Advanced Systems Lab

Erik Jonsson (*jerik*) and Michael Bang (*mbang*)

November 15, 2013

Contents

0.1 Notation

In this report the following notation is used:

- Confidence Intervals for Normal[1] Distribution: 10 ± 5 - the 90% confidence interval is [5,15].
- Confidence Intervals for Unkown and Asymmetrical Distributions: $10 \pm [-3, 5]$ - the 90% confidence interval is [7,15].
- Standard deviation: The standard deviation is represented by the error bars in the figures and graphs.

[1]For all metrics which are assumed to be normally distributed there is a histogram of the values in the appenix.

1 Implementation description

In our message passing system we have decided to make the client as simple as possible by placing as little logic as possible here. The middleware provides a well-defined API that the clients can use to interact with the messaging service. The middleware interacts with a database using SQL. Figure 1 is a graphical representation of our system. In the following we give an explanation of the life of a request, starting from and ending at the client. The reader can follow these steps on Figure 1.

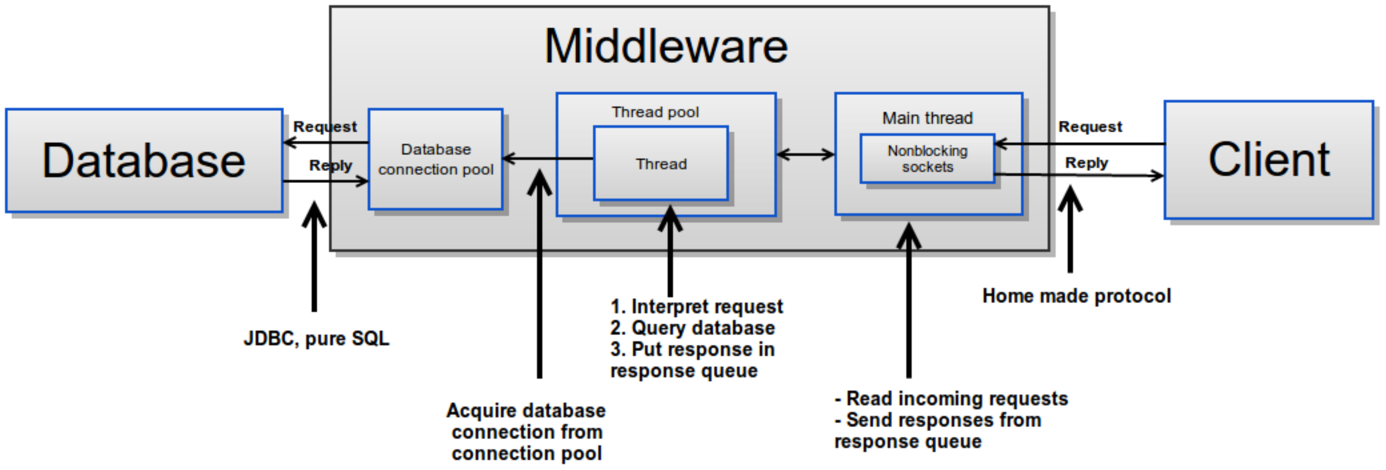


Figure 1: Graphical representation of implementation

1. Client X sends request to middleware
2. Middleware receives request on non-blocking socket in the network i/o
3. Middleware passes raw request data on to worker thread X from the worker thread-pool
4. Worker thread Y interprets the request
5. Worker thread Y fetches a database connection from the database connection-pool, and performs a query matching the client's request
6. Depending on the result of the query, worker thread Y figures out what to respond to the client and puts the response in the response queue
7. Next time the network i/o thread goes through the response queue, it will find response Y and send it to client X

There can, of course, be more than one middleware in our system. In this case the figure would be the same, as seen from the viewpoint of a client, because

each client is connected to exactly one middleware.
In the following subsections we explain our design decisions with regard to each component of our system: client, middleware, and database.

1.1 Client

As noted earlier, we have tried to keep the implementation of our client as simple as possible. This means that our client implementation only consists of code that implements the API described in Appendix A. Basically, our client code is a library that can be used in other Java applications to communicate with our middleware.

The client has the following important Java **packages** and *classes*:

- **asl**
  - *ThorBangMQ* - 'API': implements sendMessage, popMessage etc.
  - *Message* - Representation of a message
- **asl.network**
  - *SocketTransport* - Transport layer using sockets
- **asl.infrastructure**
  - *MemoryLogger* - Logs to memory with the possibility to dump logs to a file later
- **asl.infrastructure.exceptions**
  - *InvalidClientException* - Invalid client id
  - *InvalidQueueException* - Invalid queue id
  - *InvalidMessageException* - Invalid message id
  - *ServerException* - Unknown exception at server, something is very wrong

1.2 Middleware

In our middleware we have chosen to use non-blocking sockets instead of blocking sockets since it as a few properties that we like, one of which is that we don't have to spawn a new thread for every client that connects. This means that it has a very small impact on our server when a client connects, and that we can support many more clients simultaneously since the amount of threads in our program doesn't have to be linear to the amount of clients currently connected. There is an obvious disadvantage to choosing non-blocking sockets over blocking sockets though, which is that it can be harder to think about, implement, and work with.

In an attempt to make our implementation simpler we decided to perform all network i/o in one thread of our program. Since we want to be able to handle multiple requests simultaneously, we pass data from the network i/o-thread to worker threads for interpretation and handling. These worker threads interpret incoming data and perform the database queries needed to handle requests. Responses to these requests are put in a response queue, which the network i/o-thread will empty as often as it can, by sending sending replies directly to clients.

By moving interpretation and handling of requests to a thread different from the network i/o-thread, we have a place to perform our (blocking) calls to the database without blocking incoming requests. This is not entirely true though, since if there are more simultaneous requests than there are worker threads, the incoming request will be placed in a the thread pool queue and will effectively be blocked, though not it is not technically blocking the flow of the program.

We have chosen to use a thread-pool for our worker threads in order to avoid the overhead of creating and deleting threads all of the time, and at the same time bound the number of worker threads our program will spawn. This should make our system able to handle many requests than we have threads, at the cost of slower response times. It is important to note that this also means that the amount of simultaneous requests that our implementation can handle is equal to the minimum of the number of worker threads and the number of database connections given that no requests are invalid requests or HELLO requests (see A.1) since all but those go to the database. This is the case since a worker thread always needs a database connection to do its work (unless it received a malformed request, which shouldn't happen during our tests).

