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.

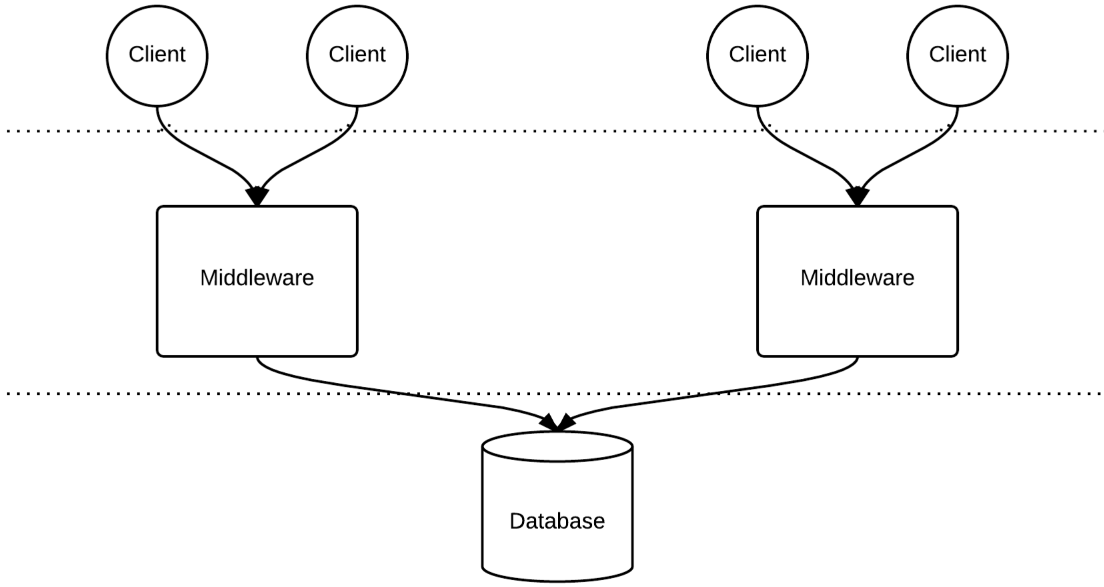


Fig. 1: The Three Tiers

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 aftewards 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.

- 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 :: containts 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 bewteen 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 waas 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. (TODO talk about text vs varchar) 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 arrrival 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*.

[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)

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.

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.

[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)

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.

What is being logged?

The only thingthat 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.

[6] http://jdbc.postgresql.org/download.html
[7] http://dag.wiee.rs/home-made/dstat/

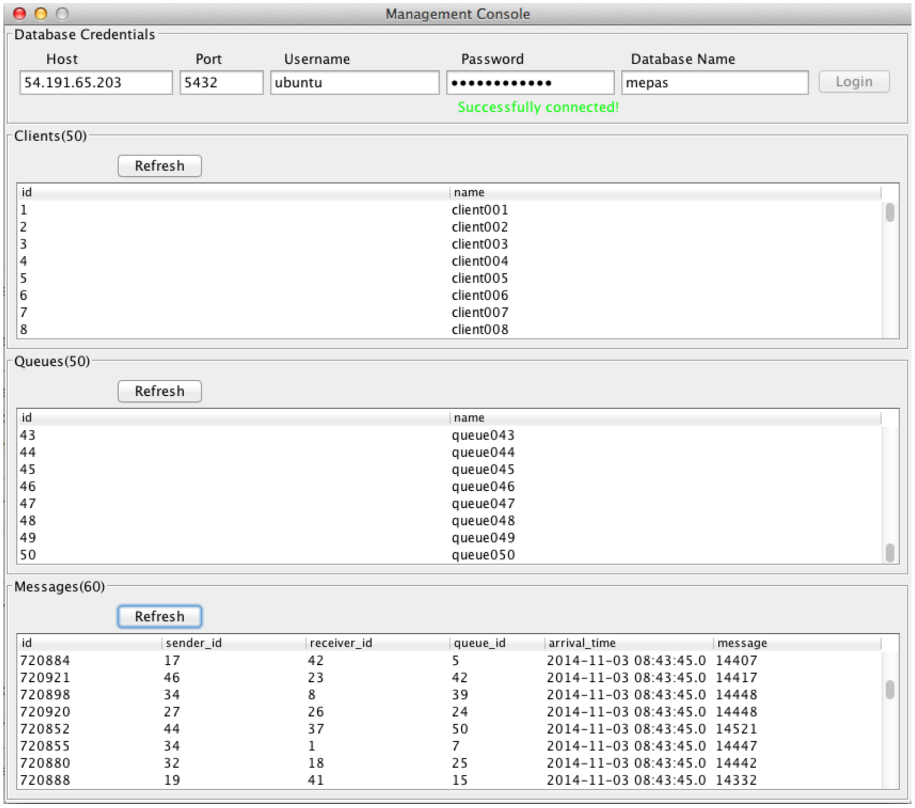


Fig. 6: Management Console Screenshot

Middleware

The middleware implements the messaging system, it receives requests/messages from clients and has to use the database in order to persist those messagess, 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 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 constrast 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 clien 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. Everytime 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 implemenation was achieved by using *InputStream*'s *available()* method that can return the number of bytes that can be read whithout 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.

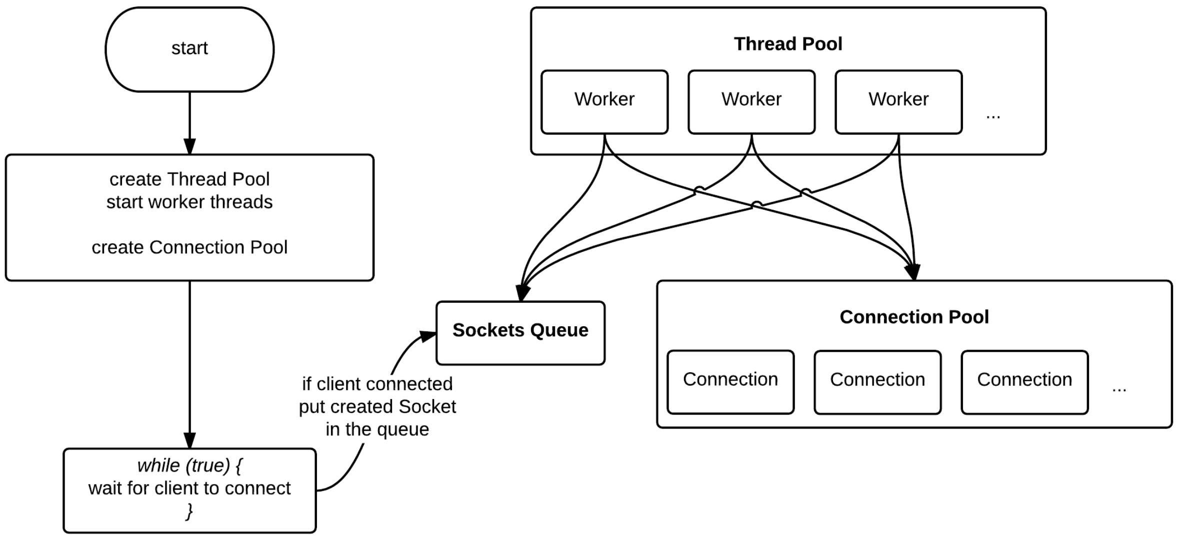


Fig. 7: Middleware Architecture

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.

