals. As we can see, every drop in the throughput graph is followed by a spark, meaning that the requests in the system are not lost, but it indicates that a request is started in the previous interval but completed in the next interval. The variances in the response times are about *6ms*, which is quite small and can be attributed to the shared nature of *AWS EC2* environment.

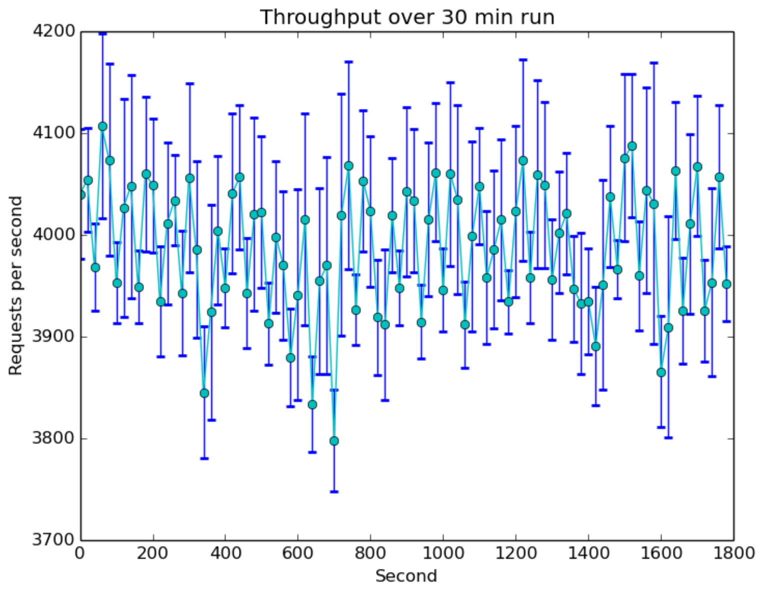

Figure 9: Throughput of the system over 30 minutes run for 4000 requests per second

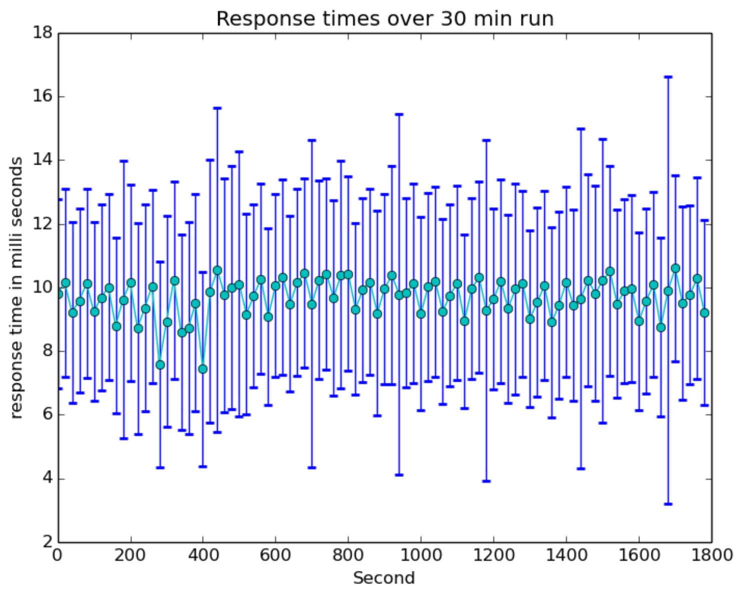

Figure 10: Response time of the system over 30 minutes run for 4000 requests per second

3.2 2^k Experiment

In this experiment, I would like to analyze different possible factors on system throughput. I considered the factors in table 3.2. Values have been chosen in a way that throughput for Level 1 is expected to be higher than the throughput for level -1. For instance, having 2 middleware instance and smaller message sizes will help to have more throughput. Throughputs values are averaged between three times repetition of the experiments each one for 5 minutes duration, statistics and logs can be found in *logs/factorial*.

Factor	Level -1	Level 1
A: Message Length	1200	200
B: Number of middlewares	1	2
C: Number of clients	32	64

Table 4: Factors for 2^k throughput experiment

The results of the 2^k throughput experiment have been shown in 5. Based on the results, number of clients have the most substantial affect on throughput. The reasoning is that, as it was shown in middleware experiment, one middleware instance can easily handle upto 18500 requests per second and middleware design is highly scalable, so having more middleware instances does not help to have more throughput as only one middleware instance is already enough. Furthermore, message length only has a negligible affect on throughput as network transmission time constitutes only small fraction of the final response time values.

