Advanced System Lab – 2014

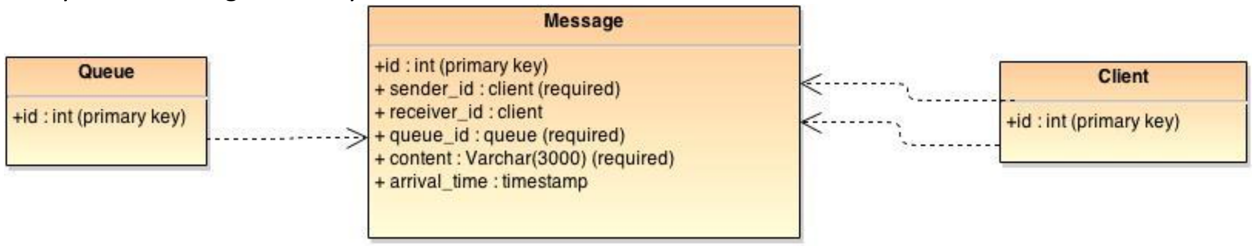
Project report – milestone 1

Amrollah Seifoddini Banadkooki

1. Database

- **Schema**

I tried to keep the database as simple as possible. So, no unnecessary field or index added. Here is the Entity Relation Diagram of my scheme.


Queue
+id : int (primary key)
Message
+id : int (primary key)
+ sender_id : client (required)
+ receiver_id : client
+ queue_id : queue (required)
+ content : Varchar(3000) (required)
+ arrival_time : timestamp
Client
+id : int (primary key)

- **Indexes**

As postgres automatically creates indexes for primary key fields, I only need to specify extra indexes which are for faster message retrieval.

```
CREATE INDEX sender_idx ON message (sender_id);
CREATE INDEX queue_idx ON message (queue_id);
CREATE INDEX receiver_idx ON message (receiver_id);
```

The sender_idx is helpful when client query for messes for a particular sender. Even though this type of query is rare in my client load, but it worth having it. Queue_idx is very important because we heavily query message table for messages with a particular queue_id. Receiver_idx is necessary for performance because in all of my queries on message table, I have a filtering based on receiver_id as well.

- **Interface (queries, stored procedures)**

For boosting the performance of accessing the database, I created stored procedures for every query that system runs even if they are very simple. So, there is nothing to say about application level queries because there is none. Here all of stored function are listed with a brief description.

/* Functions */

The create_client function is for creating clients on their first visit. Clients introduce themselves with id and system create or identify them. If a client already exists, the id will be returned.

```
CREATE OR REPLACE FUNCTION Create_Client(IN client__id INT)
RETURNS INT AS $body$
BEGIN
    INSERT INTO client(id) values($1);
    RETURN client__id;
EXCEPTION
    WHEN unique_violation
    THEN RETURN client__id;
END;
$body$ LANGUAGE plpgsql;
```

Create_queue function creates a queue and returns its id. Clients create queue via this function.

```
CREATE OR REPLACE FUNCTION Create_Queue(OUT id   INT)
RETURNS INT
```

```
AS 'INSERT INTO queue values(default) RETURNING id'
LANGUAGE SQL;
```

Delete_queue function deletes a queue given by the id and returns its id. Clients can delete queues via this function.

```
CREATE OR REPLACE FUNCTION Delete_Queue(id INT = NULL)
RETURNS INT
AS 'DELETE FROM queue WHERE id=$1 RETURNING id'
LANGUAGE SQL;
```

Insert_message function is for inserting messages into database and return its id. It takes all the possible fields of a message and store it in DB. Receiver_id can be NULL here. The arrive_time field will be automatically set by postgres when the message is committed to db.

```
CREATE OR REPLACE FUNCTION Insert_Message(sender_id INT, receiver_id
INT, queue_id INT, content varchar(3000))
RETURNS INT
AS 'INSERT INTO message(sender_id, receiver_id, queue_id, content)
values($1,$2,$3,$4) RETURNING id'
LANGUAGE SQL;
```

The msg_result type is defined to bring back the result of message select operation as a single row. The system will never returns more than one message, so this is what we need here. It includes all the fields of message.

```
CREATE TYPE msg_result AS (id INT, sender_id INT, receiver_id INT,
queue_id INT, content varchar, arrive_time timestamp);
```

This function is used for receiving a message given its queue_id and receiver_id. It will return the most recent message in the queue identified by queue_id; that either its receiver_id is equal to the parameter passed to function, or its receiver_id is NULL meaning it is a broadcast message.

```
CREATE OR REPLACE FUNCTION Get_Message_By_Queue(receiver_id INT,
queue_id INT)
RETURNS msg_result
AS 'SELECT id,sender_id,receiver_id,queue_id,content,arrive_time FROM
message WHERE queue_id=$2 AND (receiver_id IS NULL OR receiver_id=$1)
ORDER BY arrive_time DESC LIMIT 1'
LANGUAGE SQL;
```

