Advanced Systems Lab - Milestone I

Lukas Elmer (elmerl@ethz.ch), Matthias Ganz (ganzm@ethz.ch)

November 15, 2013

Contents

Abstract

This document describes the message queuing system which was build. Architecture and design choices are shown and explained. Further test scenarios and test loads are defined. Resulting test output is described, analyzed and interpreted.

1 Messaging System

In this section the system under test (also named as **middleware** or **broker**) is described.

1.1 Overview

Figure 1 shows a generic setup of the messaging system. Multiple middleware components are connected to a single database and severs a couple of clients. Application state is persisted on the database, therefore a middleware component can be considered as stateless.

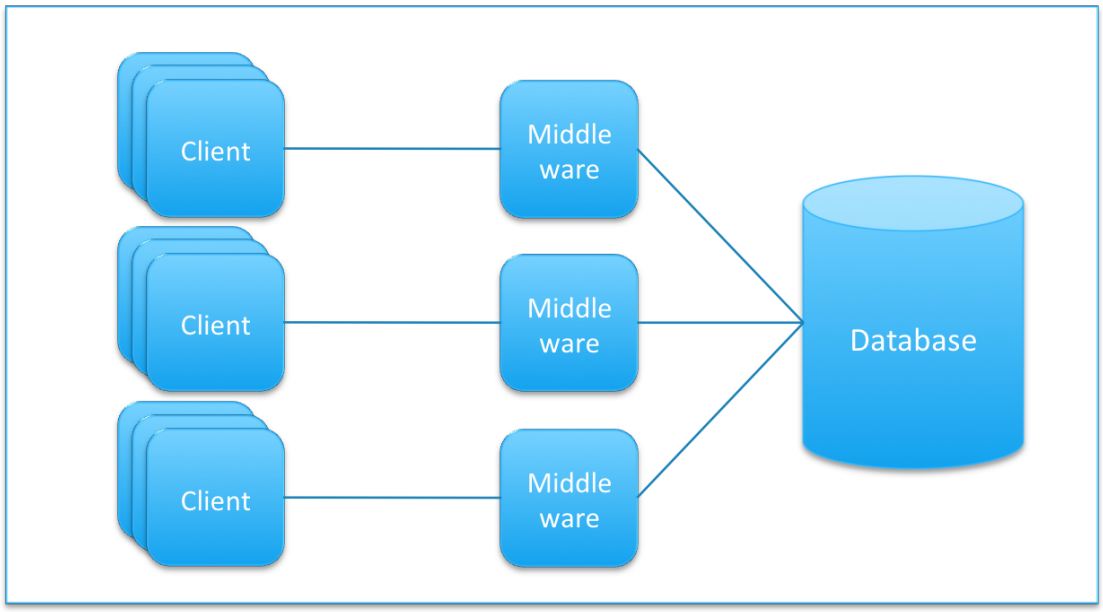

Figure 1: System Overview

Figure 2 shows important software components of a single middle ware. A nonblocking network interface (NIO) handles communication to the clients. Received data is put to a single thread safe request queue. A configurable number of workers are constantly reading from the request queue. If there is nothing to do these threads are blocked (no busy wait). As soon as a worker gets a piece of work (request raw data) it then performs the following tasks:

- decoding: The raw request is parsed and converted into a request java object
- process: The request is processes. The database is accesses and a response object is created
- encode: The response object is serialized and placed into the response queue of the network interface.

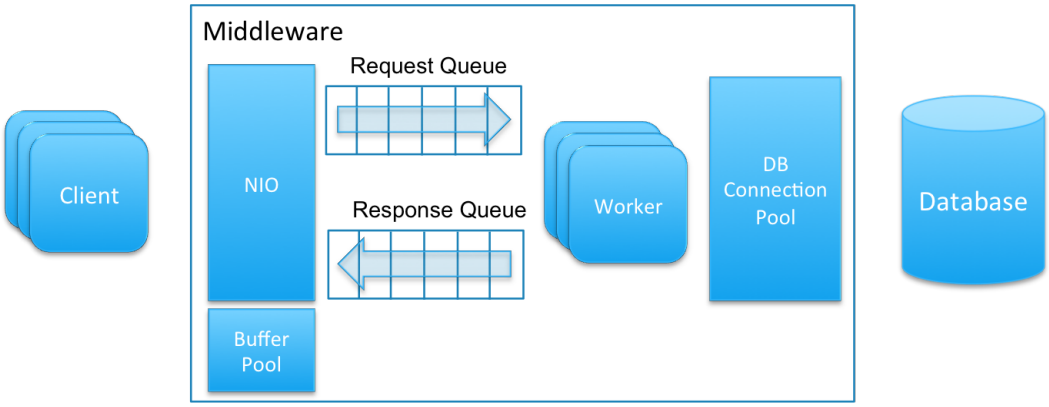

Figure 2: Middleware's main Components

1.2 Client to Middleware Interface

In this section a code snippet from the client class is shown. It describes the interface the middleware provides.

```
package ch.ethz.mlmq.client;

...

/**
 * The client interface
 */
public interface Client extends Closeable {

        /**
         * Initializes the client and connects it to the Broker
         *
         * @throws IOException
         */
        void init() throws IOException;

        /**
         * indicates whether the client is connected or not
         *
         * @return
         */
        boolean isConnected();

        /**
         * Register as a new client.
         *
         * @return client connection information
         */
        ClientDto register() throws IOException, MlmqException;
```

```
/**
 * Creates a queue with a specific name.
 *
 * @return the queue created
 */
QueueDto createQueue(String queueName) throws IOException, MlmqException;

/**
 * Tries to find the where we can send personal messages to a client
 *
 * @param queueName
 * @return Queue may be null if not found
 */
QueueDto lookupClientQueue(String queueName) throws IOException, MlmqException;

/**
 * Tries to find a client for a specific name
 *
 * @param clientName
 * @return
 */
ClientDto lookupClient(String clientName) throws IOException, MlmqException;

/**
 * Tries to find the queue where we can send personal messages to a client
 *
 * @param clientId
 * @return Queue may be null if not found
 */
QueueDto lookupClientQueue(long clientId) throws IOException, MlmqException;

/**
 * Deletes a queue
 *
 * @param id
 */
void deleteQueue(long id) throws IOException, MlmqException;

/**
 * Sends a message to a specific queue.
 *
 * @param queueId
 * @param content
 *          raw message content
 * @param prio
 *          number from 1 to 10 (10 indicates maximum priority)
 */
void sendMessage(long queueId, byte[] content, int prio) throws IOException,
    MlmqException;

/**
 * Sends a message to multiple queues.
```

```
 *
 * @param queueIds
 * @param content
 *          raw message content
 * @param prio
 *          number from 1 to 10 (10 indicates maximum priority)
 * @throws IOException
 */
void sendMessage(long[] queueIds, byte[] content, int prio) throws IOException,
    MlmqException;

/**
 * Sends a private message to a client
 *
 * @param clientId
 * @param content
 *          raw message content
 * @param prio
 *          number from 1 to 10 (10 indicates maximum priority)
 */
void sendMessageToClient(long clientId, byte[] content, int prio) throws
    IOException, MlmqException;

/**
 * Request/Responses are posted to the private client queue it is sent to
 *
 * As soon as a client performs a Request it receives a context identifier
 *
 * Any response received from a client needs to be in the queue of the receiving
     client and needs to have the same context identifier
 *
 * @param client
 * @param content
 *          raw message content
 * @param prio
 *          number from 1 to 10 (10 indicates maximum priority)
 * @return returns a context identifier which can be used as a filter criteria
     when reading messages
 */
long sendRequestToClient(long client, byte[] content, int prio) throws
    IOException, MlmqException;

/**
 *
 * @param clientId
 * @param context
 *          context identifier
 * @param content
 *          raw message content
 * @param prio
 *          number from 1 to 10 (10 indicates maximum priority)
 * @throws IOException
 */
```

```
        long sendResponseToClient(long clientId, long context, byte[] content, int prio)
            throws IOException, MlmqException;

        /**
         * Query for queues with pending messages.
         *
         * This method returns number of message in the client's queue and a list of
             queues which are not empty.
         *
         * @param queues
         *           output parameter containing Queues which contain messages. This
             method simply appends the results to this list
         * @param maxNumQueues
         *           maximum number of returned QueueDto's
         * @return number of messages in the client's personal queue
         */
        int queuesWithPendingMessages(List<QueueDto> queues, int maxNumQueues) throws
            IOException, MlmqException;

        /**
         * Reads the first message without removing it.
         *
         * @param messageQueryInfo
         * @return message, may be null
         */
        MessageDto peekMessage(MessageQueryInfoDto messageQueryInfo) throws IOException,
            MlmqException;

        /**
         * Reads the first message and removes it.
         *
         * @param messageQueryInfo
         * @return message, may be null
         */
        MessageDto dequeueMessage(MessageQueryInfoDto messageQueryInfo) throws
            IOException, MlmqException;
}
```

1.3 Database Design

Figure 3 gives an overview of the database design.

