Advanced Systems Lab

Milestone 1 Report

Karolos Antoniadis

This is the report of the first milestone of "Advanced System Lab" project. The report starts with an introduction of the system created for this milestone. Afterwards, in Section 2 we describe the general design of the system, including the database, the middleware and the clients. In Section 3 we describe how we tested the system. We follow with a description of the experimental setup and how the experiments were conducted in Section 4. In Section 5 we continue by describing the experiments that were done and their evaluation. We conclude the report in Section 6.

1 Introduction

Goal of this milestone was to create a message passing system supporting persistent queues and a simple message format. Furthermore to experimentally evaluate it and determine its performance characteristics. The desired message passing system consists of three tiers. The first one implements the persistent queues using a database, which from now on will be referred as the "database tier" or "database" (db). The second tier implements the messaging system and is responsible of all the logic related to system management, also it is the one tier that is using the database in order to implement its functionality. We will refer to this tier as the "middleware tier" or "middleware" (mw). Finally, the third tier that implements the clients that send and receive messages using the middleware, this tier is going to be referred to as "clients tier" or simply "clients". Figure 1 depicts the three tiers and they way they are connected to each other. As can been seen in the figure there is only one database while there can be more than one middlewares that are identical to each other and are connected to the database, as well as many clients connecting to different middlewares.

2 System Design and Implementation

In this section we describe the design of our system. We start by describing the code structure of our implementation and its main interface and afterwards we look more thoroughly at every tier and how its functionality was implemented.

Code Structure and Interfaces Overview

All the code for the client and the middleware was implemented in subpackages of *ch.ethz.inf.asl*. The package structure can be seen in Figure 2.

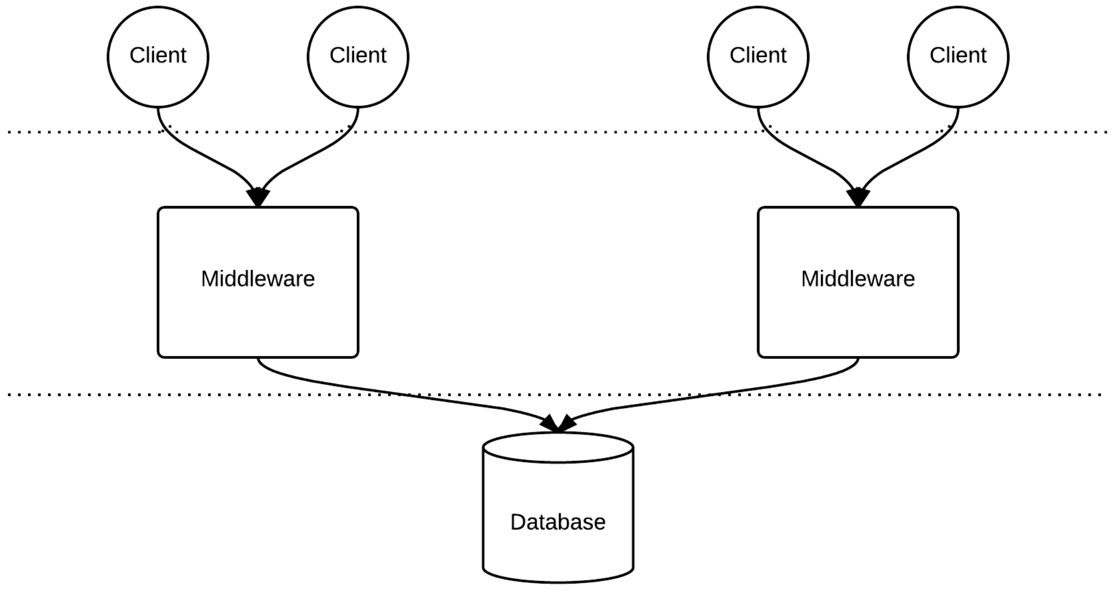


Fig. 1: The Three Tiers

ch.ethz.inf.asl
- client :: contains classes related to client code
- common :: package containing common classes to be used by both the clients and the middleware
  - request :: contains all the possible request classes and the *Request* abstract class
  - response :: contains all the possible response classes and the *Response* abstract class
- console :: contains the management console code
- exceptions :: contains relevant exceptions used by the application
- logger :: contains the *Logger* class used for instrumenting the system
- main :: contains the Main class that is used to start the clients and the middleware
- middleware :: package containing classes related to the middleware
  - pool :: package containing pool implementations
    - connection :: contains the implementation of a connection pool
    - thread :: contains the implementation of a thread pool
- utils :: contains general utility methods for the application

Fig. 2: Package Structure

While designing the system we came to the realization that the communication protocol, meaning the messages that are being sent, for example send message, receive message etc. are the same between the clients and the middleware and between the middleware and the database. They are the same in the sense that when a client wants to send a message he has to issue some kind of send message request to the middleware. Similarly when the middleware wants to serve a send message request or the client he can issue a send message to the database. Because of this we created the interface that can be seen in Figure 3. This interface can be found in the *MessagingProtocol* interface and is implemented by both *ClientMessagingProtocolImpl* and *MiddlewareMessagingProtocolImpl* classes. The difference between the two implementations is that in the client implementation when for example *sendMessage(...)* is called an underlying connection is used to a send a message from the client to the middleware that informs the middleware of the desire of the client to send a message. While on the other hand when the middleware calls *sendMessage(...)* the middleware is calling a stored function from the database to actually “save” the message in the database. More on how this interface was implemented by the client and the middleware is given in their corresponding subsections.

```
int sayHello(String clientName);
void sayGoodbye();
int createQueue(String queueName);
void deleteQueue(int queueId);

void sendMessage(int queueId, String content);
void sendMessage(int receiverId, int queueId, String content);
Optional<Message> receiveMessage(int queueId, boolean retrieveByArrivalTime);
Optional<Message> receiveMessage(int senderId, int queueId, boolean retrieveByArrivalTime);
Optional<Message> readMessage(int queueId, boolean retrieveByArrivalTime);
int[] listQueues();
```

Fig. 3: Messaging Protocol Interface

As can be seen from Figure 3 the retrieving messages methods use the *retrieveByArrivalTime* parameter. If this parameter is true then the message retrieved is the oldest one. The *readMessage()* method is different to its corresponding *receiveMessage()* method since it only reads a message from the system but is not actually removing it.

Database

The PostgreSQL database management system was used, specifically PostgreSQL (release 9.3.5). It was need for the system to persistent store information so a database was used to store the needed information for the clients, the queues and the messages. For this reason three tables were created as can been seen in Figure 4 with their fields and their respective SQL types. As can been seen in this figure the fields *sender_id*, *receiver_id* and *queue_id* are all foreign keys of the *message* table. The first two are associated with the *id* of the *client* table, while *queue_id* is connected to the *id* of the *queue* table.

client
id *serial primary key*
name *varchar(20) NOT NULL*

queue
id *serial primary key*
name *varchar(20) NOT NULL*

message
id *serial primary key*
sender_id *integer REFERENCES client(**id**) NOT NULL*
receiver_id *integer REFERENCES client(**id**)*
queue_id *integer REFERENCES queue(**id**) NOT NULL*
arrival_time *timestamp NOT NULL*
message *text NOT NULL*

Fig. 4: Tables

As can be seen all of the fields except the *receiver_id* of the *message* table cannot contain the *NULL* value. This was a deliberate choice since it is possible for a message to be sent with no particular receiver in mind and such a message could possibly be received by any other client (except the client that sent the message). In such a case, i.e. a message has no specific receiver, the *receiver_ id* contains the *NULL* value.

The *message* table has also two check constraints associated with it. Those constraints are:

1. *CONSTRAINT check_ length CHECK (LENGTH(message) <= 2000)*
2. *CONSTRAINT check_ cannot_ send_ to_ itself CHECK (sender_ id != receiver_ id)*

The *check_ length* constraint checks that a message cannot contain a message with too much content, in this case one with more than 2000 characters, this constraint was removed during the "increasing message size" experiment. The second constraint was added because it was considered meaningless for a client to send a message to himself. It is also considered meaningless for a client to receive a message he sent (in case the *receiver_ id* is *NULL*), this is also checked in the stored function and is explained later on.

In order to increase the performance of the database, indexes were used. PostgreSQL creates by default indexes on the primary keys[1]. The extra indexes that were introduced are the following:

1. *CREATE INDEX ON message (receiver_ id, queue_ id)*
2. *CREATE INDEX ON message (sender_ id)*
3. *CREATE INDEX ON message (arrival_ time)*

The first index was introduced to make faster the retrieval of message since most of them are based on a *receiver_ id* and on a *queue_ id*. Note that the field *receiver_ id* appears first on this multicolumn index, this was not a random choice since it is known[2] that the in a multicolumn index the leftmost column can also be efficiently used solo. The case where *receiver_ id* is used alone and not in combination with *queue_ id* is the listing queues query that lists the queues where a message for a client exists. The second index was created to speed up receiving of messages from a specific sender. The third index was introduced since some of the receiving messages functions receive messages based on the arrival time.

Code for the creation of the tables and the indexes can be found in the `auxiliary_functions.sql` file in `src/main/resources`.

Stored Functions

Stored functions were created using the PL/pgSQL procedural language to reduce the network communication time between the middleware and the database. Also stored functions have the advantage that they are compiled already by the DBMS and their query plan has been generated so they can be reused and therefore increa*se* performance. The code for the stored functions can be found in `read_committed_basic_functions.sql` file in `src/main/resources` and all of them are used to be able to implement the interface shown in Figure 3.

The stored functions *read_ message* and *receive_ message* specifically check that if a message has no receiver it is not being returned to the client that sent it since this cannot be catched by the *check_ cannot_ send_ to_ itself* constraint because in those cases the *receiver_ id* is *NULL*.

Stored functions were not used everywhere, only where it made sense. For example in cases where the same SQL queries did not need to be executed many times, simple queries were sent to the database instead, e.g. management console.

[1] "Adding a primary key will automatically create a unique btree index on the column or group of columns used in the primary key." (http://www.postgresql.org/docs/9.3/static/ddl-constraints.html)

[2] "...but the index is most efficient when there are constraints on the leading (leftmost) columns." (http://www.postgresql.org/docs/9.3/static/indexes-multicolumn.html)

Transactions and Isolation Levels

In this subsection we discuss isolation levels and why they are important for the correctness of our system. In order to do so let us see a simplified version of the internals of the *receive_message* stored function, seen in Figure 5, that takes two parameters, the *p_requesting_user_id* and the *p_queue_id* and is trying to find a message for the requesting user in the given queue.

```
SELECT id INTO received_message_id FROM message WHERE queue_id = p_queue_id AND
receiver_id = p_requesting_user_id LIMIT 1;
RETURN QUERY SELECT * FROM message WHERE id = received_message_id;
DELETE FROM message where id = received_message_id;
```

Fig. 5: Simplified Version of *receive_message* Body

Functions in PostgreSQL are executed within transactions[3]. Transactions are known to be atomic, in the sense that they either "happen" completely, i.e. all their effects take place, or not at all. But still problems could arise! The default isolation level in PostgreSQL is READ COMMITTED[4] which roughly states "...a SELECT query (without a FOR UPDATE/SHARE clause) sees only data committed before the query began; it never sees either uncommitted data or changes committed during query execution by concurrent transactions."[4]. So with such an isolation level it is possible for two concurrent transactions to read the exact same message, only one of them will delete it, but both of them will return it. This of course is not acceptable since we want a message to be read by only one client. In order to solve this problem there are at least two approaches:

1. Use *FOR UPDATE*[5] and therefore prevent other transactions from selecting the same message.
2. Change isolation level to *REPEATABLE READ* which is stronger than *READ COMMITTED* and roughly states "This level is different from Read Committed in that a query in a repeatable read transaction sees a snapshot as of the start of the transaction, not as of the start of the current query within the transaction."[4]. In case another transaction deletes the message in the meantime the transactions is going to fail by giving back an error.

The "problem" with the second approach is that a transaction could find concurrent update errors and will have to be re-executed: "...it should abort the current transaction and retry the whole transaction from the beginning."[4]. For the above reasons we used the first approach since it made our application code easier, i.e. not having to repeat a transaction.

We have to mention here that the *SELECT* command combined with *FOR UPDATE* and *ORDER BY* could have some problems: "It is possible for a SELECT command running at the READ COMMITTED transaction isolation level and using *ORDER BY* and a locking clause to return rows out of order."[5]. This is because ordering of the rows occurs before locking them, so it is possible that when the rows are locked some columns might have been modified. This is not a problem in our implementation since we delete the selected row.

Connecting Java and PostgreSQL

For the connection between Java and the database the JDBC41 PostgreSQL driver[6] was used.

[3] "Functions and trigger procedures are always executed within a transaction established by an outer query" (http://www.postgresql.org/docs/current/interactive/plpgsql-structure.html)

[4] http://www.postgresql.org/docs/9.3/static/transaction-iso.html

[5] "FOR UPDATE causes the rows retrieved by the SELECT statement to be locked as though for update. This prevents them from being modified or deleted by other transactions until the current transaction ends." and "That is, other transactions that attempt UPDATE, DELETE, SELECT FOR UPDATE, SELECT FOR NO KEY UPDATE, SELECT FOR SHARE or SELECT FOR KEY SHARE of these rows will be blocked until the current transaction ends. " (http://www.postgresql.org/docs/9.3/static/sql-select.html#SQL-FOR-UPDATE-SHARE)

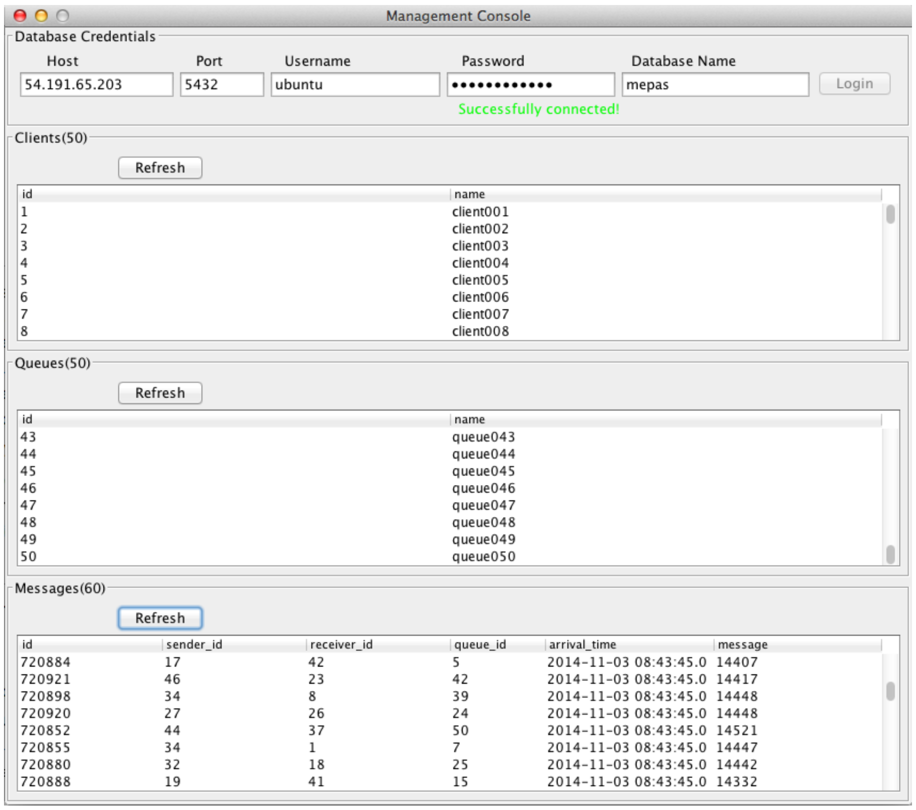
[6] http://jdbc.postgresql.org/download.html

What is being logged?