This function behave very similar to get_message_by_queue except that it is for retrieving a message by giving its sender_id and receiver_id. It is rarely used in the client workflows, but it's pretty useful in case client wants to query for messages from a particular sender. This select statement is similar to last function except that queue_id is replace by sender_id.

```
CREATE OR REPLACE FUNCTION Get_Message_By_Sender(receiver_id INT,
sender_id INT)
```

```
RETURNS msg_result
AS 'SELECT id,sender_id,receiver_id,queue_id,content,arrive_time FROM
message WHERE sender_id=$2 AND (receiver_id IS NULL OR receiver_id=$1)
ORDER BY arrive_time DESC LIMIT 1'
LANGUAGE SQL;
```

This function deletes a message given its id and if it is successful, return message id.

```
CREATE OR REPLACE FUNCTION Delete_Message(id INT = NULL)
RETURNS INT
AS 'DELETE FROM message WHERE id=$1 RETURNING id'
LANGUAGE SQL;
```

This function is used by clients to query for queues that have messages waiting for them. It will return the list of queue_ids as a table. It looks through message table and find those with receiver_id equal to the given parameter or with NULL value.

```
CREATE OR REPLACE FUNCTION List_Pending_Queues(IN id INT)
RETURNS TABLE(
      queue_id INT
)
AS 'SELECT DISTINCT queue_id FROM message WHERE receiver_id IS NULL or
receiver_id=$1'
LANGUAGE SQL;
```

This function lists all the queues in the database. Clients use this to get the possible queue_ids and then choose one queue_id for sending their message to. This method is also used in DB monitoring tool for counting the number of queues in DB at each time.

```
CREATE OR REPLACE FUNCTION List_Queues()
RETURNS TABLE(
      id INT
)
AS 'SELECT id FROM queue'
LANGUAGE SQL;
```

This function returns list of clients in the database. Clients use this function to get possible receiver_ids and then choose one id for sending their message to. They also might select none of them, meaning that the message would be broadcast. This method is also used in DB monitoring tool for counting the number of clients in DB at each time.

```
CREATE OR REPLACE FUNCTION List_Clients()
RETURNS TABLE(
      id INT
)
AS 'SELECT id FROM client'
LANGUAGE SQL;
```

This utility function counts the number of messages in database. It is used in monitoring utility which shows the state of DB periodically.

```
CREATE OR REPLACE FUNCTION Count_Messages()
RETURNS INT
AS 'SELECT COUNT (*) FROM message'
LANGUAGE SQL;
```

This function is somehow a debugging method for middleware to determine the proper error message for client in case of failure. If a user try to read rom/write to a queue which does not exist, database query fails but then middleware execute this function with the queue_id to see if missing the queue was the reason of failure of something else is broken. The results of this function is queue_id if there is a queue with that id, null otherwise.

```
CREATE OR REPLACE FUNCTION Queue_Exist(IN q_id INT )
RETURNS INT
AS 'SELECT id FROM queue WHERE id=$1'
LANGUAGE SQL;
```

- **Design**

For the design I followed the project description which says client, queues and messages should be persistent. So, I created tables for each of them to be persistent independently. If for example clients didn't have a separate queue, they would be gone after deleting the last message. And also finding the clients list in database would be a heavy select task. For simplicity and keeping the size of database small, I kept client and queue tables as minimal as possible, so they only have ID field. The arrive_time field for message is a timestamp field which is filled by DBMS at the time of insert; so, it's efficient and fast in sorting as well. Also, having Foreign Keys from message table to client and queue tables helps to for keep the database consistent and prevent illegal deletes. For example, if a client try to delete a queue while it still has some message in it (it is referenced by some message rows); DB will not delete the queue and returns an error.

For improving the query execution time, I created postgres stored functions inside the database scheme. This way, we save the time for creating and parsing the long statements for getting or inserting a message. Even though that I'm using postgres stored functions for these queries but still I have to run a select query for executing a stored db function. Parsing that query into a query plan for postgres is done is two levels, first in JDBC and second in DBMS itself. This two levels take a bit of time even for my simple queries which only calls a db stored function. So, I decided to again crack it up a notch and further improve the database performance by using prepared statements for most frequent queries. In my implementation of clients, sending, getting and reading messages are most frequent queries. And according to postgres logs and pg_stat_statement statistics, getting message is the most expensive query, and insert/read message are equal in terms of execution time and they are way cheaper than getting message. That is because getting message involved deleting a message and updating indexes heavily. Also getting pending queue query is expensive but since it's not that frequent, I did not use prepared statement for that. With this design, I only need to pass the new parameters to the prepared statement and execute it for every get/insert request no matter which client it comes from. In the image below it shows how statement cache in database can help us in this design to achieve higher throughput.