With regard to database connections, we have chosen to use a connection-pool to avoid the overhead of setting up a new connection to the database each time we want to perform a query.

The middleware consists of the following important Java **packages** and *classes*:

- **asl**
  - *Main* - Entry point of the program
  - *IntervalLogger* - Logs test data every x second, configurable
  - *GlobalCounters* - Holds global counters used during tests
  - *Message* - Representation of a message
  - *ServerSettings* - Configurable parameters of the server
- **asl.infrastructure**
  - *MemoryLogger* - Logs to memory with the possibility to dump logs to a file later

- **asl.infrastructure.exceptions**
    - *InvalidClientException* - Invalid client id
    - *InvalidQueueException* - Invalid queue id
    - *InvalidMessageException* - Invalid message id
    - *ServerException* - Unknown exception at server, something is very wrong
- **asl.network**
    - *SocketTransport* - Transport layer using sockets
- **asl.persistence**
    - *PostgresPersistence* - Use Postgres as storage
    - *InMemoryPersistence* - Use local memory as storage
    - *LyingPersistence* - Don't store anything

It should be noted that in our implementation of *Pop message* we are using 'Peek message'. This causes two accesses to the database whenever a client performs a *Pop message*-request that returns a message. We only realized this after doing our tests, which is why we kept the implementation as described.

The flow of an individual request can be seen in Figure 2. Some things to note about Figure 2:

- The last step in the ClientRequestWorker, *Add To Response Queue*, does this by using the interface *asl.network.ITransport* which is passed to the ClientRequestWorker by the Socket I/O component. The ITransport-inferface declares just one function *void Send(String s)*. The production implementation of this interface, *SocketTransport*, keeps a reference to the response-queue which is continuously polled by the non blocking socket I/O. This way the Client Request Worker and the underlying networking implementation is decoupled.
- The Socket I/O and Job Creation component does more in one iteration than just check the response and requests queues. It also accepts client connections though this was omitted in the figure for clarity.

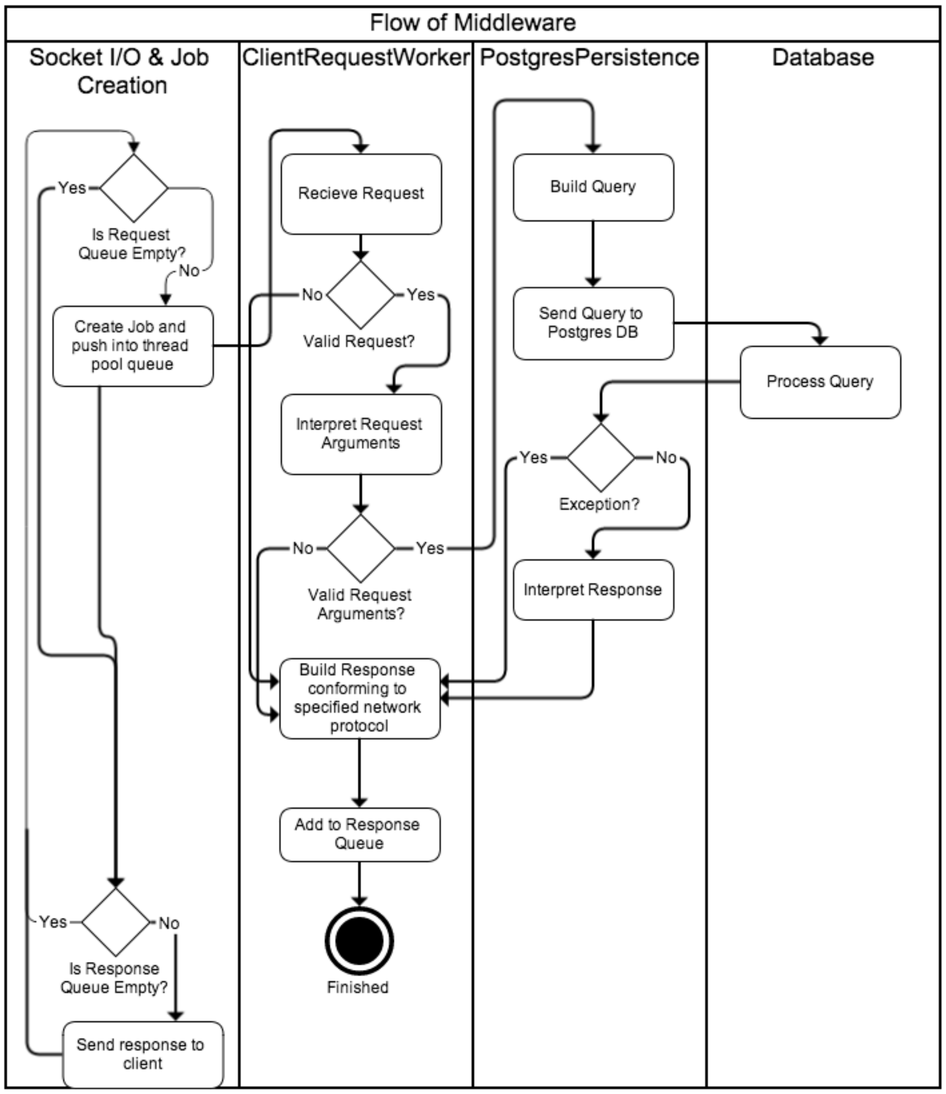


Figure 2: The flow of the middleware's non-blocking sockets I/O and the path of an individual request.

1.3 Database

We have chosen to use a simple database schema with three tables: messages, queues, and clients. This schema is illustrated on Figure 3. We have used the following two multicolumn-indexes on the *messages*-table: *(receiver_id, queue_id, time_of_arrival)*,*(receiver_id, queue_id, priority)*,*(receiver_id, queue_id, priority)* and *(receiver_id, queue_id, priority)*.