The only thing that is being logged in the database while it is being used is he the CPU, network and memory utilization using the dstat[7] tool.

Management Console

A management console was also created (it is implemented in the *Manager* class under the *console* package) to easily check the contents of a database in a remote machine. The console is a GUI application and can be seen Figure 6. The user of the console just has to provide the host address and port number of where the database is running, as well as the username, password and database name. Then by clicking "Login" and the appropriate "Refresh" buttons he can check the current data of the client, queue or message table. For retrieving the data from the database simple "*SELECT * FROM ...*" queries were issued on the database, no stored functions were created for this since they are only used for the console and quite sparingly.



Database Credentials

Host	Port	Username	Password	Database Name
54.191.65.203	5432	ubuntu	••••••••••••	mepas

Successfully connected!

Clients(50)

id	name
1	client001
2	client002
3	client003
4	client004
5	client005
6	client006
7	client007
8	client008

Queues(50)

id	name
43	queue043
44	queue044
45	queue045
46	queue046
47	queue047
48	queue048
49	queue049
50	queue050

Messages(60)

id	sender_id	receiver_id	queue_id	arrival_time	message
720884	17	42	5	2014-11-03 08:43:45.0	14407
720921	46	23	42	2014-11-03 08:43:45.0	14417
720898	34	8	39	2014-11-03 08:43:45.0	14448
720920	27	26	24	2014-11-03 08:43:45.0	14448
720852	44	37	50	2014-11-03 08:43:45.0	14521
720855	34	1	7	2014-11-03 08:43:45.0	14447
720880	32	18	25	2014-11-03 08:43:45.0	14442
720888	19	41	15	2014-11-03 08:43:45.0	14332

Fig. 6: Management Console Screenshot

In order to use the management console the *main()* method from the *Manager* class needs to be executed.

Middleware

The middleware implements the messaging system, it receives requests/messages from clients and has to use the database in order to persist those messages, as well as retrieve the messages from the database to return to the clients. Obviously if we want to be able to support more than one client

[7] http://dag.wiee.rs/home-made/dstat/

the middleware needs to be multi-threaded. The interface that the middleware has to implement can be seen in Figure 3.

Our middleware follows a non-blocking approach using simple Java IO. This seems hard to believe at first but nevertheless this is the case as will be explained later on. But before doing so, let us see some of the possible approaches that can be used to implement a middleware.

- In this approach the middleware would have some threads, also known as worker threads, on the middleware and every one of them waits for a connection from the client. When a connection is established the thread blocks and waits for a request from the client. When it receives the request it, it executes the request meaning it issues the corresponding operations to the database and then returns the response back to the client. Afterwards it closes the connection and waits for the next client connection. Although this approach can support an arbitrary number of client it is quite wasteful and slow since for every request-response interaction the client has to establish a connection with the middleware.

- With this approach we have a worker thread for every client. A worker thread is created when a client connects and then it is used for this client until the end. The advantage of this approach in contrast to the previous is that there does not have to be an establishment of a connection for every request between the client and the middleware. But this solution seems to have scalability issues since the number of clients the middleware could possibly support is bounded by the number of threads the system can support.

- This approach uses Java New IO. The rough idea is having a thread, called selector thread, that blocks until a new connection from a client is established or data from some already established connection are received. When data from a connection are received the reading of the data can be passed to a worker thread that is going to do the actual reading and the one that is going to send the response back to the client. This solution has none of the above problems.

Our approach followed a different way. Its main idea is to have a queue of sockets corresponding to connections from the clients. Every time a client connects to the middleware the socket is being added to this queue. Then there are also some worker threads operating in a round-robin approach on this queue and check if there is something to read from the underlying input stream of the socket. If yes they read the data, use the database to perform their operation and send the response back to the client. This approach has none of the problems described in the first two approaches. Our implementation of this approach is non-blocking since a worker thread never blocks to wait for data from a specific connection, if there are no data in a connection it just puts the connection back to the queue and continues with the next connection. The non-blocking implementation was achieved by using *InputStream*'s *available()* method that can return the number of bytes that can be read without blocking. *available()* is of course a non-blocking method which means a worker thread issues a blocking *read()* method call only when *available()* showed that a number of bytes can be read without blocking. In Figure 7 the architecture of our middleware is depicted while in Figure 8 it is shown how a middleware's worker thread operates. As can be seen in Figure 7 the worker threads of the middleware interact with the sockets queue, as well as with the connection pool in order to get a connection to the database for issuing their request. Note furthermore that the "waiting for a connection" part of the middleware is blocking.

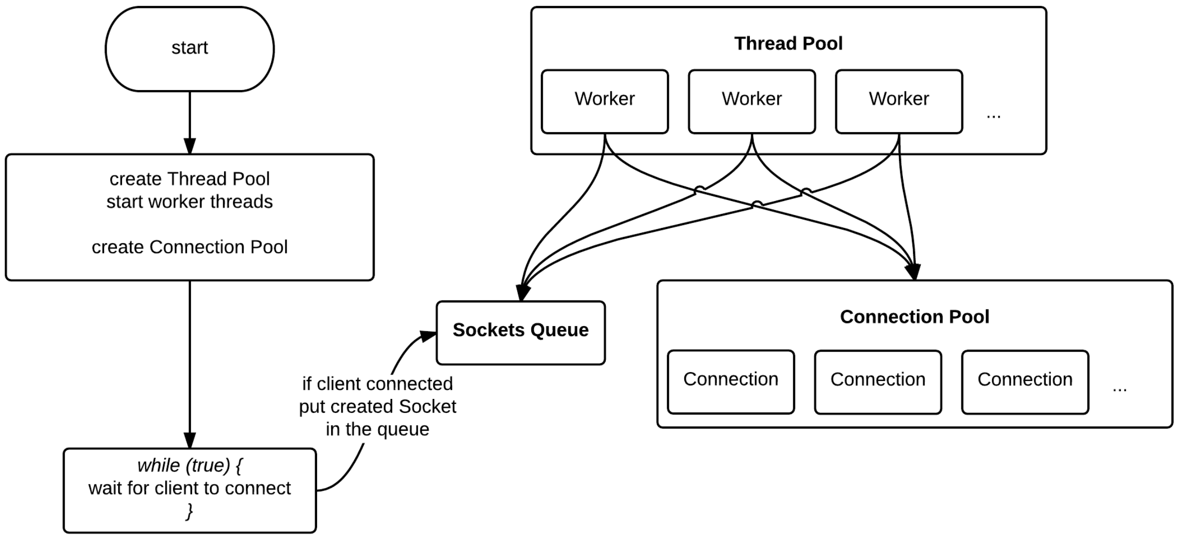


Fig. 7: Middleware Architecture

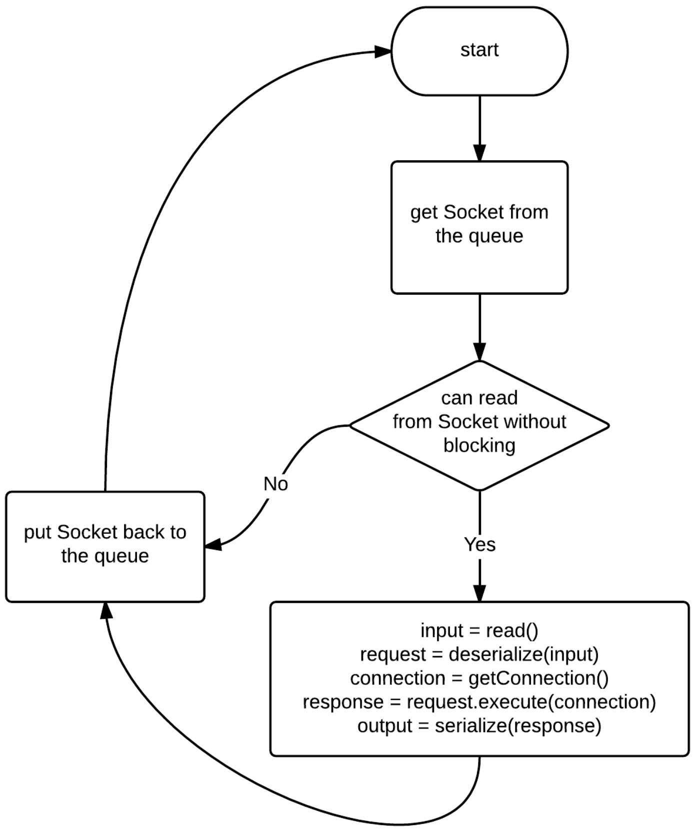

start
get Socket from
the queue
can read
from Socket without
blocking
No
Yes
put Socket back to
the queue
input = read()
request = deserialize(input)
connection = getConnection()
response = request.execute(connection)
output = serialize(response)

Fig. 8: Worker Thread of the Middleware

Middleware Implementation

The implementation of the middleware is located under the package with the same name: *ch.ethz.inf.asl.middleware*. This package contains the following classes:

- *Middleware*: this is the class that needs to be instantiated to start the middleware. It is constructed using a configuration, what is needed for the configuration is explained later on. By calling its *start()* method the middleware initializes the thread pool and waits for upcoming connections from clients. When a client connects his corresponding connection is inserted into the sockets queue.

- *MiddlewareRunnable*: this class corresponds to a worker thread. It is being constructed with the middleware's sockets queue as well as the connection pool created from the middleware.

- ***InternalSocket***: the middleware socket queue does not actually contain immediate *Socket (javat.net.Socket)* objects but actually *InternalSocket* objects. An internal socket contains internally a normal Java socket as well as some extra methods. For instance it has a method that returns the last time this socket was worked on (*getLastTime()*), this method is helpful for logging purposes, i.e. we can now know how much a socket was waiting on the sockets queue before it was picked by a worker thread. This class also supports reading data in chunks! Because we use the *available()* method as was said previously it is possible to have only 1 byte available per time and in such cases we need somehow to accumulate the read bytes until we have read the full request. *InternalSocket* achieves this operation by calling *addData(byte[] readData)* every time we read some bytes and we can get the accumulated bytes until now by calling *getObjectData()*.

- *MiddlewareMessagingProtocolImpl*: this class implements the *MessagingProtocol* interface and is the one that actually calls the stored functions.

Thread & Connection Pool

Our own thread and connection pools were implemented, their code can be found under the *ch.ethz.inf.asl.middleware.pool.thread* and *ch.ethz.inf.asl.middleware.pool.connection* packages in the *ThreadPool* and *ConnectionPool* classes respectively. Let us see each of these classes:

- *ThreadPool*: this class is instantiated given the number of threads the desired pool needs to have. This class creates the given amount of threads and executes them, every one of those threads waits to receive another thread submitted by the user of the thread pool to execute it. In order to execute a thread we just have to call the pool's *execute(Runnable command)* method that just adds the *command* in the queue to be executed. The *execute()* method might block if the underlying queue is full. The queue of the threads that is used internally by this classes to "save" the commands is a Java's *ArrayBlockingQueue* queue, that is, a thread-safe bounded queue implementation that orders elements **FIFO** (first-in-first-out).

- *ConnectionPool*: this class can be constructed given the maximum amount of connections we need and the login credentials of the database. Afterwards a call to *getConnection()* returns a *Connection* object. Until the maximum number of connections is reached, new connections are being created on every call to *getConnection()*, afterwards they are being re-used. The *getConnection()* method blocks when there are no more available connections at the moment. When the *close()* method is called on a *Connection* object returned by *getConnection()* the closed connection is not actually closed but just returned back to the pool to be re-used. This was achieved by creating the *InternalConnection* inner class that contains a Java's *Connection* object internally and also extends the *Connection* class. Whenever a call to an *InternalConnection* method is issued the corresponding *Connection* object method is called. Except in the case when the *close()* method is called in which case the specific connection is returned back to the pool. The connection pool can also be closed by calling its respective *close()* method which is going to close all the connections currently existing in the pool, or it can be used in a try-with-resources statement since it implements the *AutoCloseable* interface. Internally the connection pool uses *ArrayBlockingQueue* similarly to the thread pool implementation. This means that requests to *getConnection()* that are waiting are waiting in a FIFO way.

Serialization of Messages

A question that might arise by reading this report until now might be how are requests and responses formatted before being transmitted between the clients and the middleware. Answer: they are serialized. All the requests and all the responses implement the *Serializable* interface, so they can easily be serialized to a *byte* array. This array's length is calculated and the resulted length is transformed to exactly 4 bytes (1 *int*). The length is concatenated with the serialized byte array, with the length being in the front. This concatenated array is what is being transmitted between the client and the middleware. The length is added so when the middleware starts reading data from the client it can read the length of the upcoming object (look *InternalSocket*) and know when to stop reading. When it reads the whole data, it removes the length part and deserializes the remaining byte array to get the *Request* object. The middleware creates the response in the exact same way before sending it to the client. Although this is not necessary for the client, since the client blocks until he reads the response. But it was done nevertheless for consistency reasons so the clients and the middlewares send and receive in the same way.

Where do requests get queued up?

From the above description of the system it is easy to see that there are two main parts of the system were a request could stuck waiting. The one is in the sockets queue waiting until its socket is being worked on by a worker thread. The second part is waiting for the connection, after getting the request the middleware might have to wait for others worker threads to finish their database requests until it is able to proceed with its own. Until then it waits in the connection queue of the connection pool.

What is being logged?

Every worker thread in the middleware logs its own data. This was done so they worker threads do not contend with each other while logging. Afterwards the logs can be merged and be sorted by time to be analyzed.

A snippet of some middleware logs can been seen in Figure 9.

```
16045 5 WAITING THREAD
16167 25 # OF ENTERS
16167 1 # TO READ
16340 110 GOT CONNECTION
16365 25 DB REQUEST SEND_MESSAGE
16366 321 READING INSIDE
```

Fig. 9: Middleware Log Snippet

In the snippet of Figure 9 the left column corresponds to the time in milliseconds (ms) since when this worker thread starting working. So, we can infer that this snippet was taken from the 16th second of the log. All the time values in the snippet are counted in milliseconds. We can see 6 types of logs in the snippet, those are all the possible types that are logged. Let us see each one of them:

1. "WAITING THREAD": corresponds to the time the connection was waiting until it got picked up by a worker thread, in the snippet this was 5ms.

2. "# OF ENTERS": corresponds to the times this socket has been picked by a worker thread but at the time did not contain any data. In this case the specific socket has been picked up by a worker thread 25 times and all of them except the last one the socket did not contain any data for reading.

3. "# TO READ": is the times the socket had been picked by a worker thread (after it started having data) in order for it to read its whole request. As it was said it is possible to have very few bytes available every time a socket contains data, which means those bytes are read and then the socket is returned back to its queue. This number merely shows how many times the socket entered the queue from the moment it had data for reading until all its data (one request) were read. In the above snippet the number is 1 which means the moment the whole request was read at once (all the data were there) by the worker thread.

4. "GOT CONNECTION": corresponds to the time it took a thread to get a connection from the connection pool. In the snippet case this is 110ms.

5. "DB_REQUEST": corresponds to the time it took for a database request. In the snippet's case it took 25ms for a "SEND_MESSAGE" request. Request could also be "RECEIVE_MESSAGE" or "LIST_QUEUES". The value of this log actually also contains the time for request to be sent to the database, executed in the database and then sent back to the middleware.

6. "READING INSIDE": corresponds to the time it took the thread to finish its job from the beginning when it picked a socket that had data till the end. In the snippet's case this is 321ms.

