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 on 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 with describing the experiments done and their evaluation. We conclude the report in Section 6.

1 Introduction

This report describes **mepas**, the system developed for "Advanced Systems Lab" project. Goal of this milestone was to create a message passing system platform 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 form now on we will refer to as the "database tier" or "database". 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 [TODO] 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 identical middlewares connected to the database. There can be many clients connection to different middlewares as well.

2 System Design

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

Code Structure and Interface Overview

All the code for the client and the middleware was implement in subpackages of *ch.ethz.inf.asl*. The package structure can be seen in Figure 1.

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 general *Response* 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. 1: Package Structure

While designing the system I came to the realization that the communication protocol, e.g. send message, receive message between the clients and the middleware could be the same between the middleware and the database. Because of this, the interface that can be seen in Figure 2 was created. This interface can be found in src/main/ ... and is implemented by both ClientMessagingProtocolImpl and MiddlewareMessagingProtocolImpl. The difference between the two implementations is that in the client implementation when for example sendMessage() is called, a message is sent 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.

```
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. 2: Messaging Protocol Interface

MENTION that serializtion is used

Why are we having one Request/Response class per request/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 more extensible if needed to add new requests and for simplifying the work of the middleware. As can been seen the Request class contains the abstract execute method that receives as a parameter a MessagingProtocol implementation. Now when the middleware receives a request, after deserializing it it can just do 'request.execute(...)' and it knows that the correct execute method is going to be called. For example 'SendMessageRequest' implements execute as following. Therefore by taking advantage of polymorphism the Middleware doesn't need a long and error-prone list of if-else like this 'if request is SendMessage do this ... else if ... '.

In order to create one more request for the system, the method has to be inserted in the MessagingProtocol and then implement it in ClientMessagingProtocolImpl and MiddlewareMessagingProtocolImpl and the correct REquest - Response classes to be implemented and that is it. There is no need to go around and introduce one more enum value or put one more 'else-if' at some part of the code.

Code conventions

Initially we were planning to run the system at Dryad cluster for testing purposes where we were said Java 7 would be installed. For this reason I re-implemented part of the Optiona 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 close() is going to be called.

Database

Channging checkpoints cnfiguration file:

s; sync files=17, longest=0.035 s, average=0.006 s 2014-11-03 12:10:07 UTC LOG: checkpoint starting: xlog 2014-11-03 12:10:20 UTC LOG: checkpoint complete: wrote 1179 buffers (7.2%); 0 transaction log file(s) added, 0 removed, 3 recycled; write=13.352 s, sync=0.050 s, total=13.425 s; sync files=11, longest=0.012 s, average=0.004 s 2014-11-03 12:10:34 UTC LOG: checkpoints are occurring too frequently (27 seconds apart) 2014-11-03 12:10:34 UTC HINT:

—

Consider increasing the configuration parameter "checkpoint_segments".

——

checkpoint_segments = 1000 # in logfile segments, min 1, 16MB each checkpoint_timeout = 1h # range 30s-1h checkpoint_completion_target = 0.5 # checkpoint target duration, 0.0 - 1.0 checkpoint_warning = 30s # 0 disables

