Advanced Systems Lab (Fall'15) – First Milestone

Name: *Pedro Mendez Montejano*
Legi number: *13-923-115*

Grading

Section	**Points**
1.1	
1.2	
1.3	
2.1	
2.2	
2.3	
3.1	
3.2	
3.3	
3.4	
3.5	
3.6	
Total	

1 System Description

1.1 Database

1.1.1 Schema and Indexes

The database schema was designed accordingly to the systems requirements. In the description of the system three main components can be clearly identified, the users, queues and messages. Taking those elements as base components of the system, the application works with three main tables in the database, each one corresponds to the main components in the system $SeeFigure1$.

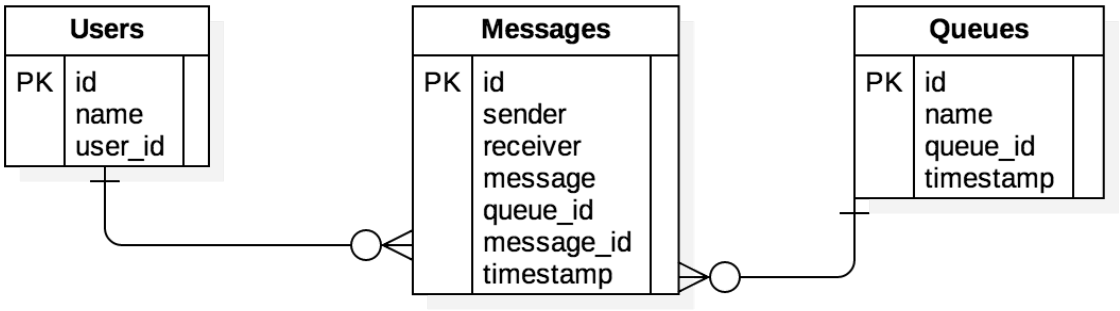


Figure 1: Entity-Relation-Diagram of the database.

The table user contains three columns. The id column is the auto increment value assigned from the database for each new row in the table. The column name contains the name of the users, this value is marked as unique, because each user must, have and a unique user name. Finally the userid column, is a second id that binds a user with the middleware and its assigned by it each time a new user connects to the system.

The table queue contains four columns. As in the previous table it has the id auto increment column. The column queuename, has the name of the queues in the database also marked as unique. The column queueid, also shares an id generated by the middleware, this value is assigned by the middleware when a users requests to create a new queue. Finally the column timestamps as the name implies, are the timestamps of the queue when this one is created.

The table messages, is the biggest but not necessarily complex table. Each column corresponds with each component of the message object. In the requirements is specified that each message must have a sender, receiver, the queue it belongs to, timestamp, the message content and the message id. Therefore, the table has a column for each of these parameters.

The indexes of the database are the following:
In the table users, there is a single index of the users, because that is the attribute used by the stored procedures to a find a particular user.
In the table queues, as in the previous table a single index is used to optimize the search over the id of the object queue.
Finally in the table messages, three indexes were created, one to search over the message id to find a particular message, an index to search over the receiver of a message, a use case of this index is when a user requests a message entitled to him a search over this value must be done. Lastly an index to search over the queid of which that message belongs, an example of the need

of this index is when a user requests to delete a queue and all the messages stored in that queue must be found in order to eliminate them as well.

1.1.2 Stored Procedures

The database messaging contains a total of 14 stored procedures, some created as testing tool and other as part of the functionality of the messaging system. A list of each store procedures in the system and a corresponding description of its function is presented next.

delete_messages_from_queue(queue_id): This functions are executed when user wants to delete a queue. Therefore each message that belongs to that queue is eliminated. This function expects the queue_id value. The value is obtained by the get_queue_id(queue_name) function.

delete_queue(queue_name): This function deletes a queue. However this cannot be done if there exist a message in the messages table that has on its queueid column the value of the id of the queue that is about to be deleted. This protection function is enabled as an initial relation established on the database.

delete_user(user_name): This function is on charge of deleting a user from the table users. Nevertheless, the database also has a protection mechanism regarding the messages on the database and the user id of the user that is about to be deleted. This constrained, stops the execution if there exists a message for this user.

get_message(receiver_name): This stored function retrieves a message for the corresponding receiver. If the message doesnt exists it returns a empty message, with empty values on each column.

get_message_from(receiver_n,sender_n): This function gets a message from the database where the value of the sender and the receiver matches the ones passed as arguments to the function. If the message does not exist it returns an empty message, with empty values on each column.

get_queue_id(queue_name): This function returns the id value of the queue specified in the name parameter. If the queue does not exists it returns empty.

get_user_id(user_name): This function returns the user id of the user been requested. If the user does not exist it returns empty.

insert_new_queue(name,id,timestamp): This function is executed when a user requests the creation of a new queue. The queue name, the id created by the middleware and the timestamps of creation are provided as parameters to the function.

insert_new_user(name,id): This function is executed, when a new user connects to the middleware, in order to add this new user to the database. The name and the middleware id are provided in the function.

insert_new_message(message,sender,receiver,id,timestamps,queue_id): This is the function executed when a user sends a new message to the database. The queue id is obtained using the get_queue_id(queue_name) function and the return value depends on the name of the queue where the user want to allocate the message. The rest of the parameters are the default values of a normal new message.