Obviously it should be the case that the time of time("READING INSIDE") >= time("GOT CONNECTION") + time("DB REQUEST").

Similarly to the database the CPU, memory and network utilization are logged.

Starting the Middleware

A middleware can be started by executing the *main()* method of the *Main* class giving two program arguments, the string "middleware" and the file path containing the configuration of the middleware. For example assuming we have an executable JAR file named "mepas.jar", then we can start the middleware by doing "java -jar mepas.jar middleware middleware.properties" where the properties file is similar to the one shown in Figure 10.

```
databaseHost=172.31.12.119
databasePortNumber=5432
databaseName=mepas
databaseUsername=ubuntu
databasePassword=mepas$1$2$3$
threadPoolSize=10
connectionPoolSize=20
middlewarePortNumber=6789
```

Fig. 10: Middleware Properties File

Most of the properties in Figure 10 are self explanatory, obviously the middleware needs the database credentials in order to start as well as the thread pool and connection pool sizes. The "middlewarePortNumber" corresponds to the port where the middleware is going to accept connections from.

Stopping the Middleware

The *Middleware* creates a thread when started based on the *MiddlewareStopper* inner class that is always running on the background. This thread blocks and waits in the standard input for a “STOP” string. When this string is entered the middleware is gracefully stopped by stopping the worker threads and by closing the connection pool and middleware’s *ServerSocket*. This way of stopping the middleware is useful for the experimental setup as will be explained in Section 4.

Clients

The implementation of the clients can be found in the *ch.ethz.inf.asl.client* package. It contains the following three classes:

- *Client*: is the class that needs to be instantiated to start the clients. Every client corresponds to a thread that is being executed, a *ClientRunnable* thread.
- *ClientRunnable*: corresponds to a client. The way a client communicates with the middleware is implemented in this class.
- *ClientMessagingProtocolImpl*: this class implements the *MessagingProtocol* interface. It is responsible for serializing the requests before sent, deserializing the responses when they are received and actually sending and receiving the requests and the responses to the middleware.

Workload

While thinking of the workload of the clients we wanted to have a stable workload and one that does not arbitrarily increases the size of the *message* table in the database. So we came up with the following workload, where every client executes what is shown in Figure 11.

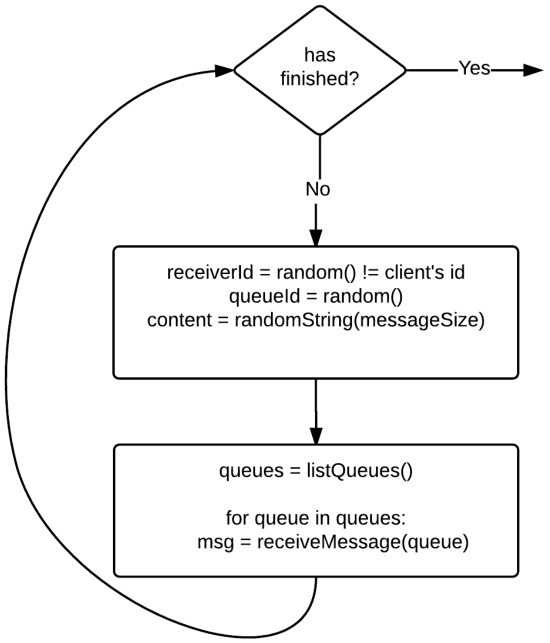

has finished?
Yes
No
receiverId = random() != client's id
queueId = random()
content = randomString(messageSize)
queues = listQueues()
for queue in queues:
msg = receiveMessage(queue)

Fig. 11: Client’s Workload

The clients have no thinking time, they just send requests to the middleware as fast as possible. Also in every iteration of a loop a client sends one message, list the queues once and might call receive message multiple times for different[8] queues. This way we noticed that the number of messages in the database was always in the around a hundred. This workload also has the benefit of using three different types of requests. Also notice that our clients do not use the *sayHello()*, *createQueue()* or *deleteQueue()* methods. This was done to simplify our experiments. Since the *message* table has foreign keys to the *client* and *queue* tables it was possible when a client was trying to send a message to another that the other client has not yet been created. In such a case an error is logged and a failed response message is returned to the client. For this reason we chose to avoid that types of requests.

Initially clients were not sending random strings as content to each other but instead where sending a counter that was increased after every send. This was helpful for verifying that the system was operating correctly. This counter was replaced with random string during the "increasing the message size" experiment which is presented later in this report.

What is being logged?

As with the database and the middleware, clients also log the CPU, network and memory utilization using dstat.

```
27581 28 SEND_MESSAGE (88, 37, HW8Z2)
27604 23 LIST_QUEUES
27629 25 RECEIVE_MESSAGE (49, 40, 1FJFY)
27653 24 RECEIVE_MESSAGE (68, 78, ITCI6)
27679 26 RECEIVE_MESSAGE (71, 85, 9XL3S)
```

Fig. 12: Client Log Snippet

Every client thread logs for itself and at the end the logs are combined. This was done so there is no contention between the clients while logging. In Figure 12 we can see a snippet on what a client logs. As can be seen on the left column the time since the logging started is logged similarly to the middleware logs. The second column contains the time in milliseconds it took for the given request to be served. This time includes the time for the request to be sent in the middleware, served by the middleware and sent back to the client. For example it took 23ms for the listing of the queues in the above log. The parentheses next to "SEND_MESSAGE" and "RECEIVE_MESSAGE" correspond to (queueId, senderId, first five characters of the content) and (queueId, receiverId, first five characters of the content) respectively. Only the first five characters of the content are used to reduce the generated logs size.

Starting the Clients

Similarly to the middleware the clients can be started by executing the *main()* method of the *Main* class. Two arguments need to provided, the first one should always be "client" while the second one corresponds to the file path of the configuration of the clients. Such a configuration can been seen in Figure 13.

[8] This is because of the way the *list_ queues* stored function is implemented. It returns distinct queue ids.

```
middlewareHost=172.31.8.29
middlewarePortNumber=6789
numberOfClients=50
totalClients=50
totalQueues=50
startingId=1
messageSize=20
runningTimeInSeconds=600
```

Fig. 13: Client Properties File

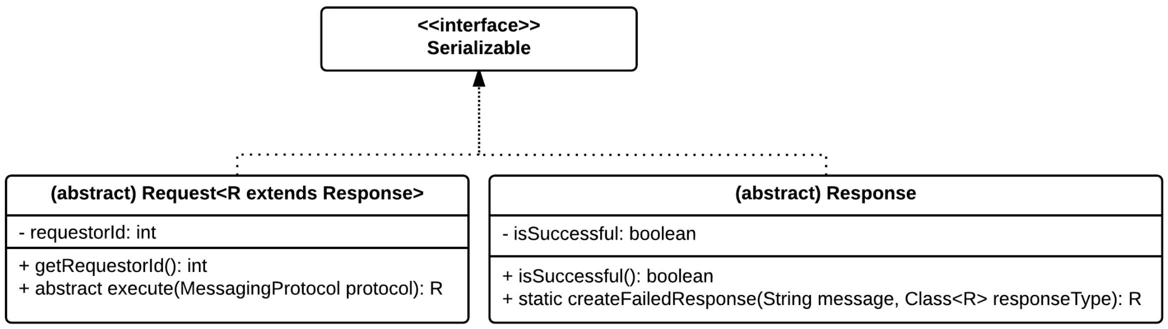
Obviously the client needs to be aware of where the middleware resides, therefore "middlewareHost" and "middlewarePortNumber" exist in the client configuration. The "numberOfClients" corresponds to the "numberOfClients" (number of threads) that are going to be executed by these *Client* execution while "totalClients" are the total clients currently in the system, i.e. where a client can send a message. Since many *Client*'s executions could possibly be running from different machines the "startingId" is used so a specific *Client* execution knows how to assign ids to its clients. The "messageSize" corresponds to the length of the content of a message that is being sent in characters. After "runningTimeInSeconds" the clients stop working and leave the system.

Also note that when starting the clients all the clients are connected to one specific middleware. It is not possible to have clients from a *Client* execution that are connected to different middlewares.

General Remarks

One Request and One Response Class Per Request Type

For every type of request the client can send to the middleware there exists a respective class for this request. And for every specific request there is a corresponding response class. The related to requests and responses classes can be found under the *ch.ethz.inf.asl.common.request* and *ch.ethz.inf.asl.common.response* packages. In Figure 14 we can see the *Request* and *Response* abstract classes with some of their basic methods. Note that the *Request* class uses a generic type R, that is a subclass of *Response*. This was done to avoid possibly bugs like issuing a send message request and expecting a list queues response.



<<interface>>
Serializable
(abstract) Request<R extends Response>
- requestorId: int
+ getRequestorId(): int
+ abstract execute(MessagingProtocol protocol): R
(abstract) Response
- isSuccessful: boolean
+ isSuccessful(): boolean
+ static createFailedResponse(String message, Class<R> responseType): R

Fig. 14: Request and Response Classes

The classes that extend *Request* are all of the format *xRequest* where x is the type of request. Similarly the classes that extend the *Response* class are all of the format *xResponse* where x is the type of request related with this response. It might seem weird having one request and one response class for every possible request and response. But this was done to make the code extensible in case it is needed to add new types of requests. It was also done in order to simplify the implementation of the middleware. The implementation can be simplified because in order to create one more request for the system, the method has to be inserted in the *MessagingProtocol* and then be implemented in both the *ClientMessagingProtocolImpl* and *MiddlewareMessagingProtocolImpl*. Also one subclass of *Request* and one of *Response* have to be be implemented. There is no need to go around and introduce one more enum value or one more 'else-if' statement at some part of the code

The way the described flexibility is achieved is because of the *execute()* method. As can been seen the *Request* class contains the abstract *execute()* method that receives as a parameter a *MessagingProtocol*. This method is being overridden by all the subclasses of *Request* and every one calls its correspond protocol method. For example the *execute()* method in the *SendMessageRequest* does *protocol.sendMessage(...)* while the *execute()* method in *ListQueuesRequest* does *protocol.listQueues()*. Now when the middleware receives a request, after deserializing it it can just issue *request.execute(...)* and the corresponding protocol method is going to be called. Therefore by taking advantage of polymorphism the middleware does not need a long and error-prone list of "if-else-..." statements like this: 'if request is of type send message do this ... else if request is of list queues do this ... '.

Java 7

Initially we were planning to test our system using the Dryad cluster which has Java 7 installed. For this reason we re-implemented part of the *Optional* class found in Java 8. Also almost everywhere the try-with-resources, a feature that appeared in Java 7, is being used so we can be assured that the respective *close()* method is called, even in case of an exception.

Building with Ant

Ant was used to build the project. Ant's build file is `build.xml` and can be found in the root directory of the project. It contains the following targets:

- compile: which just builds the system.
- jar: creates the executable JAR.
- test: runs all the tests of the system. Beware that all the tests might take some time to get executed and also a database needs to exist in the system for some of the tests to successfully get executed.
- clean: removes the generated class files and the executable JAR.

3 Testing

Correctness of our system was of foremost importance, therefore testing played an impo rtant role while developing the system. Many parts of the system have been tested with unit tests (the Intellij IDEA 13 coverage tool shows that 83% of the classes were covered). Although testing the system took its fair amount of time we do believe it was worth it since it helped us find bugs while still working locally with the system that if they appeared when running experiments would be more hard to locate. For testing the TestNG[9] testing framework was used and also the Mockito[10] framework was used for mocking. TestNG is similar to JUnit while Mockito allows the developer to easily and fast mock objects that would be quite expensive to construct. For example, the configuration files were mocked in the end-to-end tests using Mockito.

All the tests are located in the `src/test` directory under the package *ch.ethz.inf.asl*. With the exceptions of the *endtoend* and *testutils* packages, all the other packages are the same as with the non-test code packages and under them the corresponding tests can be found. The tests that

[9] http://testng.org
[10] https://github.com/mockito/mockito

belong to the *DATABASE* and *END_TO_END* groups, defined in *TestConstants* class in the *testutils* package, are using the local database which is being accessed by the constants given in the same file.

Stored Functions

Since the stored functions are in some sense the core of our system, they have been tested thoroughly.

The first tests can be found in *SQLFunctionsDatabaseTest* class and actually check that the stored functions actually do what they are supposed to do. For testing them the database is populated with some fake data taken from the file `src/test/resouces/populate_database.sql` and the stored functions are applied to this populated database, after the stored functions are applied we verify the expected results.

The second tests can be found in *SQLFunctionsConcurrentCallsDatabaseTest* class and they check that with the given isolation levels as explained in the previous section, the stored functions still operate correctly. This test actually creates many concurrent readers that issue receive message requests and at the end it is verified that no message was read more than once and that all messages were read.

End-to-End Tests

There are two end-to-end tests for our system. Both of them exist under the *endtoend* package. The first one exists in *EndToEnd* class while the other one in *EndToEndWithMessages*.

EndToEnd Class

This test is as close as possible to how the system is being executed and uses the *Client* and *Middleware* classes. It creates two middlewares and 4 clients all running on the local machine. In this scenario there are 2 clients connected to each middleware. The clients are being executed for 20 seconds and they communicate with each other by sending and receiving messages. At the end of their execution it is verified that number of requests sent by the clients were actually received by the middleware and no more. And that the number of responses sent from the middlewares were actually received by the clients. In order to check the requests and responses that were being sent and received we had to inject some end-to-end testing code in the normal non-testing code, e.g. method *getAllRequests* in the *Middleware* class. This was done halfheartedly since it mixes tests with the code, but at the end this test was useful since after every change in the system by running this test we could be assured that everything was still in place.

EndToEndWithMessages Class

This test uses one middleware and 2 clients that send and receive specific messages with each other. It is verified that every client actually receives the messages sent by the other and with the expected content. In order to so do it uses the *ClientMessagingProtocolImpl* class immediately and not the *Client* class.

Encountered Bugs

Here we present a view of the bugs we found by running the created tests.

- Inside a stored function we had *RETURN QUERY SELECT id INTO received_message_id* ... and although this was not raising any problems with PostgresSQL it was throwing this error message when tested: “PSQLException ERROR: cannot open SELECT query as cursor”.
- In the *receive_message_from_sender* stored function we had *SELECT * FROM WHERE sender_id = p_sender_id AND receiver_id = p_requesting_user_id OR receiver_id IS NULL*. Our tests were failing and we realized we were missing a parenthesis, it should be

... sender_id = p_sender_id AND (receiver_id = p_requesting_user_id OR receiver_id IS NULL) instead.

- Quite some *NullPointerException*. One of them was in the *Message*'s *equals()* method we had *this.receiverId.equals(other.receiverId)* and *receiverId* could possibly be *NULL*.
- As was said previously the idea of having the clients create themselves and the queues initially lead to problems because a client could have tried to send a message to a client that is not in the system yet. This was immediately noticed after implementing this functionality and running the end-to-end test.

General Encountered Problems

Unfortunately not all bugs were found by the tests. The most sneaky one was the following that allowed clients to receive the exact same message. This was done because the *receive_message* stored function was like the one shown in Figure 15.

```
SELECT id INTO received_message_id FROM message WHERE queue_id = p_queue_id ...;
RETURN QUERY SELECT * FROM message WHERE queue_id = p_queue_id AND ...;
DELETE FROM message where id = received_message_id;
```

Fig. 15: Buggy *receive_message*

The problem with the shown receive is that a message is found and selected but then when returning it with *RETURN QUERY SELECT * FROM* another message could possibly be returned. So although the *FOR UPDATE* explained earlier in the report was protecting us from two clients deleting the same message, it was not protecting us from two clients returning the same message. This bug was solved by just changing the return statement to: *RETURN QUERY SELECT * FROM message WHERE id = received_message_id*. The aforementioned bug was not found using tests but was noticed through the use of counter in the content of the messages, it was quickly noticed that different clients received messages with the exact same counter.