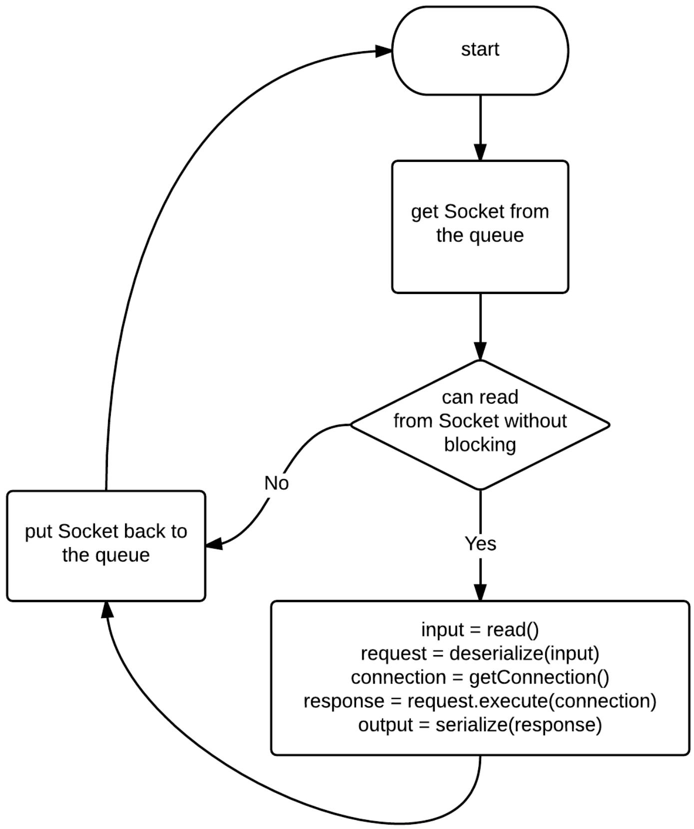


Fig. 8: Worker Thread of the Middleware

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 instatiated 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 implemenation 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 tha 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 created length is transformed to bytes as well, to exactly 4 bytes (1 *int*). The transformed length is concatenated with the serialized byte array, with the length being in the front. This concatanted array is what is being transmitted between the client and the middleware. The length is added to 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 formats 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, it was done 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 contatining 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.

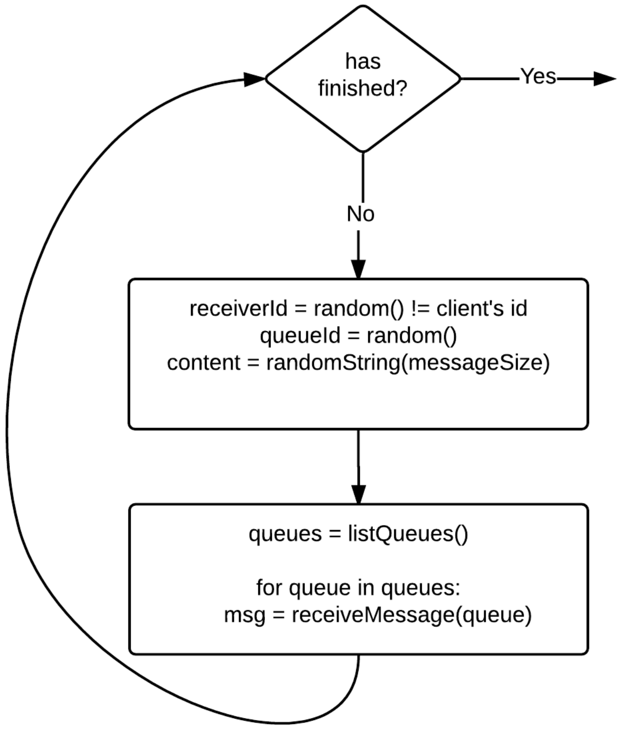


Fig. 11: Client's Workload

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 diferrent[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.