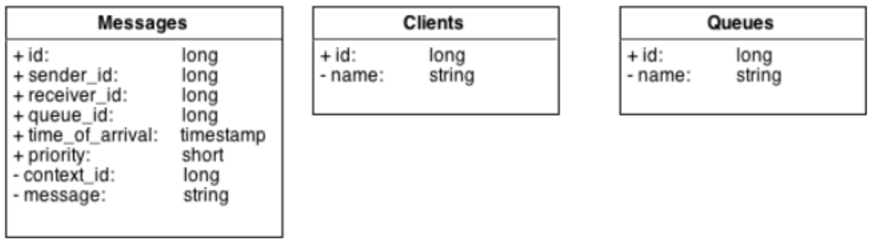


Figure 3: Database schema

When clients send messages to multiple queues at once, we have chosen to replicate messages in queues rather than having a many-to-many relationship in the database. This decision could affect performance since we're inserting more messages than we would, had we chosen to use a many-to-many relationship.

Communication between the middleware and our database is done using JDBC, and we make SQL queries directly from our Java code. We decided not to use stored procedures, even though it requires slightly less data to represent each query, because we found it easier to implement and debug SQL directly in our Java code.

1.4 Communication protocol and API

We have chosen to use a very simple messaging protocol between the clients and the middleware. It uses an end of message token to differentiate messages from each other as we found it simpler to implement, compared to defining packets with headers containing message length and so on. We decided that our end of message token should be null, i.e. "\0". All messages should be encoded in UTF8.
Interpretation of requests and responses using our protocol is implemented in both server and client, in the SocketTransport classes. The full description of the communication protocol and API can be found in Appendix A.

2 Testing infrastructure

Our testing infrastructure is written in Python and does the following:

- Starts and stops servers
- Starts and stops experiments
- Fetches logs
- Generates graphs

Figure 4 gives a graphical interpretation of our testing infrastructure.

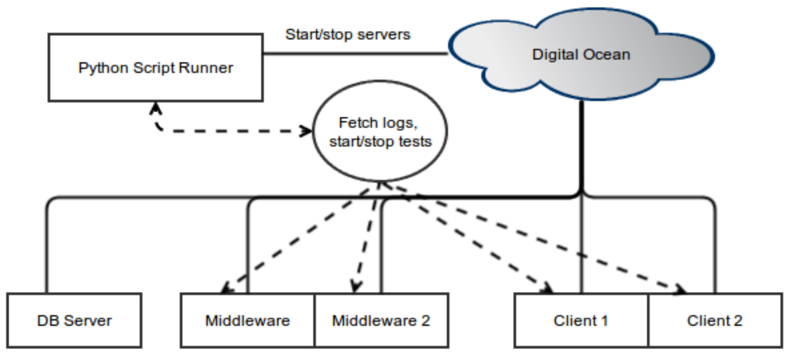


Figure 4: Graphical representation of our testing infrastructure

The infrastructure consists of 4 files:

main.py Interface to start tests, see servers currently running etc.

infrastructure.py Functions doing the actual infrastructure work.

droplets.py Functions to start and stop servers (specific to DigitalOcean cloud provider).

gnuplot.py Plots data.

Tests are defined by creating a folder in the test-definitions directory and creating three files: *conf.txt*, which defines the configuration of a server, *test.txt*, which defines the configuration of the clients and the number of servers and clients the test should be deployed on, and *gnuplot.sh* which generates gnuplots when called with a test-result file as an argument.

Other than the files described above, our testing infrastructure relies on the following python libraries:

python-digitalocean which can be installed via pip or found at *https://pypi.python.org/pypi/python-digitalocean*.

psycopg2 which can be found at *https://pypi.python.org/pypi/psycopg2*.

It should be noted that our testing infrastructure is very simple in that, for instance, it doesn't check for errors while running; it just assumes that everything executes successfully and continues on, even though that is not always the case. We found this to be a somewhat of a mistake since it has caused us quite a bit of trouble during our testing.

3 Test definitions

This section contains a description of the tests that we performed during our measurements of our system. These tests can be configured in various ways in order to test different properties of our system.

3.1 Send and Pop Same Client

In this test all client threads continuously send 'SendMessage' and 'PopMessage' requests to the server (described in Appendix A). Each client thread sends a message to itself, then pops the message it just sent, and then repeats. This test is designed to keep the amount of messages in the database stable, while potentially putting a lot of load on the database - depending on the configuration of the parameters of the test. The test is configurable in: the number of (identical) threads to be spawned, time the test should run, the size of messages sent, the number of queues being used, and the amount of time each thread should sleep between sending requests. In all tests where nothing else is mentioned, the waiting time between requests is 0.

Tests of this type perform the same amount of 'SendMessage' and 'PopMessage' requests, which means that the amount of messages in the system is kept stable. We expect that this test could put a lot of load on the server, especially if the amount of sleep-time is very low.

3.2 Standard test

This test is an implementation of the test described in the project description, which we're supposed to be using for the traces. Our interpretation of the description, and our implementation, is as follows: A number of one-way clients bounce a single message randomly between each other, incrementing a counter in the message. Each one-way client keeps performing *Pop message*-requests, checking if he got the message. A number of two-way clients each have a partner assigned, with whom they keep sending messages back and forth. Each two-way client keeps sending *Pop message*-requests until he receives a message from his partner, and then sends a reply back. From this description we should note that the amount of *Send Message*-requests doesn't change when the number of one-way clients increases, but it does when the number of two-way clients increases. Also to be noted, the number of *Pop message*-requests increases when incrementing either of the two types of clients.

The test is configurable in the amount of one-way clients and two-way clients it should use. Only too late to fix it did we realize that it would have been nice to be able to configure the size of messages sent, and the number of queues the messages are distributed over.

Tests of this type perform many more 'PopMessage' requests than 'SendMessage' requests, but most of these 'PopMessage' requests will not actually find a message to pop, causing the total number of messages in the system to be kept stable.

3.3 Push peek pop

In this test a single client performs a configurable amount of a single type of request. This test is used for micro benchmarking each type of request. The test can be configured in: the type of requests sent and the amount of requests sent. This test can't be configured from the command line, and must be performed manually.

Depending on the request being benchmarked, the amount of messages in the database may change over time. *Send Message* requests, for instance, will increase the amount of messages in the system, while *Pop message* will decrease the amount of messages in the system.

3.4 Timing measurements on the middleware

On the middleware we perform some timing measurements, measuring how much time is spent in different parts of our system. These names that we use for these parts are:

IPersistence which uses JDBC to query our database. Time spent in this class includes: think time for a database query and network delays from our middleware to the database and back again.

Client Request Worker which interprets user requests. Time spent in this class includes: string manipulation to interpret user requests, calling of methods in IPersistence, creating a response for the client and putting it in the response queue. The time spent in IPersistence is subtracted from the time measured in this class such that time measured for this class doesn't include the time it takes to perform calls in IPersistence.