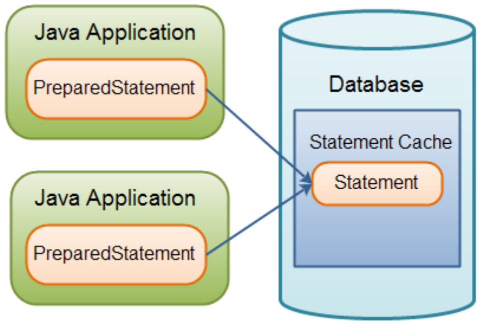


Figure 1: The caching of PreparedStatements in the database. (Source: http://tutorials.jenkov.com/jdbc/preparedstatement.html)

For deployment on Amazon, we used a postgres 9.3 with customized configurations. For example, I limited the number of connections to 200, increased the cache size to %90 of machine memory, increase buffer. Also I activated some statistic configuration for logging. In the middleware I also cache query results in JDBC. For assuring consistency and ACID features in database, I forced the isolation level of "Serializeable". Even though it makes the DB a bit slower but my database is fast enough in the whole that I don't care about his small overhead.

- **Performance characteristics of the database (throughput, response time, scalability)**

With combining the designs described above and running the postgres database on m3.large machine on Amazon, I managed to get a really good response time and throughput for database tier such that it's mostly below 10 milliseconds under load of 1200 clients.

For evaluating database performance characteristics I did an experiment which in that I insert and get messages from db on the db machine itself. So, there is not internet link of waiting for connection involved, but of course I still used prepared statement for testing. I kept running the test for 30 seconds and according to the result, throughput of database is 2880 message per second. In this test, size of database was around 750,000 messages. The average response time for inserting message is 1.2 ms and for getting message (which include deleting message) is 5.6 ms. I did the same test with database size of 1200,000 message and 4000,000 messages. The response time for inserting message was 1.7 and 2.4 respectively. Obviously, the response time of database increased as the database grows, and then throughput goes down as well. But these response time is still negligible for database of size less than several hundred million messages, compared to long waiting time in the middleware. So, I conclude that database is fast enough for my application with reasonable database size and it cannot be the bottleneck. This means throughput and response time will not have significant change as the database grows normally until reasonable size (ex. 20,000,000 messages). If we need a database of that big, we should consider clustering and separating different tables over different database instances.

2. Middleware

- **Overall design**

The middleware tier is connected to clients in one side and is connected to database tier in other side to execute the commands coming from clients. So, it has a main thread listening on a port for clients and uses a connection pool to communicate database. For handling the clients, it uses a threadpool of workers which is much smaller than number of clients in high load.

- **Connection pool to the database**
For the connection to database, I use connection pool of JDBC. So, connection are created in advance and is shared between all workers. Every worker that get a connection, handle the client request and then release the connection back to the pool. The connection pool size is set to 4 times of number of workers. This is because as I told in DB design section every worker gets 3 prepared statement for handling the send, read_from_queu and read_from_sender commands. And every statement uses one dedicated connection and will not release it until the end of worker's life which is when the experiment is done and make thread of middleware exists. Also, every worker needs one connection for handling all other requests. This means that we should have 4*worker_count to be sure that we never run out of connection for clients. We definitely cannot have less than (3*worker_count + 1) connections and it is not necessary to have more htan 4*worker_count since every worker is dealing with only one client at a time and afte finishing the task, if the connection was not from a prepared statement, it will be released. So, the closer we get to 4*worker_count, the less waiting is for workers for the connection in normal queries.
- **Connections to the clients**
Communication to clients are done via java sockets. Upon receiving a client, middleware opens a socket foe individual communication to that client and then add that socket to queue of clients.
- **Queuing mechanism**
I designed the middleware to be able support a lot of clients at the same time. For this purpose every middleware instance which has a queue (from BlockingQueue of java) that keeps the client sockets and share it with several workers that handle client requests. One the middleware starts, it initialize several workers (number of them is set in configuration file) in different threads. These workers run with a threadpool structure. Then the middleware start listening for client connections in an infinite loop, once it got one and accept it, it will create the socket for communication to that client and put the socket into queue. Also, client_id is put in separate blocking queue because workers need both of these to communicate properly with the right client.
Each worker is running in an infinite loop. In each iteration, it get the head of client queues and client queue_ids, then start reading the socket of client for any commands. One client sent a command and worker receive it, the command is forwarded to commandHandler function where it is dispatched to relevant method for execution. If command needs some data, commandHandler wait for client to send them as well. Some commands like register, unregister, create_queue and list users/queues don't need more data. After the task is finished and result is returned to worker, it will write that into client socket. Then worker put the client into queue again and go to next iteration to pick the next client. Since there are several workers, client does not have to wait a lot in the queue as long as the size of queue remains reasonable. The size of this queue is very important for improving performance. If a worker is blocking on client socket for a command and client does not send anything for a specific timeout seconds, then worker put the client back in the queue and go for next iteration. This is necessary for avoiding infinite blocks. When a client send the un-register command it will be removed from the queue.