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

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_MESSSAGE” 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.

```
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 assing 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.

General Remarks

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 targes:

- 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.

Initially we were planning to test our system using the Dryad cluster which has Java 7 installed. For this reason wes 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.

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.

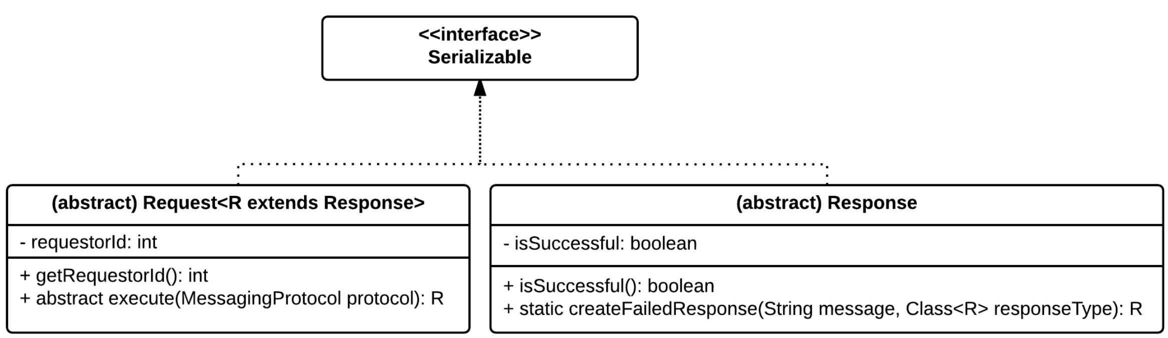


Fig. 14: Request and Response Classes

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 overriden 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 ... '.

3 Testing

Correctness of our system was of foremost importance, therefore testing played an important role while developing the system. Parts of the system have been tested with unit tests. 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 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 thoughroughly.

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`

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

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.

TODO mention other isolation level

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

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 reponses 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

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 paranthesis, 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 intiailly lead to problems because a client could have tried to send a message to a client that is not in the syste 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 afforementioned 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 Experiments Analysis

The experimental setup was solely done using Python together with these packages: boto[11] for interacting programmatically Amazon Elastic Compute Cloud (EC2), pexpect[12] for connecting to the EC2 instances used in the experiment and executing the appropriate programs, psycopg2[13] for connecting to the PostgreSQL database and getting it ready for our experiments.

INFORM: only once a middleware is used per instance

Setup

Before we presenting the exact details on our experimental setup we present in Figure the general overview of how an experiment is conducted.

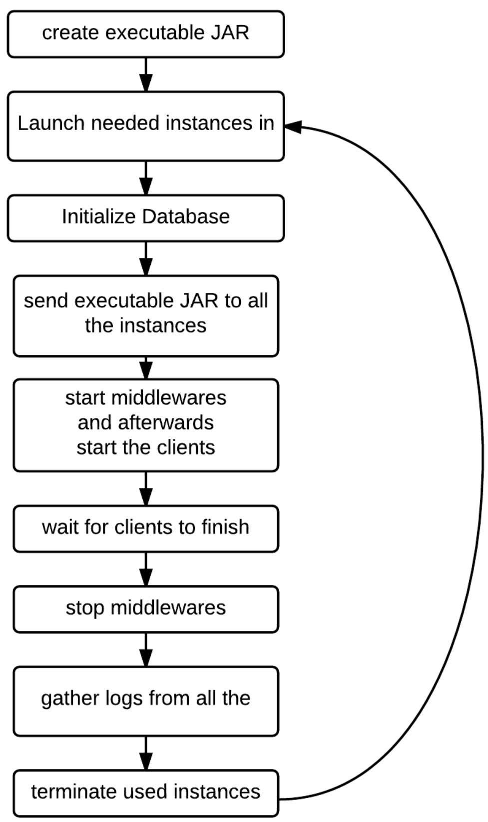


Fig. 17: Flow of an Experiment