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 3 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. 3: 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 mich 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 SQL functions and is expalined below (TODO).

In order to increase the performance of the database, indexes were used. PostgreSQL creates by default indexes on the primary keys[1]. The extras 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*. The field *receiver_id* appears first on the index as its most commonly used[2] and in some cases its used alone, e.g. when listing the queues where a message for a user 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.

[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)

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 times 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 increasing performance. The code for the stored functions can be found in `read_committed_basic_functions.sql` file in `src/main/resources`.

Sta tests sinithos aplos calo ta statements etsi ... (den kalo ta callable statements) Stored procedures not used everywhere only where it made sense, 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 that takes as parameters the *p_requesting_user_id* and the *p_queue_id* and is looking for a message for the requesting user in the queue with the given id for receiving a message with no isolation levels in mind:

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;

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 as will be explained. 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."[5]. So with such an isolation level it is possible for two concurrent transactions to read the same message, one of them to delete it and both of them will return the same message. This of course is not acceptable behaviour since we want a specific message to be read by only one user. In order to solve this problem there are at least two approaches:

1. Use FOR UPDATE[6]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

[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] "Read Committed is the default isolation level in PostgreSQL." (http://www.postgresql.org/docs/9.3/static/transaction-iso.html)

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

[6] "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 "hat 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)

read transaction sees a snapshot as of the start of the transaction, not as of the start of the current query within the transaction."[7]. 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 transaction could find concurrent update errors and will have to be re-executed[8]. For the above reasons I used the first approach since it made my application code easier, i.e. not having to repeat a transaction.

Talk about FOR UPDATE and ORDER BY — have a look

Connecting Java and PostgreSQL

For the connection between Java and the database the JDBC41 PostgreSQL driver[9] was used.

What is being logged?

Setting up the Database

This is coolness!!

Wha

Management Console

A management console was also created (it is implemented in the *Manager* class under the *console* package) to easily check the contents a database in a remote machine. The console is a GUI application and can be seen Figure 4. 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 check the current data of the client, queue or message table. For retrieving the data from the database "*SELECT * FROM ...*" queries were issued on the database.

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

[8] "...it should abort the current transaction and retry the whole transaction from the beginning."(http://www.postgresql.org/docs/9.3/static/transaction-iso.html)

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

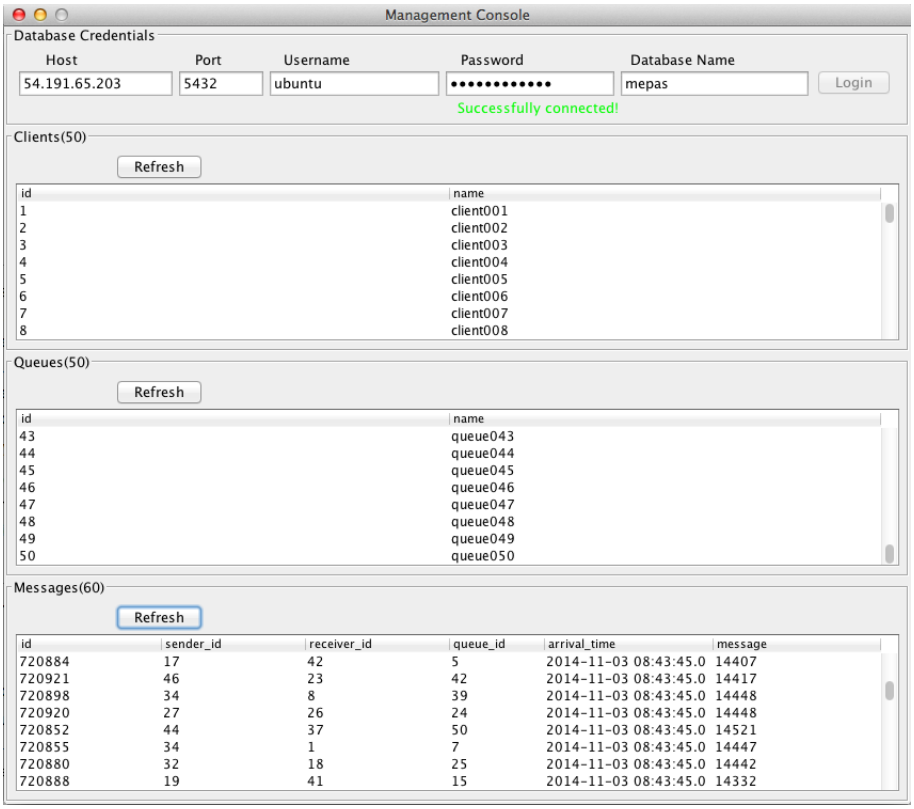

Fig. 4: Management Console Screenshot

Middleware