- **Explanation for the design** (what you wanted to achieve, design decisions, expected behavior)
With this design, I wanted to allow a huge number of clients to be connected to system and get their requests answered, even though some of them have to wait a long time (up to several seconds) for receiving the answer. The queueing mechanism does this properly. Also, with a small number of workers, the number of db connections is relatively low. For 8 worker we need around 30 connection for optimal performance. If the queuing mechanism was not used and I had created a thread for each client that connect to middleware, then for handling 2000 clients, scheduler of operating system would have been busy all the time and every client get only a little time slot and the context switch between threads

would be huge overhead which decreases the throughput. But with this design we won't be having this problem in large number of clients. Also, the database connection time in very low in this case since I use prepared statements. This means that the most of response time for clients will be for waiting in the queue.
- **Performance characteristics of the middleware** (throughput, latency, scalability)
The latency of middleware is measure by removing the DB latency from overall time which a message spend in the middleware. It was different in different hours of day based on load on amazon serves but it was always something around 10 times of DB response time. So, in this case, it was 60 ms and DB response time was 5 ms. For scaling the middleware, we can add more workers per instance ort add more nodes to the system. The best performance for small number of clients can be achieved by setting the number of workers ass the number of clients but after that, there will be no improvement. Also, for large client number it is not feasible to increased number of workers to that point as there is not enough DB connection. Every node needs atleast 3*worker_count db connection but I set the postgres connection limit to 200. So, any worker number after 60 is meaningless. It is recommended to have a small number (for example 2*cpu cores) of workers per middleware instance and for scaling add the number of nodes instead.

3. Clients
- **Design**
I designed clients as a class which runs in a thread. The main thread in clients' machine create many instances of clients, assign each one of them to one of the available middlewares in a round-robin approach (for balancing the load) and the run them inside a thread. Every client first wait for some seconds to give all clients the required time for registration in middleware and database. Then it goes through an infinite loop for workload generating until it exceed the runtime_duration which is a parameter in configuration file. There is not pausing in this loop, so, clients generate a lot of load.
- **Workloads**
In that loop, clients are divided in three category, populate_client (which only send message), consumer_client (which only gets or read message), and normal clients (which do both sending ad receiving). For experiments, I only use normal clients but for filling the database or decreasing its size I used populater and consumer clients too. Evert normal client, first query for list of users in the system, then query for list of queues in the database, then it choose a receiver and a queue randomly out of them and send a message to that client on that queue. But client does this message sending with probability of %90 because otherwise database size grows rapidly. Also, client does not query for client list and query list in every iteration; it only does that every couple of seconds which is in set in configuration. In the next part, client asks for queue where they have messages pending for it. Then it choose a queue randomly out of them and query for the top recent message in that queue. Here client either issue a get query or a read query which does not remove message from queue. To keep the size of database roughly even during system runtime, client get messages on %90 percent of time and read messages on %10 of the time. This is because I wanted to have read message in my queries and I had to limit their execution frequency with this probability. In practice, it keeps the database size roughly constant and help to get more precise measurements.
- Instrumentation, scalability, deployment
Clients can issue one of these commands: register, un-register, send msg, get msg, read msg, create queue, delete queue. The operation of these commands are trivial but a more comprehensive description of them along with parameters and expected response from server is available in protocol.txt file. Clients can be as many as number of allowed socket or thread per process. Even though one cannot expect to have the same response time with 2000 clients in one machine. Scalability of client tier is done with increasing client numbers per machine or adding a new client machine. For deployment

and running experiments, I only used ant; so most of the parameters are passed via commandline to clients. For example the command ant -Dserver_host=172.31.9.249, 172.31.9.250 -Dcl_number=40 -Dclient_type=client -Doffset=0 -Drun_time=10 run-clients will create 40 normal clients in current machine and run them for 10 minutes, middlewares are listed under server_host and offset says that these are client number 1:40. This way we can run individualize clients as many as we need.