[11] https://github.com/boto/boto
[12] https://github.com/pexpect/pexpect
[13] http://initd.org/psycopg/

All the experiments were executed in the Amazon Elastic Compute Cloud (EC2).

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 need to set up the database in a correct state before the beginning of an experiment.

The experimental code can be found in `ExperimentRunner.py`, there the code exists.

In order for the experiment to run notice that the following have to been done in the machine doing 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[14]. By doing so a user does not have to manually press “Yes” when a host is unknown.
- The experiment runner reads the database's password

Setting up the Database

This is coolness!! SHOULD BE hre

Wha

experimentRunner (careful to have enough space for this instance 200GB just to be sure)

-> create .pgpass file, install postgresql ?? is needed

All the experiments were conducted in Amazon EC2.

In order to retrieve the IPs of the instances we used boto[15]. By naming the instances with “client”, “middleware” or “database”

The three most important classes needed for the experiments are the *Client*, the *Middleware* and the *Database* class. Let us see each one of them.

Talk about pexpect and how awesome it is!

[14] http://askubuntu.com/questions/87449/how-to-disable-strict-host-key-checking-in-ssh
[15] https://github.com/boto/boto

EC2 Instances

Initially we created an instance based on "Ubuntu Server 14.04 LTS (HVM), SSD Volume Type" Amazon Machine Image (AMI) and we installed the following:

openjdk-7 jdk
dstat
iperf
htop

We created an image of our created instance to be used for the generation of future instances. The created image can be used to create instances for clients and middlewares.

We created a general security group to be used by all instances that allowed everybody to pass by.

Similarly we installed in another instance PostgreSQL, dstat ... to be used for databases and we created an image of it. For the database the following things need to also be changed (Configuration file allowing outside hosts ...)

After doing so we had the instances ready for the experiments.

Do the following http://superuser.com/questions/331167/why-cant-i-ssh-copy-id-to-an-ec2-instance ssh-add privatekye file to login to the ec2 instances without having to do 'ssh -i ~/... ' every time!! AWESOME!!

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 trace somehow is a unique experiment this method was created to ..
- *getThroughput()*: returns througput.