1.1.3 Design decisions

The main goal during the design of the database was to comply with the system requirements in the system description. Furthermore once the requirements were fulfilled, I focused on a small schema in order to keep the size of the database small.
For the improvement of the operations on the database, the postgres functions (Stored procedures) were implemented. Postgres functions written in PL/pgSQL execute the queries like prepared statements; they reused cache query plans, this make and improvement in the execution time, since they decrease the planning overhead for the execution.[2]

In addition of the postgres functions also indexes were created in the database. Indexes improve the performance of the database specially when small amounts of data are expected to be retrieved. Since indexes on postgres are partially or totally cached accessing data form a cached source is always faster to access then from disk. This improves overall the performance of the database.

1.1.4 Performance characteristics

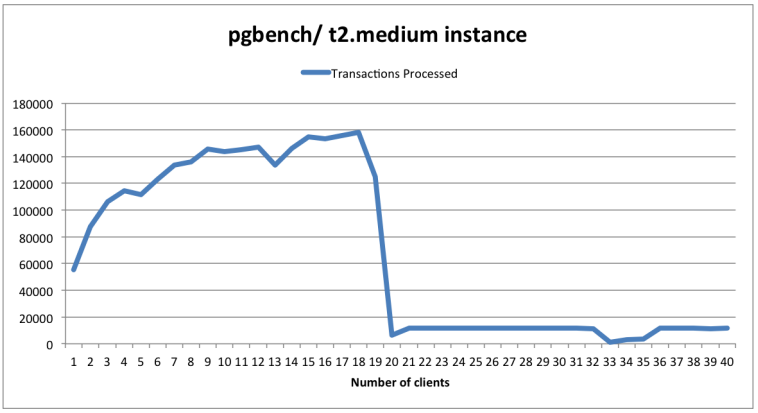


Figure 2: Database performance evaluation in a t2.medium amazon instance, increasing the number of clients in order to observe the maximum amount of transaction it can process.

In order to have a first impression about database performance, and even before setting my own experiments to the messaging system. I decided to run some benchmarking tests in the database system. In addition to get a good analysis of the performance of my database, the network time factor had to be removed. Therefore I decided to execute locally pgbench, which is a benchmarking tool, to find out the performance of my database.

The database server is a t2.medium instance of the amazon EC2, it has 2 vCPUs, 4GiB of Memory and it uses a 20 GB SSD for storage. The postgres database configuration is the following: max. number of connection is 100, it has a shared buffer of 120 MB. This is a basic postgres default configuration.

For benchmarking pgbench has a default size configuration database with three tables, like the messaging systems. The normal size of the database is 15MB. However the size can scale with the scaling factor of pgbench. The number of rows in one of the pgbench tables is 100,000 and it grows with the scaling factor as well.

The benchmarking was performed for a three minutes period with different amount of clients in the database. The scaling factor of the benchmarking was of 100. As it is shown in Figure2 the database achieves it maximal performance with 19 clients achieving a total of 124968 request,

this database was loaded with concurrent request and with a scaling factor of 100. Therefore given load that the Messaging system can generate, its performance should not be affected at least by the database tier. This because this component can handle more load than the generated by the system.

1.2 Middleware

1.2.1 Design overview

Overall the Middleware system has three main components, the Middleware server, the Client Handler and The Database Connector Server *SeeFigure*3. The Middleware server is main thread on the system. It is always listening to new connections from new clients. Before the running of the main listening connection socket the Middleware server sets up the connection with the database.

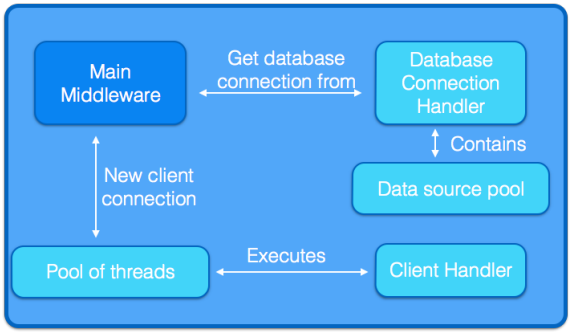


Figure 3: Main middleware component schema.

In order to establish this connection the Middleware needs the following parameters: the address of the database server, the port where the database is listening, the user to connect to the database, the password, the databases name and finally the number of connection to handle. The latest parameter defines the behavior of the queuing of new incoming connections and the pool in the connection to the database; this behavior is explained more in detail in the Queuing and connection pool database section.

Whenever the middleware obtains a new connection from a new client, it pairs this connection with a socket and a connection to the database. The Database connector server gives the connection for the incoming user, after getting the connection a new thread from the Executor service in the Middleware is launched. The new thread takes the socket of the incoming connection and the connection to the database. The initial number of connections to handle limits the number of simultaneous threads in the Middleware.

The behavior of the executor service is explained more in details in further sections. The Client Handler is the component that is instantiated by the new thread in the executor service. Each thread of a Client Handler is in charge of a single user, therefore the system has a total number of handlers equal to the numbers of users been handle by the Middleware.

The Database Connection server: Since the Middleware gets the parameters to enable the connection the database. It passes those parameters to the Database connection server in order to instantiate a new set of connections in a pool for the database.

The Client Handler: This component of the middleware has all the logic to fulfill every request a client can do to the middleware. It uses the socket given by the Middleware to interact with the client and the database connection from the pool to interact with the database server. As result each client can successfully send messages and these will be stored in the database.