- **Performance characteristics of the client**
For evaluating the number of messages an *isolated client* can send per second I designed an experiment where clients send go through sending procedure but never write into middleware socket. This means there is not network involved and they send as many message as they can in the specific period of time. Then I count the number of sent messages from client (regardless of what and when happened to them in middleware). So, after averaging I get 80 for each client. This means on my setup, each client can send up to 80 messages per second on a t2.small machine of Linux 64bit AWS. In this experiment, there were 40 clients running on different threads on a single machine. In other words, the load on the middleware from this machine would be 80*40=3200 message per second which obviously it cannot handle.
I did another experiment to see how many messages network can send per second. In the new experiment, clients also write into middleware socket (corresponding commands and data of message) but they don't wait until the result comes back. They keep sending messages in this way. After aggregating the log and averaging over number of clients, I get the number 77 message per second for every client. So, it means that network is not as fast as clients since we have 80 message per second if they don't send over the network. But anyways, the whole client tier is still way faster than middleware. We should take this into account that all 40 clients were running on one machine, so I expect that real number of messages per second for a client running on a single machine would be higher than aggregated number (40*77) that we got here. Because in that case, we don't have overhead of context switch between client threads. Based on this result, I can definitely say that think time of clients is negligible compared to other times involved in the process. Then, clients are fast enough for generating a high load on middleware and they cannot be a bottleneck in the performance.

- **Sanity checks on workload generation and correctness of responses**
When the clients generate load by inserting message or getting message from middleware, it goes through a validation process. Risky part of code is always put in try and catch clause to assure smooth running of clients which will not exit on small errors in one client. In those cases, all other clients are still running and generating load. The request and response between client and middleware should follow the specified protocol which described in protocol.txt file. Upon successful response client receive OK message and otherwise a proper error description will be returned to client. For example if a client tries to read an empty queue it will get a relevant error. Also, every message which is received is passed to a parser that convert it to a message object from Message class and if there is any problem is the message it will be caught there. Also, the same thing happens when worker convert the result or message to a string and pass it over network.

4. Overall design
My system has one database node (m3.large AWS), two middleware nodes (t2.medium AWS) and two clients node (t2.small AWS). For building, creating jar file, running middlewares, clients and DB-monitoring and even log analyzing I use ant.

- Preparing and performing experiments
For deployment and uploading the system on the AWS machines, I used WinSCP script because I'm using Windows. Also, for downloading the log files I use WinSCP scripts.

For each experiment I set relevant parameter while running ant. These parameters are passed to client and servers and this way the desired behavior is configured for testing. The log_parser class in my analytics package in this project is used to parsing log files and extracting response time and throughput and some other detailed measures. I aimed for %95 confidence interval when aggregating logs. Variance is also calculated and all of these values stored as .csv file. Later on I use Gnuplot for drawing graphs out of this file. All the winscp and gnuplot scripts are available in docs folder of project. I used elastic-IPs for my main nodes in the Amazon, so the IPs were fixed for most of the experiments.

5. System stability

For the stability test, I ran two middlewares and two client machines and one database machine for 30 minutes. Each middleware runs 4 worker threads which handle client queries. Each client machine runs 40 client as different threads. As always, I use only one Database machine. The middleware machines are of type t2.medium Amazon Linux 64 bit (which has 2 CPUs, 4GB memory and 20GB SSD disk space). Client machines are t2.small machines. The graphs below show how stable system behaved in terms of throughput and response time. The logs for this run in available in log folder beside this report.
 Note that warmup and cool down phases are still visible in this graph. I normalize the measures metrics over every minute, because in Amazon, different machine clocks might be not synchronized and if I average over seconds, the different graphs might not match with each other (ex. throughput and response time). Warmup and cool down phase is around two minutes here. Because clients are still registering themselves during warm up phase and load is not complete. In the cool down phase they are un-registering and again load in not normal. Variance of measurement are also displayed along the graph.

