Advanced Systems Lab (Fall'15) – First Milestone

Name: *Your name*
Legi number: *Your legi number*

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 systems 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 messaging database, each one corresponds to the main components of the system $SeeFigure1$.

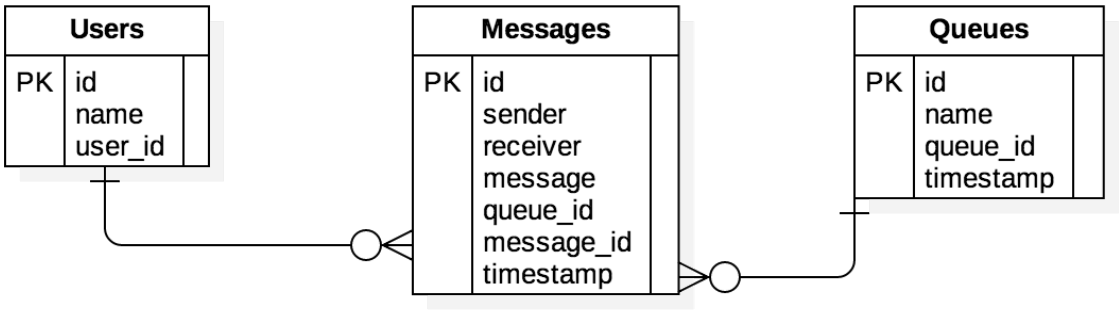

Figure 1: Entity-Relation-Diagram of the database.

The table user contains three columns. The id column is the auto increment value assign 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, this one is assigned by the middleware 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 timestamp as the name implies, is the timestamp of 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 use in the stores procedure to a find a particular user.
In the table queues, as in the previous table a single index is also 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, when a user request a message entitled to him a search over this value must be done. Lastly and index to search

over the queid of which that message belong, an example of the need of this index is when the user request to delete a queue and all the messages stored in that queue must be found in order to eliminate them.

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 procedure that is used as part of the messaging system, and a corresponding description of its function is presented next.
delete_messages_from_queue(queue_id): This functions is executed when user requested 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 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 store functions 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 functions gets a message from the database where the value of the sender and the receiver matches the ones passes as arguments to the function. If the message doesnt exists it returns a 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 doesnt exists it returns empty. get_user_id(user_name): This functions returns the user id of the user been requested. If the user doesnt 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 timestamp of creation is provided as parameter to the functions.

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

insert_new_message(message,sender,receiver,id,timestamp,queue_id): Tis is the function executed when a users send a new message to the database. The queue id is obtain 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 of 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 procedure) 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.[1]

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 tattly cached accessing data form a cached is always faster then form disk. This improves overall the performance of the database.

1.1.4 Performance characteristics

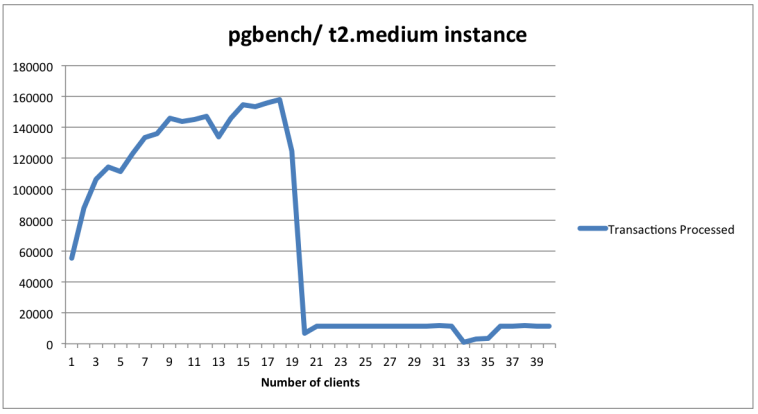

Figure 2: Database performance evaluation in a t2.medium amazon instance, increasing the number of client in order to observe the maximum amount of transaction it can process.

Even before setting my own experiments to the messaging system. I decided to run some benchmarking tests to the database system. In order to have a first impression about its performance. In order 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 be affected at least by the database tier.

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. Its always listening to new connections from new clients. Before the running of the main listening connections socket the Middleware server sets up the connection with the database.