In the second part of this experiment, I tried to analyze the same factors affect on response time. Factors chosen for this experiment are shown in table 6. Again, factors have been chosen in a way that response time for Level -1 is expected to be smaller than response time for factors in Level 1 because by having larger message length and smaller middleware insatnce response time is expected to increase as for larger message sizes, it is expected to have more networking overhead. In additon, having less clients means less load for the system and hence shorter response times. The experiment is ran for 5 minutes, and response time values are averaged every 20 seconds. The results of the experiment are shown in table 7. Based on the results, again one can see that number of clients have the biggest affect on response time. The same

I	*A*	*B*	*C*	*AB*	*AC*	*BC*	*ABC*	*TP*
1	-1	-1	-1	1	1	1	-1	3909.3
1	1	-1	-1	-1	-1	1	1	4040.9
1	-1	1	-1	-1	1	-1	1	4851.8
1	1	1	-1	1	-1	-1	-1	4784.0
1	-1	-1	1	1	-1	-1	1	3896.6
1	1	-1	1	-1	1	-1	-1	3966.2
1	-1	1	1	-1	-1	1	-1	2174.6
1	1	1	1	1	1	1	1	2301.5
29925	260.30	-1701.1	-5247.1	-142.10	132.70	-5072.3	256.70	ToTal
3740.625	32.538	-212.637	-655.888	-17.762	16.587	-634.038	32.087	ToTal/8
Variation	0.120294 %	5.137535 %	48.880291 %	0.035850%	0.031263 %	45.677778 %	0.116989 %	

Table 5: Sign table for throughput

reasoning as the throughput experiment also applies here. Also, number of middleware nodes have a very small affect which is meaningful as increasing the number of middleware nodes could decrease the latency for a small amount.

Factor	Level -1	Level 1
A: Message Length	200	1200
B: Number of middlewares	2	1
C: Number of clients	32	64

Table 6: Factors for 2^k response time experiment

I	*A*	*B*	*C*	*AB*	*AC*	*BC*	*ABC*	*RT*
1	-1	-1	-1	1	1	1	-1	6.9675
1	1	-1	-1	-1	-1	1	1	6.7175
1	-1	1	-1	-1	1	-1	1	7.8725
1	1	1	-1	1	-1	-1	-1	8.2642
1	-1	-1	1	1	-1	-1	1	29.858
1	1	-1	1	-1	1	-1	-1	31.567
1	-1	1	1	-1	-1	1	-1	16.161
1	1	1	1	1	1	1	1	16.461
124.1654	2.2556	-26.4396	63.9286	-1.0806	1.7624	-31.1664	-1.7374	ToTal
15.52068	0.28195	-3.30495	7.99107	-0.13508	0.22030	-3.89580	-0.21717	Total/8
Variation	0.088181 %	12.116054 %	70.833924 %	0.020240%	0.053834%	16.835450%	0.052316%	

Table 7: Sign table for response time

3.3 System Throughput

In order to measure the maximum throughput of the system, I used all the information I learned from my system in previous experiments. In section 1.1.4, I observed that the system can achieve the best throughput for *16-24* database connections. In addition, based on the experiment in section 1.2.4, one middleware instance can easily support up to 18000 requests per second. Furthermore, in section 3.2, I observed that number of clients has the most prominent affect on throughput. Hence, for this experiment I came up with the following set up. I will use 16 database connection, 30 Fronend Pool workers, and a workload of combination of the all of the system operations meanning that I considered *pop*, *peek*, *send*, *query* each one with equal portion, then I measured throughput and response time for different number of clients. I considered *think_time* of zero for this experiment. Throughput and response times are shown in figures 11 and 12.