1.2.2 Interfacing with clients

The interface between users in the Middleware is the Client Handler. Nevertheless, the Client Handler uses a special object that helps during the interaction with the client. This is the Protocol component; it defines a message object, a queue object, a user object and a protocol number. The value on the protocol number defines the type of operation been requested by the user. The values and their corresponding protocol number are the one in Table1.

Operation #	Desciption
99	Initialize connection with client.
0	Read message from the queue.
1	Read message intended for the user.
2	Read message sent by a user.
3	Create a new message.
4	Create a new queue.
5	Send a message to a particular receiver.
6	Delete queue.
7	End connection

Table 1: List of operations that can be performed by the client and their protocol number

1.2.3 Queuing and Connection pool to database

In the Middleware there are three queuing components: The Pool of executor in charge of executing a Client Handler per connection, the connection pool in charge of managing the connection to the database, and the queuing message system it self *SeeFigure*4.

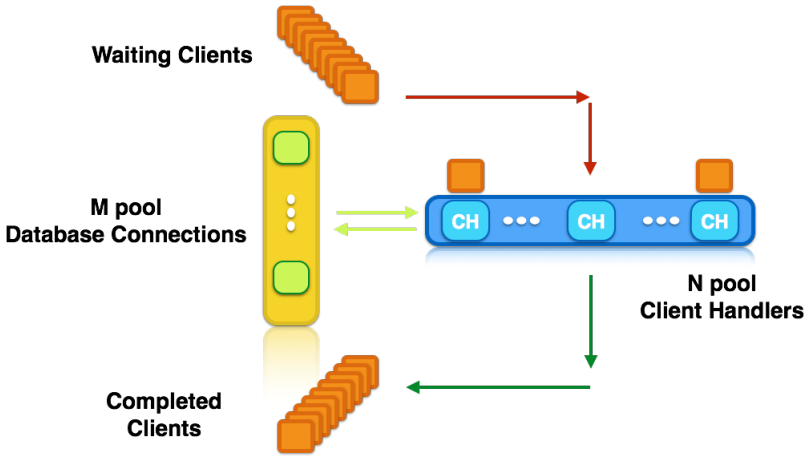


Figure 4: Behavior of the system, M and N is the number of connection in each pool, the may be equal or different. The number of connections in the pool affects the througput in the system.

The Executor service acts as a blocking queue. In this case this is a bounden queue since it has a maximum number of thread in this case is the number of users that it can handle

concurrently. As part of the design of the system I wanted to use this technique in order to prevent resource starvation. When a system uses large queues and a relative small pools then the usage of the CPU and the OS resources, and the overhead due to context-switching decreases. Nevertheless this may also affect the throughput of the system.

The connection pool in charge of the connections has a fixed number of connections ready for the clients. Each connection is assigned to a Client Handler to interact with the request coming from the client. When a Client Handler is done with a client and it calls the closing method for the connection, this one never actually closes, instead it returns to the pool of connection in order to be used by another Client Handler. This improves the performance of the system, since there is no need to setup a new connection for new Client in the system.

1.2.4 Performance characteristics

Overall the evaluated metrics in the system are highly tied to the configuration of the system. A fixed set of parameters is considered a model and by changing a single parameter the systems becomes a completely different model with different throughput and response time.

In this system the metrics like the throughput correspond to the number of requests per second the middleware can handle. The type of request depends on the clients. The response time or latency is the time it takes to fulfill a request generated by the client, depending on where this is measured we can also have latency in the middleware and latency in the database. In order to provide a description of the characteristics of the system, its parameters will be fixed in a specific setting describing normal use case of the system.

The system configuration consist of 5 machines from which 1 is the database machine, 2 are middleware nodes, and 2 are client machines running 30 clients each. The operation been performed by the clients is to send messages in the system. This behavior was decided in order to simulate a heavy operation situation of the system. This will give a clear description of the operation of the system under a stress condition. For this performance analysis the system has a single connection to the database and 30 handlers on each middleware node. This will show how the system react with a single connection been shared among the clients and how the performance affect the measured metrics for a period of 3 minutes.

Further on in the report other experiments where run, changing more values in certain parameters and behavior on the metrics differ.

In summary on middleware 1 the following result are found:
Total number of clients: 30
Average total of waiting time for all clients: 69.3144702642 ms
Average total of waiting time for all Client Handlers: 66.959272843 ms
Average Throughput in the Middleware: 465.727272727 req/s
Total # of requests in Middleware: 128683
Utilization Law X=465.727272727 S=0.00213703441791 U=0.995275211178

On middleware 2 we observe a almost equal behavior and this is because both have the same configuration:
Total number of clients: 30
Average total of waiting time for all clients: 68.8650415678 ms
Average total of waiting time for all Client Handlers: 66.8341319684 ms
Average Throughput in the Middleware: 472.386861314 req/s
Total # of requests in Middleware: 129926

Utilization Law X=472.386861314 S=0.00210889275434 U=0.996213229069

Overall on the system we had 60 clients and by summing both throughputs we a have a total of 938.1 requests per seconds with a latency of 68 ms this is including network delay time. In addition we can observe the system is being busy must of the time given the result on the utilization formula.

In this part for this performance results are merely presented on numbers. However detailed plots can be found in the experiment section. The variations on results are result of the configuration of the main components affecting the system. Those are the queuing in the handlers pool, the connection pool of the database, the load in the system and the database. Nevertheless conclusions about each component are found on each experiment in the experiment section.