Another bug, not so important as the previous, that was also not found by tests but noticed while working with the system was not closing all the connections when closing the connection pool. This was because the connection pool closing method was implemented as shown in Figure 16.

```
for (int i = 0; i < connections.size(); ++i) {
InternalConnection connection;
try { connection = connections.take();
...
```

Fig. 16: Buggy *close()*

The problem with the implementation of Figure 16 was that the *connection.size()* was returning different size in every iteration of the loop.

4 Experimental Setup and Analysis

In this section we present the general flow of how our experiment were conducted. Then, we present how we set up our instances and the implementation details on how we automatized the experiments. At the end of the section we describe how we analyzed the logs generated by our experiments.

Flow

Before presenting the exact details of our experimental setup we present the general flow of how an experiment is conducted, which is depicted in Figure 17.

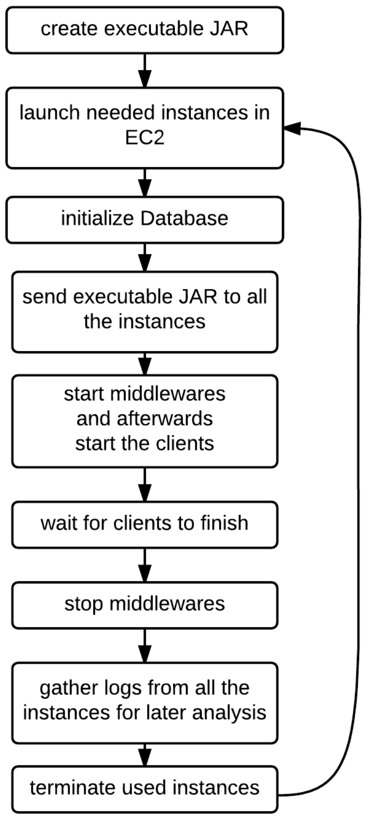

create executable JAR
launch needed instances in EC2
initialize Database
send executable JAR to all the instances
start middlewares and afterwards start the clients
wait for clients to finish
stop middlewares
gather logs from all the instances for later analysis
terminate used instances

Fig. 17: Flow of an Experiment

EC2 Instances

All our experiments were conducted using Amazon Elastic Compute Cloud (EC2) instances. We never executed more than one middleware in one instance and never executed more than two executions of the *Client* class in one instance as well. Also all instances of an experiment were located in the same availability zone[11]. We also never mixed clients, middlewares or the database in the same instance.

In order to setup the instances to use them for experiments we launched an instance based on "Ubuntu Server 14.04 LTS (HVM), SSD Volume Type" Amazon Machine Image (AMI) and for client and middleware instances we installed the "openjdk-7 jdk" package to be able to execute the JAR. For the database instance we installed "postgresql" and for both of them we installed "dstat" that was used for CPU, memory and network utilization logging.

In order for PostgreSQL to accept TCP connections and allow anybody from outside the instance to access our database we changed the line *listen_addresses='localhost'* to *listen_addresses='*'* in the `/etc/postgresql/9.3/main/postgresql.conf` configuration file, as well as added the line *host all all 0.0.0.0/0 md5* to the file `/etc/postgresql/9.3/main/pg_hba.conf`.

[11] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html

After those instances were created we generated their corresponding AMI's so we could easily launch client, middleware or database instances without having to re-do the installations described above.

All the instances used the same general security group that allowed access from any IP for all inbound and outbound connections. This security group was used when launching our instances.

Note that in Figure 17 we show that instances are launched and terminated for every run of an experiment. This might be costly since when using EC2 instances you pay per hour even if you use an instance for just a minute. Nevertheless, this approach for doing the experiments simplified a lot our flow.

EC2 instances can vary in their type[12]. The types[13] that were used in our experiments and some of their characteristics are shown in the table below.

Type of Instance	vCPU	Memory (GiB)
t2.small	1	1
t2.medium	2	2
m3.large	2	7.5
r3.2xlarge	8	61

Implementation

The experimental setup implementation was solely done using Python together with these packages:

- boto[14] for interacting programmatically with EC2 instances. Its main use was to launch and terminate instances on demand.
- pexpect[15] for connecting to the EC2 instances used in the experiments and executing appropriate commands. Using pexpect we connected to an EC2 instance using SSH and issued interactively commands. For example we could start the middleware issuing a "java -jar ..." command and at some later point in time send the string "STOP" to gracefully stop the middleware.
- psycopg2[16] for connecting to the PostgreSQL database and getting it ready for our experiments.

These are the most important classes for the experiment setup residing in the `experiments/code` directory.

- *EC2Instantiator*: contains methods to launch new instances, as well as terminate instances. Specifically it can create instances that can be used by clients or by a middleware as well as instances that are going to be used for databases.
- *Client*: contains methods to start clients in a client instance as well as a method to inform us on whether the clients have finished.
- *Middleware*: has methods to start the middleware in an instance as well as a method to gracefully stop the middleware.
- *Database*: contains methods that are needed to get the database in a valid initial state before the beginning of an experiment.

[12] http://aws.amazon.com/ec2/instance-types/
[13] "The t2 types of instances are designed to provide moderate baseline performance and the capability to burst to significantly higher performance as required by your workload." (http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/t2-instances.html)
[14] https://github.com/boto/boto
[15] https://github.com/pexpect/pexpect
[16] http://initd.org/psycopg/

In order for the experiments to get executed, notice that the following have to been done in the machine running the experiments:

- The private key used to connect to the EC2 instances needs to be added in the `.ssh` directory. This can be done issuing this command *ssh-add privateKeyFile.pem*. It is much easier to "ssh" a machine when the SSH key has been added, since doing *ssh ubuntu@host* is enough to connect to the instance.
- Strict host key checking needs to be disabled[17]. By doing so a user does not have to manually press "Yes" when a host is unknown.
- The experiment runner assumes database's password exists in a `.pgpass` file in the local directory that contains a line like the following: **:*:*:*:mepas$1$2$3$*. By doing so you can issue a "psql" command without having to provide a password.

The experimental code can be found in `ExperimentRunner.py.` In order to configure an experiment a Python dictionary is used in the *ExperimentRunner* file. The configuration looks similarly to the one shown in Figure 18 with added comments explaining the values that are not self-explanatory.

The configuration shown in Figure 18 cannot be used for all kind of experiments. For example it cannot be used in an experiment trying to increase two values at the same time. In such cases a small addition was done inside the *for* loop of our experiments, the *for* loop actually corresponds to the loop shown in Figure 17. For instance if we needed to run an experiment increasing both the number of threads and the number of connections we could just use the given configuration and also add the statement *conf["connectionPoolSize"]=variable* in our *for* loop. The *for* loop was also added in a function so we can use multithreading to run experiments concurrently. Note however that all the experiments running at the same time cannot use more than 20 **running** instances since this is the limit given by Amazon on a specific availability zone.

Analysis

The code that analyses the generated logs from the experiments can be found in the `ResultReader.py` file in the `experiments/code` directory. It contains the following methods:

- *getTrace()*: since the trace is a different kind of an experiment with respect to the others it also has its own function for analyzing it. This function was used to break the trace in intervals of one minute and calculating throughput and response time.
- *getThroughput()*: this function is used to calculate the throughput of an experiment.
- *getResponseTime()*: this function is used to calculate the response time of an experiment.
- *getTimeSpentOnEachComponent()*: this function uses the middleware logs to find the average time spent on each component of the middleware, e.g. waiting to be picked up by a worker thread, waiting to get a connection etc.
- *getCPUUsage()*: used to calculate the average CPU utilization of an experiment given the corresponding CPU usage file generated by dstat.

All the aforementioned functions use extensively UNIX commands like awk, sed, grep, cat, wc and others. This was done to achieve faster analysis of the logs. For example for calculating the number of lines of a file a "wc" command is called from within Python instead of opening and reading every single line of the file. Another example would be removing the warm-up and cool-down phase of an experiment which can be simply done like this: *awk -F'\t' '$1 >= warmUpInSeconds && $1 <= (lastTimeInSeconds - coolDownInSeconds) { print; }' file.*

[17] http://askubuntu.com/questions/87449/how-to-disable-strict-host-key-checking-in-ssh

```
conf = {
"nameOfTheExperiment": "name", # corresponds to the directory where all logs are saved
"username": "ubuntu", # username for the EC2 instances
"placement": "us-west-2c", # availability zone to be used by all instances
"databaseType": "m3.large",
"clientInstances": (2, "m3.large"), # number of client instances and their type
"middlewareInstances": (2, "m3.large"), # number of middleware instances and their type
"databaseUsername": "ubuntu",
"databasePassword": "mepas$1$2$3$",
"databaseName": "mepas",
"databasePortNumber": 5432,
"middlewarePortNumber": 6789,
"runningTimeInSeconds": 600,
"threadPoolSize": 20,
"connectionPoolSize": 20,
"totalClients": 100,
"totalQueues": 50,
"messageSize": 20,

# “mappings” maps a client instance to a middleware instance in the form (a, b)
# where a corresponds to the index of a client instance and b to the index of
# a middleware instance. Indexes start from 0. For instance in the following mapping
the first tuple states that we connect the first client instance with the first middleware.
"mappings": [(0, 0), (1, 1)],

# “clientsData” contains pairs in order of the form (a, b) where a corresponds to the numberOf-
Clients per
# instance and b the startingId of every client. The first tuple of “clientsData” corresponds to the
first client
# instance, second to the second, etc.
"clientsData": [(50, 1), (50, 51)],

"variable": "threadPoolSize", # value that is going to be changed for the experiments
"values": [5, 10, 15] # possible values in this case correspond to thread pool sizes
}
```

Fig. 18: Experiment Configuration

Plots

For plotting the results of our experiments we used Gnuplot[18]. The plotting code can be found in the files with the “.gnu” extension under every experiment in the `report` directory.

5 Experiments

In this section we present the experiments conducted including their evaluation. In all the experiments client’s throughput and response time was presented unless stated otherwise. Also all the receives of messages in our experiments were successful.

Stability

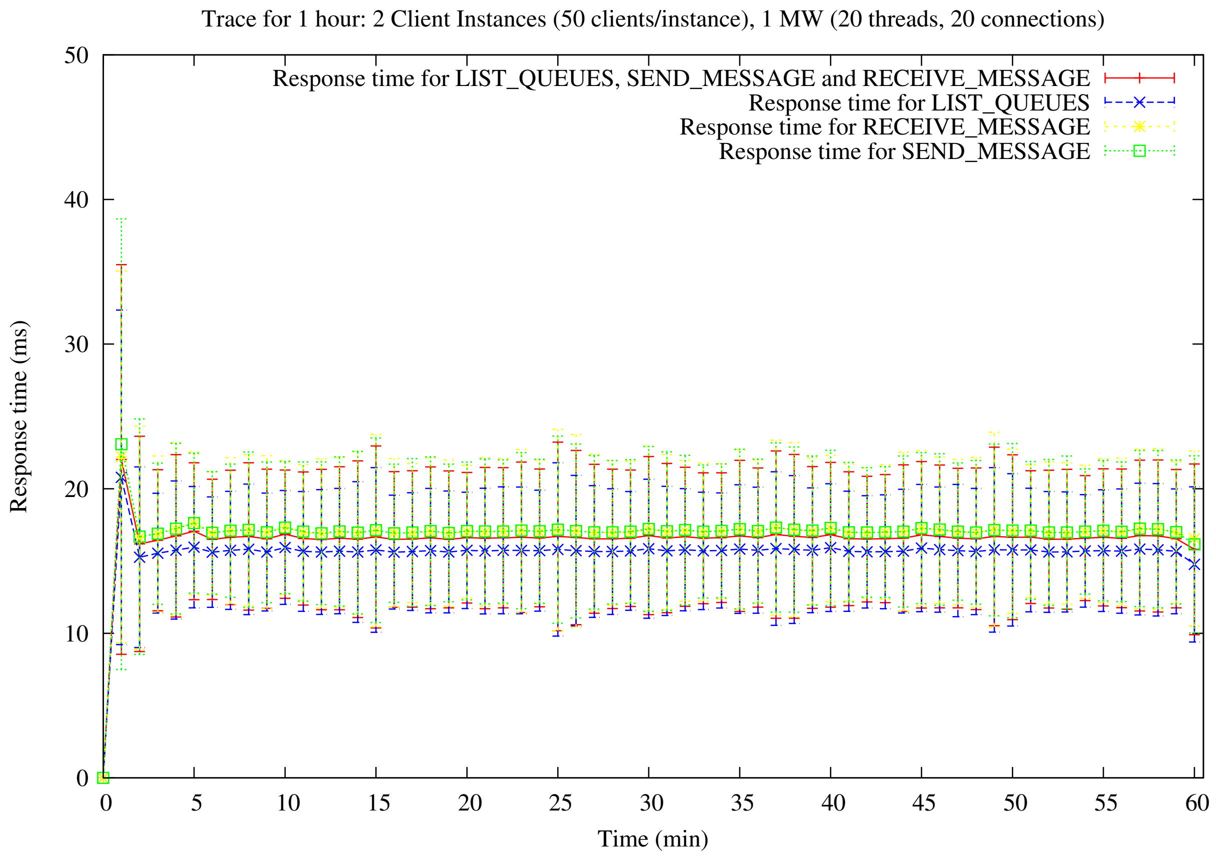
In order to check our system’s stability we ran an one hour trace with the following configuration:

[18] http://www.gnuplot.info/

- 2 t2.small client instances with 50 clients each
- 1 t2.small middleware instance with 20 middleware threads and 20 connections
- 1 t2.medium database

Note that this configuration uses instances of type T2. Those are problematic instances for running experiments since they have burstable performance. T2 instances receive CPU credits every hour which can then be used to give more processing resources to the instances. Before having the experimental setup we explained in the previous section we a had a slightly different one. We were not terminating instances after every experiment but were reusing them instead. The trace was executed in instances that were running for quite some time without being used and therefore their burst-ability did not interfere with our trace.

In Figure 19 the throughput and response time values of our traces can been seen. In this figure response time was averaged over the interval of one minute while throughput was calculated per second and averaged over the interval of one minute.