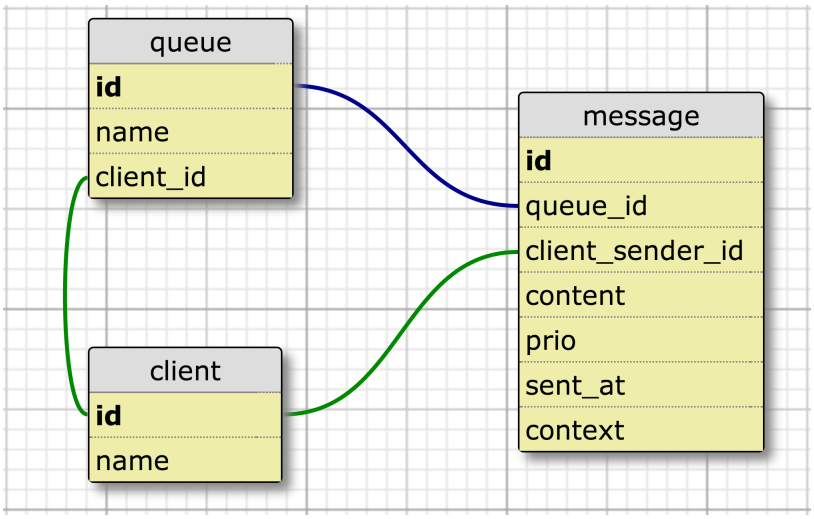

Figure 3: Database Schema

1.3.1 Client

Attribute	Indexed	Datatype	Description
id	PK	Integer	AutoIncrement Primary Key
name	Yes	Varchar(50)	Unique name of a client

1.3.2 Queue

Attribute	Indexed	Datatype	Description
id	PK	Integer	AutoIncrement Primary Key
name	X	Varchar(50) Unique	Client Name
client_id	X	Nullable Integer	References a client. This queue is a ClientQueue if not null.

1.3.3 Message

Attribute	Indexed	Datatype	Description
id	PK	Integer	AutoIncrement Primary Key
queue_id	X	Integer NonNull	Foreign key to the queue where this message is sent to
client_sender_id	X	Integer NonNull	Issuer of this message
content		Bytea	Message raw data
prio	X	SmallInt	Priority of the message
sent_at	X	Time	Time when this message was sent
context	X	Integer NonNull	Nullable context integer. Ties a request response message pair together

1.4 Middleware to Database Interface

The interface to the database is abstracted through the the classes located in package *ch.ethz.mlmq.server.db.dao*.

- MessageDao - handles access to the message table
- ClientDao - creates and deletes clients
- QueueDao - creates and deletes queues

These classes encapsulate all database access performed bye the middleware bye using PreparedStatements[1]. PreparedStatements tend to be faster than normal Statements because they can be precompiled on the database system.

1.4.1 Stored Procedures

Database operations which would have required more than one (Prepared)Statements per request are implemented as StoredProcedures. This reduces unnecessary database roundtrips.

[1] `http://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html`

2 Design Decisions

In this section a selection of the more important design decisions is presented. It is described why these choices were made and how they affect the behaviour of the system.

2.1 Blocking vs. Nonblocking Network IO

Java basically allows two ways how to handle tcp/ip connections. There is the more straight forward way with blocking io and there is another API which allows nonblocking operation on sockets.

The implementation of the middle ware features a single threaded networking software layer which implements the reactor pattern by using a *java.nio.channels.Selector*. The basic idea is to let the Selector keep track of connections. By a connection based state variable we let the Selector know in which events we are interested in.

Advantages By using nonblocking io over the blocking method, we avoid that the number of threads in the system need to be scaled with the number of clients. It is assumed that system scales better in terms of concurrent connections than the blocking method for the simple fact that thread switches takes time and memory.

2.2 Database Connection Pooling

Establishing a jdbc database connection takes time. A Connection Pool keeps a list of database connections which are permanently opened. As soon as a connection is needed by a worker, it simply queries the pool for one.

The pool needs to be accessed in an exclusive fashion. However the cost to avoid concurrent pool access should be way lower than creating a database connection each time one is needed.

A nice effect of having a permanent database connection is that it allows the use of PreparedStatement. These are sql statements which can be precompiled on the database system.

2.3 Mutual Exclusion on the Database

A problem which arose later in the project was a bug where there were two messages in a queue and two clients tried to dequeue messages from it simultaneous. Dequeuing a message is performed in two separate sql statements. The first statement peeks a message and the second one tries to delete the peeked one. The bug occured where two client peeked the same message but only one of them could delete it. A simple way of resolving that issue would have been to put both statements described into a single prepared statement and additionally set the database isolation level to *serialized*.

We decided to fix this bug by locking critical records. If a client tries to dequeue a message it selects and locks it with a "Select ... For Update' sql statement. The selected record is now exclusively locked for the current transaction. Other clients which try to get a lock on the very same record end up being blocked by the database system until the first client commits its transaction.

Another option Having a connection pool could have been avoided if every client was assigned a dedicated database connection. There would have been no need to synchronize worker threads to retrieve a connection. This however decreases the number of tuning parameter. Database connection would not have been optimally used. Connections would have been idling for each time consuming operation a worker performs.

2.4 Buffer Pooling

Serializing and deserializing messages which are up to 2kb long requires a buffer of at least the same size. In order to avoid excessive garbage collection it was decided to implement a pool containing message buffers.

Pooled buffers are obtained from the networking interface before receiving a message. After processing the message and right before we lose the reference to the buffer it is put back to the pool for later use.

Having this pool should significantly increase the period between garbage collection.

Note See implementation of *ch.ethz.mlmq.nio.ByteBufferPool*

2.5 Message Transmission

Messages are passed on the initiative of a client (There is no way for the messaging system to push information to a specific client). A clients issues a requests and expects a response within a certain time. Each requests generates a response. In case of an error an error response is replied by the middleware.

A message consists of a 4 byte length field and a variable length field containing the payload.

To send a messages (requests and responses) the first 4 bytes indicate the binary length of the message. After that the request (or response) is serialized.

2.6 Message Serialisation

The first approach to serialize messages was using the java's built in stream based serialisation infrastructure. This however did not work well together with 2.4. Default java serialisation won't let us plug in our buffer pool. Further the decision to implement the networking interface with java nonblocking io however forced us to use java's *ByteBuffer* class to handle binary data.

Theses issues brought us to the decision to implement custom serialisation and deserialisation supporting pooled ByteBuffers.

2.6.1 Example

A SendMessageRequest for instance is serialized as following.

1. 4 bytes - lenght of the message payload
2. 4 bytes - type of the message (e.g. SendMessageRequest)
3. 8 bytes per recipient queue plus 4 bytes list lenght overhead
4. 4 bytes priority
5. 0-2000 byte message content

2.7 Load Shedding

With the current design requests are queued in a single request queue. If workers are slow requests begin to queue up. If the queue is full adding another request fails. The request from the client is silently dropped. This behaviour is commonly known as load shedding. The system tries as quick as possible to get rid of load which cant be handled any more. The point where the system starts trashing will be moved further away.

3 Performance Relevant Features

3.1 Overview

During a brainstorming session, a broad spectrum of performance relevant features were extracted, see figure 4. Then, the primary features (PF, orange) and the secondary features (SF, green) were chosen according to the group membmers domain specific knowledge and presumptions. They were also validated during discussions with other group members and the teaching assistant.

3.2 Primary features

According to figure 4, the primary features are as follows:

1. **# Clients** The amount of clients interacting with the system.
2. **# Brokers** The amount of brokers interacting with the system.
3. **Database connection pool size** The amount of database connections per broker to the database.
4. **Worker pool size** The amount of workers per broker.
5. **Send Message** When many insertions occur (send a message).
6. **Dequeue Message** When many deletions occur (read a message and remove it from the queue).

3.3 Secondary features

According to figure 4, the secondary features are as follows:

1. **Peek Message** When many selects occur (read a message, but don't remove it from the queue). This is mostly covered by the dequeue message feature and therefore only seen as a secondary feature.
2. **Message Size** The average size of the messages payloads.
3. **Use Shed Load** Whether the fault tolerance pattern Shed Load [2] is used or not.
4. **Garbage Collection** The time it takes for the garbage collection.
5. **Indices** Whether indices are used or not.
6. **Request Queue Length** The amount of messages on a broker which are buffered until the are processed.
7. **Response Queue Length** The amount of messages on a broker which are buffered until the are sent back to the client.