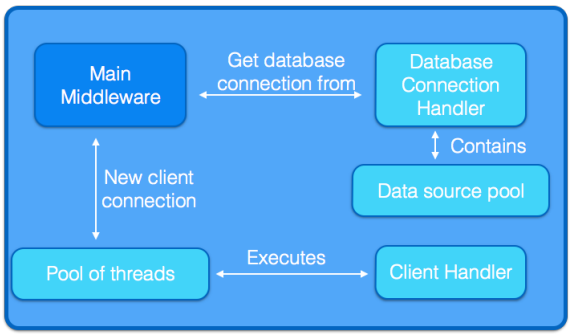

Figure 3: Main middleware component schema.

In order to establish this connection the Middleware needs the following parameter: 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 get 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

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

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 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 following:

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.

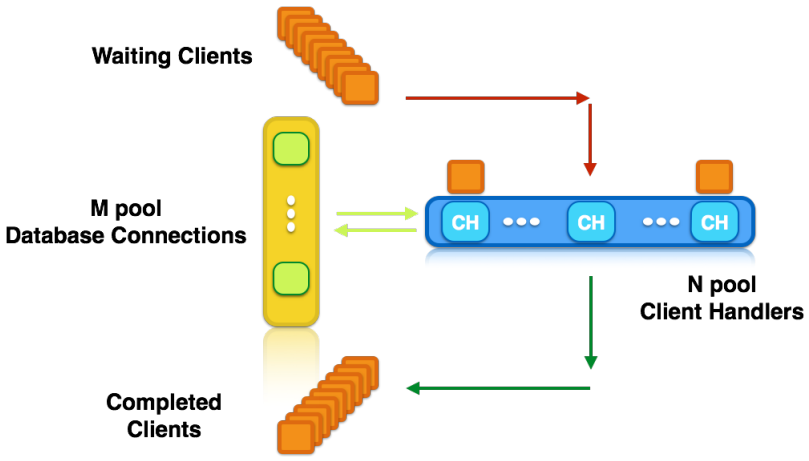

Figure 4: Behavior of the system, M and N is the number of connection in each pool, the may be equal or differetn. The number of connections in the pool may affect the througput in the system.

The Executor service acts as a blocking queue. In this case this is bounden queue since it has a maximum number of thread in this case users that it can handle. As part of the design of the system I wanted to use this technique in order to prevent resource starvation. When a system uses of 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

1.3 Clients

1.3.1 Design and interface

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. The main parameters for running are the following: server address of the middleware, listening port of the middleware, and the client name. Once they start running they could accept the can design 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 one defined in section 1.2.2.

Furthermore inside each client there is embedded two message types one with a length of 200 and 2000 characters, this as 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) *SeeFigure*5.

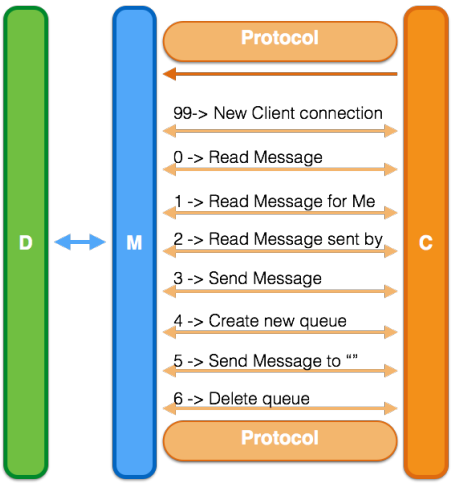

Figure 5: Client, middleware and database interaction procesdure. This diagram show the operations that a client can requesta and the behavior between thr C:Client, M:Middleware and D:Database.

1.3.2 Instrumentation

1.3.3 Workloads and deployment

1.3.4 Sanity checks

2 Experimental Setup

2.1 System Configurations

. Overall there are three tier components in the system, 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. In addition as part of the experiments performed in the system, when the operation that the client is not mentions is implied that it was sending new messages to queues. This was decided as part of the performance analysis while executing a heavy resource operation task.