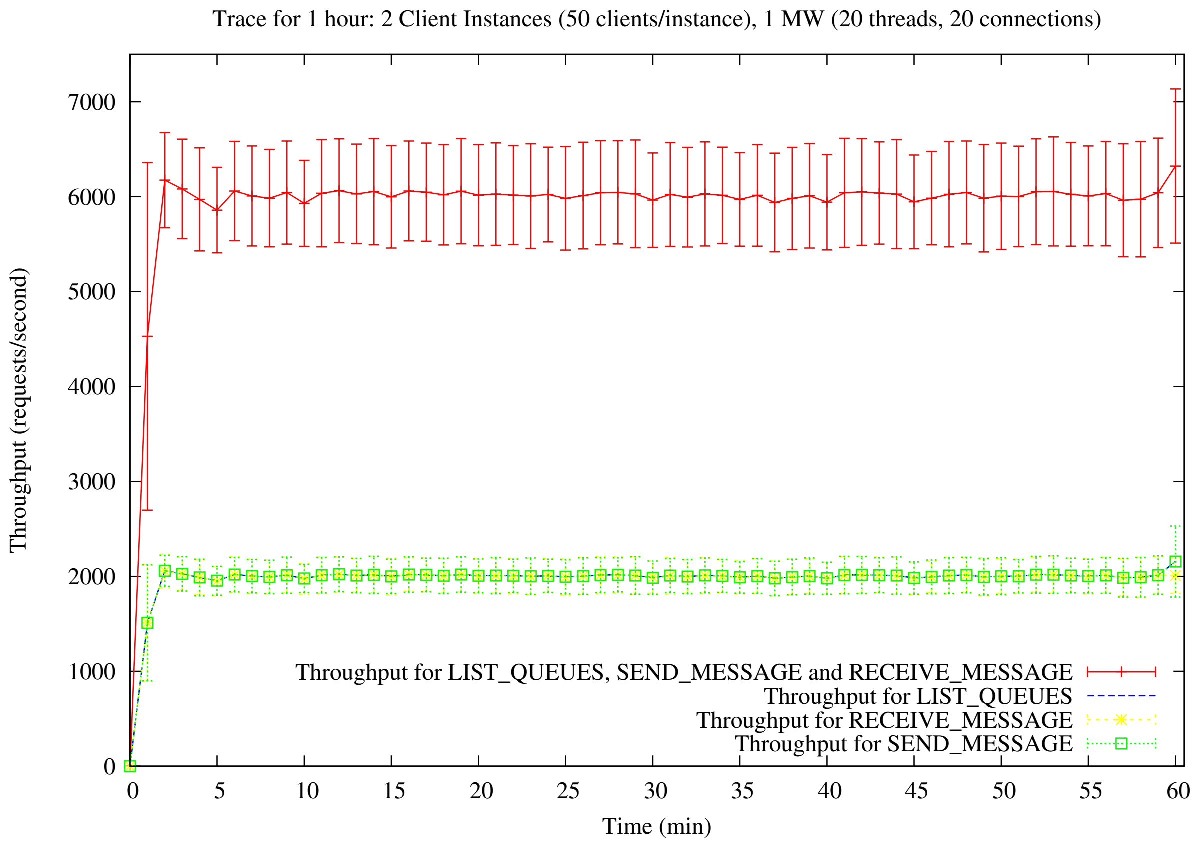


Fig. 19: Response time and throughput of an one hour trace with 2 client instances (50 clients/instance) and 1 middleware instance (20 threads, 20 connections)

Based on the results of the trace, we calculate the average of all types of requests which is 16.668, as well as the time spent on a specific request shown below:

Type of Request	Average Response Time (ms)	Standard Deviation
RECEIVE_MESSAGE	17.10725	5.51819
SEND_MESSAGE	17.1355	5.543025
LIST_QUEUES	15.7621	4.689385

We were also interested in the time spent on each component during the trace, those times were calculated and are presented below:

Time spent on ...	Average Response Time (ms)	Standard Deviation
waiting for a MW thread	13.3207	4.17608
waiting to get a DB connection	0.000689706	0.216034
executing a DB request	3.23399	2.38918
working inside the MW thread	3.33539	2.51786

The time to receive a connection is almost nonexistent, this is because the amount of worker threads is the same with the number of connections, so when a worker thread asks for a connection one can be immediately given back to the thread. The time waiting for a worker thread is quite high and was to be expected since we have 100 clients and only 20 worker threads. So at any point in time 80 clients connections could possibly be waiting. Network time is also really low and this makes sense since the throughput between two instances is (iperf).

As we can see our system is stable under load. In the Figure 19 we see that the throughput for the RECEIVE_MESSAGE, SEND_MESSAGE and LIST_QUEUES requests is pretty much the same. This is because every client issues the exact amount of SEND_MESSAGE and LIST_QUEUES requests and because a RECEIVE_MESSAGE is only called if a message is going to be received. So the amount of SEND_MESSAGE requests should be almost the same with the RECEIVE_MESSAGE requests. The average response time of the LIST_QUEUES is a bit lower than the response time of the two other types of requests. Our assumption was that this is because LIST_QUEUES does not modify the database, neither with an INSERT or a DELETE. To verify this claim we checked the database request average response time for every type of request. The results are shown below:

Type of Request	Average Database Response Time (ms)
RECEIVE_MESSAGE	3.60231
SEND_MESSAGE	3.71801
LIST_QUEUES	2.38218

Finally we note that in our trace 99.7% of the requests were served in less than 50ms, 97% in under 25ms and 86% under 20ms.

As was mentioned earlier in this report, we have a **closed system**. This means that interactive response time law should apply to our results. The interactive law states that $R = \frac{N}{X} - Z$ where R is the response time, N is the number of clients issuing requests, Z is the think time and X is the throughput. Our response time is about 16.668ms, we have 100 clients and our think time is 0. In reality Z is a positive value since we have to serialize and deserialize objects but this time is considered negligible. This means that throughput should be $X = \frac{N}{R+Z} = \frac{100}{16.668+0} = 5.999$. The value 5.999 corresponds to the number of requests per millisecond. Per second it is 5999 which is close to the average throughput (of all the trace) that is 5897 requests per second. All the upcoming results have been verified using the interactive law.

Warm Up and Cool Down

For all the following experiments, every data point in the plots corresponds to a 10 minute execution. From those 10 minutes the first 2 minutes have been removed, as well as

the last 1 minute. Those minutes have been removed as they are considered the “warm up” and “cool down” phase respectively. Those number were derived by checking the trace and seeing that after 2 minutes our system stabilizes and until one minute before the end the system is stabilized.

Throughput Plots

In all the following plots the throughput was calculated per 20seconds and averaged over the time of the experiment (excluding “warm up” and “cool down” phases of course).

Increasing the Message Size

In this experiment we increased the size of the messages that are sent between the clients. While running those experiments we realized clients were receiving the following failed response:

```
ch.ethz.inf.asl.exceptions.MessagingProtocolException:
failed to send message ERROR: could not extend file "base/16389/16427.6": No space
left on device Hint: Check free disk space.
```

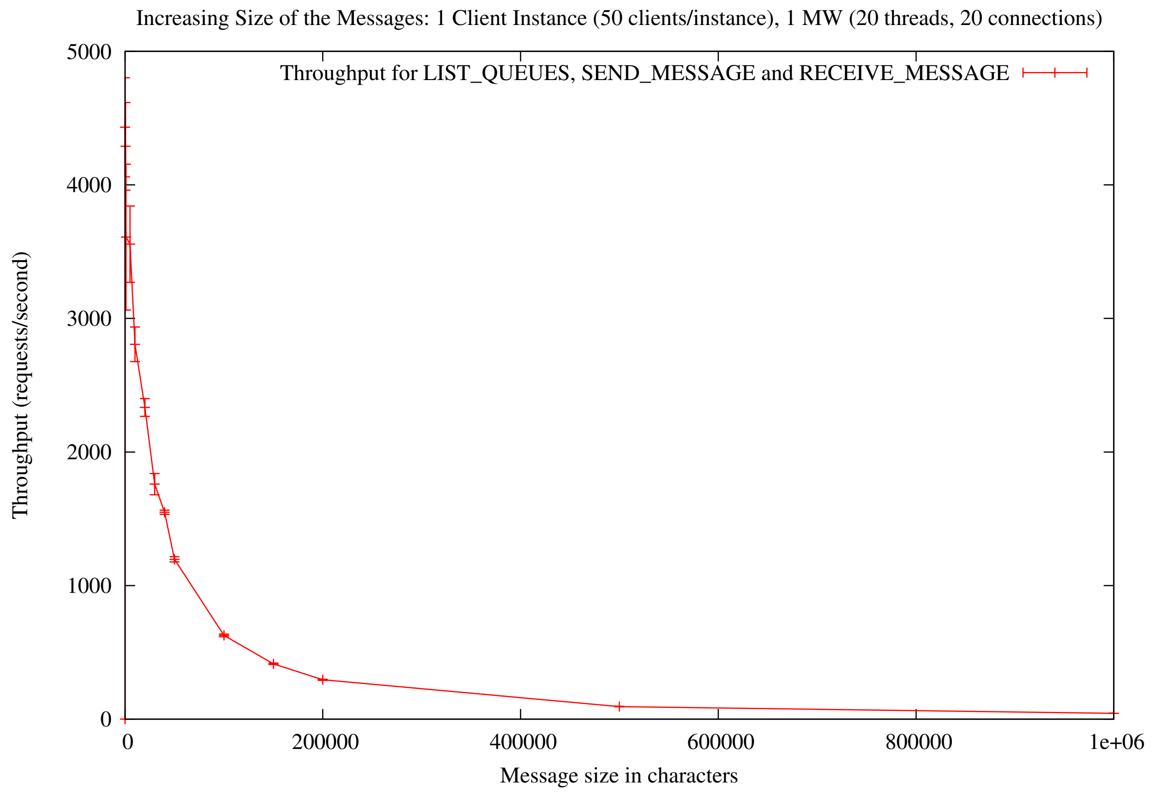
We realized the problem was the amount of available memory in our instance’s disk. We solved this problem by increasing the available size of our disks and rerun the experiments. Afterwards no such failed response was received. The disk space was increasing because we are not VACUUMing[19] our database and it is known that tuples that are deleted are not physically removed from their tables until a VACUUM is issued.

Configuration: The experiment was concuded with:

- 1 m3.large client instance with 50 clients
- 1 m3.large middleware with 20 middleware threads and 20 connections
- 1 m3.large database
- message sizes in number of characters that taken were: 1, 500, 1000, 5000, 10^4, $2 \cdot 10^4$, $3 \cdot 10^4$, $4 \cdot 10^4$, $5 \cdot 10^4$, 10^5, $15 \cdot 10^4$, $2 \cdot 10^5$, $5 \cdot 10^5$ and 10^6

Hypothesis: We expect to have increased response times as well as decreased throughput. This is because by increasing the message size network time needed to send and receive a message will increase but mostly because the database is going to become slower when handling more data.

[19] http://www.postgresql.org/docs/9.3/static/sql-vacuum.html



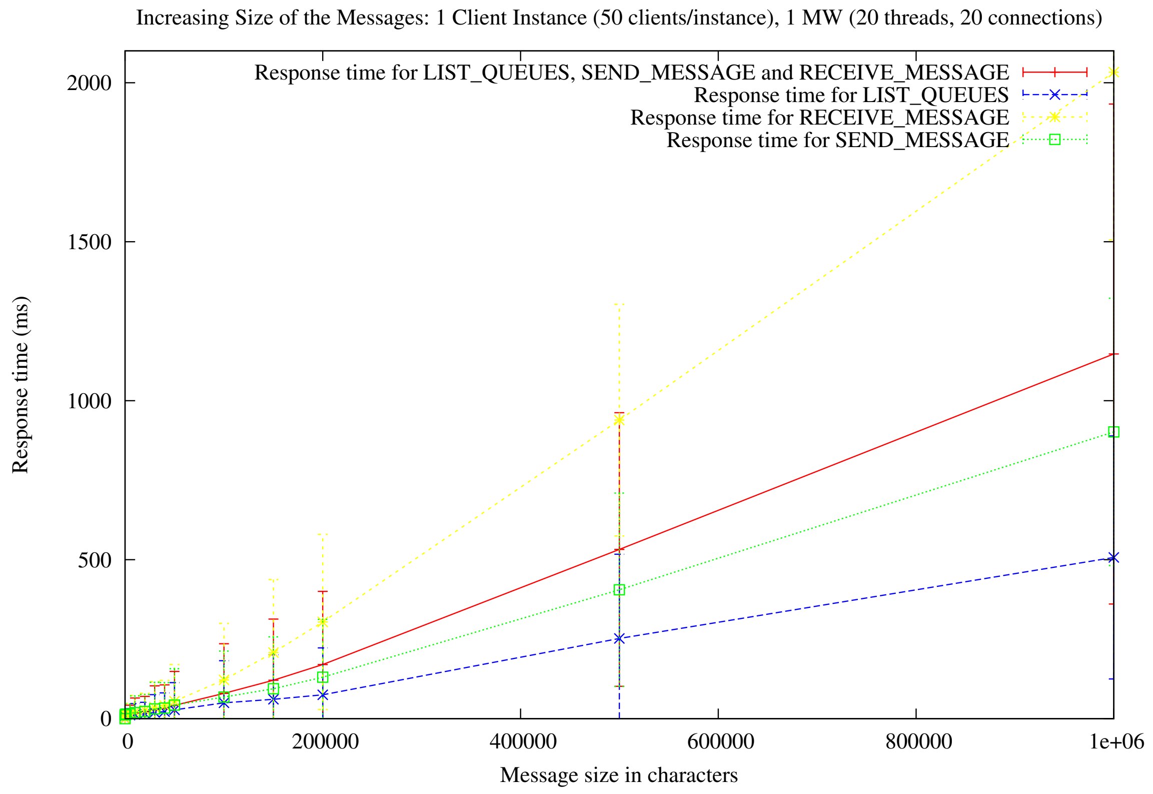


Fig. 20: Increasing the Message Size

As we can see in Figure 20 there is a huge increase in the response time as well as a huge decrease in the throughput as was expected. We also see that LIST_QUEUES requests have less response time than both "SEND_MESSAGE" and "RECEIVE_MESSAGE". This was expected but we weren't expecting such a huge response time for LIST_QUEUES requests since they do not move any messages around. With further investigation why this was the case we found the following message in the PostgreSQL logs: "HINT: Consider increasing the configuration parameter "checkpoint_segments". 2014-11-05 21:29:32 UTC LOG: checkpoints are occurring too frequently (2 seconds apart)". A checkpoint is: "a point in the transaction log sequence at which all data files have been updated to reflect the information in the log. All data files will be flushed to disk."[20]. So this could explain why LIST_QUEUES had such a huge response time. Also the checkpoints can explain why RECEIVE_MESSAGE requests have such a greater response time than SEND_MESSAGE requests since a RECEIVE_MESSAGE request never "puts" something in the database, it is not this request that fires up the checkpoint but it is a SEND_MESSAGE instead. To verify our claims we did one more experiment. In order to prove this claim we run once more the experiment with 10^6 characters per message but this time we changed the configuration of our database concerning its checkpoints. So we changed the file `/etc/postgresql/9.3/main/postgresql.conf` to contain the following:

```
# - Checkpoints -
checkpoint_segments = 64 # in logfile segments, min 1, 16MB each
checkpoint_timeout = 12min # range 30s-1h
checkpoint_completion_target = 0.5 # checkpoint target duration, 0.0 - 1.0
checkpoint_warning = 30s # 0 disables
```

With this configuration we did not receive any HINT's in our databases' log file and the response times were the following:

Type of Request	Average Response Time (ms)	Standard Deviation
RECEIVE_MESSAGE	393.903	769.763
SEND_MESSAGE	1518.57	1894.07
LIST_QUEUES	186.319	568.049
All types of requests	699.703	1357.94

while with the previous configuration we had those results:

Type of Request	Average Response Time (ms)	Standard Deviation
RECEIVE_MESSAGE	901.7	420.242
SEND_MESSAGE	2032.84	527.611
LIST_QUEUES	507.133	382.042
All types of requests	1146.8	786.268

More "checkpoint_segments" implies less often logging a checkpoint so we have better performance. The above tables proves our claim since now LIST_QUEUES and RECEIVE_MESSAGE are quite faster and much less compared to SEND_MESSAGE requests which is the one creating the logs of the checkpoints. The high standard deviation in all of those experiments is easily explained by thinking that when no log checkpoint is taking place, database requests can be served quite quickly but at the moments when checkpoints are being logged the system as a whole becomes slower and therefore we have higher database requests response times.