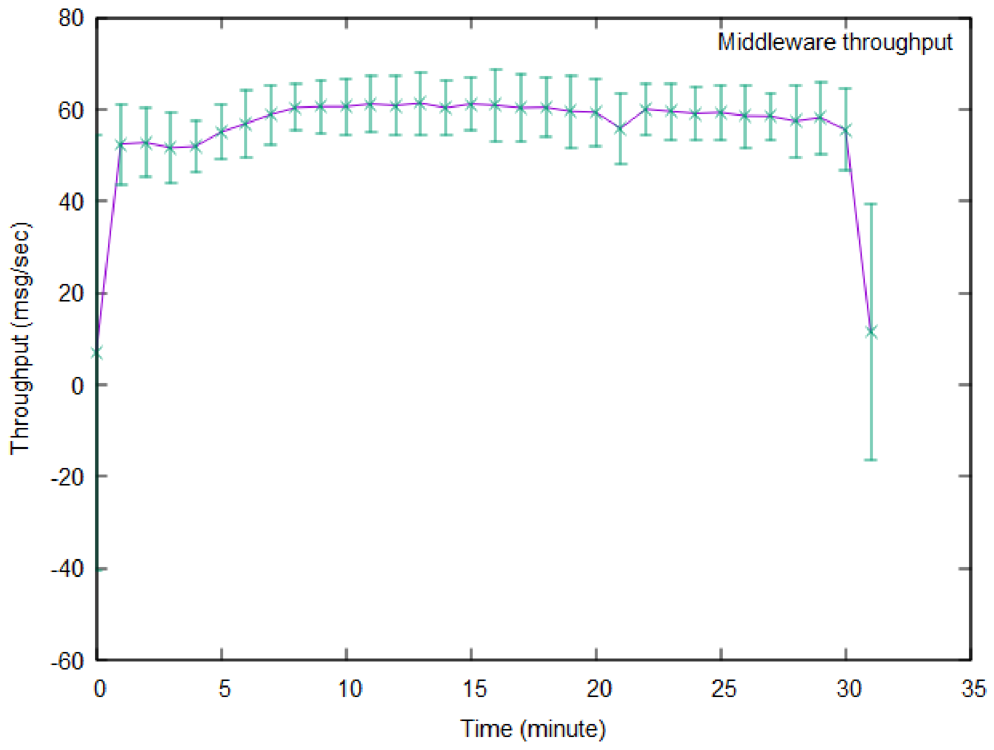


Figure 2: middleware throughput

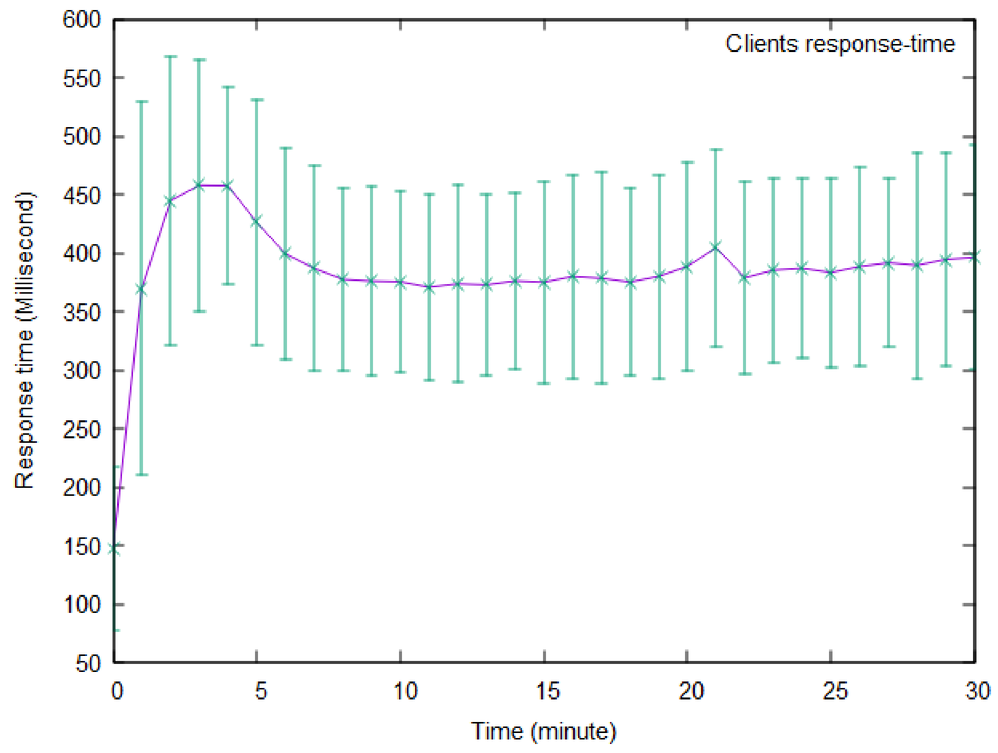


Figure 3: response time for clients

If we draw the th graph with %95 confidence interval, it would be like this:

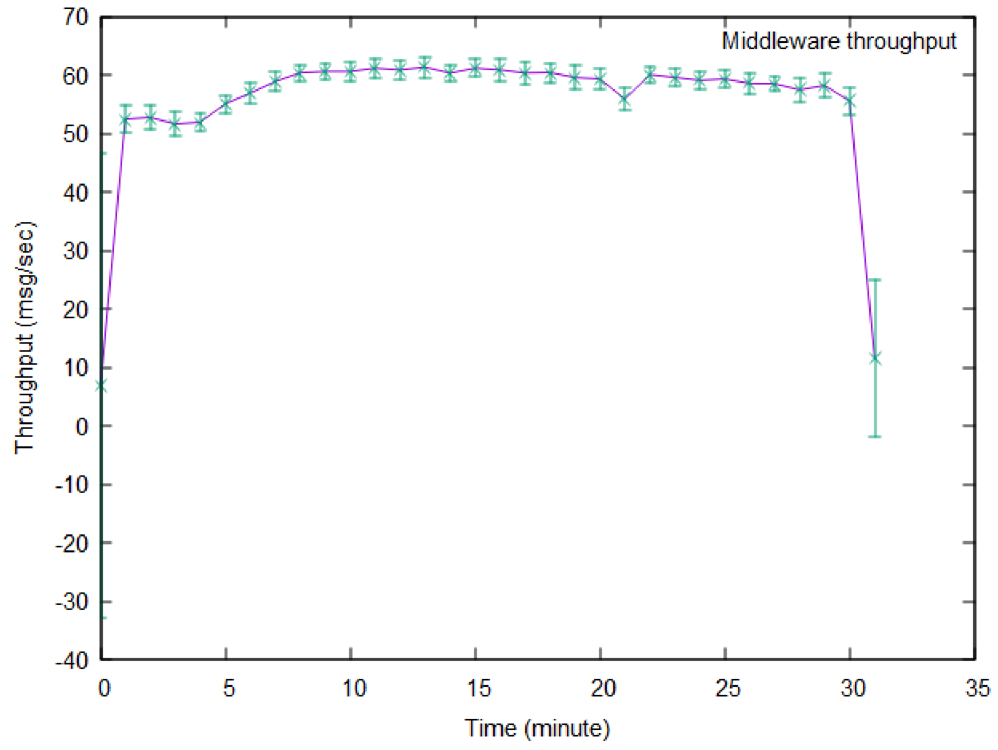


Figure 4: throughput with %95 confidence inteval

6. Systems performance

All the experiments that have done for this section are documented as log files in the log folder of project. Experiments for evaluating different message size and client number is done with one client machine, one middleware and one database.
Some of the experiments were running for 5 minutes and some others have done for 10 minutes because Amazon was noisy during that time of day. Most of them are repeated at least two times and averaged and also stored separately as log files. The response time is calculated based on the time a client issue a send/get/read command and the time it receives the response from middleware. Throughput is calculated as the number of send/get/rea_ message commands that is successfully handled in the middleware in each second.

- Maximum throughput (describe the exact configuration of the system)

After analyzing the time a message spend in different components of the system, I realized that middleware queue in the bottleneck of my system. So, by increasing the workers I tried to shorten this queue and achieve maximum throughout. So, this is related to my speed up experiment which is described at end of this section. There are 80 clients running on one machine, one middleware node with 8 worker threads and one database with 600,000 messages already in it. So, with this configuration I achieve max throughput of 203 messages per second. Response time for this throughput is 202 milliseconds. If we add more clients and also add some workers we get a better throughput. I fixed worker number and increased clients' number until I reached the maximum throughput of 270 messages per second. And response time is 380 for this latter test.

- Scalability of the whole system (explain configurations used to explore the scalability)

I tested both scale out and scale up. All test ran for 10 minutes. The table below, shows the result of scale up experiment where I still have one middleware machine, one client machine and one database. So, I try to see how much we can increase the clients by adding more workers.

Configuration /Metric	4 workers 80 clients	16 workers 320 clients	64 workers 1280 clients
Response time (ms)	256	932	11920
Throughput (Message per sec)	190	209	50

As we can see in this table, any increase of workers after 16 is not enough for handling this many of clients. Also, we see that from 4 workers to 16 workers, we have only a small throughput improvement but response time increased a lot.

For scale out test, I added more server and clients machines to system and measured metrics. Number of middleware workers is fixed to 4. Every 80 clients run on a different machine. So, in last test, there are 4 middleware machines and 4 client machines. For calculating overall throughput, I sum up throughputs of all middleware nodes.

Configuration /Metric	1 middleware 80 clients	2 middleware 160 clients	4 middleware 320 clients
Response time (ms)	256	595	310
Throughput (Message per sec)	190	166	600

Based on this results, I can see that system is scaling well for 4 nodes but not for 2 nodes. For 4 servers throughput should be 750 but now 600 is close enough. For 2 servers, it should be around 380 but now it is too far. I guess it was a noise in the Amazon load and by repeating the test we get the result according to expectation.

- **Response time for messages of different sizes**

This experiment is done 10 minutes several times with 80 clients running on one machine, 4 workers running on one middleware and a database with already 600,000 message in it.

Metric/Message size	**200**	**2000**
Response time (ms)	255	300
Throughput (Message per sec)	194	165

As we can see, throughput decreases with bigger message sizes and response time increases. We expected to see this because there is more data to pass by network and it takes more memory in middleware and slows everything down. Note: this is the only test that we have messages of size 2000. In all other experiments message size is 200.

- Response time for different number of clients

This experiment is done 5 minutes several times with clients running on one machine, 4 workers running on one middleware and a database with already 600,000 message in it.