1.3 Clients

1.3.1 Design and interface

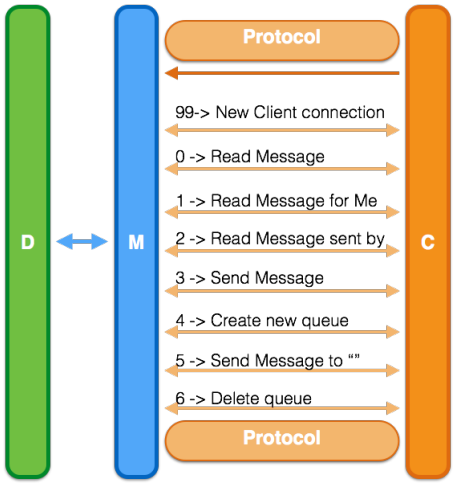


Figure 5: Client, middleware and database interaction procedure. This diagram show the operations that a client can request and the behavior between thr C:Client, M:Middleware and D:Database.

The design of the clients is simple. They accept different running modes, those are defined by the number of parameter they get during the running and deployment. The main parameters for running are the following: server address of the middleware, listening port of the middleware, and the client name. Once it starts running it can accept which operation to perform by console and fulfill the entire fields in the same manner. Nevertheless for experimenting purposes the clients are initialize with more parameter to avoid the interaction with a real user and automatize the operations that it performs. The extra parameter are the following: a running time that defines how long the clients are going to run, the workload under it will send a new request to the middleware, the operation number that is going to perform, this could be any of the ones defined in section 1.2.2.

Furthermore inside each client there is embedded a single message with a length of 200 characters. The size of the message can change with a scaling parameter that is supplied during the launching moment in the client. This changing size feature is in the system since is defined in the system description. Additionally, they count with a list of pre-set user name and queue names from where they can choose randomly during the execution of certain client operations. (e.x. Sending a new message) $SeeFigure5$.

1.3.2 Instrumentation

The instrumentation can be divided as the one added before compiling to the system and external tools used after running of the system. The clients in there source code, instrumentation logging mechanisms where added, in order to record data during running time, like throughput during unit time and records of the operations performed by it.

As external tools of instrumentation I used python script to parse the logs and perform all the measurements of my system. The scripts use the time stamp value of each of the operations logged in the log files. In addition to make an assumption about the time it takes the db to perform an operation a log entry is performed in the middleware level, in order to identify the log of the db a line in the log files has a db tag identifier which the parser is able to detect and use to estimate the database response time.

For details on how my instrumentation tools were used se the Workload and Deployment section.

1.3.3 Workloads and deployment

The workload behavior in the system is self-adjusting, since each client was not having a thinking time for sending a new request. In addition based on the design of system as a closed system, clients were not able to send more requests to the middleware if these ones where not having a response of the previous request. Therefor the load was based in the number of concurrent client being attended or served by the middleware. In addition the time between request was a consequence of the type of operation been requested, plus the time of processing by the middleware, network delay and time of processing in the database.

Exmaple of Deployment and Instrumentation Tools

```
Running The Experiment
# fab -R local fullAmazon1:experimentID=resp_time_1_mid1_cl5_msg1,dbServer=52.30.174.216,dbName=messaging,dbUser=postgres,
dbPassword=squirrel,noOfConnections=15,listeningPort=5432,noConnDB=5,duration=180,serverPort=5433,serverAddress=52.30.110.167,
operationType=5,workload=0,noClients=15,messageType=1

Parsing the Results
# fab -R local parsing:pathOfLogs=/Users/pedrini/Documents/workspace/Middleware/script/logs_exp_resp_time_1_mid1_cl5_msg1/
Output Example
Total number of clients: 15

Average total of waiting time for all clients 36.2926324829
Standard Deviation total of waiting time for all clients 0.44978481201
Standard Error total of waiting time for all clients 0.120210047451

Average total of waiting time for all Client Hanlders 34.5220723635
Standard Deviation total of waiting time for all Client Hanlders 0.48504220643
Standard Error total of waiting time for all Client Hanlders 0.129632982471

Average Throughput in the Middleware: 438.239285714
Standard Deviation of Throughput in the Middleware: 44.6265843199
Standard Error of Throughput in the Middleware: 2.6717237184
Total # of requests in Middleware: 123226

Utilization Law X=438.239285714 S=0.00227224773993 U=0.995788226511
```

Figure 6: Full run of experiment and parsing command

To deploy the system parameters as number of clients, operation performed by the clients, number of connections and size of messages were specified in the deployment scripts See Figure 6. These scripts were described previously and were part of python instrumentation scripts used to parse and perform operations with the log resources.

1.3.4 Sanity checks

The operation of sending new messages to the system is part of the sanity check of the system. For most of the experiments I started with and empty database. Therefore at the end of each experiment I was able to observe how many transaction my middleware was reporting. Once I got the number I performed a cross validation with the total number of live tuples in my database.

In addition giving the logs of the clients and the number of transactions they were reporting, the sum of the total of request of all the clients had to match the ones in the middleware. The python script performed these validations. For checks regarding sensible operations in my system, like I/O error or connection errors the system has pertinent Exception handlers embedded. In addition if an error occur during the execution the system has a logger to register the error.

Furthermore to control transaction among client and middleware I implemented another sanity check tool which validates that each answer coming from the middleware correspond to the one requested by the client. If this not the case then the system trigger and alert on the logs.

Finally the construction is modular each critical component is well separated which allows a quick response time in case of a component fails.