- *getResponseTime()*: find the response time.
- *getTimeSpentOnEachComponent()*: finds the average time spent on each component of the middleware. For example, the amount of time that was spent waiting to get picked by a worker thread, the amount of time waiting to get a connection, etc.

All the afforementioned functions use extensively UNIX commands like awk, sed, grep, cat, wc and others. This was done to achieve better performance. 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 to count them up. Another example would be removing the warm-up and cool-down of an experiment which by using awk can be simply done as follows:

awk -F'\t' '$1 >= warmUpInSeconds && $1 <= (lastTimeInSeconds - coolDownInSeconds) { print; }' file.

5 Experiments

BALE HYPOTHESIS .. .kai blah

All successfull receiveals bla blah

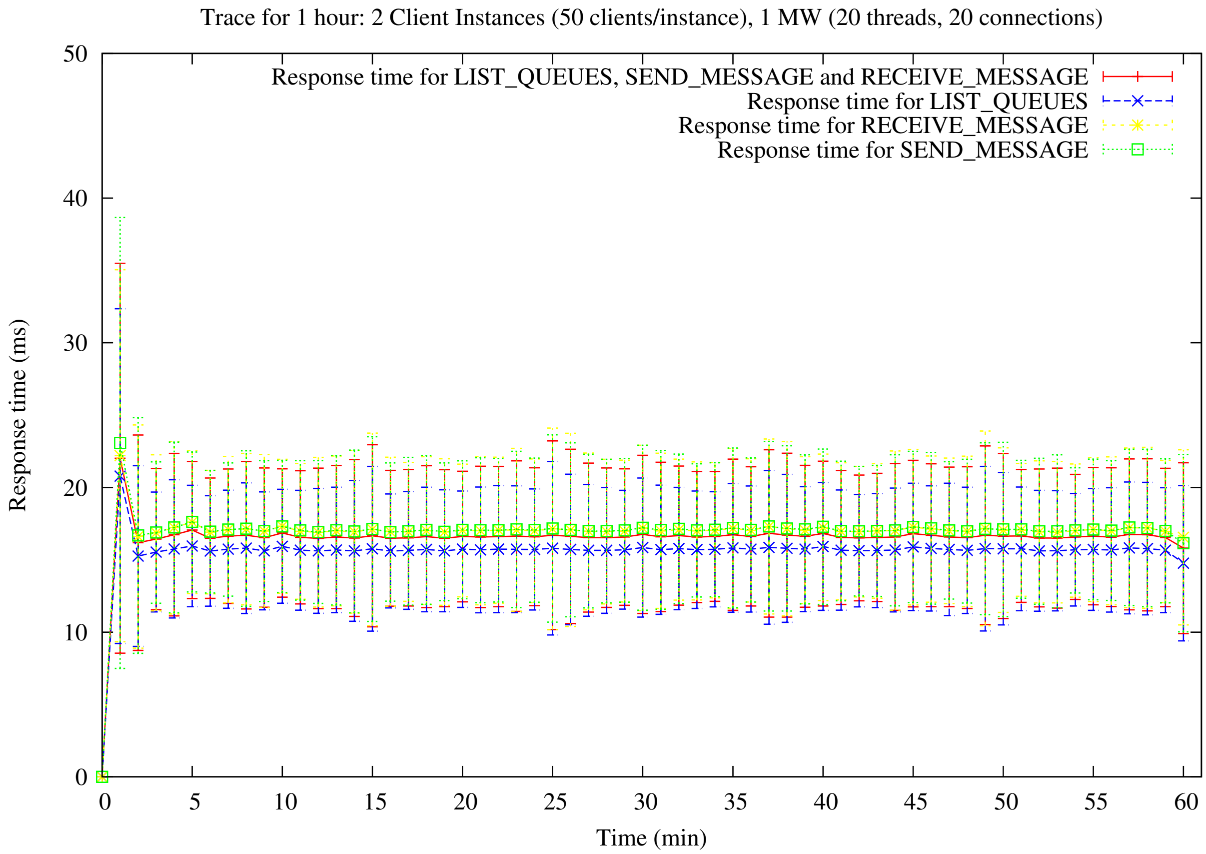
—————— In the report mention that in the throughput all the requests were successful, I had no failed responses.

ALl the response times and througputs correspond to the client side unless stated otherwise.

Stability

Clients and MW was t2.small and db t2.medium.

As can be seen in Figure ... Used getTrace method from ResultsReader.py to extract the data. Response time was averaged over the interval of one minute. While throughput was calculated per second and averaged over the interval of one minute. I.e. In minute i corresponds to the averaged time from (i - 1, i]. The data were generated using the getTrace from ResultReader



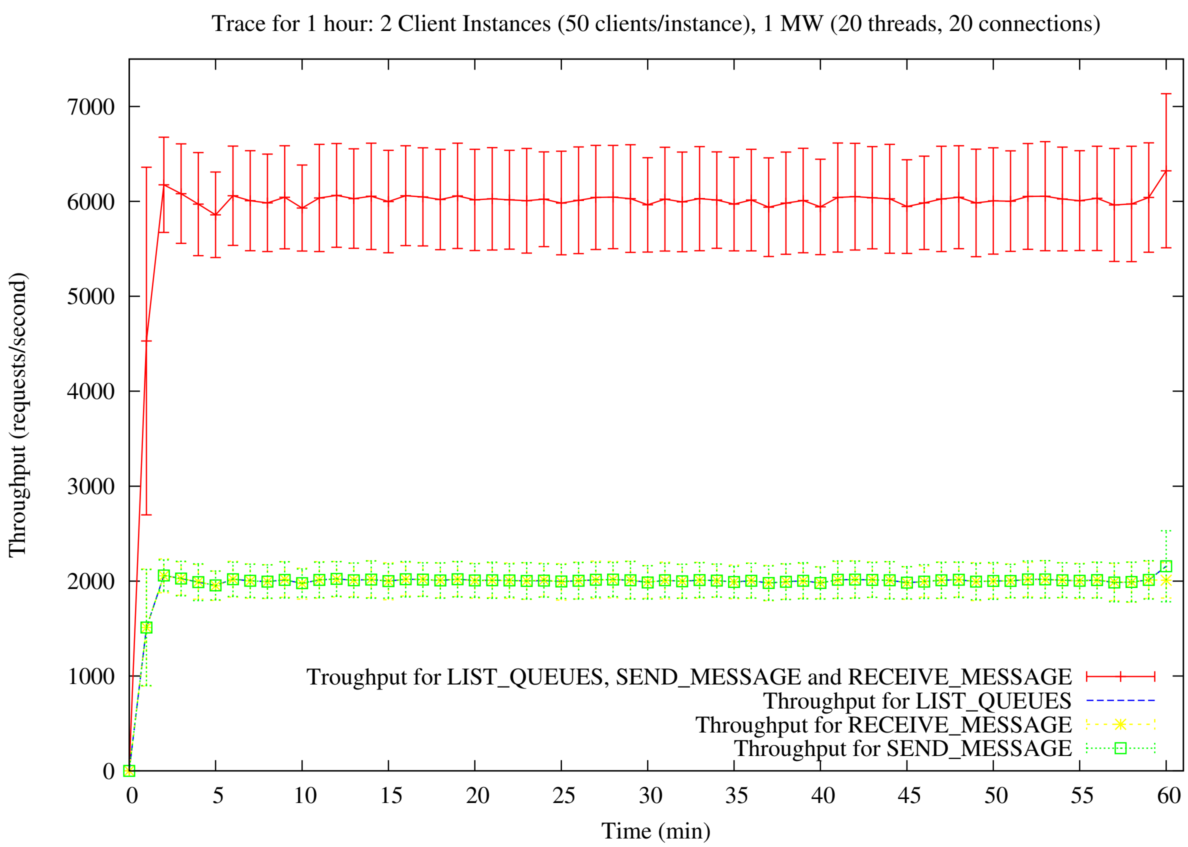


Fig. 18: Response time and throughput of an one hour trace with 2 client instances (50 clients/instance) and 1 middleware instance (20 threads, 20 connections)

BECAUSE THROUGHPUT goes like this I decided to just plot the througput for all the three types of requests siince most througputs for the requests are the same. For every send you have a list and for every send a receive.

Let's see where time was spent

adfs	dsafasdf
CONNECTION: 0.000689706, 0.216034	
REQUEST: 3.23399, 2.38918	
IN WORKER THREAD QUEUE: 13.3207, 4.17608	
TIMES A SOCKET IS WORKED for a REQUEST TO BE READ: 1, 0	

TIMES TO ENTER: 1.00164, 0.163046 TIMES (NOTHING) INSIDE: 0.0201363, 0.570286 TIMES (DOING) INSIDE: 3.33539, 2.51786

Averages and SD

25 17.10725 5.51819 RECEIVE MESSAGE

25 17.1355 5.543025 SEND MESSAGE

25 15.7621 4.689385 LIST QUEUES

16.668 average response time in total Gia ola ta requests