[2] http://dl.acm.org/citation.cfm?id=SERIES12798.1557393

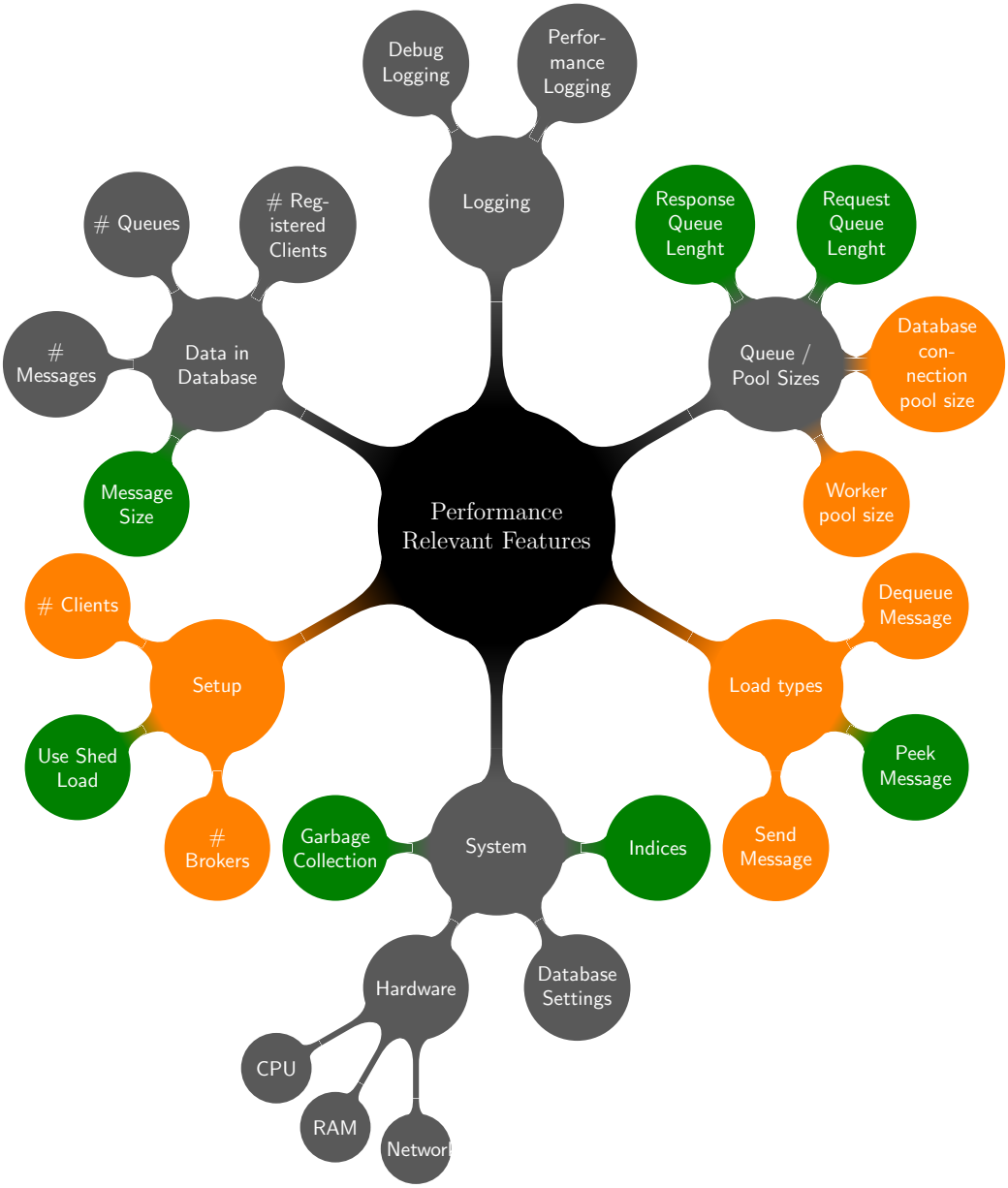

Figure 4: Performance relevant features mind map

4 Experiments

4.1 Test Protocol

The whole test protocol can be found on the Testmasters[3]. One will find a complete list of experiments which were conducted on the Amazon cloud. Additionally, a short summary of the test protocol can be found in the directory *ethz-asl-mlmq/doc/test_protocol*.

4.2 Testload

To generate test load for the messaging system a mixture of different types of clients have been chosen. This section describes these clients and their behaviour.

4.2.1 OneWayClient

As stated in the project description each OneWayClient sends an initial message to an arbitrary other one. Having it's initial message sent they periodically check their personal queue for incoming messages. If there is one the message is dequeued and forwarded to a random receipient.

Sent messages contain a simple counter which is increased each time a message is relayed.

4.2.2 PairedClient

PairedClient's always come in pairs. They keep on sending requests and responses to each other. At start it is determined which client is the requesting client and which is the responding client.

4.2.3 PublicQueueProducer

This kind of client writes the same message again and again to a public queue.

4.2.4 PublicQueueConsumer

This kind of client tries to read messages at a constant rate from a public queue.

4.3 Microbenchmarks

The goal of the microbenchmarks is to understand single components and units.

4.3.1 Plots

This section starts with some plots from the microbenchmarks.

[3] https://testmaster-asl-eth.renuo.ch and https://testmaster2-asl-eth.renuo.ch

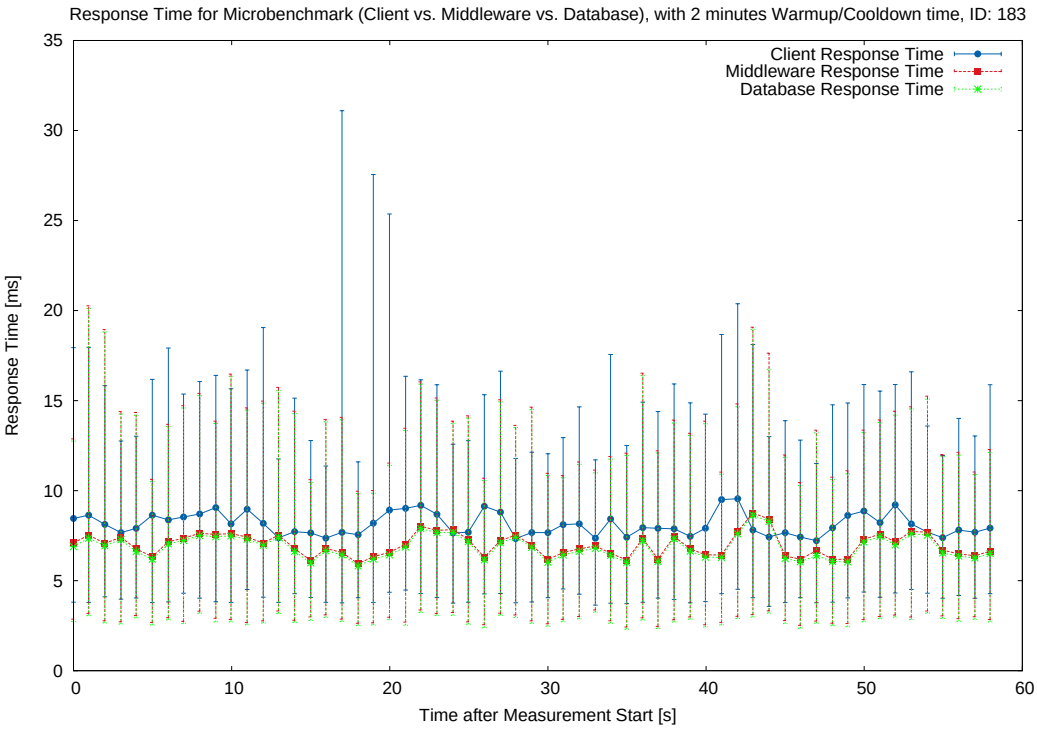

Figure 5: Response time of microbenchmark. The error bars are the 2.5% and 97.5% percentiles.

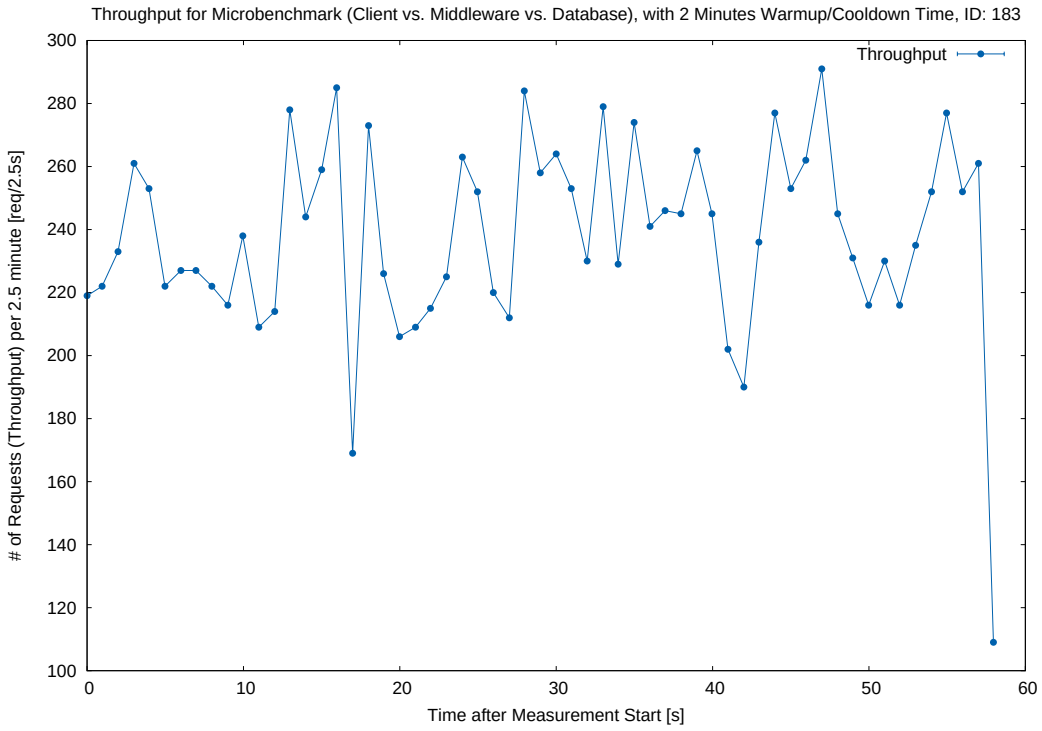

Figure 6: Throughput of microbenchmark in 1000 requests.