2 Experimental Setup

2.1 System Configurations

Overall the system has a three tier configuration, as it was described in the requirements. The tiers are the database machine, two middleware machines and two client machines. All the machines are t2.medium Amazon EC2 instances. In order to build the Middleware and Client applications Ant was used.

2.2 Configuration and Deployment mechanisms

For configuration of each client as mention in 2.1 the Middleware and the Client were built using Ant. In order to deploy the system including the Database and applications in each tier Python scripts were used. A main script with different function was written in order to perform different operations within the system.

Moreover, fabric a python library that works as command line tool and ssh application was used as part of the deployment process see Figure 6. In addition as part of the experiments performed in the system, when the operation that the client is not mentioned is implied that it was sending new messages to queues. This was decided as part of the performance analysis in the system in order to observer the effects of heavy task.

2.3 Logging and Benchmarking mechanisms

For the logging mechanism the library lo4j was used. The configuration files is passed as parameter to the Clients and the Middleware application. Both components log the type of request they are preforming, with a timestamp and the number of the request.

The client just before sending the request makes a log entrance and once it get the response enters a second log. In the middleware a log entrance is made when the Middleware gets the request from the client, then when it sends a request to the database and when the request returns

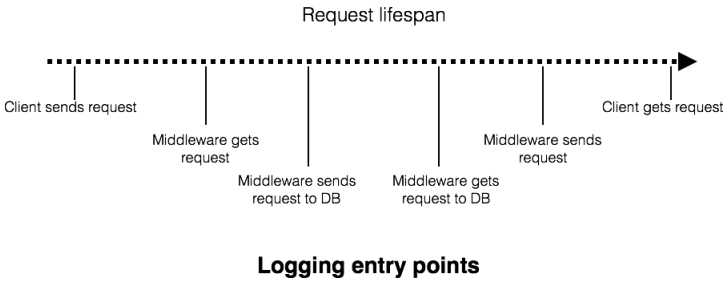


Figure 7: Different logging entry points during the lifespan of a single request in the messaging system

from it, this one is to estimate the response time of the database from the middleware perspective. Finally after sending the response to the client it enters another log entrance. See Figure 7.

Finally for benchmarking a python script was written in order to parse the results of the logs provided by the middleware and the clients. In addition the middleware has embedded another logging functions to log the throughput of the middleware every second during the execution of it. With those numbers in the log files and the parsing methods, a performance analysis of the system was made.

3 Evaluation

3.1 System Stability

In order to perform a stability test the configuration of the system was the following, the database was already initialize it had 14750 messages stored; it had a size of 324MB. Two middleware nodes, one on each t2.medium instance, 15 clients connected to each middleware node, each group of 15 client were running in a t2.medium instance as well, which makes a total of 30 clients sending and receiving data. The period of running of the experiment was for 30 minutes and with 5 repetitions in total.

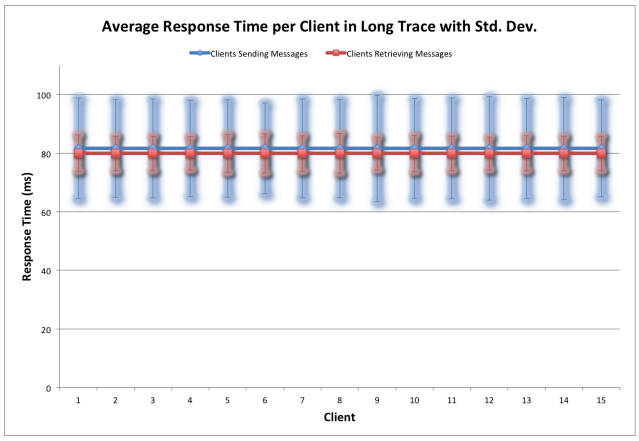


Figure 8: Plot with response time in different clients perfmorming distinct operations in the middleware

From the result plotted in Figures 8 and 9, it can by observed in the beginning a stable throughput is observed in the system. In addition we can see the decrease of the throughput at the end in the cool down face. Furthermore there are some sparse down spikes in the throughput,

which I point to the effects of the java garbage collector and the I/O activity in the middleware caused by the writing of the log files.

In addition to the analysis of the throughput we can observe the difference in response time in

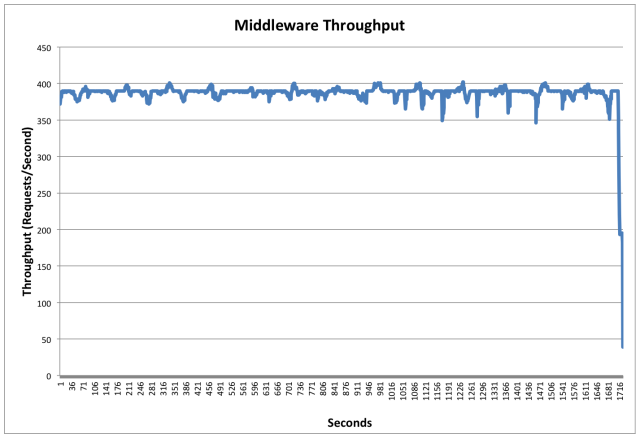


Figure 9: Throughput during stability run over 30 minutes

the two different operations being performed by the clients. Clearly the system take more time in create a new message and save it on the system than in retrieving a single message. This makes sense since when a message is about to be created it check if the queue where is going to be stored exist or not and if it does not it creates that queue. This operation incurs in more time of processing then the retrieving a single message operation.