Talk about list queues(Since this was the trace and we just wanted to verify that our system is stable when it is being executed for a fair amount of time we did not really do any assumptions about the results. There are some things that actually can be asily explained, the time to receive a connection is technically 0, this is because we have the same amount of worker threads to connections. The time waiting for a worker thread is quite high and was to be expected since we have 100 clients and only 20 workwer threads. So at any point in time 80 clients connections could possilby be waiting. Network time is also really low and this makes sense since the throughput between two instances is (iperf). List queues check db time and why they are faster, not doing so much with the database ... verify this by checking request time for all types of requests.

DB REQUEST per type of request

grep "DB REQUEST\tLIST_QUEUES" middlewareInstance1/m*.csv | awk -F'\t' '{ sum += $2; n++; } END { if (n > 0) printf sum /n }' 2.38218

grep "DB REQUEST\tSEND_MESSAGE" middlewareInstance1/m*.csv | awk -F'\t' '{ sum += $2; n++; } END { if (n > 0) printf sum /n }' 3.71801

grep "DB REQUEST\tRECEIVE_MESSAGE" middlewareInstance1/m*.csv | awk -F'\t' '{ sum += $2; n++; } END { if (n > 0) printf sum /n }' 3.60231

TIME WISE (for all request ... < 50ms are 99.7% of the requests

< 50 0.997697

< 25 0.97127

< 20 0.869102

< 23 0.952159

< 22 0.9357

VERIFY with **closed system** STUFF

Warm Up and Cool Down

All the experiments data points were executed for 10 minutes, for the following results the warm up that was removed was 2 minutes while the cool down was 1 minute.As we saw from the trace ... This was done by using sed.

Increasing the Message Size

througput was averaged per 20seconds

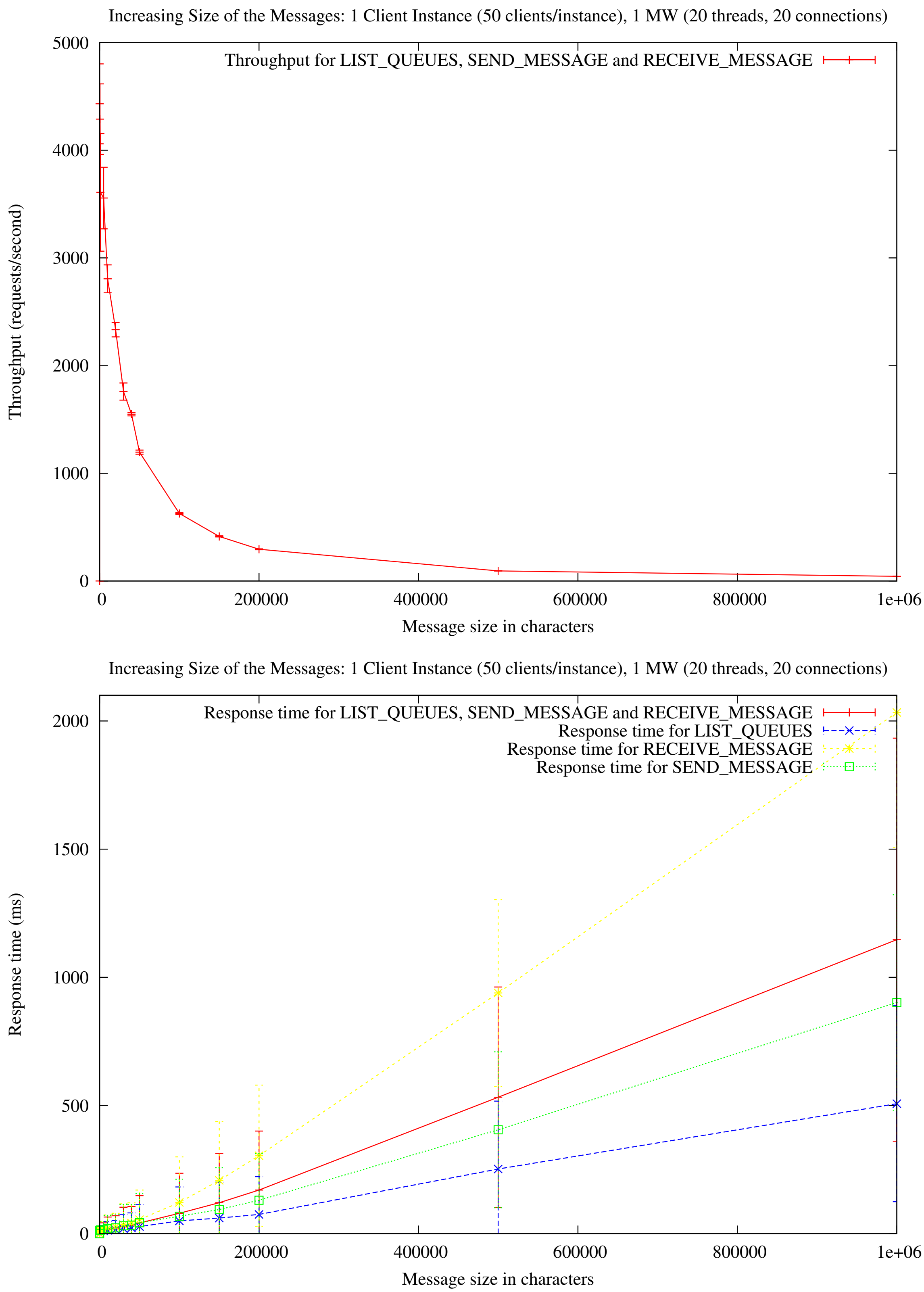


Fig. 19: Increasing the Message Size

When running experiment increase number of message I got the following error from the client logs:

ch.ethz.inf.asl.exceptions.MessagingProtocolException: failed to send messageERROR: could not extend file "base/16389/16427.6": No space left on device Hint: Check free disk space. Where: SQL statement "INSERT INTO message (sender_id, receiver_id, queue_id, arrival_time, message) VALUES (p_sender_id, p_receiver_id, p_queue_id, p_arrival_time, p_message)" PL/pgSQL function send_message(integer,integer,integer,timestamp without time zone,text) line 3 at SQL statement at ch.ethz.inf.asl.client.ClientMessagingProtocolImpl.receiveResponse(Unknown Source) at ch.ethz.inf.asl.client.ClientMessagingProtocolImpl.sendMessage(Unknown Source) at ch.ethz.inf.asl.client.ClientRunna
Source) at ch.ethz.inf.asl.client.ClientRunnable.run(Unknown Source) at java.lang.Thread.run(Thread.java:745) ubuntu@ip-172-31-14-163:~/logs$