2.3 Logging and Benchmarking mechanisms

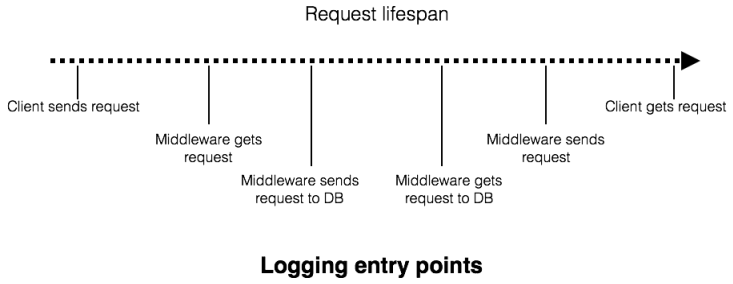

Figure 6: Different logging entry point during the lifespan of a single request in the messaging system

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 make 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 return 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 6.

Finally for benchmarking a python method is the tool is script was written in order to parse the result of the logs provided 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 method numbers of the performance of the system were gathered.

3 Evaluation

3.1 System Stability

Figure 7

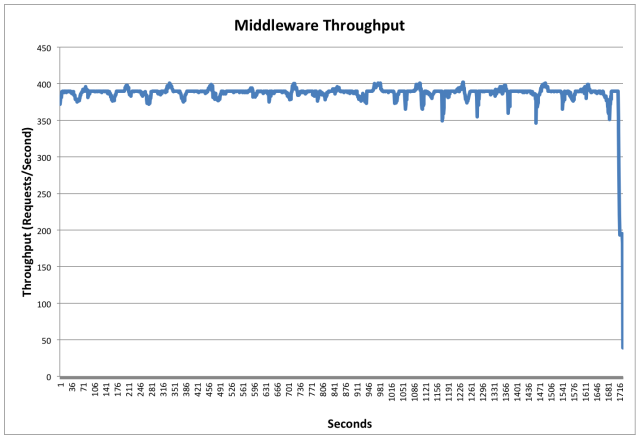

Figure 8

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 node, 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 10 minutes and with 5 repetitions in total.

From the result plotted in Figures 9 and 8, we can observe since the beginning a stable running since the warm-up face in the throughput of the system. In addition we can se 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 effect of the java 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 the two different operations being performed by the clients. Clearly the system take more time in create a new message and store 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 doesnt it creates that queue. This operation incurs in more time of processing then retrieving a single message.

3.2 System Throughput

For the Throughput analysis the configuration of the system was the following the same type and number of machines described in the system configuration are used. In order to find the maximum throughput increased the load in the system and the number of machines simultaneously.

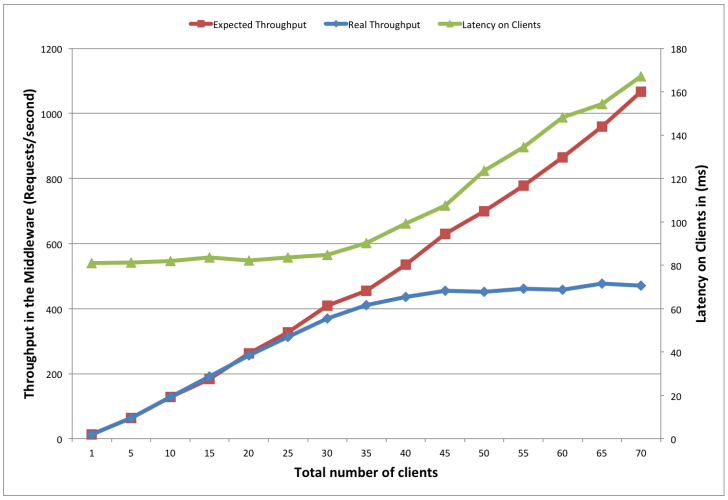

Figure 9

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 showed in Figure 9 obtained 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

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 1

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 2

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 the double the 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 1.

In the scaling out setting I wanted to achieve the same number 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 value for the parameters, which were the mention previously. The result o the scale out setting can be observer in the Table 2.

As I expect on my hypothesis my system on the scaling out setting was behaving like the scale up experiment. 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.

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

Table 3

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 4

3.4 Response Time Variations

3.5 2^k Experiment

For the 2^k experiment the two factors I decided to explore ere 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 allowed in the system in the handler parameter the throughput was increasing. Furthermore I wanted to see the impact on the number of connection in 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 3, I define my two variables x_a and x_b

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

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

After the analysis of my result 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, also 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 account for 64.60371004 Requests per second. Finally the interaction of these two combined have an effect of 0.482320861 Request per Second.See Table 4.

3.6 Conclusion

References

[1] Postgres. Postgresql sql-prepared statements.