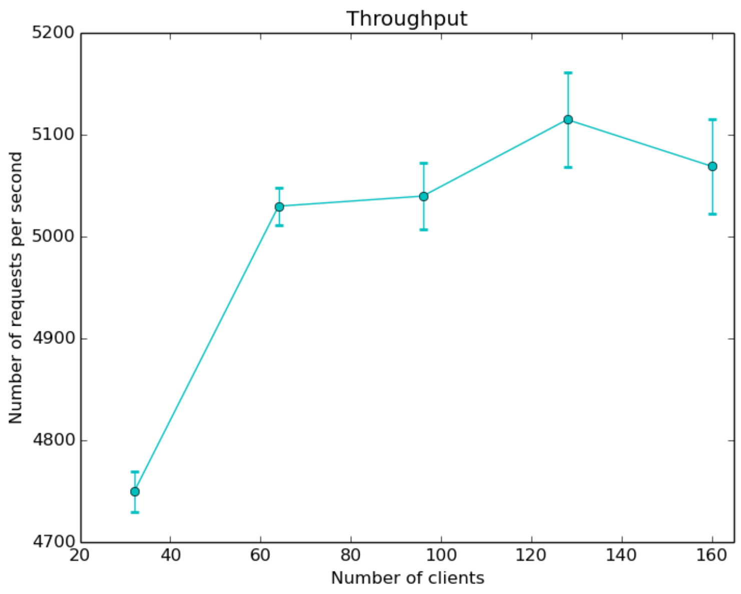

Figure 11: Throughput of the system for different number of clients

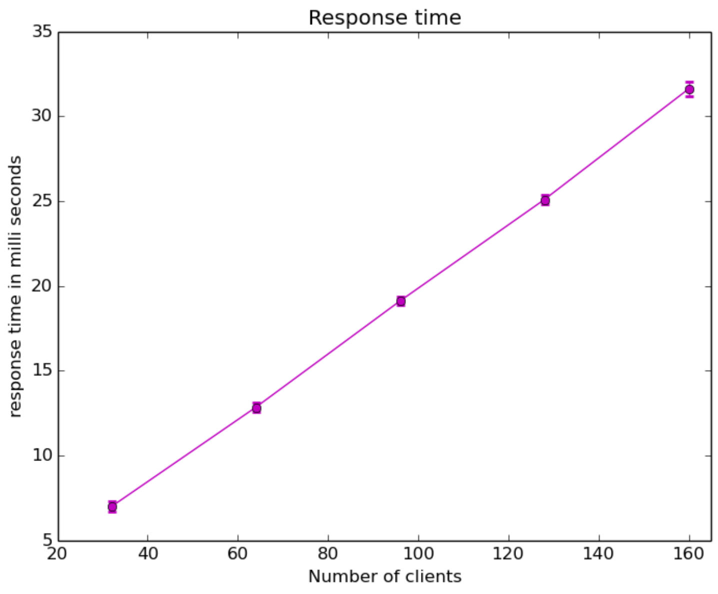

Figure 12: Response time of the system for different number of clients

As expected the max throughput is around 5000 requests per seconds which is around the max throughput of the database. This experiment shows that database is the bottleneck in this design. Also, we should consider that we have *Query* operations in this experiment which is not considered in the database experiment, however, it should be roughly of the same cost as *Peek* operation. In addition, in this experiment I considered 16 database connections for each middleware instance which also helps to have optimal throuhput.

3.4 System Scalability

In this section, I would like to explore the scalability of the system based on the results obtained in the previous experiments. In section 1.1.4 , we observed that 12 to 24 database connections can achieve maximum performance. Furthermore, in section 3.3, it was shown that above 80 clients should be able to create enough load to saturate the database. For this experiment, I choose combination of the *peek*, *pop*, *query*, *send* operations, each one with *25%* contribution to the whole load to simulate the realistic load on the system. I chose 96 clients, because first, based on the experiment in 1.1.4 they can generate enough load to saturate the database, second, this number is divisable by 4, 8, and 12 which I need in this experiment to distribute the number of clients equally between middlewares nodes and 4 different operations. To sum up, I run this experiment for 96 clients, and different number of middlewares *1,2,3*, and each middleware has either 6 or 10 database connections. The logs for this experiment can be found in folder *logs/scalability*. Obtained throughput and response times are shown in 13, and 14. The results show that up to 20 database connections are enough to have the maximum performance. This is consistent with the experiments conducted in 1.1.4, in which we observed that 15-24 database connections will result in maximum performance. The second point from the figure 13 is that throughput for the configuration of 3 middleware instance with 6 or 10 database connections per middleware are roughly the same. This means that with 30 database connections(10 database connection per instance), the system cannot saturate all of them. However, in the first configuration with 3 middleware instance and 18 database connections, database connections are saturated and the system uses all of them optimally. By looking at the response time graph, we observe that with 6-10 database connections in one middleware instance, response time values are very high, meaning that the system has too few database connections to handle the requests. Adding more database connections improve the reponse times, and one could have the optimal performance for 12-20 database connections. Furthermore, I should mention that number of middleware instances does not really affect the throughput or response times. As shown in section 1.2.4 one middleware instance can handle much more requests than database, and database is the bottleneck in this design.