3.2 System Throughput

For the Throughput analysis five machines were used 1 database, 2 middleware and 2 client machines. Each client machine was running several client instances sending new messages to the system. With a fixed number of Client Handlers and connections to the database. Nevertheless in order to find the maximum throughput, the load in the system was increasing by the increment of running client on each client machine. The machine were t2.medium instances.

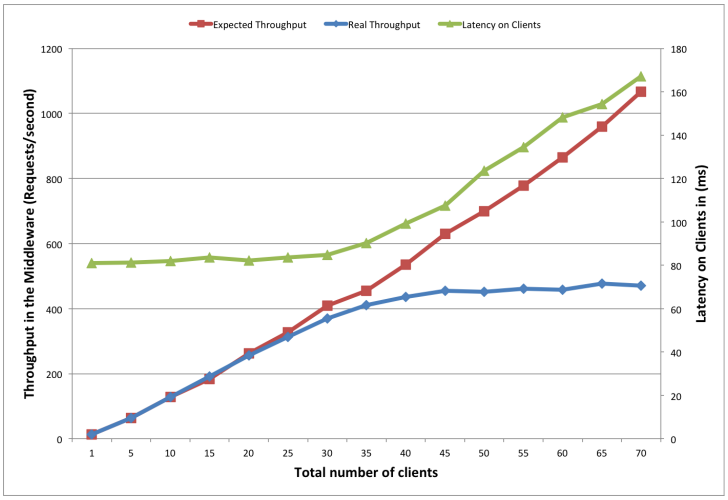


Figure 10: Number of clients vs Throughput and Latency, maximum througput level achieved with incremental on response time.

Assuming enough resources on my system I performed experiments with increments of five clients more on each middleware on each repetition. My hypothesis for this experiment was to

observe a linear increase in the throughput and to observe a constant response time.

Nevertheless, given the results obtained and showed in Figure 10 after preforming constant increases for periods of ten minutes I was able to find the maximum throughput on my system. In addition I was able to observe the effect that the increment of load in the system affects the response time. In the figure I plotted what I expected to be the increment in the throughput. However when the middleware is handling more than forty clients the throughput stops increasing and remain more less constant. Nevertheless the Latency or average response time that the clients are having just continues increasing.

Even when the Middleware is configured to handle more clients with an initial configuration running a hundred Client Hanlders to attend the requests from clients, the response time increases because of the number of connections to the database, which in this experiment is fixed to 10, are not enough to satisfy the amount of operations the clients are requesting from the middleware. Since after each database operation a connection is being taken from the pool and when all the connections are occupied the client requests must wait until one connection gets free. This as consequence increases the response time in the clients.

Is important to mention that the data and plot reflect only one of the middleware and the response time of a single machine running clients. Nevertheless given the fact that in both machines the clients are preforming the same operation and both middleware have the exact same configuration the same analysis and results apply for the other components.

3.3 System Scalability

For the scalability tests I performed a scale-up and a scale out experiments. My hypothesis for these experiments was to expect the same behavior for both and observe a constant function measured metric response time or latency, in my clients.

In the scale up setting the configuration of my system was a single machine running client, one middleware machine with a middleware node and my database machine. The parameters I fixed in this experiment were the number clients with 15 clients, 5 Client Handlers in the middleware and 5 connections to the database in the middleware. The experiment was running for 10 minutes, and each client was running for 3 minutes and all the clients where sending new messages of 200 character to the system.

For the scaling setting I was increasing by doubling value of the parameters in my system. Therefore as part of my hypothesis I was expecting to see a double increment of my throughput in the system. The result of the scale up can be observer in the Table 2.

Setting /*Metric*	5 Client Handlers 15 Clients	10 Client Handlers 30 Clients	20 Client Handler 60 Clients
Latency (ms)	80.22654714	80.25648836	81.48056189
Throuhput (Req/sec)	194.4534884	388.494186	766.6140351

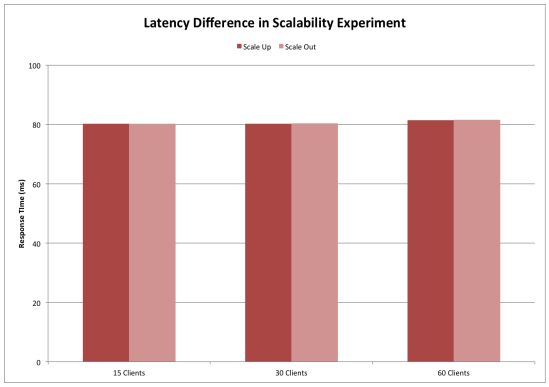
Table 2: Scalep up setting with single machine, increasing constanlty connections and client handlers

In the scaling out setting I wanted to achieve the same numbers with double incremental of my parameters. Therefore I used the same configuration for my 1 machine running clients, 1 middleware and 2 database running 15 clients, with 5 Client Handlers and 5 connections to the

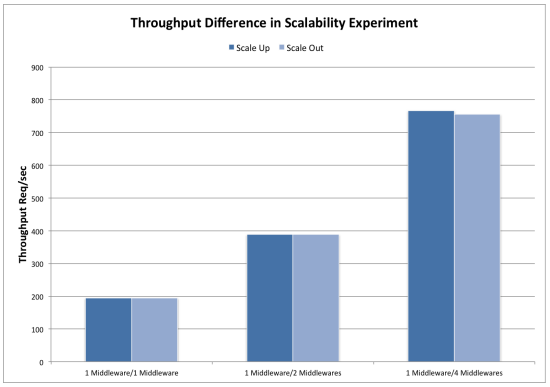
database and perform double value incremental having setting with 2 machines running clients, 2 middleware servers, then a final setting with 4 client machines, and 4 middleware server. Each single machine replicated was running with the same fixed values for the parameters, which were mentioned previously. The result of the scale out setting can be observer in the Table 3.