4.3.2 Send Message and Receive Message

The time to send a message has been measured:

Test Run ID	Component	Median [μs]	Mean [μs]	StdDev [μs]
183	Client	6832	7365.251888	4187.470317
183	Middleware	5467	5694.925436	2628.921725
183	Database	5424	5652.727388	2576.826615

The time to receive a message has been measured:

Test Run ID	Component	Median [μs]	Mean [μs]	StdDev [μs]
183	Client	8954	9796.484939	4927.697663
183	Middleware	7617	8073.361544	3467.291457
183	Database	7575	8029.920168	3461.8049

Calculation of the expected time for send message:

Component	Type	Expected Time [μs]	Expected Time [%]
Client and Network Client-Middleware	Send	1365	19.9795081967213
Middleware after deserialization	Send	43	0.629391100702576
Database and Network Middleware-Database	Send	5424	79.3911007025761

And the following results for the receive message:

Component	Type	Expected Time [μs]	Expected Time [%]
Client and Network Client-Middleware	Receive	1337	14.9318740227831
Middleware after deserialization	Receive	42	0.469064105427742
Database and Network Middleware-Database	Receive	7575	84.5990618717892

Interpretation The think time of the client should be very near to 0ms - so we use the same value here as for the broker. Thus, we suspect that the difference between client and middleware is the network. We assume that the network time between the client and the middleware is the same as the middleware and the database. We consider the median for the measurements.
This way, we can estimate the network as $2*(client-middleware)$, and the client as $middleware$. Also, the database should be $database-network/2$. Thus we conclude the following final results for the send message:

Component	Type	Expected Time [μs]	Expected Time [%]
Client	Send	43	0.629391100702576
Middleware	Send	43	0.629391100702576
Database	Send	4016	58.7822014051522
Network	Send	2644	38.7002341920375

And the following final results for the receive message:

Component	Type	Expected Time [μs]	Expected Time [%]
Client	Receive	42	0.469064105427742
Middleware	Receive	42	0.469064105427742
Database	Receive	6196	69.1981237435783
Network	Receive	2590	28.9256198347107

4.3.3 Time Caused By Code

As described in the previous section, the time used for the code is 0.042 ms and 0.043 ms for sending and receiving respectively. In percent this is less then 1% for sending and receiving, and thus this factor is relatively unimportant compared to the database and the network.

4.3.4 Network performance

Even though the network connection throughput is specified by Amazon, only a measurement tells the truth. To confirm this, a network performance test was executed with iperf. We sim-

ulated a bi-directional data transfer between a middleware/client and the database. We get 841Mbits/s upload to the database and 93.1Mbits/s download to the middleware/client if iperf is run in bidirectional mode (send and receive in parallel). If only one direction is used, then the throughput from middleware/client to database is 959Mbits/s and from database to middleware/client it is 312Mbits/s. We also notice that the variance for the single values are rather big.

```
------------------------------------------------------------
Client connecting to 54.194.38.184, TCP port 5001
TCP window size:  133 KByte (default)
------------------------------------------------------------
[  3] local 172.31.6.201 port 60902 connected with 54.194.38.184 port 5001
[  5] local 172.31.6.201 port 5001 connected with 54.194.38.184 port 52555
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec   122 MBytes   102 Mbits/sec
[  5]  0.0-10.0 sec   970 MBytes   814 Mbits/sec
[  5] 10.0-20.0 sec  1007 MBytes   844 Mbits/sec
[  3] 10.0-20.0 sec   135 MBytes   113 Mbits/sec
[  5] 20.0-30.0 sec  1.00 GBytes   861 Mbits/sec
[  3] 20.0-30.0 sec   121 MBytes   102 Mbits/sec
[  5] 30.0-40.0 sec   978 MBytes   820 Mbits/sec
[  3] 30.0-40.0 sec   111 MBytes  93.0 Mbits/sec
[  5] 40.0-50.0 sec  1009 MBytes   846 Mbits/sec
[  3] 40.0-50.0 sec  88.0 MBytes  73.8 Mbits/sec
[  5] 50.0-60.0 sec  1.00 GBytes   862 Mbits/sec
[  3] 50.0-60.0 sec  89.1 MBytes  74.8 Mbits/sec
[  3]  0.0-60.1 sec   666 MBytes  93.1 Mbits/sec
[  5]  0.0-60.0 sec  5.88 GBytes   841 Mbits/sec
```

Figure 7: Bidirectional Test

```
------------------------------------------------------------
Client connecting to 54.194.38.184, TCP port 5001
TCP window size: 22.9 KByte (default)
------------------------------------------------------------
[  3] local 172.31.6.201 port 60905 connected with 54.194.38.184 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec   430 MBytes   361 Mbits/sec
[  3] 10.0-20.0 sec   342 MBytes   287 Mbits/sec
[  3] 20.0-30.0 sec   347 MBytes   291 Mbits/sec
[  3] 30.0-40.0 sec   426 MBytes   357 Mbits/sec
[  3] 40.0-50.0 sec   342 MBytes   287 Mbits/sec
[  3] 50.0-60.0 sec   342 MBytes   287 Mbits/sec
[  3]  0.0-60.0 sec  2.18 GBytes   312 Mbits/sec
```

Figure 8: Unidirectional test, database to middleware/client

```
------------------------------------------------------------
Client connecting to 54.194.42.48, TCP port 5001
TCP window size: 22.9 KByte (default)
------------------------------------------------------------
[  3] local 172.31.8.61 port 52558 connected with 54.194.42.48 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  1.13 GBytes   974 Mbits/sec
[  3] 10.0-20.0 sec  1.11 GBytes   956 Mbits/sec
[  3] 20.0-30.0 sec  1.11 GBytes   957 Mbits/sec
[  3] 30.0-40.0 sec  1.11 GBytes   957 Mbits/sec
[  3] 40.0-50.0 sec  1.11 GBytes   956 Mbits/sec
[  3] 50.0-60.0 sec  1.11 GBytes   956 Mbits/sec
[  3]  0.0-60.0 sec  6.70 GBytes   959 Mbits/sec
```

Figure 9: Unidirectional test, middleware/client to database

Sending one message uses about 20 Bytes without payload, because we use custom serialization (see 2.6 Message Serialisation). Thus, our message size will be between 20 Bytes and 2020 Bytes. With a network connection that supports 100Mbit/s we can send 625'000 (without payload) up to 6188 (with payload) per second. In this project we focus on messages without or only little payload, and thus the network bandwidth is not a bottleneck for the throughput for the tests.

4.3.5 Component Limits

In the microbenchmarks, the limit for the amount of messages a single client was determined:

Duration [s]	**Requests sent**
100	23465
1	**234.65**

4.3.6 Testmaster

To run the tests, the Testmaster is used to compile the Java code, generate the config files, deploy it to the Amazon AWS machines and start the tests. When the test finishes, it then copies all log files after all Java JVM's have stopped. This whole process takes 8-9 minutes when the Java instances stop immediately.

4.4 2 Hour Test

To verify that the system runs stable over a long period of time an experiment is performed which puts a constant load onto the system for 2 hours.

During the project, the system evolved and thus some test runs were invalidated in this process. Here, the final two 2h test runs are presented. The other 2h test runs can be found and analysed on the Testmaster[4].

[4] https://testmaster-asl-eth.renuo.ch/

4.4.1 Experiment setup

The experiment is performed on the Amazon cloud with constant load. For the setup configuration, the project specifications were used specifications (80 clients).

Component	Amazon instance type and availability zone
database	m1.medium, eu-west-1a
middlewares	m1.small, eu-west-1a
clients	m1.small, eu-west-1a

Middleware In total 6 middleware components are running On 2 Amazon instances.

Clients The following mix of clients are used for the 2 hour test.

Client type mix	#
OneWayClient	1x30 (machines x jvms)
PairedClient	1x16 (machines x jvms)
PublicQueueConsumer	1x10 (machines x jvms)
PublicQueueProducer	1x5 (machines x jvms)

4.4.2 Hypothesis

Since the system will not be saturated we expect no changes over time concerning response time and throughput. The mixture of clients generating load is chosen such that there is a constant number of messages in the system. Increased SQL query execution time should not be observable.

4.4.3 Results

There were multiple 2h tests performed. The final two 2h tests are the ones with id 174 [5] and 176 [6]. Both run stable and generated the following plots:

[5] https://testmaster-asl-eth.renuo.ch/test_runs/174
[6] https://testmaster-asl-eth.renuo.ch/test_runs/176

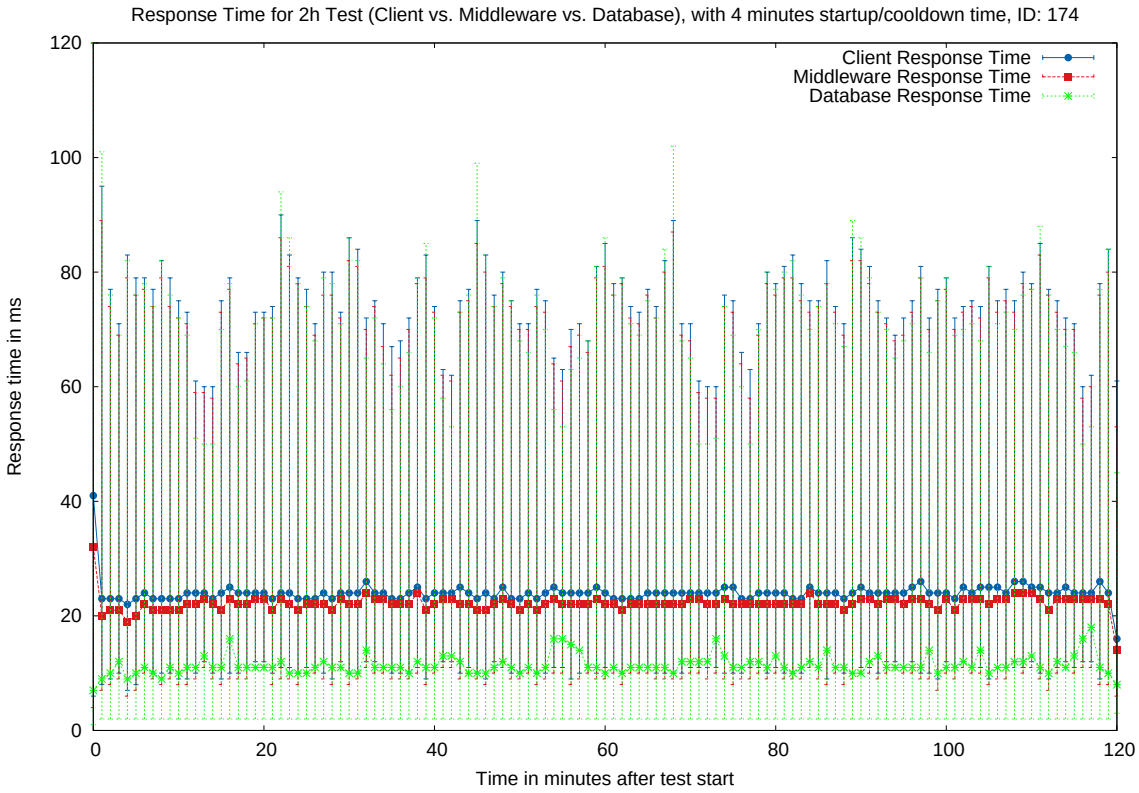

Figure 10: Response time of 2h Test. The error bars are the 2.5% and 97% percentiles