Socket I/O & Queuing which handles communication with clients. Time spent in this class includes: reading from and writing to a client socket, handing data read from client sockets to a worker thread, and the time requests spend waiting to get handed to a worker thread.

Figure 5 is a graphical representation of these measurements points.

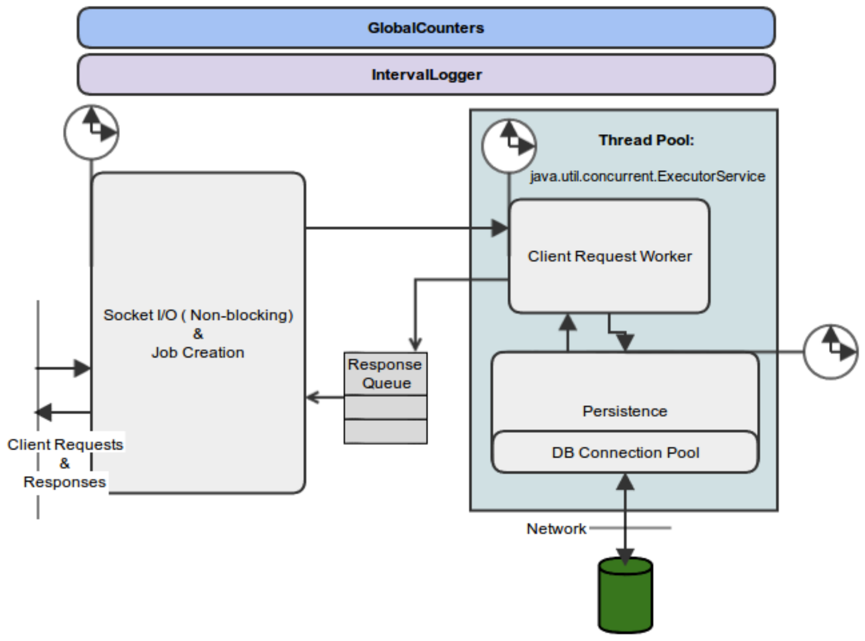


Figure 5: Graphical representation of timing measurements done on middleware

4 Tests and results

In this section we describe the tests we have performed, present the results and give a short commentary on what can be seen on the graphs. We interpret and analyze these results in Section 5.

In our tests we have focused on peeking, popping, and sending messages. We decided to have this focus because we work under the assumption that these types of requests are vastly more frequent than all other types of requests (create queue, remove queue etc.), and that the performance of the system as a whole will be dominated by these three types of requests.

We have performed our tests using machines from the cloud hosting provider DigitalOcean.com, from whom we got a 200$ coupon to run experiments. Running our tests on a cloud host has some obvious downsides w.r.t. the results of our tests, since we can't know for sure which hardware our tests are run on, and if we are sharing this hardware with anyone.
We have run all of our experiments on machines with: 4GB memory, SSD storage, and 2 logical CPU cores. Due to the way DigitalOcean provide their servers, we can't really tell how much computing power these logical CPUs provide, or whether there are any neighbors on the hardware. In order to compensate for

this we've run all of our experiments for at least 10 minutes each, to see whether the observed performance is somewhat consistent.

For all tests where nothing else is mentioned, the amount of messages in the database has been kept steady at around 20,000 messages, and the size of messages (payload of a *Send message* request) is very low (around 5 bytes)

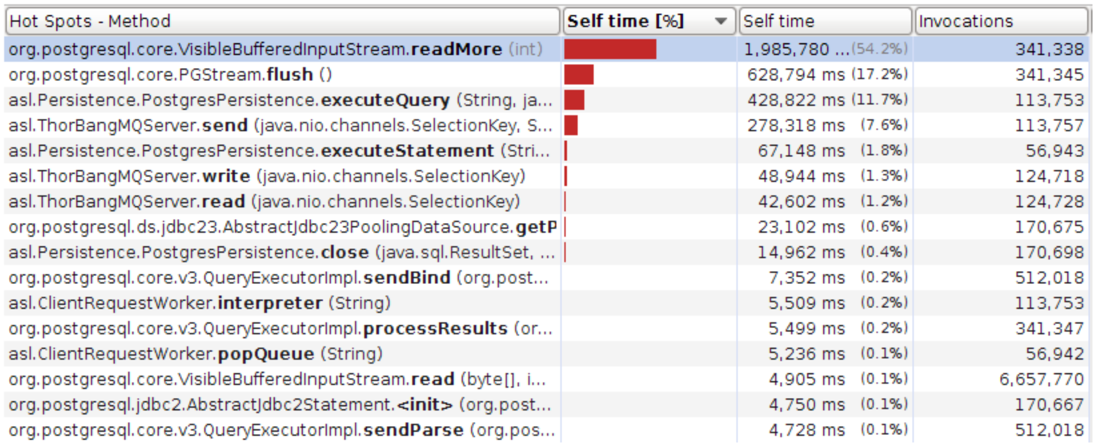
It is important to note that all of the following tests use blocking sockets (on the client side) to send requests. This means that each of our clients can not have more than one ongoing request or, put a different way, the total number of simultaneous requests in the system is at most the number of clients. In the literature this is referred to as a closed system.

In order to try to explain the performance of our system, we have identified the following parameters that we wish to test.

- Client
  - Size of messages
  - Frequency of requests
  - Total number of clients
  - Type of Requests
- Middleware
  - Number of middleware instances
  - Number of database connections
  - Number of worker threads
  - Impact of using database vs. not storing messages at all
- Database
  - Size of dataset
  - Number of queues which messages are distributed over