Setting /*Metric*	1 Middleware 15 Clients	2 Middlewares 30 Clients	4 Middlewares 60 Clients
Latency (ms)	80.22654714	80.33796634	81.48632835
Throuhput (Req/sec)	194.4534884	388.3139535	755.4543636

Table 3: Scaling out setting with fixed paramters in connections and client handlers but constant incremental of middleware nodes and client machines



(a) Latency comparison between scaling experiments increasing number of Handlers and connections to the database in the Middleware



(b) Throughput comparison between scaling experiments increasing number of Handlers and connections to the database in the Middleware

Figure 11: Results in scalability experiment

As I expect on my hypothesis my system on the scaling out setting was behaving like the scale up experiment *SeeFigure*11. Nevertheless giving these scenarios, I consider scaling up factor is more efficient giving that the allocation of new requests to free connections is done by a single middleware which can be more efficient. However this may not be the case always since this depends in how the system is configured.

3.4 Response Time Variations

In this experiment I had a setting of five machines as well, 2 middleware nodes, 2 client machines and 1 database, all t2.medium instances.

The fixed parameters in my experiment were the number of clients on each machine with a total of 15 on each. Each client was sending messages to the system. The number of Client Handlers were 15, one for each client and 5 connections to the database. Therefore these connections were shared among the clients. The duration of this experiment was of 5 minutes with many repetitions. In addition before every run the database started empty.

The parameter varying in this experiment was the length of the message been sent to the database. The reason for changing the message size of was to observe the impact of it in the response time on the clients.

My hypothesis for this experiment was to observer an increase in the response time metric in the system. The raise on it I expected to be as consequence of the increment in the message size.

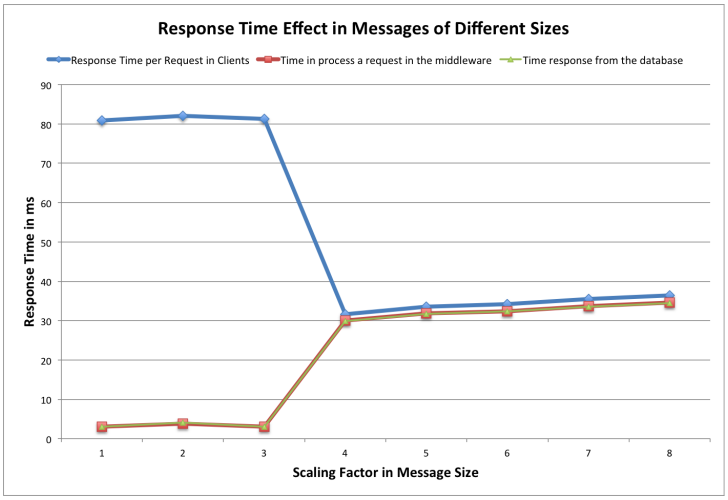


Figure 12: Response time for clients during incremental in the message size

Nevertheless, after running the experiment and plot the response latency of the clients and the response time that it took for the middleware to process a request, the first results were not the ones I expected in my hypothesis. The clients were observing a high latency when the middleware time of processing a request was really low. However, after the size of the message was scaled 3 times, 600 characters, the response time in both client and middleware was behaving as expected. See Figure 12.

The high latency behavior of the clients observed even when the middleware is processing request fast I assume is because of what I observed is thread local allocation buffers, what is common for multi thread applications is to assign to each thread a buffer that is used only by that thread to perform allocations. When the size of the message is not big enough then the thread may assign a direct allocation to the heap requiring more synchronization. Once the size of the message is big enough then the threads optimize the use of the thread local allocation buffers (TLABs) with non-direct heap allocations. Improving the response time. Nevertheless in order to be 100% certain more experiments need to be done modifying the running parameter of the Java Virtual Machine (JVM) in this case the size of the TLBA [1].

3.5 2^k Experiment

For the 2ˆk experiment the two factors I decided to explore are the number of concurrent client in my middleware (Handlers) and my number of connections in the database, the reason was because as I observed in my through put experiment by increasing the number of clients in the system in the handler parameter the throughput was increasing. Furthermore I wanted to see the impact on the number of connection has on the system.

Both parameters were explored in two levels each with the values of 10 and 20. As result I had to preform the experiment 4 times with adjusting the values of the previous parameters. Each experiment was running for 10 minutes, with t2.medium instances each client with and average workload of 12 requests per second. By performing this experiment we are able to see the impact of these factors in the performance of the system in Request completed by second (Req/sec).

After the data seen in Table 4, I define my two variables x_a and x_b

	Connection to Database	
ClientHanlder	10	20
10	129.354782609	128.763478261
20	257.597560976	258.93554007

Table 4: Throughput metric results for different settings of the 2^k experiment

$$x_a = \begin{cases} -1 & if \quad 10 \quad Connections \quad to \quad Dabatase \\ 1 & if \quad 20 \quad Connections \quad to \quad Dabatase \end{cases}$$

$$x_b = \begin{cases} -1 & if \quad 10 \quad Client \quad Hanlders \\ 1 & if \quad 20 \quad Client \quad Hanlders \end{cases}$$

After the analysis of my results my hypothesis about the number of Client Handler impacting in high way the performance of my system is true. Furthermore I was able to show that given the current configuration of my database and the number of connections it has is more than enough for my system.