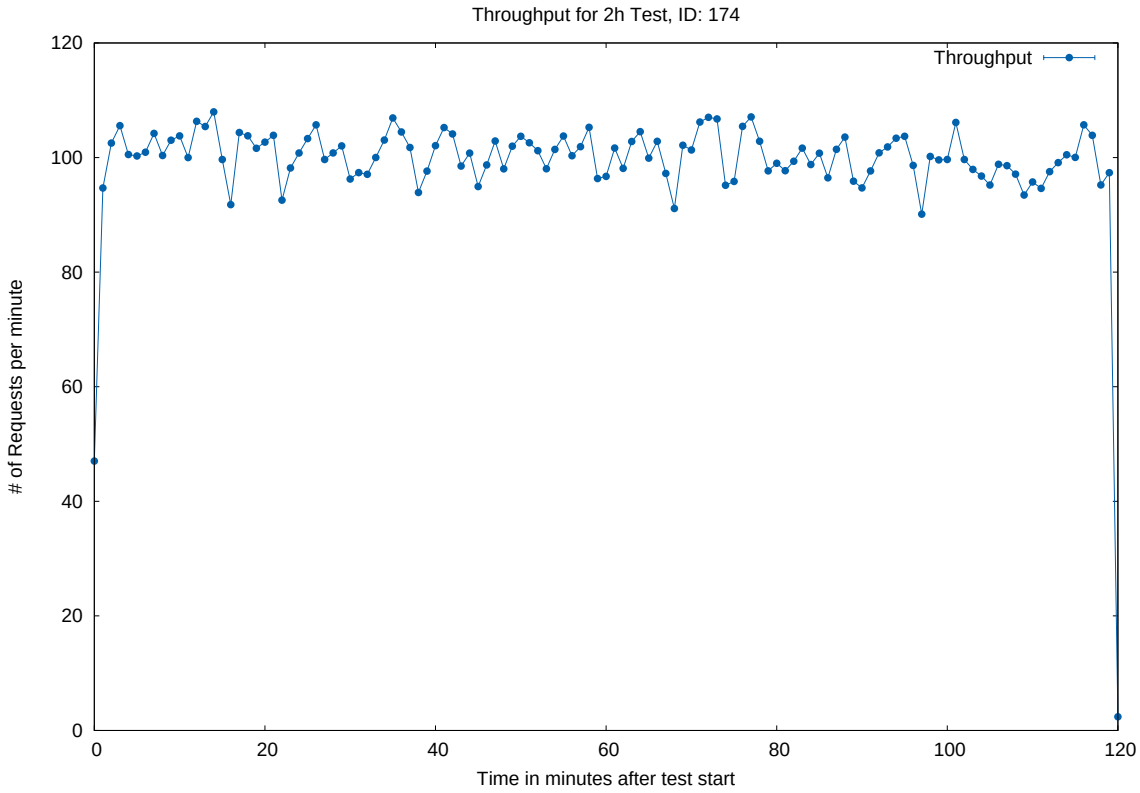

Figure 11: Throughput of 2h test in 1000 requests.

4.4.4 Interpretation

The final test runs (174 and 176) run as expected in the hypothesis.

4.5 2^k Experiment

In order to determine the influence of a number of primary factors on to the system it was decided to do a 2^k analysis on several factors described in the following section.

For each feature level an experiment of 30 minutes is performed where clients perform 100 actions (send and receive message) per second.

4.5.1 Factors and Levels

Number of Clients The number of simultaneously connected clients is varied between 8 and 30. There is a mix of different clients which produce slightly different types of load.

Client type mix	#	#
OneWayClient	16	8
PairedClient	8	4
PublicQueueConsumer	4	2
PublicQueueProducer	2	1
Total	**30**	**15**

Number of brokers The number of middleware components is varied between 4 and 8. For both levels the number of Amazon instances used is 2.

Number of workers The number of worker threads a single middleware instance has is varied between 4 and 8

Number of Database Connections The number of concurrent database connection a single middleware instance has is varied between 4 and 8

4.5.2 Hypothesis

The response time should not be majorly affected by these chosen parameters. Since the system is not under much load only small changes concerning the response time should be observable.

However the number of requests processes should vary. The main feature which obviously affects number of requests is the number of connected clients sending messages. Other minor variations will reveal bottle necks in the system.

4.5.3 Measurement Results

The following table shows median time taken and number of requests processed for all different feature levels.

Clients C	Brokers B	Workers W	Connections D	TestRun Id	Median Processing time [ms] PT	# of Processed Requests RC
15	4	4	4	2003	6,80	1 996 478
15	4	4	8	173	7,46	1 806 665
15	4	8	4	2007	6,87	2 062 704
15	4	8	8	2008	7,21	2 060 655
15	8	4	4	2009	5,11	1 385 552
15	8	4	8	2004	7,55	1 910 840
15	8	8	4	2005	5,90	1 975 889
15	8	8	8	2006	7,60	1 920 835
30	4	4	4	159	14,08	2 488 948
30	4	4	8	160	12,80	2 451 248
30	4	8	4	161	13,42	2 428 769
30	4	8	8	162	13,97	2 511 260
30	8	4	4	163	12,43	2 329 883
30	8	4	8	164	14,85	2 110 459
30	8	8	4	165	12,88	2 266 237
30	8	8	8	166	15,05	2 166 303

Figure 12: 2^4 factorial test measurements

Signs For the 2^k test the following signs are assigned to feature leavels:

Feature	**+1 Level**	**-1 Level**
Number of Clients **C**	30	15
Number of Brokers **B**	8	4
Number of Workers **W**	8	4
Number of Database Connections **D**	8	4

Figure 13: Feature Level Signs

This leads to the following sign table.

	I	c	b	w	d	cb	cw	cd	bw	bd	wd	bwd	cwd	cbd	cbw	cbwd
	1	-1	-1	-1	-1	1	1	1	1	1	1	-1	-1	-1	-1	1
	1	-1	-1	-1	1	1	1	-1	1	-1	-1	1	1	1	-1	-1
	1	-1	-1	1	-1	1	-1	1	-1	1	-1	1	1	-1	1	-1
	1	-1	-1	1	1	1	-1	-1	-1	-1	1	-1	-1	1	1	1
	1	-1	1	-1	-1	-1	1	1	-1	-1	1	1	-1	1	1	-1
	1	-1	1	-1	1	-1	1	-1	-1	1	-1	-1	1	-1	1	1
	1	-1	1	1	-1	-1	-1	1	1	-1	-1	-1	1	1	-1	1
	1	-1	1	1	1	-1	-1	-1	1	1	1	1	-1	-1	-1	-1
	1	1	-1	-1	-1	-1	-1	-1	1	1	1	-1	1	1	1	-1
	1	1	-1	-1	1	-1	-1	1	1	-1	-1	1	-1	-1	1	1
	1	1	-1	1	-1	-1	1	-1	-1	1	-1	1	-1	1	-1	1
	1	1	-1	1	1	-1	1	1	-1	-1	1	-1	1	-1	-1	-1
	1	1	1	-1	-1	1	-1	-1	-1	-1	1	1	1	-1	-1	1
	1	1	1	-1	1	1	-1	1	-1	1	-1	-1	-1	1	-1	-1
	1	1	1	1	-1	1	1	-1	1	-1	-1	-1	-1	-1	1	-1
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Weights PT	**10.24**	**3.43**	-0.07	0.11	**0.56**	0.19	0.03	-0.08	0.07	**0.52**	0.03	-0.15	0.16	0.13	-0.05	-0.10
Weights RC	2 117 045	227 093	−108 795	57 036	237	−17 122	−58 032	−34 558	17 030	18 621	−9 556	−48 050	39 516	−64 140	−17 984	47 962

Figure 14: 2^4 factorial test result

4.5.4 Interpretation

It was chosen to ignore the calculated weights for processed request counts RC because big values are more difficult to read and because both observed values correlate quite well (what is shown in 15.