The middleware implements the messaging system, it receives requests/messages from clients and has to use the database in order to persist those msssagess, as well as retrieve the messages from the database to return to the clients. Before explainging the general architecture of the middleware, the interface clients can use will be explained. This interface can be seen in Figure ?.

This interface satisfies the desired functionality of our system

The middleware follows a non-blocking approach using simple Java I/O. This seems hard to believe at first place but it is going to be explained later on. Before doing so, let us see some of the possible approaches that can be used to implement a middleware.

1. First approach would be to have some worker threads on the middleware which every one of them waits for a connection from the client. When a conneciton is established it waits for a arequest from the clients, when it receives the request it alls the middleware below it and returens the response to the client. Afterwards it closes the connection and waits for the next clent connection. This approach seems quite wasteful/slow since for every request-response interaction the client has to establish a connection with the middleware. Pseudocode of this solution can be seen:

2. Have a worker thread for every client. This solutions seems to have scalability issues since the number of clients the middleware could possibly handled is bounded by the number of conncurent ?threads the system can support.
3. Use Java New IO and use a selector thread ... blocking approach
4. My approach was to have a queue of sockets corresponding to connections for the clients and have some worker threads operating in a round-robin approach on those sockets and check if there is something to read from the underlying input stream of the socket. Benefits of this approach is that I can support more clients than the number of threads in my system, as well as avoiding to establish a connection for every request. + BufferedInputStream The non-blocking nature of this approach can be achieved by using InputStream's ? available method that can return the number of bytes that can be read whithout blocking. Available is also a non-blocking method. So a blocking read is called only when available showed that there are bytes available. Figure gives a graphical depiction of this approach. As can be seen.

Where do requests get queued up?

FIFO FIFO FIFO

What is being logged?

DB request and the type of the request

Starting the Middleware

Many identical middlewares can be started
TAK ABOUT Implementing thread pool and connection pool bymyselfss

Stopping the Middleware

GRACEFUL termination

Clients

Every client is being executed as a thread (of *ClientRunnable*).

As was said in the previous section, clients use the 'MessagingProtocol' . Clients block when waiting for a response from teh middleware. How do they clients operate?

How do check the system is correct?

Talk about graceful termination

What is being logged?

Starting the Clients

3 Testing

Correctness of our system was of foremost importance, therefore testing played an important role while developing the system. The system has been tested exhaustively (TODO) 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[10] testing framework was used and also the Mockito[11] mocking framework was used. 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* packages, are using the local database which is being accessed by the constants given in the sdame 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 this 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 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

This test is as close as possible to how the system is being used using 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 confuses 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.

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

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.

Encountered Bugs

Verification errors while debugging .. forgotten notNull Found the InstantiationException in the newInstance() thing. Because I had requests with not a nullable Constructor. (This was found while mocking to get the messages with failed response).