Furthermore my 2^k experiment show that the mean request per second is 193.6628405, in addition the impact of the connection in request per second of my database connections is very low 0.186668686 requests per second. However the value in my number of Client Handlers accounts for 64.60371004 Requests per second. Finally the interaction of these two combined have an effect of 0.482320861 Request per Second.See Table 5.

Design of 2^2 Experiment				
I	A	B	AB	y
1	-1	-1	1	129.3547826
1	1	-1	-1	128.7634783
1	-1	1	-1	257.597561
1	1	1	1	258.9355401
774.6513619	0.746674746	258.4148402	1.929283442	Total
193.6628405	0.186668686	64.60371004	0.482320861	Total/4

Table 5: Wieghts on the response variables

$$SSM = 2^2(0.186668686^2 + 64.60371004^2 + 0.482320861^2) = 16695.62732$$

Accountability of components
$A = 0.000834834\%$ $B = 99.99359165\%$ $AB = 0.005573517\%$

In addition to my fist 2ˆk experiment I decided to run a second design for a 2ˆk experiment. This was inspired by the results I obtained in my scalability test. I wanted to know how much the number of clients and the numbers of middleware nodes affect the performance of my system in this case the throughput. Clearly in my scalability experiment in increasing the load the throughput was increasing. However as part of the scalability experiment, the number of Client Handlers where increasing and the number of connections of the database as well.

In this experiment overall this previous parameters will remain fixed and the only changing parameter changing in a two level scale will be the number of client and the number of Middleware nodes. The experiment configuration is the following:

All machines are t2.medium instances, the time of the experiment is 10 minutes, with fixed values overall configuration of 20 client handlers, 10 database connections, with a message of 1600 character, all clients are running in sending mode which is sending new messages to the database into the queues.

The 2^k parameters are the number of clients with 20, and 40 and the number of middleware nodes with 1 and 2. My hypothesis for this experiment is to observer a constant performance in throughput on both settings. In addition I expect that the parameter that account more for the throughput will be the number of clients like I in the maximum throughput experiments. This as long as there is enough Client Handlers in the system to attend the clients.

	Middleware Nodes	
Clients	1	2
20	455.1	814.6061498
40	928.9178571	1735.999362

Table 6: Throughput metric results for different settings of the 2^k experiment with clients and middlewares

$$x_a = \begin{cases} -1 & if \quad 20 \quad Clients \\ 1 & if \quad 40 \quad Clients \end{cases}$$

$$x_b = \begin{cases} -1 & if \quad 1 \quad Middleware \\ 1 & if \quad 2 \quad Middleware \end{cases}$$

Design of 2^2 Experiment				
I	A	B	AB	y
1	-1	-1	1	455.1
1	1	-1	-1	814.6061498
1	-1	1	-1	928.9178571
1	1	1	1	1735.999362
3934.623368	1166.587654	1395.211069	447.5753546	Total
983.6558421	291.6469135	348.8027672	111.8938386	Total/4

Table 7: Wieghts on the response variables

$$SSM = 2^2(85057.92218^2 + 121663.3704^2 + 12520.23113^2) = 876966.0949$$

Accountability of components
$A = 38.79644729\%$ $B = 55.49285024\%$ $AB = 5.710702477\%$

After performing the experiment I find out that my hypothesis was not true, even in the setting where both configurations had the same amount of clients, the setting with 2 middleware nodes had a higher throughput. As conclusion for this experiment this behavior was due to the constant values in the client handler and connections to the database. In the scalability experiment that throughput was the same in both single middleware and multiple one because the previous mentioned parameter where also increasing as part of the scalability test and in this one they remain constant. Finally in this setting the Middleware component has a higher impact in the throughput of the system.

3.6 Conclusion

The messaging system has many components that clearly affects it performance. Furthermore different results could be observed given a different fixed parameters. On my system given a robust database, the middleware system was not enough to stress out the performance of it. The main component affecting the throughput of my system was the number of Connection Handlers. This was observed in the maximum throughput experiment and the 2ˆk analysis also supported that assumption. As result my bottleneck was in the number of handlers. However not necessary increasing this number will always improve the performance. This is because the machine has fixed amount of resources and by increasing the number of handlers also increases the switching on memory assigned asnd more parameters that may affect the system.

The workload on my system was proportional to the number of clients been handled by the system simultaneously. However the number of clients was also limited by the memory assigned for thread in the machine. In my case with a number above of 66 clients the memory was already consumed. This could be fixed by increasing the memory assigned for thread in the machine. In the system my arrival rate and my completion rate were equal since this is a closed system and I couldnt have a new request before completing the previous one.

Finally, If could design my system anew the component I will change the normal network communication instead of using the normal sockets and the Object Data streamers I would use the non-blocking and buffered oriented sockets to improve the performance and avoid for example the behavior observed in the response time experiment with small size messages. In addition I would use bigger and more powerful machines in order to generate enough load such that the database performance could be affected in a more significant way.

References

[1] Oracle. A little thread privacy please.

[2] Postgres. Postgresql sql-prepared statements.