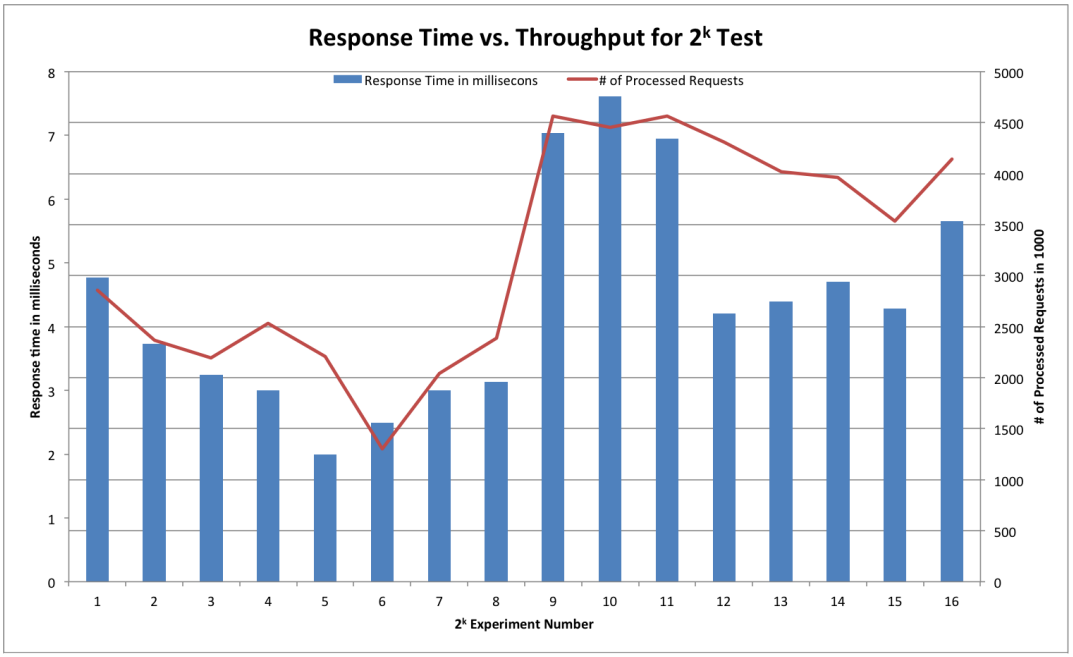

Figure 15: 2^4 factorial measured results

Mean performance concerning response time has a value of 10.24. The effect of varying number of clients from 15 to 30 has a weight of 3.43. This means when increasing the number of clients (and number of requests) the response time also increases. The other values are relatively low. Number of database connections d and number of brokers b in combination with number of database connections are the next values worth to have a look at. This means that the number of database connections has the most significant impact on response time. By varying number of brokers the total number of database connections also change.

Overall the most significant factor which affects performance the most (beside the load applied by clients) is the number of database connections. This indicates that the database is a bottle neck.

4.6 Database Load Test

To check the limits of the system a load test was performed where we start with a fixed number of middle ware instances and gradually increase the number of connected clients.

Middle Ware In total 16 middle ware instances are evenly spread across 4 Amazon instances. Each middle ware has 20 WorkerThreads and 15 concurrent database connections.

Clients Every 2 minute 20 OneWayClients try to connect. Each client tries to send and receive messages as fast as possible. In total 450 clients are spread across 15 Amazon instances (30 clients per instance).

RequestTimeout is set to 1 second. A request is considered to be successful if processed within that bound.

4.6.1 Hypothesis

By increasing the load on the system both response time and throughput should increase to a certain degree. By increasing the load more the system should become unstable and start trashing.

4.6.2 Results

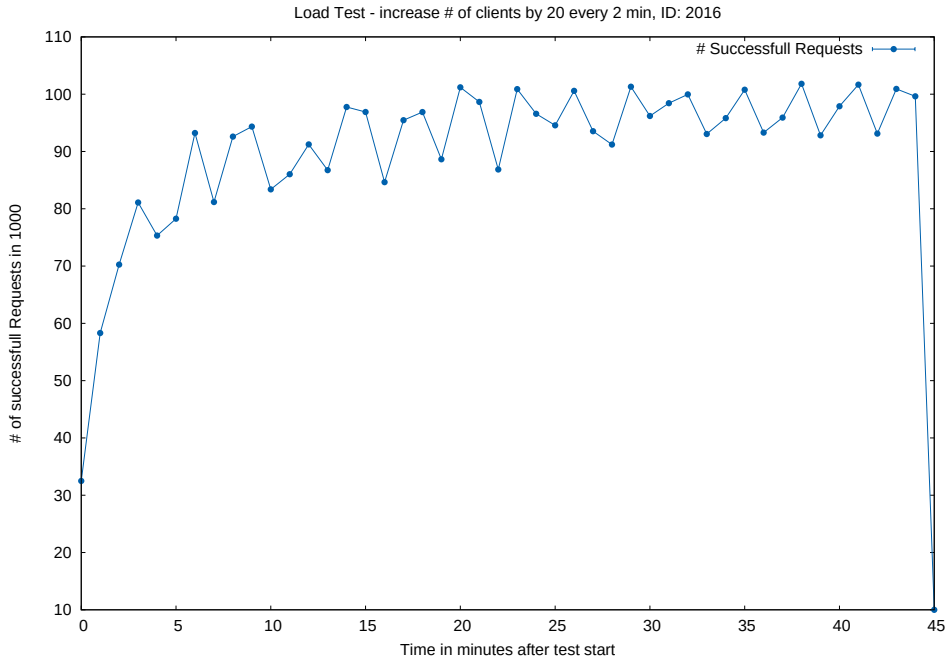

Figure 16: Throughput of load test in 1000 requests.

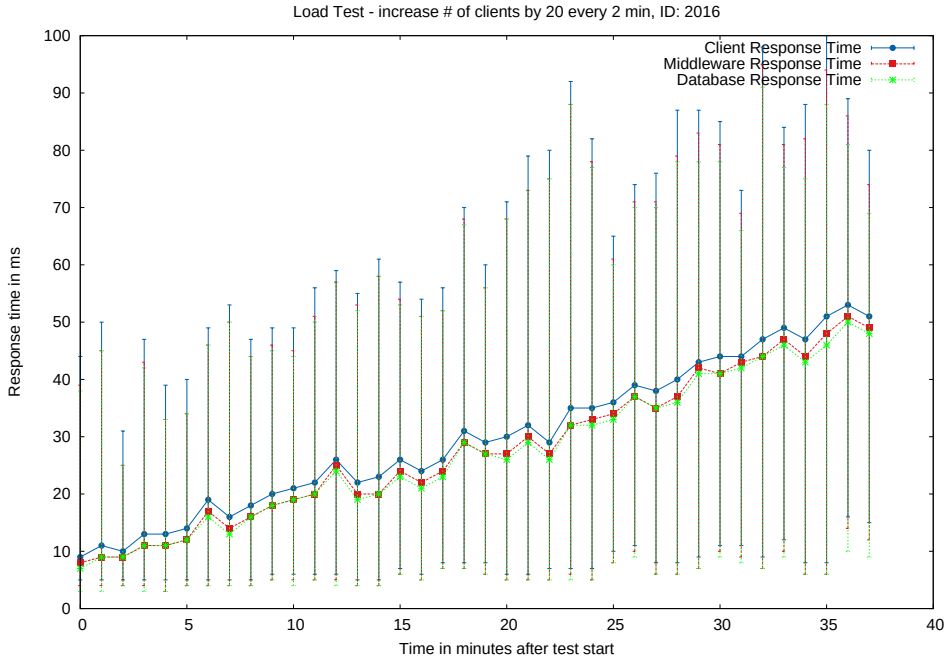

Figure 17: Response time of Load Test. The error bars are the 2.5% and 97.5% percentiles.

4.6.3 Interpretation

As expected the response time increases as the load on the system gets higher. Also worth mentioning is that the lines in Figure 17 remain parallel. This means that the all component except the database remain stable concerning response time. The database needs more and more time to process queries while load is increased.

The number of successfully processed requests reaches its peak between 5 and 10 minutes (approximate with 80 clients). The system is then saturated.

This result is quite disappointing since it does not support our hypothesis. That type of experiment was repeated several times while increasing the number of total clients and still the system seems not to be saturated. It was decided to not repeat the experiment with even more clients (because our Amazon voucher was used up).

4.7 Middleware Load Test

This test tries to find the limits of a single middleware component.

Middleware The middleware is kept at a minimal configuration where the number of workers and the number database connections is set to 1. This way it should be able to show the limitations of a single middlware instance.

Clients The number of connected clients is increased by 2 every minute. For simplicity reasons only **OneWayClients** are chosen to generate load. To check the limits of a single middleware instance a load test is performed where twith a single instance.

4.7.1 Hypothesis

Because the middleware has only one database connection it should not happen that the database becomes the bottleneck (See Database Load Test). It is expected that the middleware starts trashing as the number of clients becomes too big.

4.7.2 Results

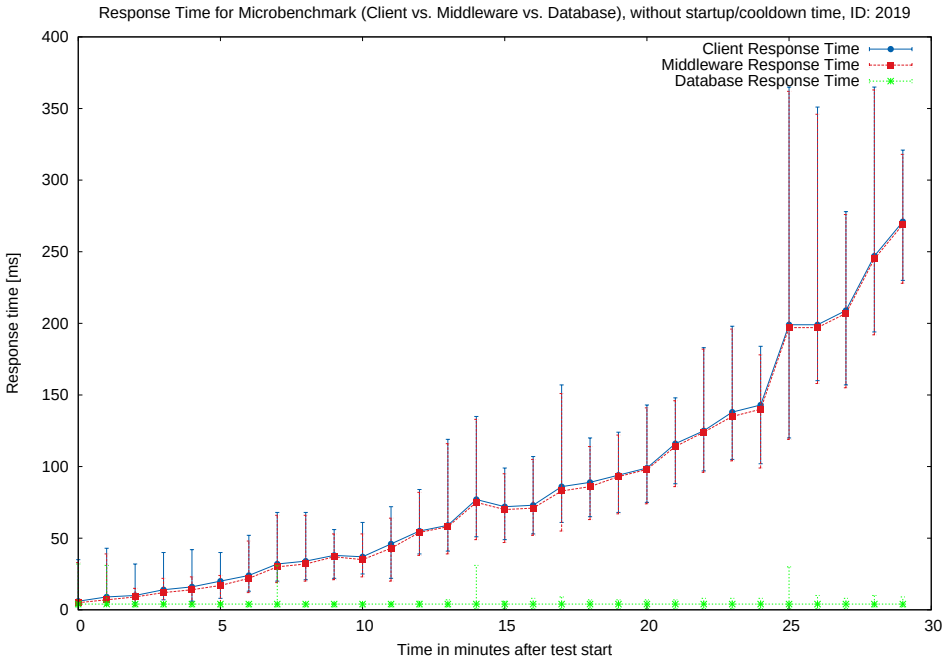

Figure 18: Response time of Middleware Load Test. The error bars are the 2.5% and 97.5% percentiles.