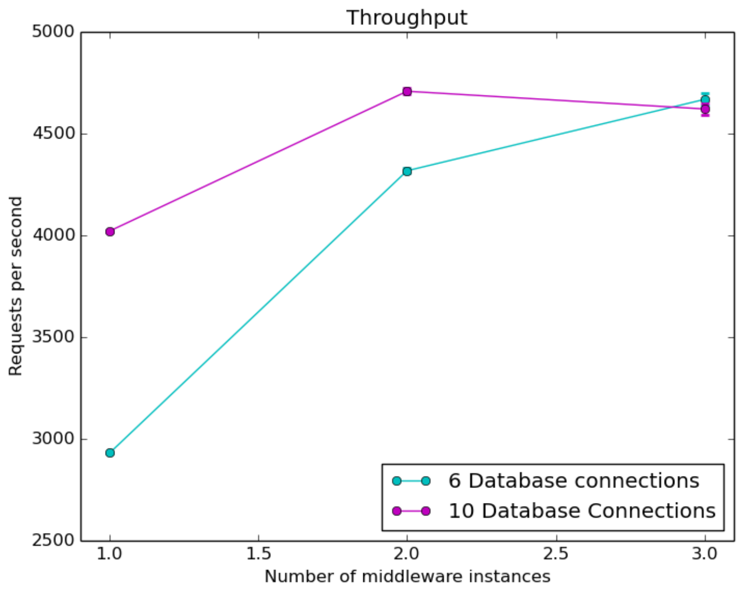

Figure 13: Throughput of the system for different number of middlewares and database connections

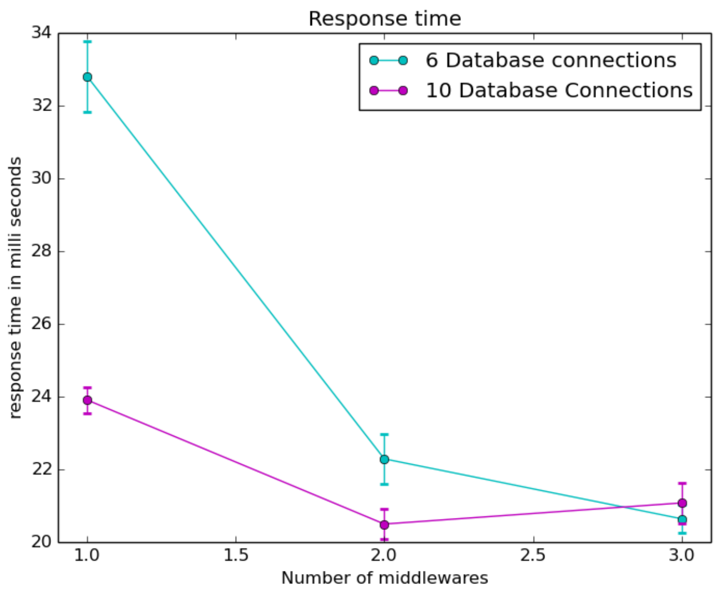

Figure 14: Response time of the system for different number of middlewares and database connections

3.5 Conclusion

In summary, system in general performs well in terms of throughput and response times as shown in the conducted experiments. In section 1.1.4, it was shown that system has the highest performance for *15-24* database connections. Furthermore, as shown in section 1.2.4, by using *java nio* package and *asynchronous I/O*, one single instance of middleware is able to handle above 18000 requests per second. However, in this design database is the bottleneck, and system performance is limited to the database performance. To make the system highly scalable, one could distribute the database along several machines, also it could be a good idea to have several instances of the *ConnectionManager* to handle the incoming requests asynchronously instead of only one *ConnectionManager* which could become a bottleneck under high loads. If desining the system anew, I would go for the same design, as it was proved in the experiments using the *java nio* packages clearly pays off. However, to simplify the design one could use only one *thread_pool*. I will also consider another table for the users to authenticate them, also this could be helpful to have the information of the users in the database because this way, one does not need to send the user informations along with their requests, this could help to use a little less bandwidth.