increased size of database

2014-11-05 21:29:30 UTC HINT: Consider increasing the configuration parameter "checkpoint_segments". 2014-11-05 21:29:32 UTC LOG: checkpoints are occurring too frequently (2 seconds apart) 2014-11-05 21:29:32 UTC HINT: Consider increasing the configuration parameter "checkpoint_segments". (in the 40000 characters per message)

DISABLE CONSTRAINT ...

$2^{k(=?TODO)}$ Experiment

Before starting I would like to talk about love the one and only one.

Factors

number of middleware threads

number of connections

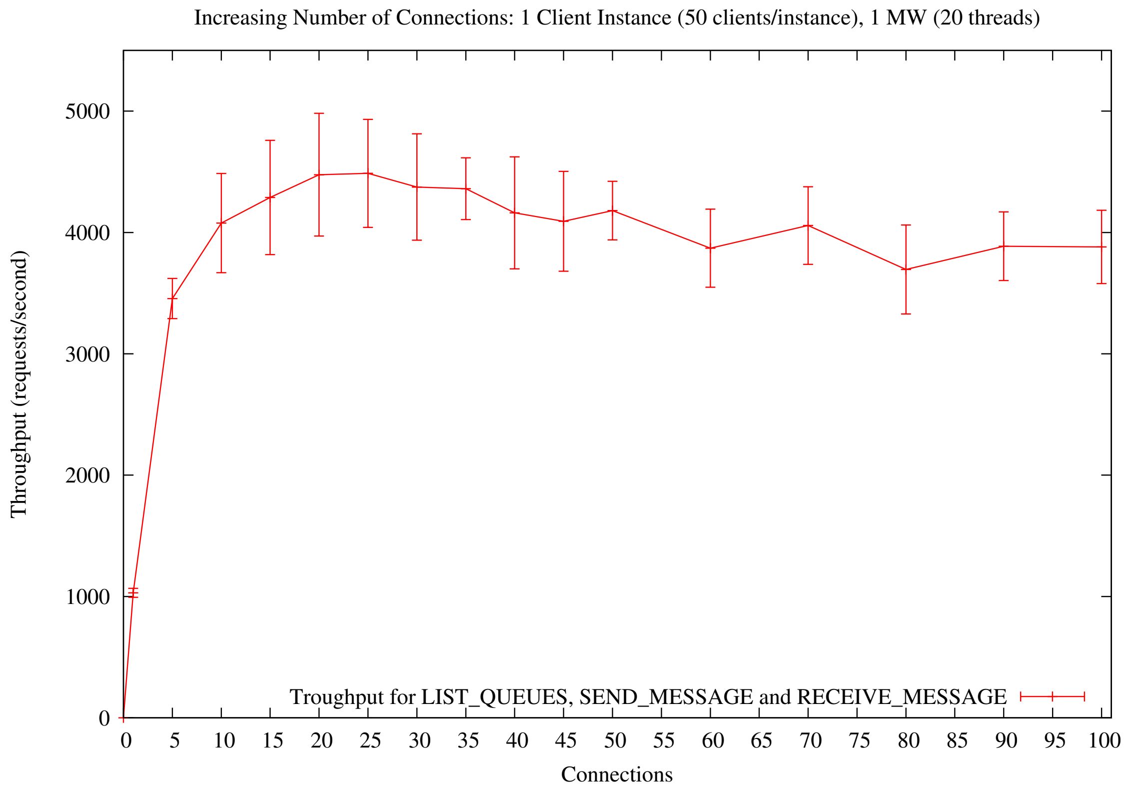
instance type of middleware

instance type of database

dfs

16 experiments in total for 10 minutes each

Increasing the Number of Connections



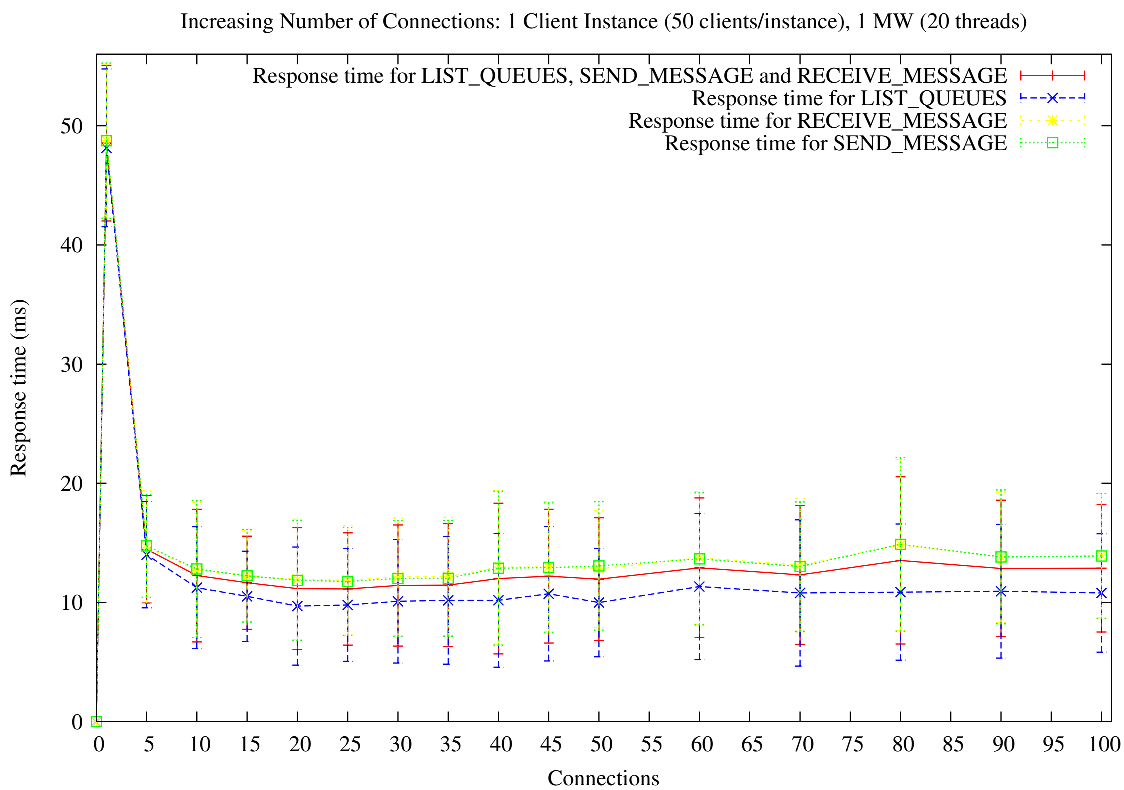


Fig. 20: Increasing the Number of Connections

Increasing the Number of Threads

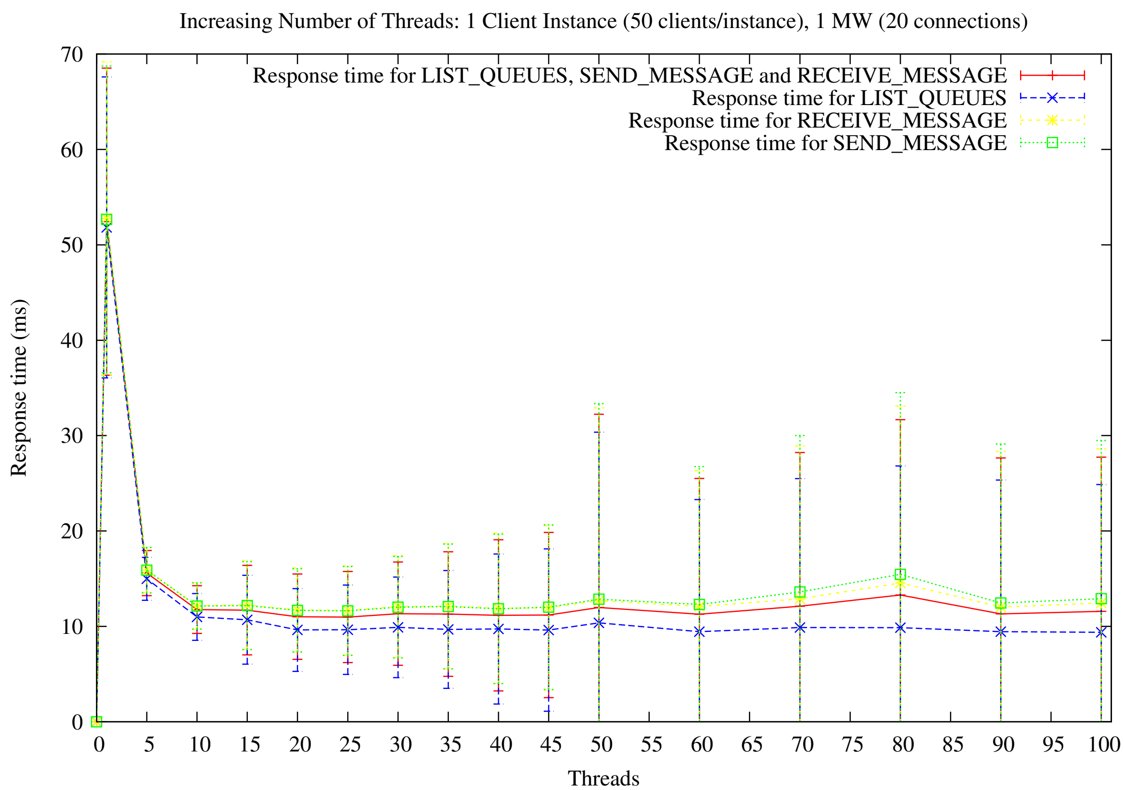
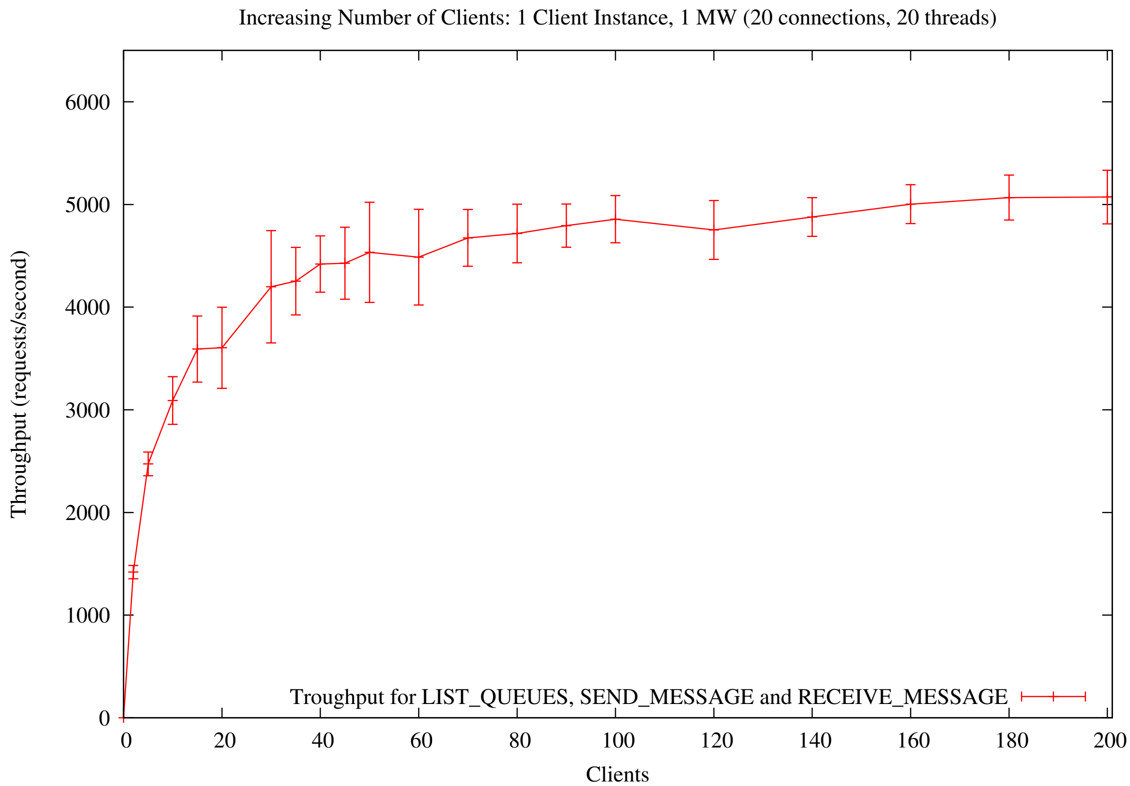


Fig. 21: Increasing the Number of Threads

Increasing the Number of Clients



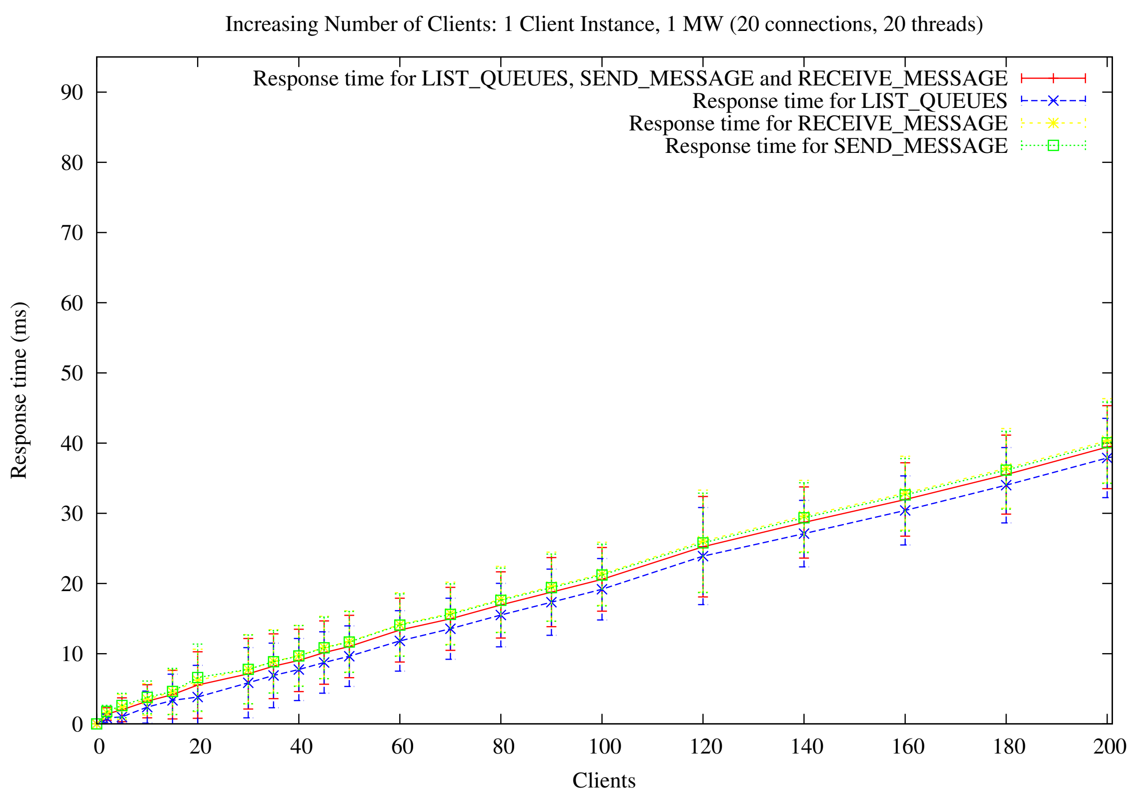
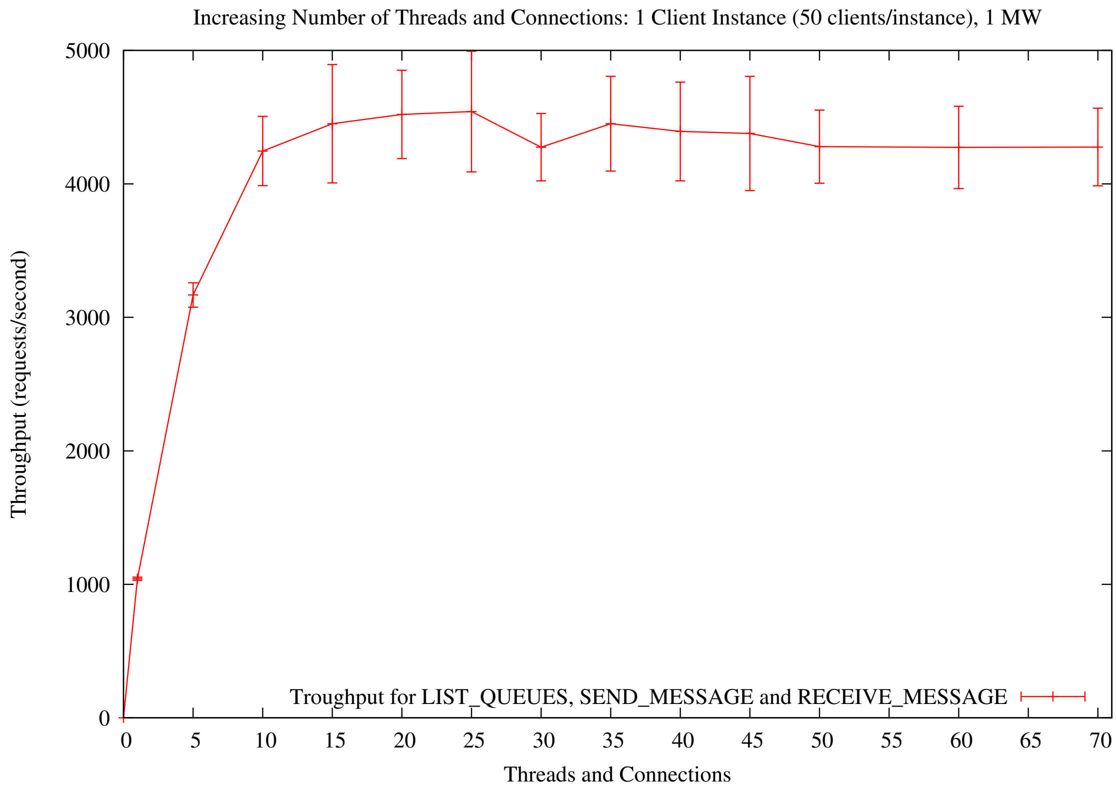


Fig. 22: Increasing the Number of Clients

Increasing Both Threads and Connections



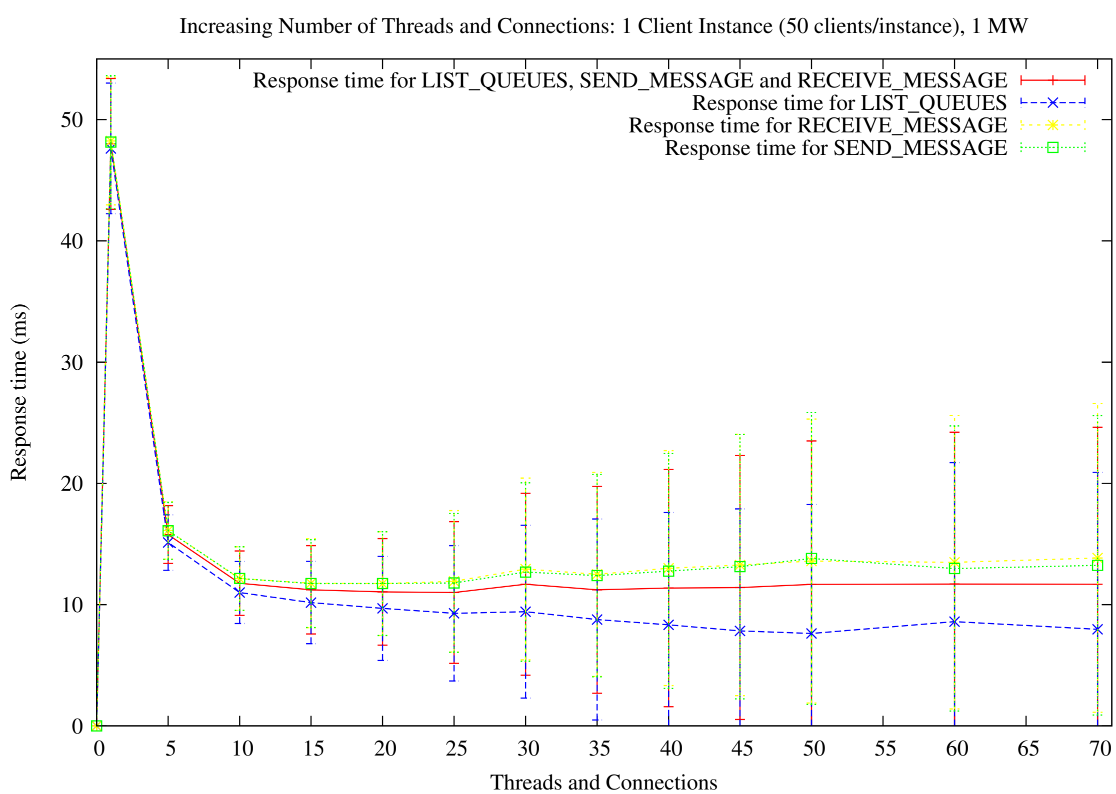


Fig. 23: Increasing Both Threads and Connections

Encountered Problems

After running the 2^k experiment we immediately noticed that when using the m3.xlarge instance type for the database the system as a whole was slower, i.e. had less throuput than when having a database of t2.medium instance type. It was expected to generally have less throughput for the clients than what we had when we executed the trace since now we have half the clients. The results of the above experiments were somehow unexpected. Why is throughput decreased when changing the database instance type form t2.medium to m3.xlarge in all the experiments except te ones where the number of threads and connection is 20? It was excepted that by just changing one component of the system with a better one, the system as a whole would become better but throughput instead decreases. And also why is it that with the number of threads and connections being 20 the throughput is better when using a better instance type for the database? A guess for this would be that by having 20 database connections and 20 threads, all of the database connections can be utilized and since the database can respond faster the system can become faster. But then why isn't it faster when having 10 threads and 10 connections?