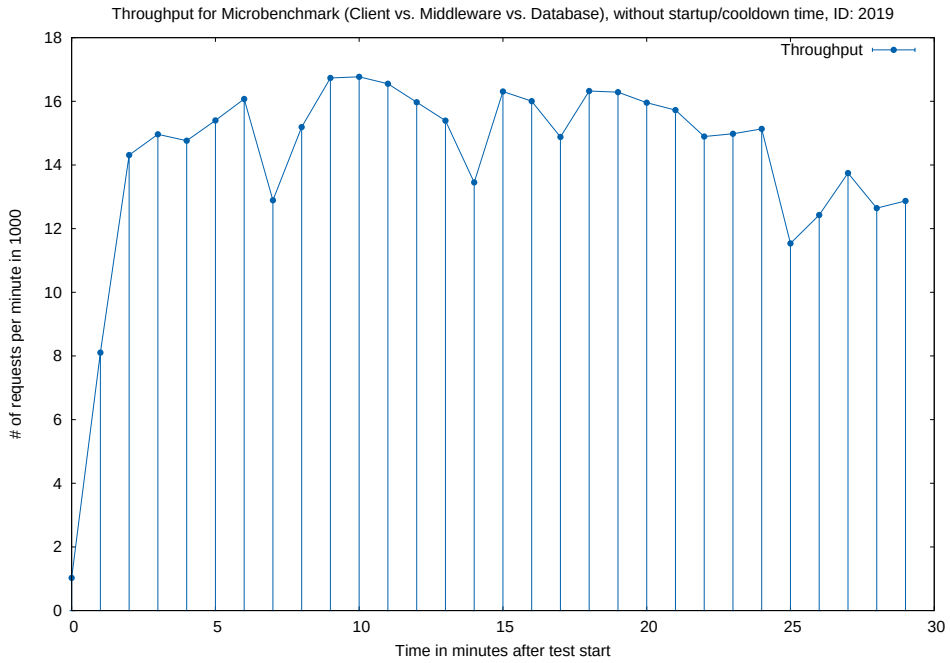

Figure 19: Througput of Middleware Load Test.

4.7.3 Interpretation

Both graphs show a significant drop in performance after 25 minutes. The queues of the middleware run full and requests are dropped. As a result of this clients time out and later try to reconnect. We expect that a number of clients disconnect at about 25 minutes (50 clients). This explains the sudden drop. A few moments later the clients connect and overload the system again.

4.8 Performance Breakup for Different Tiers in the 2h Test

To find out which components spend how much time, a performance breakup has been conducted for sending and receiving messages.

4.8.1 Hypothesis

The 2h test only generates load on the middleware, and load on the database. While the middleware should already be a little stressed out, the database shouldn't have to do too much work. We expect that the system is stable and the database queries are fast.

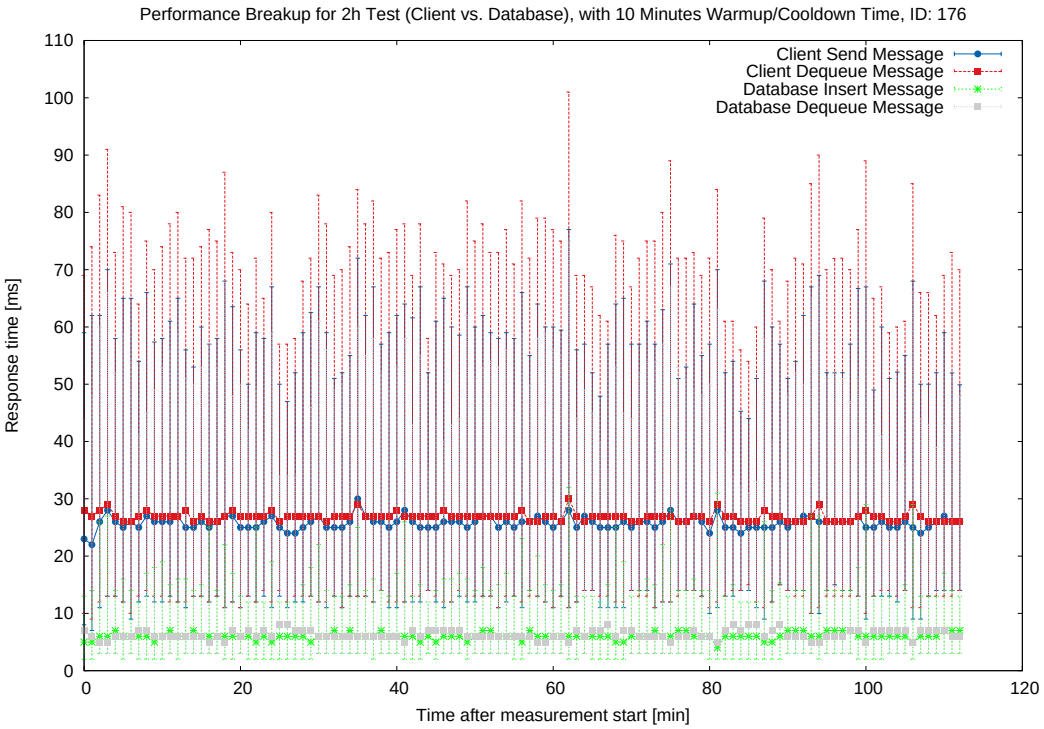

Figure 20: Performance breakup for the 2h test.

4.8.2 Interpretation

As expected, the system runs stable with a median response time between 20ms and 30ms. 95% of the deque message response time is in between 10ms and 90ms, while the send message response time stays under 60ms-70ms for 97.5% of the send message requests. Thus while sending messages is a little bit faster (about 2ms-3ms) then dequeing messages, inserting messages is more predictable.

5 How to run the System

This section describe all the important things laying around the core messaging system.

5.1 Database setup

To setup the database required for the middleware there are two options how to create one.

5.1.1 Automatic setup

You can run the java class *ch.ethz.mlmq.main.Main* which is located in the main project folder *mlmq*. Run the class without parameters to get usage informations.

Sample Start parameters may look like this:

```
java ch.ethz.mlmq.main.Main dbscript -url jdbc:postgresql://localhost:5432
-db mlmq -user myusername -password mypassword -createDatabase -createTables
```

5.1.2 Manual setup

Use your favourite database management tool (pgadmin) and execute the following sql scripts which can be found here:

```
resource/db/001_table_create.sql
resource/db/002_stored_procedures.sql
```

5.2 Configuration

Configurating software components is done with properties file. [7]

5.2.1 Main Configuration

Have a look at the example configuration located at the following place.

```
resource/config.example.properties
```

Most of the configuration should be self explanatory. An exception may be the the part which is dynamically generated part when executed via test infrastructure (see figure 21). This part of the configuration describes the configuration of an experiment.

```
common.scenario.mytype
```

Describes whether this configuration file belongs to a client or a middleware instance.

```
common.scenario.myposition = n
```

Together with *common.scenario.mytype* it determines whether to pick the n-th configuration from *common.scenario.mapping.broker* or *common.scenarion.mapping.client*.

```
common.scenario.mapping.broker=...
```

[7] See `http://docs.oracle.com/javase/tutorial/essential/environment/properties.html` for further information.

Lists the Broker scenarios to use. Where the first item corresponds to a class name followed by ip address and port

```
common.scenario.mapping.client=...
```

Lists the client scenarios to use. Where the first item corresponds to a class name followed by ip address.

```
...

################################################
# Dynamically generated per test run from here on
################################################

...

# Scenario mapping format: name1:ip,ip,ip,...;name2:ip,ip,...;name3:ip,ip,...
# E.g. scenario.mapping = broker:127.0.0.1,192.168.0.2;client:127.0.0.1,192.168.0.1
# Here: only one broker and one client
common.scenario.mapping.broker = SimpleShutdownBroker#127.0.0.1:8099;
SimpleShutdownBroker#127.0.0.1:8100,127.0.0.1:8101
common.scenario.mapping.client = SimpleSendClient#127.0.0.1;SimpleSendClient#127.0.0.1;
SimpleSendClient#127.0.0.1;SimpleSendClient#127.0.0.1,127.0.0.1

# either broker or client
common.scenario.mytype = client

# If the position is 5 and mytype is a broker, then this means that this is the 5th broker
# myposition starts at position 0
common.scenario.myposition = 0

...
```

Figure 21: Sample Config

5.2.2 Logging Configuration

Both the messaging system and the client implementation use *java.util.logging.** to log system events like errors, component startup/shutdown. This log is completely separated from the performance log (see 5.2.3), which is performed on a separate basis.

A valid logging configuration can be found here

```
resource/logging.properties
```

5.2.3 Performance Logging Configuration

As already state performance logging is separated from the default logging. Any activity which may be interesting to measure is logged via this performance logger. This logger is configured via the main configuration file (see 5.2.1) and is kept simple. You just specify a path where the performance logger puts the file

```
# folder where to write the performance log
common.performancelogger.logfilepath = performance_log
```

5.3 Start a Middleware instance/Client

As soon as both main and logging configuration files are prepared starting the middleware instance is easy. Just specify both files via command line arguments like the following:

```
java ch.ethz.mlmq.main.Main scenario -config resource/brokerconfig.properties
-l resource/logging.properties
```

Since configuration whether to start as a client or as a server is located inside the main configuration file a client is started in the same way as a middleware.