Client number\ Metric	Response time (ms)	Throughput (Message per sec)
40	227	107
80	256	190
100	290	210
200	380	270
500	2620	100
1000	10000	47
2000	27800	25

As we can see, as we increase the clients, more load is put in the system and system can easily still handle this load. So, even though the response time slightly increases due to waiting in client queue but throughput increases as well. This means the system is not saturated yet and we can put more load. But one we get to 500 clients, the throughput drops and response time increases drastically. This means we are over saturated the system now. Any client increase after this (ex. 1000 and 2000) is just pushing the system down the hill and throughput will drop to a very small number eventually. This is because the context switch between clients is so much that none of them get any useful time for sending or receiving messages. This means the actual load on the middleware has decreased. Thus, throughput decreases. Also, in the middleware they have to wait for a long time in the queue until a worker pick them and handler their task. So, it means larger response time.

- Response time for different numbers of middleware workers
This is speed up experiment, which I keep the same load but increase the number of workers to see how metric change. The load in generated by 80 clients on a single machine. Each experiment ran for 10 minutes two times.

Metric / worker number	Response time (ms)	Throughput (Message per sec)
2	270	156
4	256	190
8	202	203
16	203	204
64	204	200

As we can see, the more we increase the worker number, throughput increases and response time increases but this behavior is not linear, because after 8 workers we don't see any improvement is the system performance. This is because context switch between threads is a big overhead and almost all the clients already have short waiting time in the queue and adding new workers does not make this time shorter due to overhead of scheduling worker threads. I did experiment for 128 and 1028 workers as well and I was expecting the throughput really suffer from that, but I guess Amazon changed my VPS location and the whole system improved; so the numbers from those experiment cannot be compared to this numbes.

7. Analysis of the performance numbers

- Summary of the behavior of the system in terms of the overall design
So, my system design can handle a lot clients even though the response time may be low, but it will not break down easily. Also, the database tier and connection to that tier is very efficient. So, the system can scale for larger clients by adding new middleware nodes assuming that we have high enough worker numbers per middleware instnace.

- What would you do differently if you would have to design the system anew?
I didn't know about prepareCall method for prepared queries. So, I used prepareStatement but I would use prepareCall for the new system.

8. Answer to Some specific questions

o How long does it take to send a message?

It took around 310 ms to send a message during my stability test (with overall response time of 400 ms) but it varies on different time of day because the Amazon AWS is a share VPS and during off-hours you get better results than. But always it is less than receiving message. All the response time for different commands are logged for clients and middleware. So, one should average it over all the time but I didn't do that yet.

o How long does it take to receive a message?

It took about 400ms to get a message (include deleting it from queue). Again it depends on time of day.

o How much time is spent for each of those operations in each part of the system? Consider the client, the messaging component, the database, and the network links between them.

Since I log at different critical end point of message in my three tier system, it is easy to divide response time into different component. Also, I looked at postgres log to determine the actual execution time of send/get queries (which has to be combined with time of deletion). After subtracting tit from DB response time which is measured in middleware, we can find time spent in network link between DB and middleware.

For sending a message, it takes about 2ms in client, 3ms in middleware processing, 3ms in database, 2ms for network time between DB and middleware. The rest of response time () is for waiting in the clients queue and also network link between clients and middleware (These two times cannot be measured individually in my design since we don't know where message reach middleware until we pick the client from queue.)

For getting message (including deletion), it takes 40ms in client, 2ms in database, 10ms in database, 2ms for network time between DB and middleware, The rest of response time () is for waiting in the clients queue and also network link between clients and middleware.

o How does the messaging system behave as a function of the number of connections to the database?

The range which I can play with this parameter is not that much, because every one of my middleware works need 3 dedicated connections for prepared queries and it only needs one shared connection for all other queries. Therefore, valid range is [3*worker_count , 4*worker_count]. Any value above this range does not affect performance since those are extra connections which will be unused. Values below than this range are not possible since worker wont' work properly. And the more we get close to upper bound of this range, the better performance we get, since it decreases the waiting time for db connection in workers.

o Explore the scalability of the system as more instances of the messaging system are added. Can you find the limit? What is the bottleneck?

• Characterizing the code

<u>o What are the bottlenecks in your code?</u>

The queueing part is the most expensive part of code and the bottleneck; because it has to lock the queue for getting or putting the clients. So, clients have to wait a little bit anyways. All other parts of the code take less than 10 ms combined and DB response take up to 10 ms.

<u>o What are the data structures that play the most critical role in the behavior of the system?</u>

At first, I was using ArrayList for keeping the queue of client sockets. In order to maintain this list consistent across all worker threads, I had to put every part of code in middleware that access this list in the synchronized clause. This locking process in those parts of code was a huge bottleneck for my worker threads such that it was taking %80 of total response time by itself. So, I searched for alternatives and found BlockingQueue data structure in java. After replacing that in my code, it runs smoothly and without those long blockings. Also, I use HashMap for keeping messages that come as String in the socket. Then I construct the message object out of that hash-map. The fast key-value look up operation in the HashMap improves the speed of clients when they receive message. If I wanted to store those values in ArrayList it would be slower. I also use HashMap for storing the parsed queries. This data structure speed up the commandHandler function in middleware.