[20] http://www.postgresql.org/docs/9.3/static/sql-checkpoint.html

Number of Connections Versus Number of Middleware Threads

In our system design there is strong correspondence between database connections and middleware threads given to the database. We wanted to have a better look at the performance of the system when those two factors are combined.

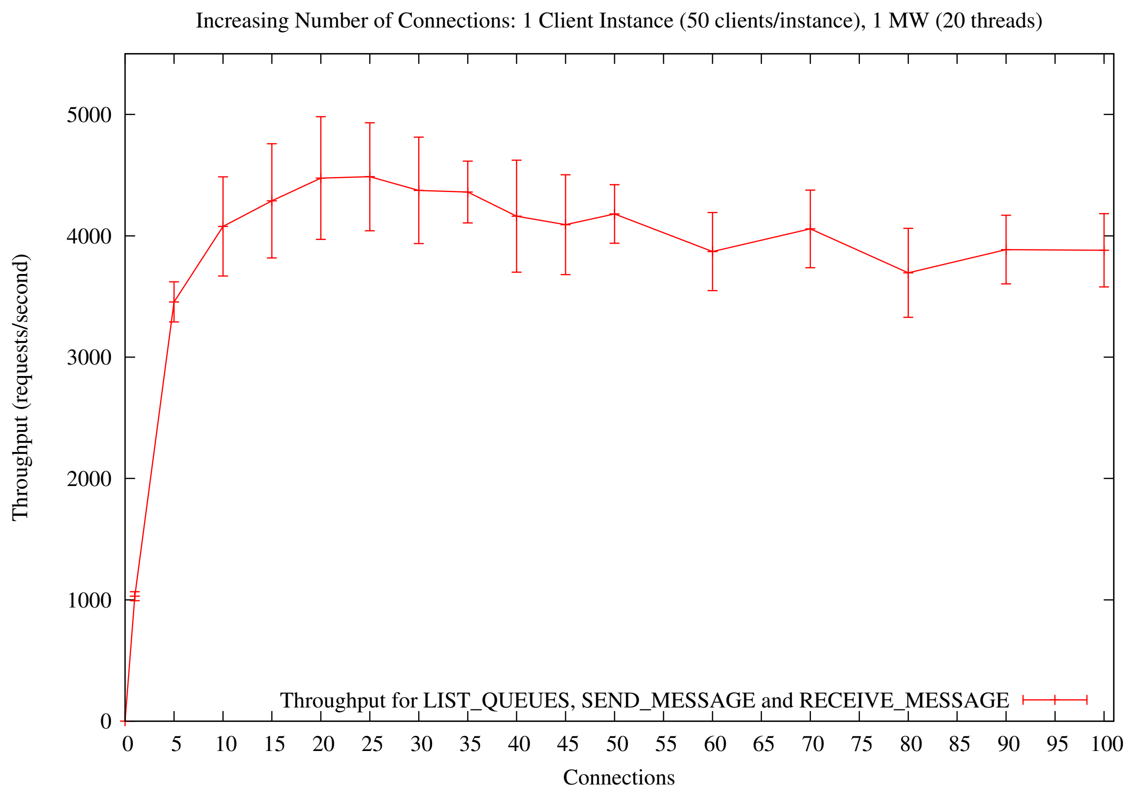
Increasing the Number of Connections

In this experiment we increased the number of connections to the database while everything else remained stable.

Configuration: The experiment was conducted with:

- 1 m3.large client instance with 50 clients
- 1 m3.large middleware with 20 middleware threads
- 1 m3.large database

Hypothesis: Since we have 50 clients and 20 middleware threads we would expect that by increasing the number of connections the throughput is going to increase and response time is going to decrease. This is excepted to happen until the number of connection reaches 20. Afterwards we except that the introduction of more connections is not really going to "help" the system since it only has 20 middleware threads and throughput is going to get saturated.



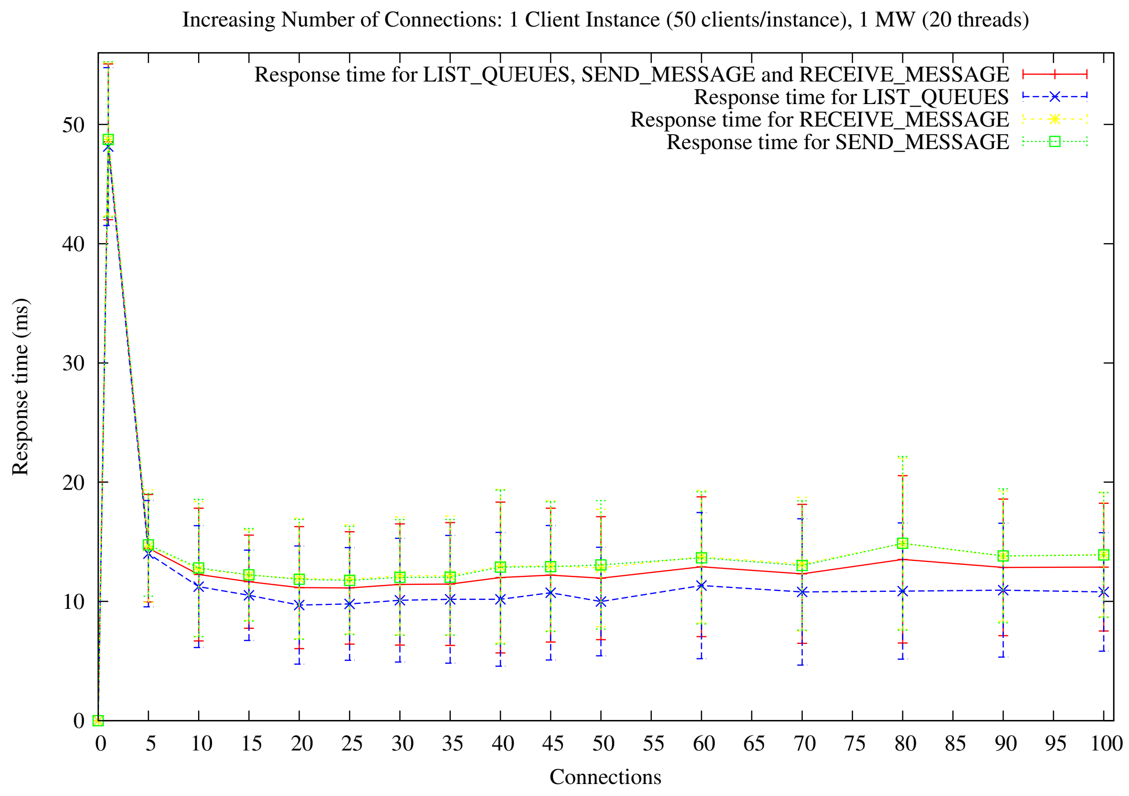


Fig. 21: Increasing the Number of Connections

Figure 21 shows the throughput and response time of our system while increasing the number of connections. The corresponding results were as expected up to 20 number of connections. After 20 connections we see a slow decrease in throughput which was not excepted. But still the reduced throughput can be explained by the way our connection pool is used. A call to *getConnection()* from our connection pool is always going to return a new connection if the number of maximum connections have not been reached yet and only then it is going to start reusing connections. So in our case when we have for instance 80 connections there are going to be created 80 connections in the pool although only 20 of them are being used at a specific point in time. This means that the database is going to be overwhelmed with more new processes: “As we do not know ahead of time how many connections will be made, we have to use a master process that spawns a new server process every time a connection is requested. “[21] Making the database slower has as a consequence of slowing up our system.

Increasing the Number of Threads

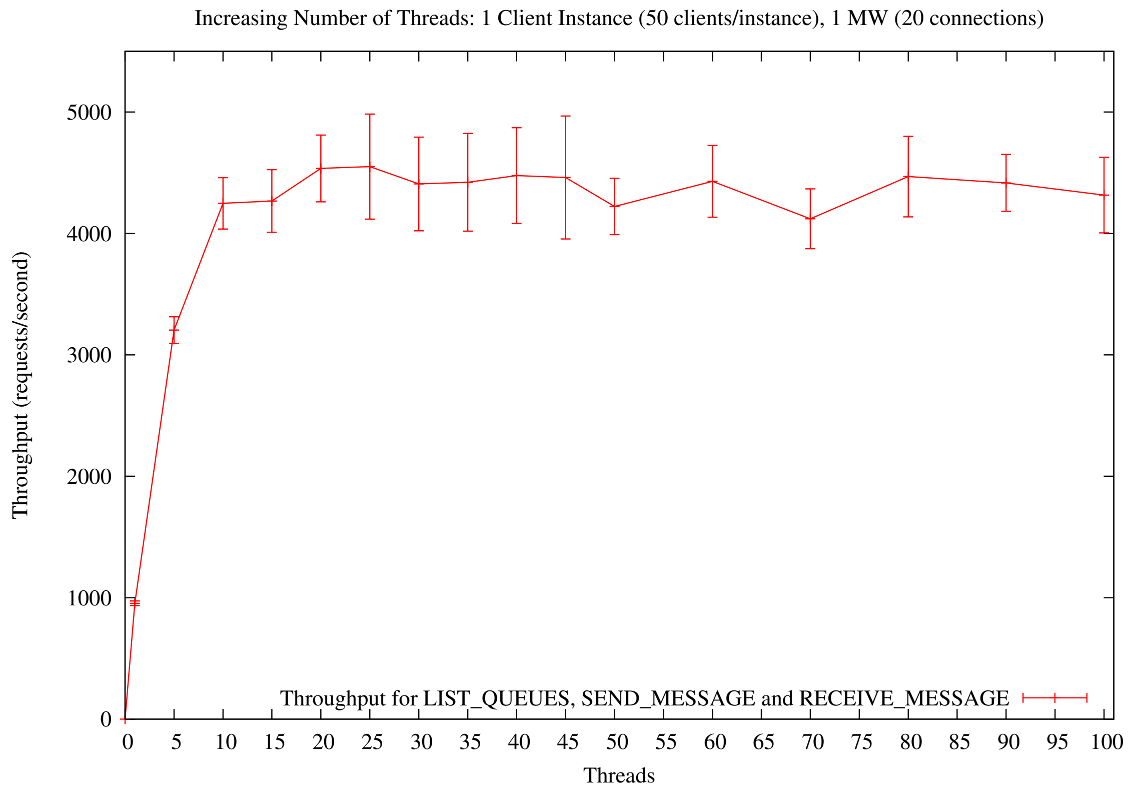
In this experiment we increased the number of middleware threads while everything else remained stable.

Configuration: The experiment was conducted with:

- 1 m3.large client instance with 50 clients
- 1 m3.large middleware with 20 connections
- 1 m3.large database

Hypothesis: Because we have 20 connections in our system we would except throughput to increase as we increase the number of threads since with more thread we will be able to utilize the given connections. In our previous experiment where we increased the number of connections we saw that there was still an increased throughput with 25 connections. So we would except here as well that a bit less than 20 threads will already have utilized our 20 connections.

[21] http://www.postgresql.org/docs/9.3/static/connect-estab.html



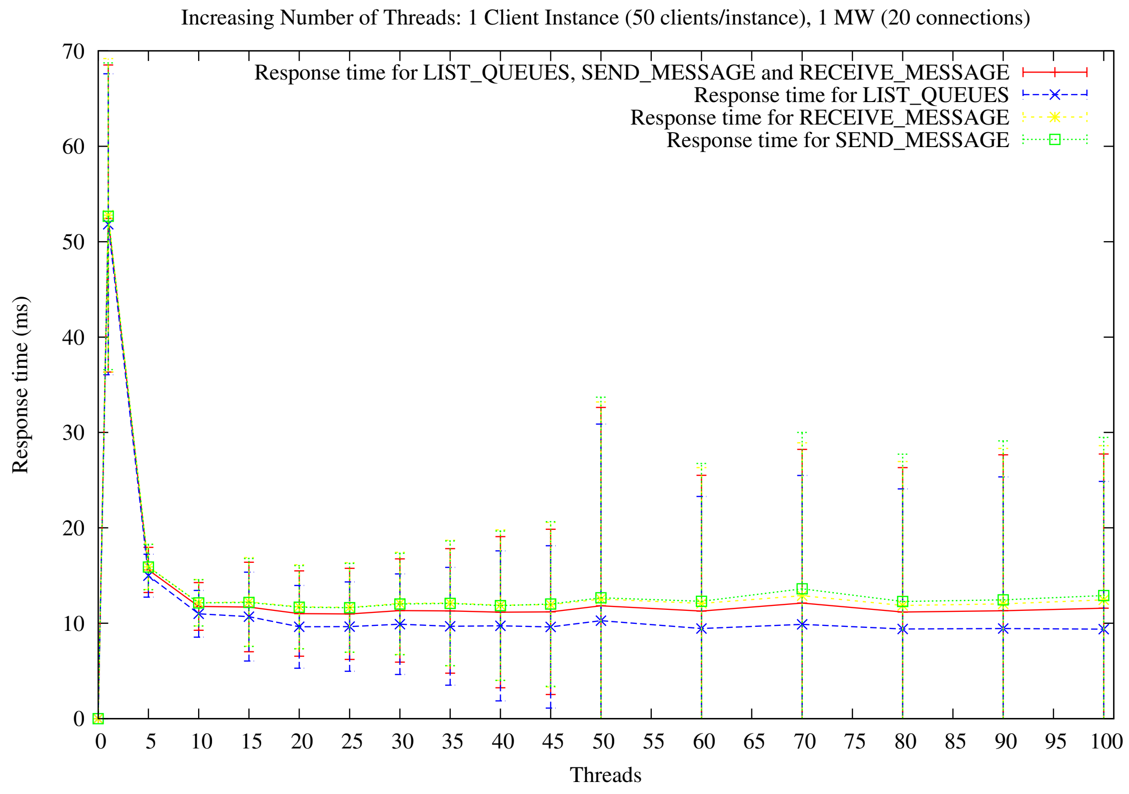
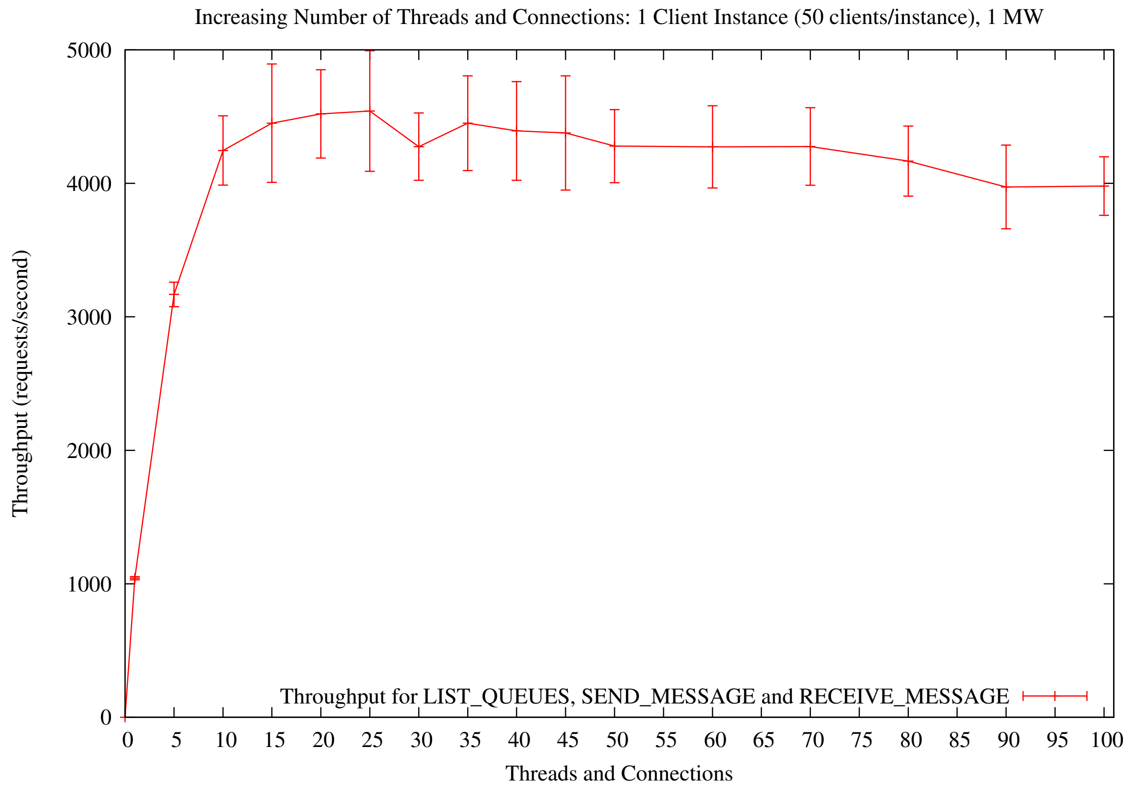