5.4 Command Line Interface

Both client and middleware instance can be controlled using basic commands. These commands are written into the so called command file which is configured in the main configuration like this.

```
common.commandofile.path = commando.txt
```

While the client or the middleware instance is running one can write a single command into this file to be executed. The middleware periodically checks whether the file has changed. If so it tries to execute the command contained in the file.

command	description
shutdown	Gracefully shuts down the middleware
logstacktrace	Writes the current stack trace of all threads in the virtual machine to the log
logmemory	Writes the current memory consumption to the log

Figure 22: Commando File Commands

5.5 Logging

This chapter describes what is logged rather than how the logging is configured. (Check 5.2.3 and 5.2.2 to see how the logging is configured).

As stated logging is performed with *java.util.logging*. This files are for debugging purposes only.

The **PerformanceLog** contains timing data which are used to reason about the timing behaviour of the system which has been build.

Except for the first line of the file the structure of the **PerformanceLog** has a csv file format.

5.5.1 Columns

represent the following values

1. Time taken for a specific action
2. UTC timestamp when a specific action was finished

3. Name of the action
4. Different runtime dependant context information. (E.g. C[.] indicates the number of connected clients on a middleware instance)

5.5.2 Action Names

The following list shows the different types of actions which are logged and their meaning. Action names were chosen to be as short as possible to reduce log file size.

- **CSndReq** - request/response rountrip measured from a client's perspective
  - **BTotReqResp** - time taken from the first received byte to the last sent response byte measured from a broker's perspective.
  - **BRcvReq** - time taken to receive a Request. This measures how long it takes to receive all request bytes on the broker.
  - **BProcReq** - time taken by the worker. Tells you how long a single worker thread is occupied processing a certain request.
    - **BDb** - time taken to perform a specific database operation. Since this is measured by the broker jdbc network communication is included here.
  - **BSndResp** - time taken to send a response.

5.5.3 Example Performance Log

An example performance log is shown in figure 23

```
BrokerConfiguration Scenario[SimpleShutdownBroker] Port[8099]
WorkerThreadCount[4] DbConnectionPoolSize[4]
4;1383684920403;BRcvReq;C[9]
0;1383684920443;BRcvReq;C[10]
93;1383684920533;BDb#getClientId;C[10]
26;1383684920561;BDb#insertNewClient;C[10]
7;1383684920568;BDb#createClientQueue;C[10]
129;1383684920569;BProcReq#RegistrationRequest:RegistrationResponse;C[10]
0;1383684920570;BSndResp;C[10]
132;1383684920571;BTotReqResp;C[10]
1;1383684920571;BDb#getClientId;C[10]
14;1383684920572;BDb#createClientQueue;C[10]
166;1383684920572;BProcReq#RegistrationRequest:RegistrationResponse;C[10]
8;1383684920573;BDb#createClientQueue;C[10]
132;1383684920573;BProcReq#RegistrationRequest:RegistrationResponse;C[10]
```

Figure 23: Sample Performance Log

5.6 Unit Tests

While developing the system we strictly relied on unittests which are located in the test folder of the mlmq eclipse project.

In order to have the database related and system unittests working you may need to adjust database connection information located in the following file. This configuration file is used by all Unit Tests which need client specific configuration.

`resource/brokerconfig.properties`

6 Test Infrastructure

To partially automate testing a Ruby on Rails[8] web application was developed to help simplify running experiments on the Amazone cloud. It is refered as Testmaster.

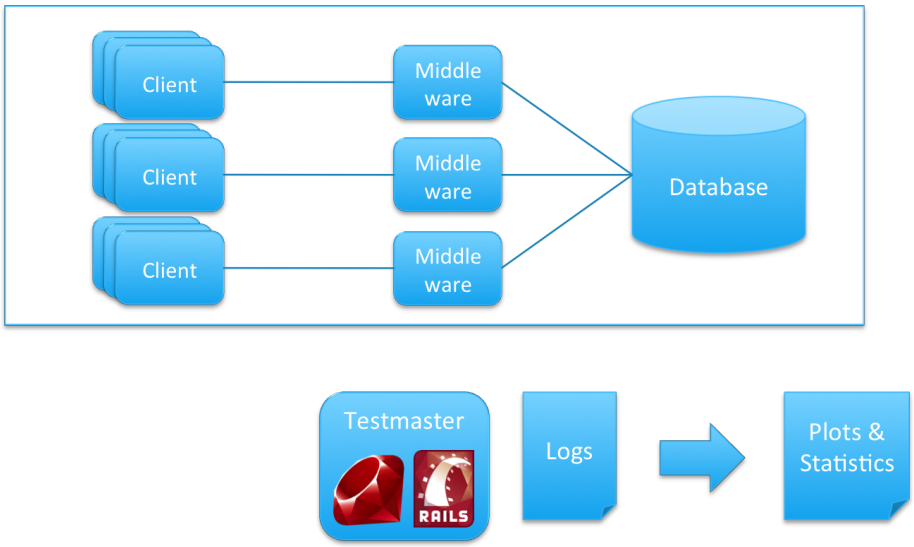

Figure 24: Test Master Overview

6.1 Workflow

6.1.1 Create an Experiment

To run an experiment at first one has to define a new testrun. This is done by hand via a web interface. One has to provide the following information:

- Define the number middleware instances are set up on how many Amazon instances.
- Define the workload. This implies choosing how many instances of each client type are needed.
- Provide client and middleware configuration.

6.1.2 Run an Experiment

This step is automatically performed without user interaction. The test master performs the following steps:

1. Start the required Amazon instances
2. Compile the latest binaries of the middleware and clients from github
3. Copy the binaries and configuration to the Amazon instances

[8] `http://rubyonrails.org`

4. Start all the binaries with the required parameters
5. Periodically check whether the experiment is still running.
6. If the experiment has finished, logging information is gathered, compressed and stored on the test master

6.1.3 Analyze an Experiment

As soon as an experiment has finished and all logging information is collected the web interface enables two buttons "Analyze" and "Downlodad".

"Download" allows to get zip file with all the log files created during the experiment.

"Analyze opens a form which provides a web interface for the Log Analyzer (see 6.2). Plots can be created with only a few clicks.

6.2 Log Analyzer

In order to semi automatically analyze performance log files which have been created through the process of an experiment, a Java program was written to simplify evaluation of these files.

The project folder (See Section 7.1) contains an Eclipse project called *log_analyzer*. All you need to do is to provide a path to the folder containing all the log files of an experiment you want to analyze. By executing the class *ch.ethz.mlmq.log_analyzer.Main* you can choose to either generate a csv file or a gnuplot file[9].

6.2.1 Example

The following line of code shows an example of calling the log analyzer.

```
java ch.ethz.mlmq.log_analyzer.Main -d logs/test_run_56 -type CSndReq#OK# -fmt csv
```

The folder found at logs/test_run_56 which contains several log files is analyzed and a csv file is created. This file contains timing information for client request/reponse operations (See 5.5.2).

[9] `Seehttp://www.gnuplot.info/`

7 What to find where

This section should help finding code, experiment data, plots etc.

7.1 Messaging System Source Code

All source code for the messaging system, clients and log analyzer can be found here:

`https://github.com/ganzm/AdvancedSystemsLab2013` and in the svn repository in the folder *ethz-asl-mlmq*.

7.2 Experiment configurations for each test run, raw data, big files

The configurations for all test runs, the raw data and the big files can all be found in this repository. Because there is some sensitive information in the configuration files (like database passwords for the middleware), please request access to this repository by writing us your github username. Please handle this data carefully, don't publish it and don't share it.

`https://github.com/lukaselmer/ethz-asl-bigfiles` and in the svn repository in the folder *ethz-asl-bigfiles*.

7.3 Testmaster Source Code

Ruby on Rails[10] Testmaster source code is located here:

`https://github.com/lukaselmer/ethz-asl-testmaster` and in the svn repository in the folder *ethz-asl-testmaster*.

It is a standard Ruby on Rails project, so the installation is easy for a Ruby on Rails developer. However, this can be very tricky if this environment is unknown. Therefore it is suggested to use the Testmaster Access described in the next subsection.

7.4 Testmaster Access

The Test Infrastructure as described in Section 6 Test Infrastructure. This is where you find all experiment data (even the failed ones). Access to this service is communicated by mail, because the service contains sensitive data (like database passwords for the middleware).

`https://testmaster-asl-eth.renuo.ch`

[10] http://rubyonrails.org/

8 Conclusion and lessons learned

Actually measure a system felt much more difficult than actually building one. Having the plots right seems more difficult than having UnitTests on a green bar.

UnitTests can't show the absence of bug. Despite of having a decent test coverage we struggled at this point. As the project moved on, more and more bugs where discovered and fixed. Each time a software bugfix was applied which potentially affect the performance of the system some of our experimental results were invalidated, or at least not entirely correct any more.

An other point which should be improved for the next project is the decision making process. We delayed what exactly to measure and to analyse for too long.

Despite the fact that much time went into building the Testmaster, in the end it was a big help. Not heaving the Testmaster in the end phase would probably have meant, that either the bugs couldn't have been fixed or that not so many experiments could have been conducted.

Another lesson learnt was that applying statistics can be tricky. Even tough the theory is not that complex and difficult, there are many ways to apply statistics. Not heaving clear specifications for what to measure and what to build in the first place made this task really difficult.

In the end, we think that we built a good messaging system. More importantly however we know how to measure systems and we think that we have a good understanding how our system performs. And of course, we are looking forward to the next part.