In order to find out what was going on we decided to have a better look on the differences between the experiment 1 (10 threads, 10 connections, MW t2.small and db m3.xlarge) and experiment 9 (10 threads, 10 connections, MW t2.small and db t2.medium). On average a request on the t2.medium database took 2.28ms while on the m3.xlarge took on average 2.73ms. Waiting time for a worker thread was 11.09ms for experiment 1 and 9.15ms for experiment 2. The differences in the waiting time can be explained since a request is slower in the first system, worker threads need more time working on a request and therefore it takes a bit more time to get a worker thread. In order to see what exactly is going on I decided to run again those two experiments and . Pgbench shows that m3.xlarge is faster. I want to kill myself right now .. FUCKING shit ... apparently there was

This was counterintuitive. By checking the time the system took in every component we show that the difference was on the database. By running pbench we saw that actually m3.xlarge had better throughput than t2.medium instance so we assumed the problem was in the newotrk. Although all our instances were running in the Oregon region (US West) we realized afterwards that in every region there are isolated locations, known as availability zones[16] and apparently the m3.xlarge instance was in another zone. We had to rerun 8 of the 16 experiments and as could be show now they make sense.

—

During the initial testing phase of the system it happened to notice some really high response time, in the magnitude of hundreds milliseconds. By follow up search it came to our conclusion that some database requests were extremely slow. After realizing that we checked the PostgreSQL log files were we found:

2014-11-03 20:47:49 UTC LOG: checkpoints are occurring too frequently (26 seconds apart) 2014-11-03 20:47:49 UTC HINT: Consider increasing the configuration parameter "checkpoint_segments".

After reading about checkpoints we realized ... so we increased the number of checkpoint_segments to 10 and this problem never occured again. We checked after every experiment ?? TODO

6 Conclusion

This report correspon . We believe a great deal of work was done ? (nah) in a nice and consistent implementation as well as deploying stuff ...

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

This is a lovely conclusion for a lovely world that used to exist but is no more. Would have started earlier with experiments ... would have not wasted time on negligibe things(asserEquals) kai paei

+ What would I ahve done different if I started anew?