Fig. 22: Increasing the Number of Threads

As we can see in Figure 22 the throughput increases as excepted

Increasing Both Threads and Connections



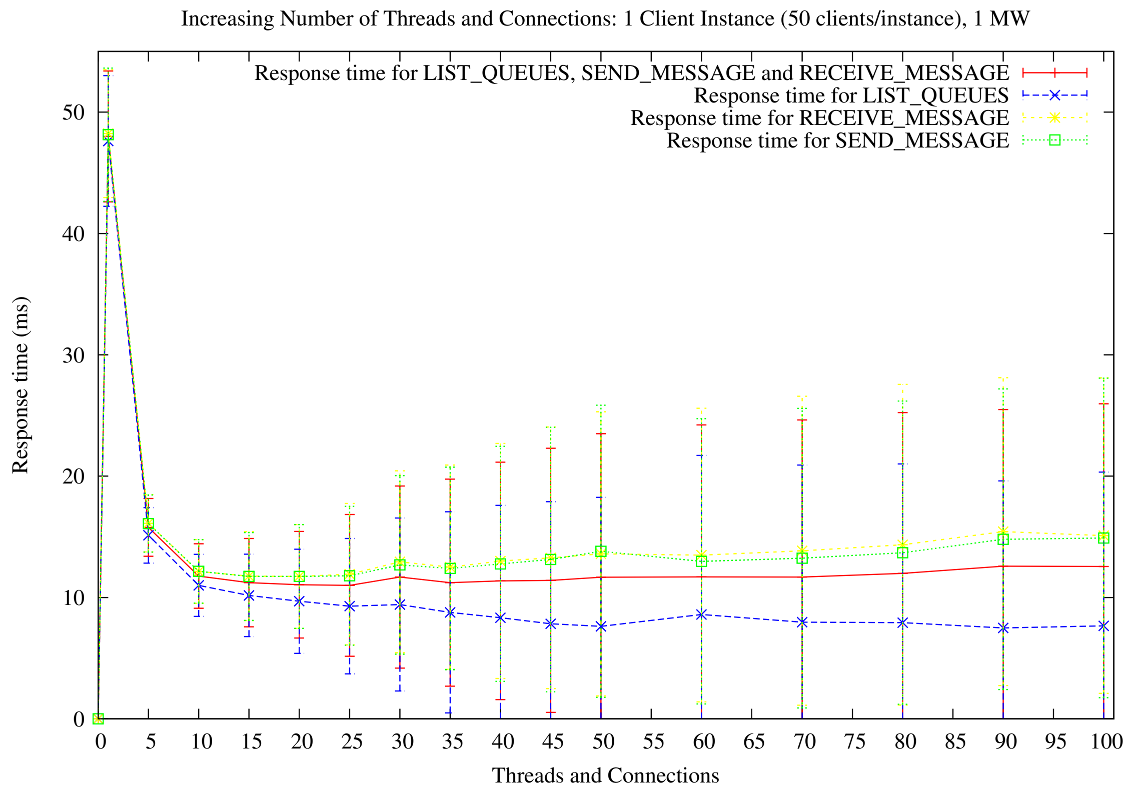


Fig. 23: Increasing Both the Number of Threads and Connections

Number of Clients in one Middleware

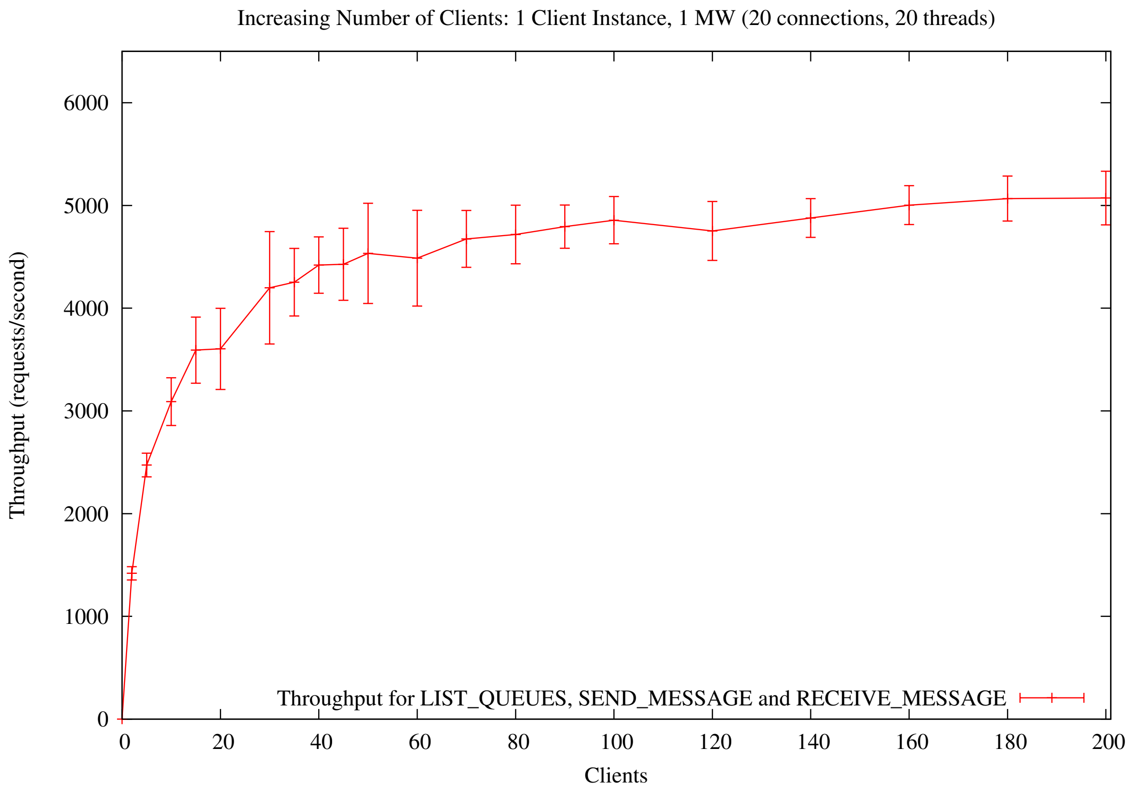
For this experiment we wanted to see what will happen if we increase the number of clients in our system.

Increasing Number of Clients

In this experiment we increased the number of clients with the following configuration:

- 1 m3.large client instance with a variable number clients
- 1 m3.large middleware with 20 middleware threads 20 connections
- 1 m3.large database

Hypothesis: Increasing the number of clients is expected to have a consequence on the throughput which is going to increase. By checking previous experiments on what was the CPU utilization of the instances running the clients we would assume that you can run quite some clients before having contention problems in your client's instance.



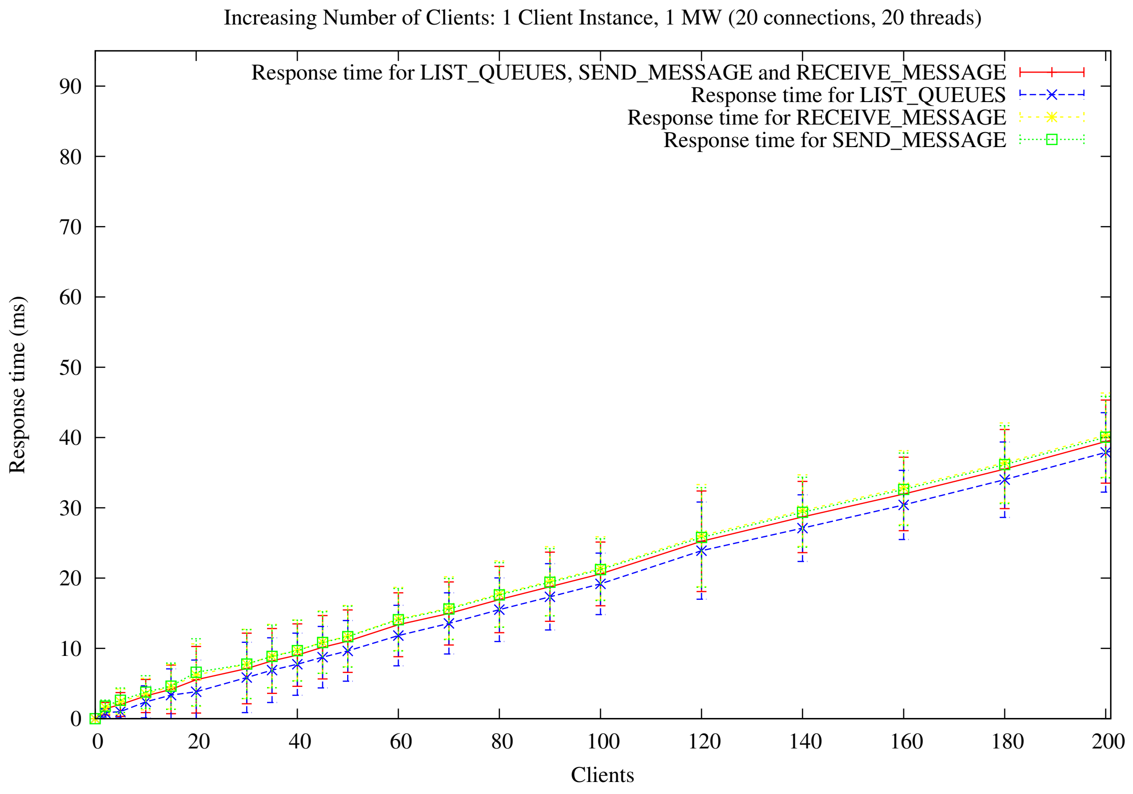


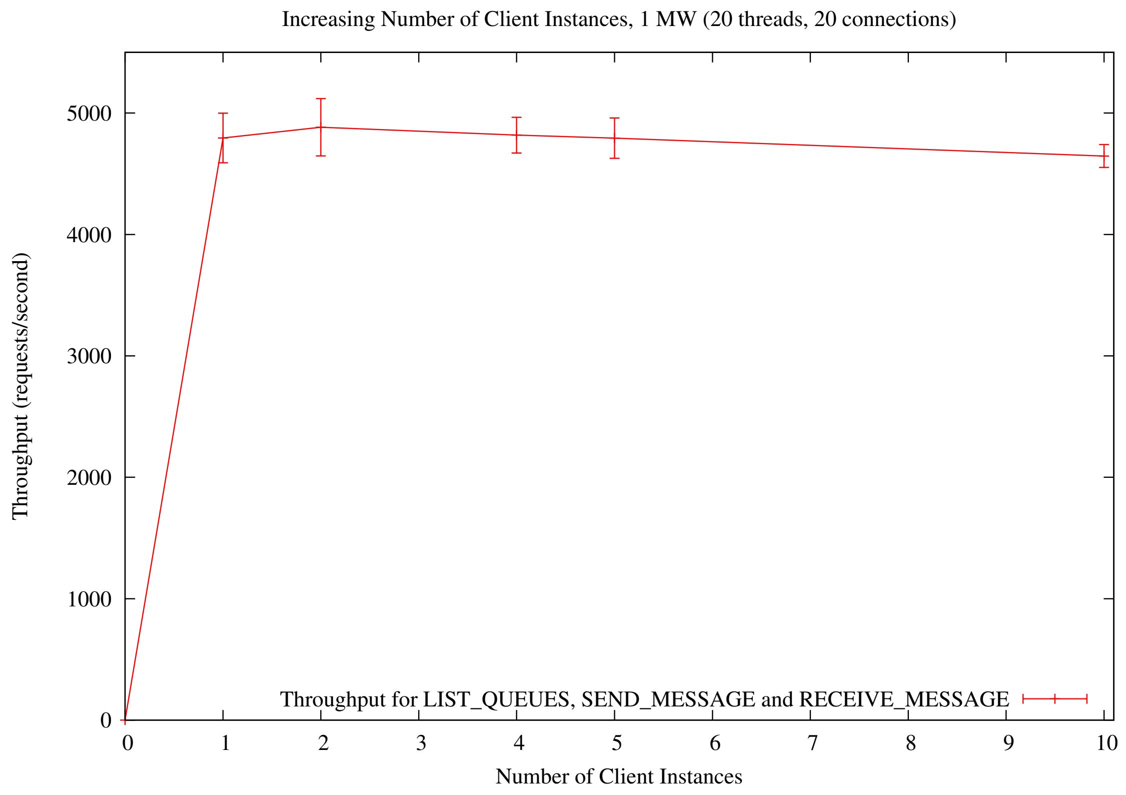
Fig. 24: Increasing the Number of Clients

Spreading Clients

and clients are not the bottleneck of our system that

In order to prove that in our previous experiment there was no contention in the clients that lead to the saturation of the throughput we conducted the following experiment where we took 100 clients and spreaded them across multiple client instances. Initially we started with one client instance having 100 clients, then 2 client instances having 50 clients each, etc. The configuration for this experiment was the following:

- Variable number of m3.large client instances
- 1 m3.large middleware with 20 middleware threads 20 connections
- 1 m3.large database



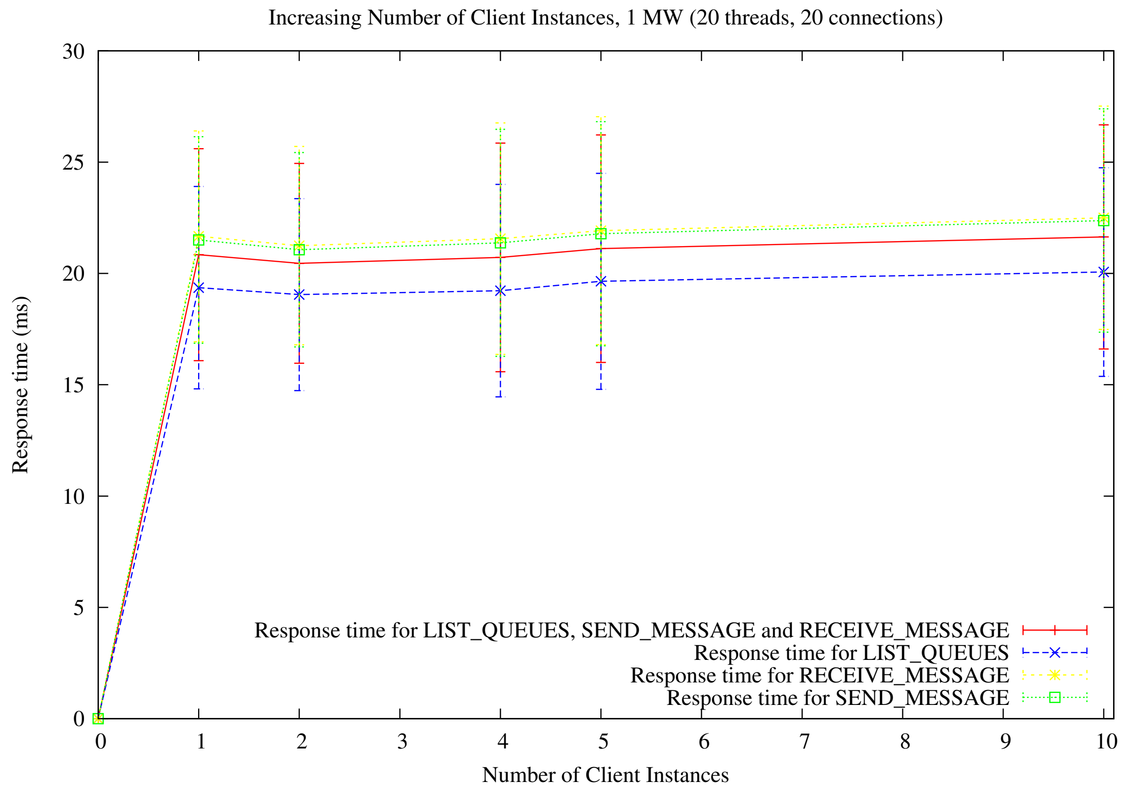


Fig. 25: Spreading Clients into Many Client Instances

As we can see

Speedup - Finding the Bottleneck

It is our belief that the bottleneck is the database. For sure it is not the clients as was explained previously. In order to prove our claim we are going to check what happens when increasing the number of middlewares. For this experiment we decided to use a r3.2xlarge instance for the database to show that even with such a powerful machine the database is going to utilize most of its CPU usage while increasing the number of middlewares.

clients and middlewares were m3.large ...