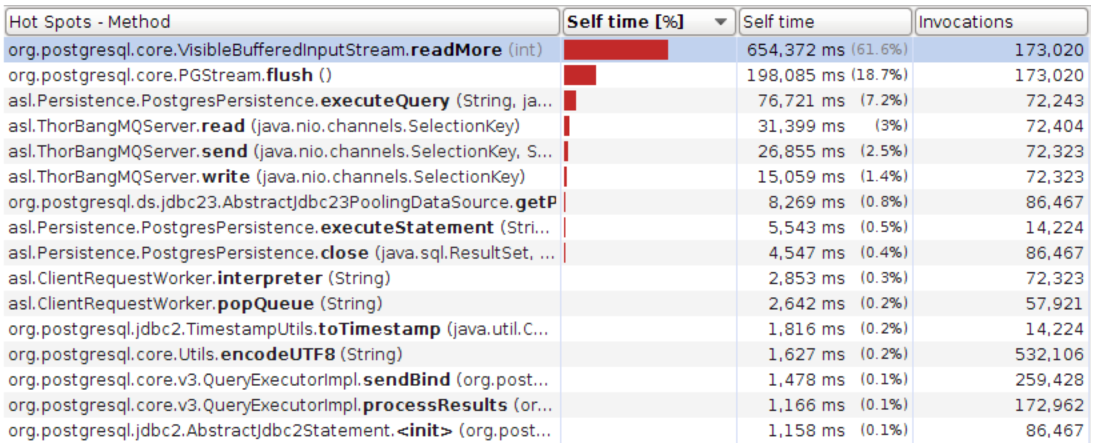
4.1 Code profiling

In order to figure out exactly where our code spends the most time and to confirm that out measurements were valid, we've used the code profiling tool VisualVM (*http://visualvm.java.net/*). We have performed code profiling during multiple different tests with multiple different configurations. Every time we ran the profiler we got almost identical results. The profiling runs shown on Figure 6 and 7 were run for approx. 3 minutes with 80 clients sending very small messages (around 5 bytes). These two tests show the general results we got with all configurations of these two types of tests. Our code spends almost all of the time, as shown on Figure 6 and 7, communicating with and waiting for the database. In fact, the top 3 most time consuming tasks are all due to communication with the database. If we put the time consumption of these tasks together, they add up to 83.1% for 'Send and Pop Same Client' and 87.5% for 'standard test'. In order to figure out exactly where our code spends the most time, we've used the code profiling tool VisualVM (*http://visualvm.java.net/*). We have performed code profiling during multiple different tests with multiple different configurations. Every time we ran the profiler we got almost identical results. The profiling runs shown on Figure 6 and 7 were run for approx. 3 minutes with 80 clients sending very small messages (around 5 bytes). These two tests show the general results we got with all configurations of these two types of tests. Our code spends almost all of the time, as shown on Figure 6 and 7, communicating with and waiting for the database. In fact, the top 3 most time consuming tasks are all due to communication with the database. If we put the time consumption of these tasks together, they add up to 83.1% for *Send and Pop Same Client* and 87.5% for *Standard test*.



Hot Spots - Method	Self time [%]	Self time	Invocations
org.postgresql.core.VisibleBufferedInputStream.**readMore** (int)		1,985,780 ... (54.2%)	341,338
org.postgresql.core.PGStream.**flush** ()		628,794 ms (17.2%)	341,345
asl.Persistence.PostgresPersistence.**executeQuery** (String, ja...		428,822 ms (11.7%)	113,753
asl.ThorBangMQServer.**send** (java.nio.channels.SelectionKey, S...		278,318 ms (7.6%)	113,757
asl.Persistence.PostgresPersistence.**executeStatement** (Stri...		67,148 ms (1.8%)	56,943
asl.ThorBangMQServer.**write** (java.nio.channels.SelectionKey)		48,944 ms (1.3%)	124,718
asl.ThorBangMQServer.**read** (java.nio.channels.SelectionKey)		42,602 ms (1.2%)	124,728
org.postgresql.ds.jdbc23.AbstractJdbc23PoolingDataSource.**getP**		23,102 ms (0.6%)	170,675
asl.Persistence.PostgresPersistence.**close** (java.sql.ResultSet, ...		14,962 ms (0.4%)	170,698
org.postgresql.core.v3.QueryExecutorImpl.**sendBind** (org.post...		7,352 ms (0.2%)	512,018
asl.ClientRequestWorker.**interpreter** (String)		5,509 ms (0.2%)	113,753
org.postgresql.core.v3.QueryExecutorImpl.**processResults** (or...		5,499 ms (0.2%)	341,347
asl.ClientRequestWorker.**popQueue** (String)		5,236 ms (0.1%)	56,942
org.postgresql.core.VisibleBufferedInputStream.**read** (byte[], i...		4,905 ms (0.1%)	6,657,770
org.postgresql.jdbc2.AbstractJdbc2Statement.**<init>** (org.post...		4,750 ms (0.1%)	170,667
org.postgresql.core.v3.QueryExecutorImpl.**sendParse** (org.pos...		4,728 ms (0.1%)	512,018

Figure 6: 3 minutes of profiling during *Send and Pop Same Client* with 80 clients, very small messages



Hot Spots - Method	Self time [%]	Self time	Invocations
org.postgresql.core.VisibleBufferedInputStream.**readMore** (int)		654,372 ms (61.6%)	173,020
org.postgresql.core.PGStream.**flush** ()		198,085 ms (18.7%)	173,020
asl.Persistence.PostgresPersistence.**executeQuery** (String, ja...		76,721 ms (7.2%)	72,243
asl.ThorBangMQServer.**read** (java.nio.channels.SelectionKey)		31,399 ms (3%)	72,404
asl.ThorBangMQServer.**send** (java.nio.channels.SelectionKey, S...		26,855 ms (2.5%)	72,323
asl.ThorBangMQServer.**write** (java.nio.channels.SelectionKey)		15,059 ms (1.4%)	72,323
org.postgresql.ds.jdbc23.AbstractJdbc23PoolingDataSource.**getP**		8,269 ms (0.8%)	86,467
asl.Persistence.PostgresPersistence.**executeStatement** (Stri...		5,543 ms (0.5%)	14,224
asl.Persistence.PostgresPersistence.**close** (java.sql.ResultSet, ...		4,547 ms (0.4%)	86,467
asl.ClientRequestWorker.**interpreter** (String)		2,853 ms (0.3%)	72,323
asl.ClientRequestWorker.**popQueue** (String)		2,642 ms (0.2%)	57,921
org.postgresql.jdbc2.TimestampUtils.**toTimestamp** (java.util.C...		1,816 ms (0.2%)	14,224
org.postgresql.core.Utils.**encodeUTF8** (String)		1,627 ms (0.2%)	532,106
org.postgresql.core.v3.QueryExecutorImpl.**sendBind** (org.post...		1,478 ms (0.1%)	259,428
org.postgresql.core.v3.QueryExecutorImpl.**processResults** (or...		1,166 ms (0.1%)	172,962
org.postgresql.jdbc2.AbstractJdbc2Statement.**<init>** (org.post...		1,158 ms (0.1%)	86,467

Figure 7: 3 minutes of profiling during *Standard test* with 80 clients, very small messages

On both Figures we also see that *ThorBangMQServer.read* and *ThorBangMQServer.send* take up a lot of time: 8.8% for *Send and Pop Same Client* and 5.5% for *Standard test*. These two calls are responsible for communication with clients, where 'read' gets clients' requests and 'send' sends replies to clients. On both Figures we also see that *ThorBangMQServer.read* and *ThorBangMQServer.send* takes up some time: 8.8% for 'Send and Pop Same Client' and 5.5% for 'Standard test'. These two calls are responsible for communication with clients, where 'read' gets clients' requests and 'send' sends replies to clients.