While writing the endtoend test I realized I was immediately closing the connection to the clent from the middlware when the client was saying goodbye so the user was waiting forever for a response from the middleware. I though it was the middleware that wasn't finishing in the test so I Started making all the thread daemon threads to see what will happen. Found bugs in equals methods ...

General Encountered Problems

Problems that werent bugs

4 Experimental Setup

Never more than one middleware per instance

The code related to experiments can be found in the `experiments/code` directory. All the code related to the experiments was written in Python but there is a huge use of the underlying system commands through Python to achieve better performance. For example for calculating the number of lines of a file a "wc" command is called from within a Python program instead of opening and reading every single line of the file to count them up.

All the experiments were conducted in Amazon EC2. The following instances were used: ...

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.

In order to retrieve the IPs of the instances we used boto[12]. 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.

- Client: contains

about properties files (maybe) :

Sunday(19/10/2014) ———— By changing the executable to read configuration files instead of the command line arguments I solved the '$' problem in the password and also I can chaange the configuration files without having to really change the deployment scripts!!

bash was used too much for exmaple to remove warm up and cool down phase, awk, sed, grep were used

command = "awk -F'\t' '$1 >= " + str(warmUpInSeconds * 1000) + " && $1 <= " + \ str(lastTimeInMilliseconds - coolDownInSeconds * 1000) + " { print; }' " + specificFile

Python seemed more appropiate for combining bash commands bla blah

Talk about pexpect and how awesome it is!

EC2 Instances

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!!

Deploying the System for Experiments

Explaing about Python boto .. gnuplot and all the other packages ... most of the time used UNIX native commands to have much faster times of whaterver ... for example for reading CSV files, cutting parts of files and so on

5 Experiments

All successfull receiveals bla blah

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

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

[12] https://github.com/boto/boto

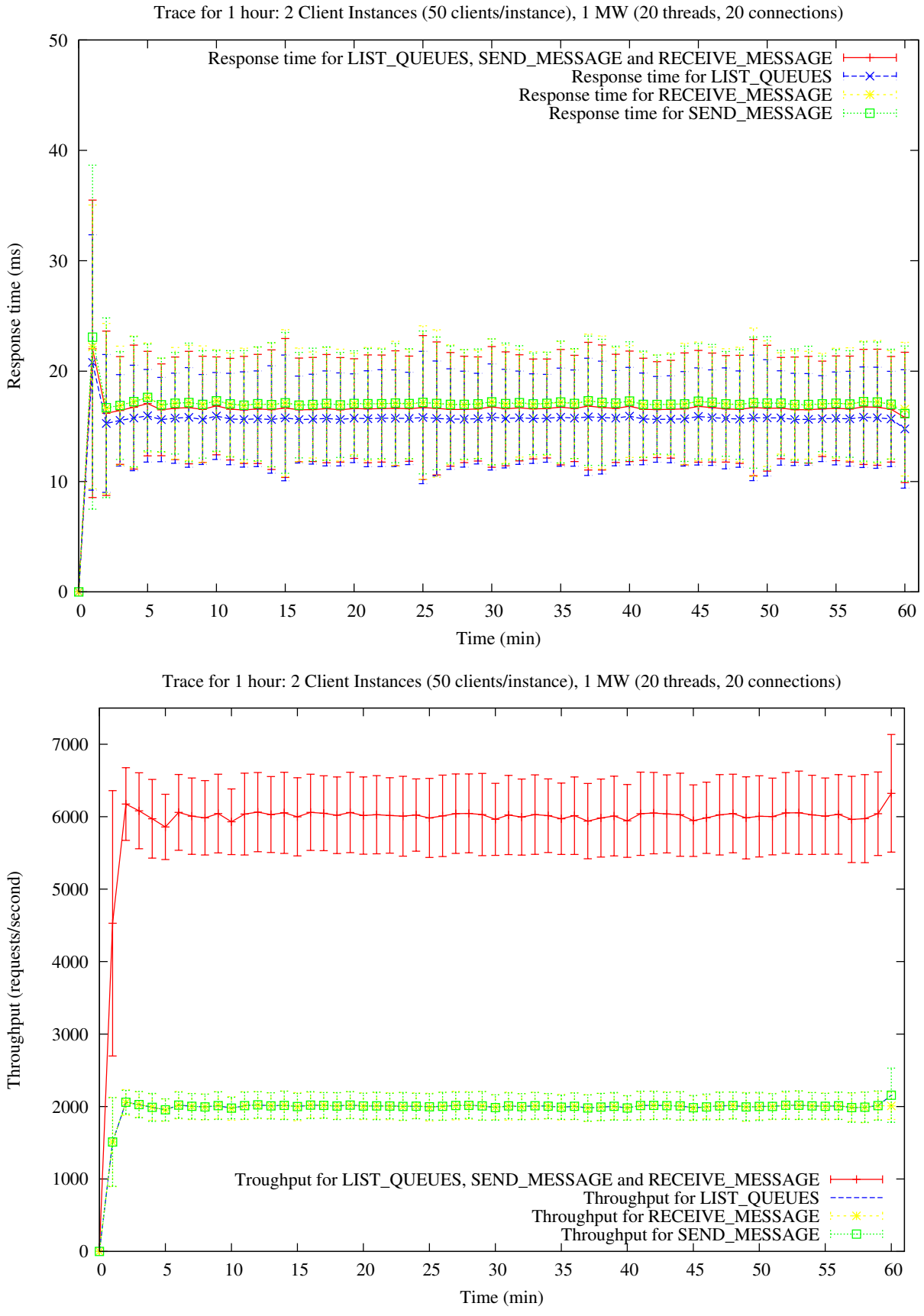

Fig. 5: Response time and throughput of an one hour trace with 2 client instances (50 clients/instance) and 1 middleware instance (20 threads, 20 connections)

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

Warm Up and Cool Down

REmoved 2 minutes from the beginning and one minute from the end in all of the following experimental results. As we saw from the trace ...

$2^{k(=?\text{TODO})}$ 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

#threads	#connections	MW instance type	DB instance type	throughput (requests/sec)
10	10	t2.small	m3.xlarge	3549
10	20	t2.small	m3.xlarge	3508
20	10	t2.small	m3.xlarge	3379
20	20	t2.small	m3.xlarge	5615
10	10	t2.medium	m3.xlarge	4585
10	20	t2.medium	m3.xlarge	4464
20	10	t2.medium	m3.xlarge	4567
20	20	t2.medium	m3.xlarge	6133
10	10	t2.small	t2.medium	4191
10	20	t2.small	t2.medium	4135
20	10	t2.small	t2.medium	4122
20	20	t2.small	t2.medium	5450
10	10	t2.medium	t2.medium	4167
10	20	t2.medium	t2.medium	4099
20	10	t2.medium	t2.medium	4212
20	20	t2.medium	t2.medium	5448

Fig. 6: Different througput values based on the number of threads, number of connections, middleware and database instance type

Increasing the Message Size

Increasing the Number of Clients

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[13] 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 is a lovely conclusion for a lovely world that used to exist but is no more.

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