Speedup calculated based on response time.

Number of Middlewares	Average Response Time for all types of requests (ms)	Standard Deviation
1	98.3874	26.3294
2	59.6998	20.969
5	57.0587	21.3355

Number of Middlewares	Throughput for all types of requests (requests/s)	Standard Deviation
1	10222.4214286	191.01678284
2	16906.9833333	81.5825047383
5	18370.6571429	501.37283824

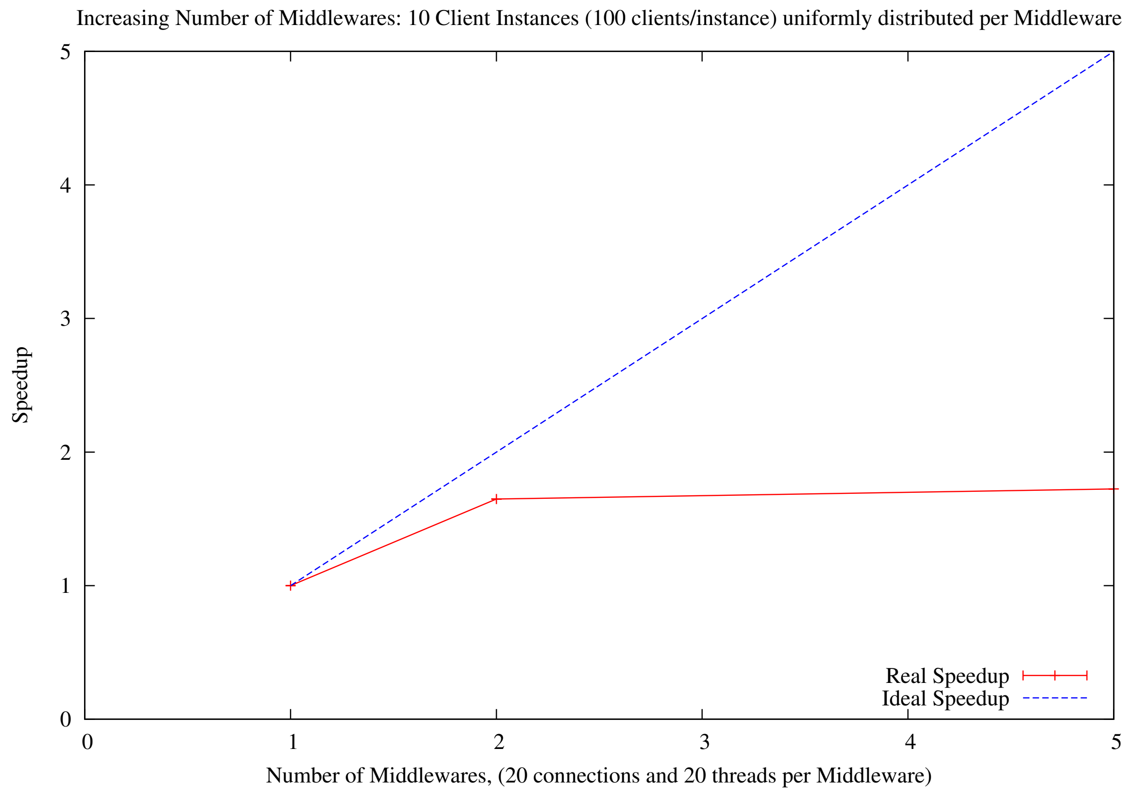


Fig. 26:

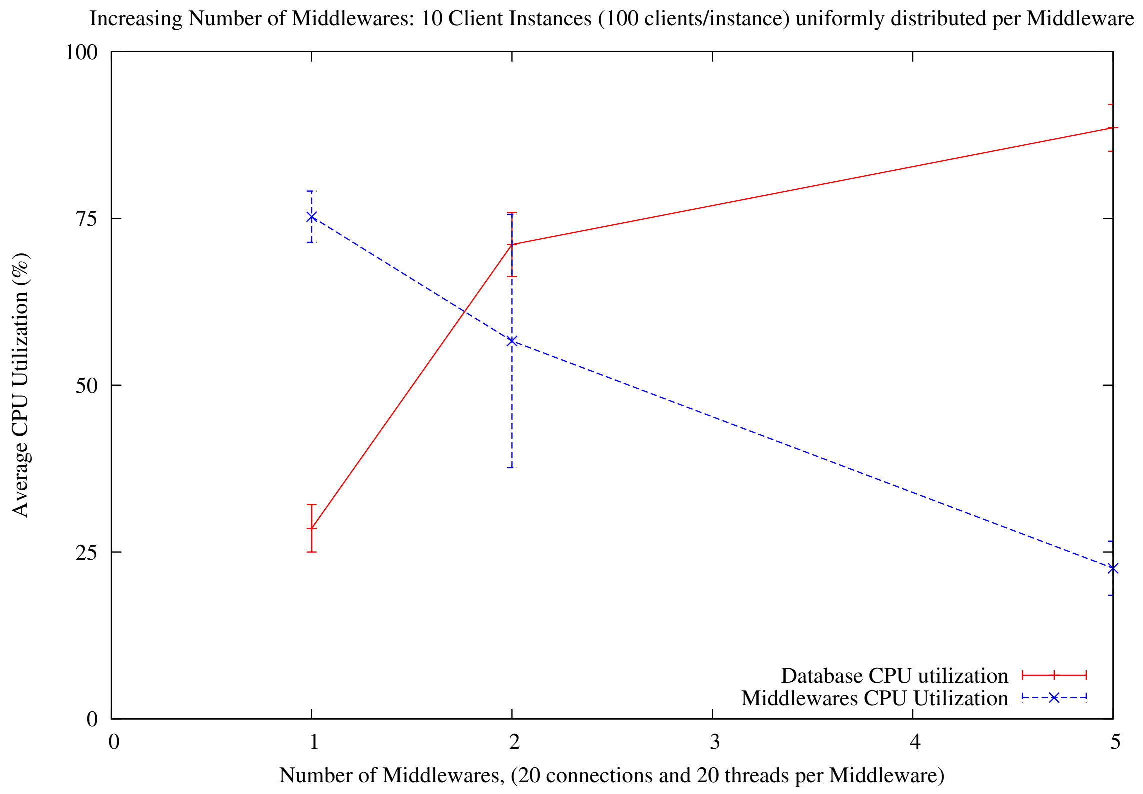


Fig. 27: CPU Utilization of Database and Middleware Instances

As we can see in Figure 27 the average middleware CPU utilization is reduced when we increase the number of middlewares while the database CPU utilization increases. It is our belief that is shows us the bottleneck in a lovely and nice way.

My next value would be to try with 10 middlewares but this was not possible because of EC2 limits on having more than 20 instances running on one availability zone.

Scale Out

clients and middlewares were m3.large ... Database was r3.2xlarge to show that the database as the bottleneck ...

had to increase number of connections to 200

Scale-up

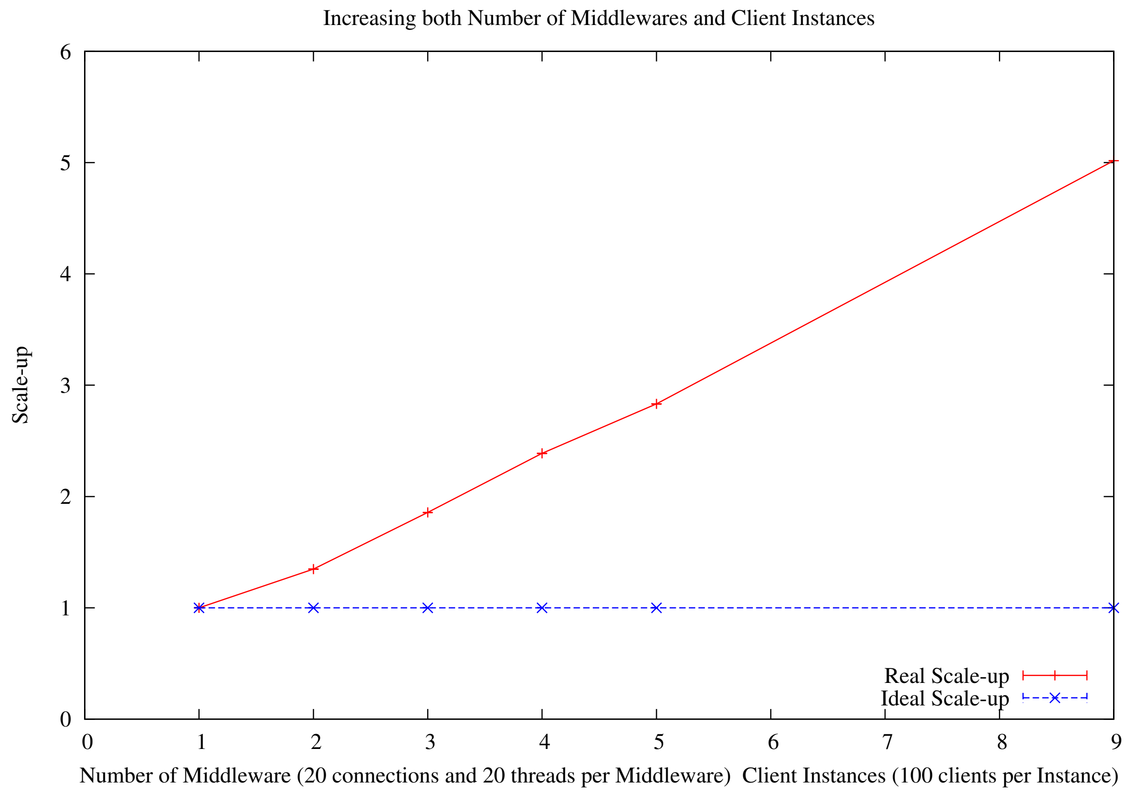


Fig. 28:

Scale-out

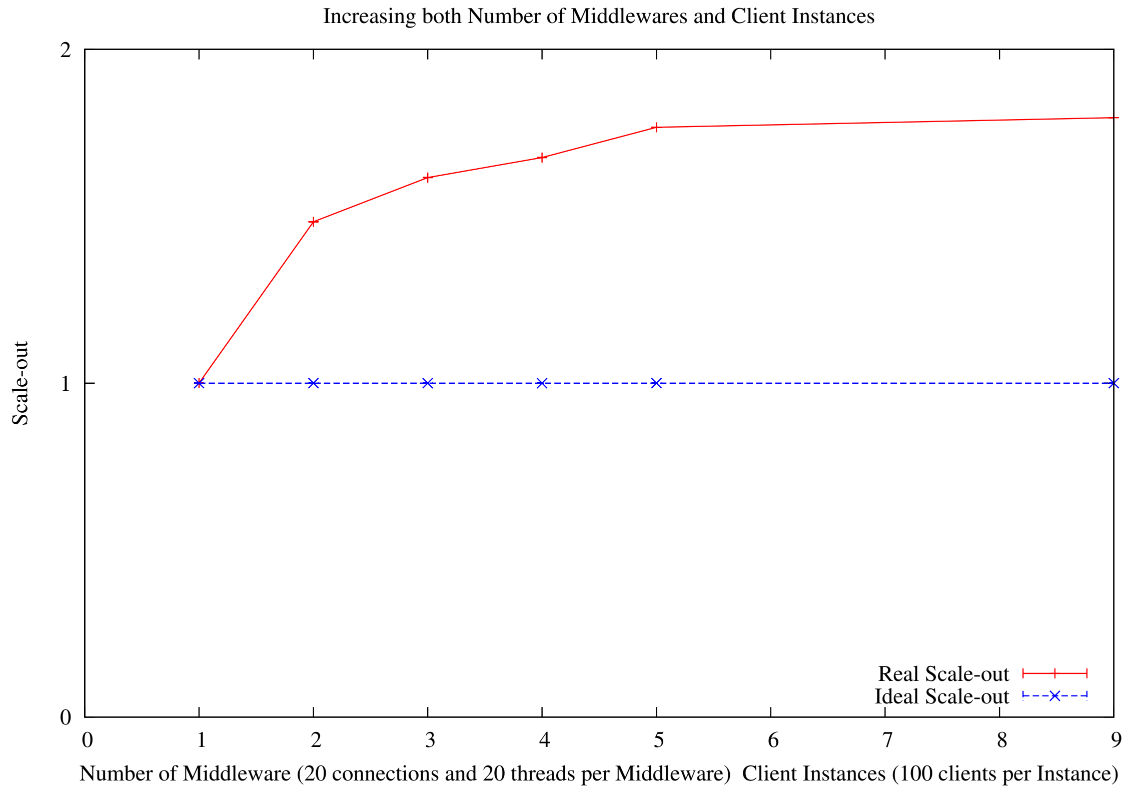


Fig. 29:

Encountered Problems

During some initial experimenting with the system we had the following problem, that seem quite naive in retrospective. We were changing only the database instance type to a better one, from m3.large to m3.2xlarge and the system's throughput was decreased! It came to our realization that the new launched database was running in the "us-west-2a" zone instead of the "us-west-2c" were the clients and the middleware were running. We were not aware that this was possible since we did not know about availability zones inside a region. This was fixed afterwards and all our experiments were executed in the same availability zone.

6 Conclusion

At the end if we could design the system anew we would not change many aspects of its design. But we would change a bit the experimental setup so it is easier to run more experiments and get back graphs instead of manually calling the *getResponse()* or *getThroughput()* methods and passing them to Gnuplot by hand. We we would also have followed a more iterative approach: changing something in the system, running the experiments, getting back the experimental results, changing something in the system, ...

We believe we had a good understanding of our system where we showed that the database is the bottlenech. We are also confident that we learned a lot about developing and benchmarking applications, some database internal characteristics, as well as deploying an application to the cloud.