The last thing we note is that 'ThorBangMQServer.write' takes up 1.3% and 1.4% of *Send and Pop Same Client* and *Standard test* respectively. This method is responsible for putting responses into the response queue (as explained in Section 1.2) and is performed by worker threads when they are done handling a client's request.

4.2 Long trace

The figures in this section show the overall stability of the system while under moderate to high system load for 4 hours. The following test are of the type *Standard test*. In this test we expect to see that the system behaves in a somewhat stable manner meaning that we have a low standard deviation of our measurements. We expect that the system will have queued jobs at all times and that the throughput will be stable and that the response times may vary since we are doing different types of calls: *Send Message* and *Pop Message*. Clients and worker-threads and database connections increases since, as described in Section 1.2, the number of database connections and worker-threads bounds the amount of requests that will be handled simultaneously by our middleware. We don't think that this will keep scaling though, since increasing the number of database connections gives the opportunity to put more load on the database which, at some point, will not be able to cope with the increased load.

The following figure have been plotted with the throughput of reads and writes individually, and have more reads than writes in the system for the reasons described in Section 3.2.

On the longest running trace we have, running a total of 4 hours, we had a total of 300 clients and 50 worker threads and database connections in the middleware. The results of this test is shown on Figure 8. Here we see that the system behaves in a very stable manner throughout the test.

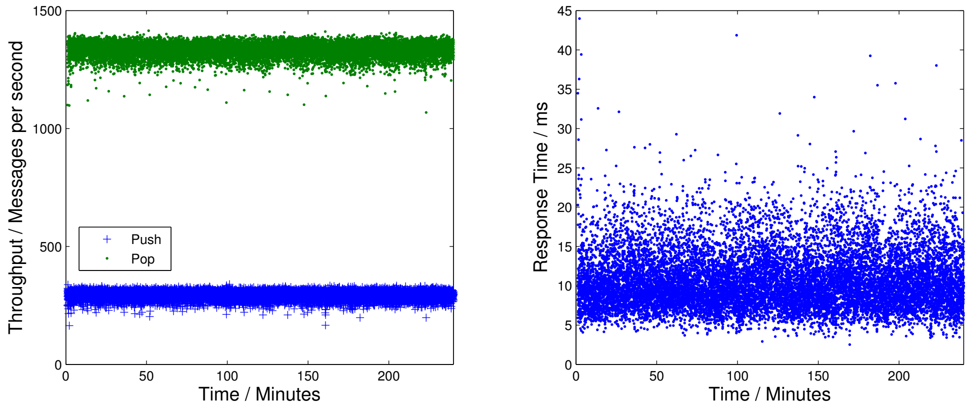


Figure 8: The throughput of the system while under load from the *Standard Test* with 200 one-way clients and 100 two-way clients. Using 1 middleware with 50 worker threads and 50 database connections. The dotted red line shows the average throughput and the solid lines the standard deviation.

Table 1 and Table 2 show the results from our long trace in numbers. We see that the confidence intervals are low, which means that the system behaves very stable. This is also evident when we look at Figure 8.

Request Type	**Mean Throughpout** (Requests per second)
Send/Push Message	$293.91 \pm [-28.91, 21.09]$
Pop Message	$1331.73 \pm [-53.73, 42.27]$
Total	$1625.63 \pm [-77.48, 59.37]$

Table 1: Breakdown of the think times in the system while under load from *Standard Test* with 200 one-way clients and 100 two-way clients. Using 1 middleware with 50 worker threads and 50 database connections.

Component	**Mean Think Time** (ms)
Socket I/O & Queuing	$1.11 \pm [-0.83, 1.80]$
CRW	$0.12 \pm [-0.10, 0.11]$
IPersistence	$9.43 \pm [-3.89, 5.52]$
Total	$10.66 \pm [-4.54, 7.06]$

Table 2: Breakdown of the think times in the system while under load from *Standard Test* with 200 one-way clients and 100 two-way clients. Using 1 middleware with 50 worker threads and 50 database connections.

The logs for this test can be found in the *4h-trace* folder.

4.3 Micro benchmarks

The tests in this section are made to test the performance of single types of requests. The following tests have been performed on a single middleware with 50 worker threads and 50 database connections. The amount of worker threads and database connections should not matter to this test though, since we use only a single client which will only send the next request when it has received a reply to the previous (closed system). This claim is tested in Section 4.14.

4.3.1 Send message

In this test a single client continuously sends *Send message* requests to the middleware. This means that the size of the dataset increases from 20,000 to 520,000 in the span of the test.

On Figure 9 we see that the amount of time spent on each request is dominated by the time it takes to get a response from the database, shown by the scattered dots in the top of the graph. In Table 3 the data for this graph is given, showing that the time spent in IPersistence is at least an order of magnitude higher than the rest of the system. We also see that the standard deviation of time spent in IPersistence is much higher than it is for the two other measurements.
Figure 10 shows the response time throughout the 500,000 *Send message* requests. On this graph we see that the response time is rather stable, except for the few scattered points in the top of the graph.

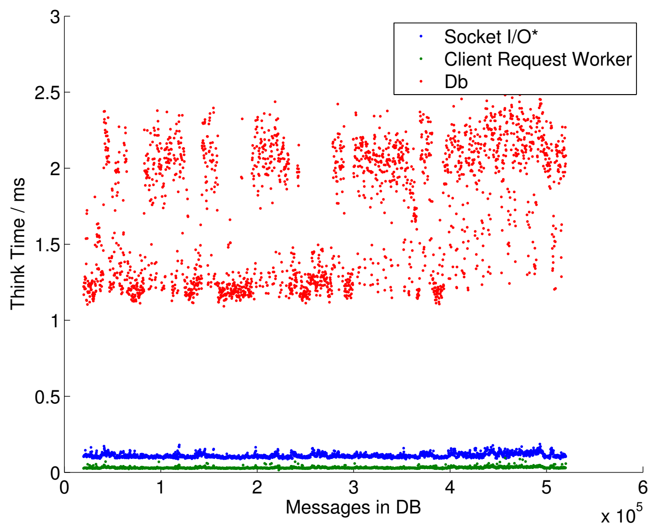


Figure 9: The think time for send message request. Each data point represents an individual request. A total of 500,000 requests sent sequentially each after the other.

Component	**Average Think Time** (ms)	**Standard Deviation (ms)**
Socket I/O & Queuing	$0.111 \pm [-0.0163, 0.0299]$	0.0145
CRW	$0.0310 \pm [-0.0052, 0.0098]$	0.0057
IPersistence	$1.714 \pm [-0.5508, 0.5903]$	0.427

Table 3: Average think time, confidence interval, and standard deviation for 500,000 *Send message* requests

Type of Request	**Response Time** (ms)
Pop	3.997 ± 0.021
Peek	2.022 ± 0.038
Push	1.856 ± 0.61294

Table 4: The response times for the requests: *Pop message*, *Send message*, and *Peek message*. Using 50 worker threads, 50 db connections, 1 client and 1 middleware.

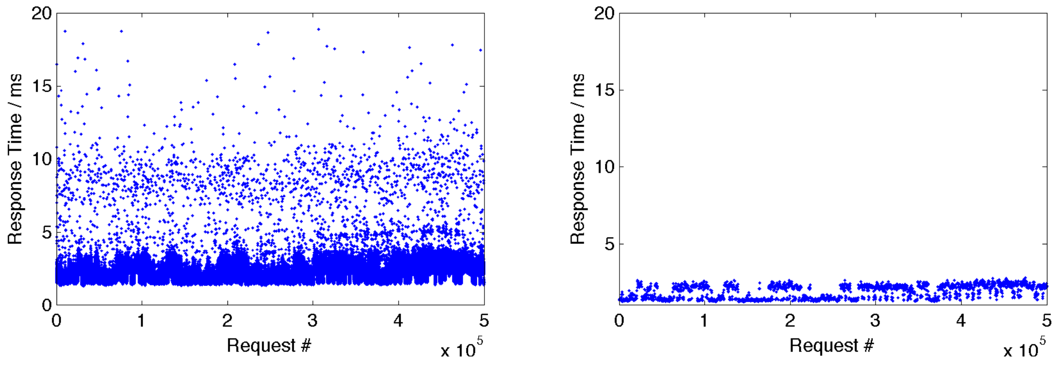


Figure 10: The response time for send message request. Each data point represents an individual request. A total of 500,000 requests sent sequentially each after the other.

The logs for this test can be found in *micro-benchmarks/sequential_requests*.

4.4 Response Times for Requests

The logs for this test can be found in *micro-benchmarks/sequential_requests*.

4.5 Breakdown of Think Times

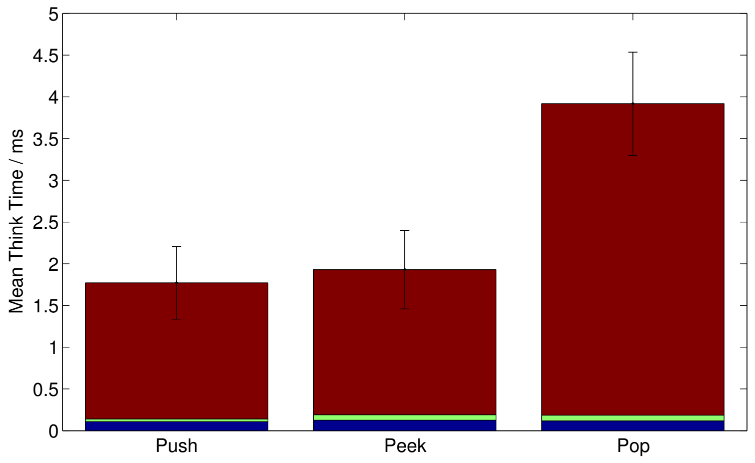


Figure 11: Breakdown of the thinktime for Send Message, Peek, and Pop requests. Using 1 client sending these requests sequentially to 1 middleware with only this client.

The logs for this test can be found in *micro-benchmarks/sequential_requests*.

4.5.1 Pop message

In this test a single client continuously sends a total of 500,000 *Pop message* requests. This test was run after the *Send message* test described in the last subsection, which means that there was a total of 520,000 messages in the database when this test was started. In this test the amount of messages in the database goes from 520,000 to 20,000.

On Figure 12 we see that the think time of the database (IPersistence) dominates the think time of the middleware. We also see that it has as specific pattern over time. We see that the think-time decreases and becomes more stable when the number of messages in the database decreases. On Table 5 we see the numbers with 90% confidence intervals.

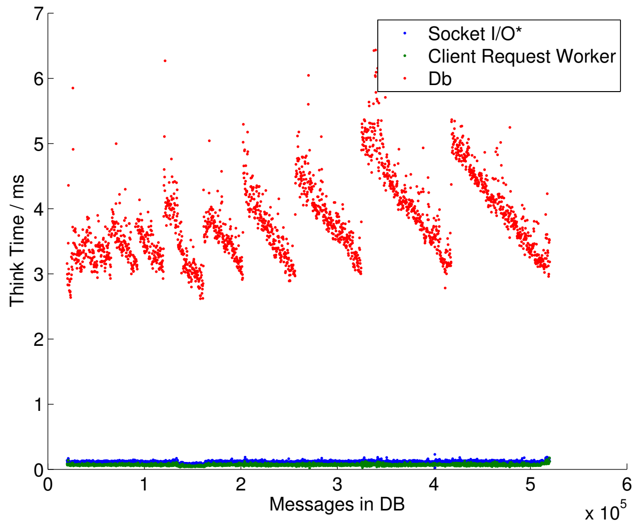


Figure 12: The think-time of the middleware during 500,000 *Pop message* requests. Each data point represents an individual request.

Component	**Mean Think Time** (ms)
Socket I/O & Queuing	0.118 ± 0.021
CRW	$0.069 \pm [-0.0192, 0.0227]$
IPersistence	3.810 ± 1.164

Table 5: Think times of the three different components of our system which we measure.

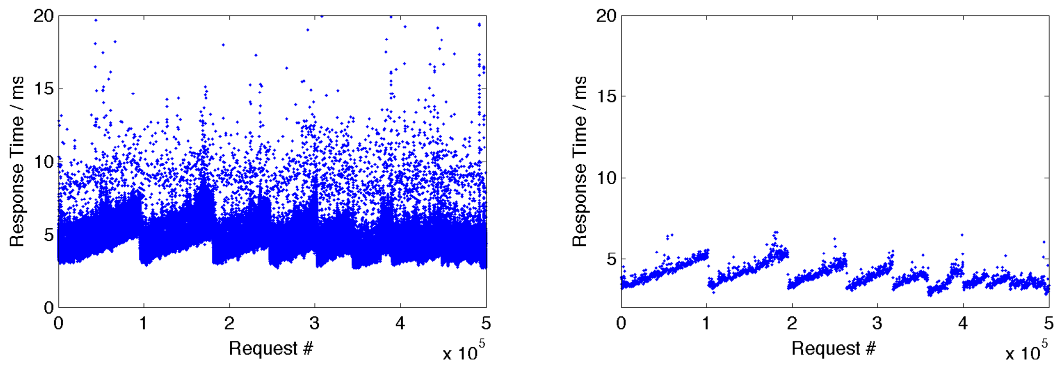


Figure 13: The response time for *Pop message* request. Each data point represents an individual request. A total of 500,000 requests sent sequentially each after the other.

The logs for this test can be found in *micro-benchmarks/sequential_requests*.

4.5.2 Peek message

In this test a single client continuously sends *Peek message*-requests to the middleware. The size of the dataset stays the same since *Peek message* doesn't actually modify the dataset.

On Figure 14 we see that the response time of the database dominates the think-time of the middleware. We see that the response time of the database varies consistently between around 1.25 and 2.5.

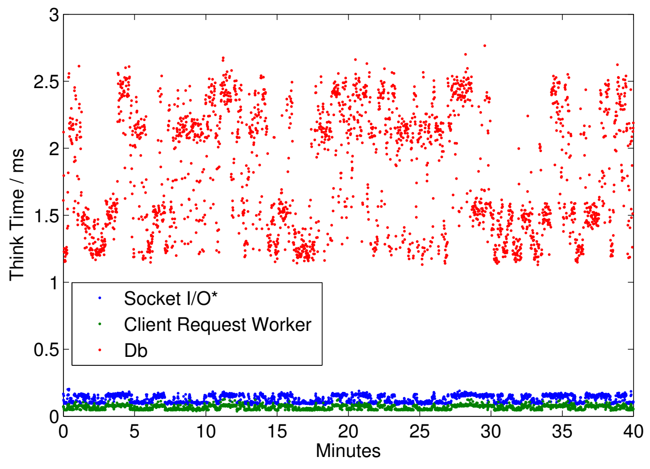


Figure 14: The think time in the server for *Peek message* request. Each data point represents an individual request. A total of 500,000 requests sent sequentially each after the other.

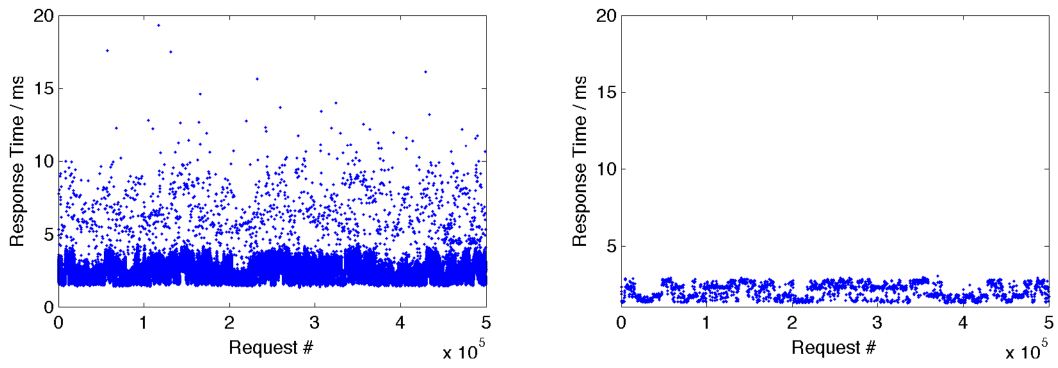


Figure 15: The response time for *Peek message* request. Each data point represents an individual request. A total of 500,000 requests sent sequentially each after the other.

Component	**Mean Think Time** (ms)
Socket I/O & Queuing	$0.1292 \pm [-0.0342, 0.0381]$
CRW	$0.0658 \pm [-0.0194, 0.0220]$
IPersistence	$1.8264 \pm [-0.6189, 0.6532]$

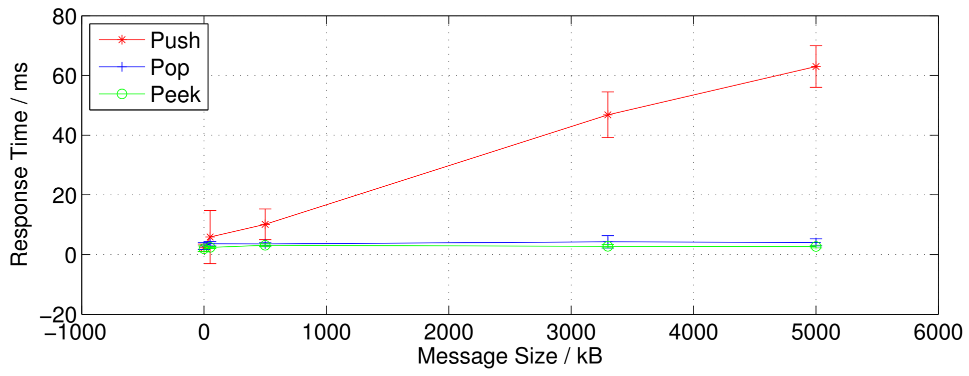
Table 6: Average think time, confidence interval, and standard deviation for 500,000 *Send message* requests

The raw data from these tests can be found in the log-folder, in the subfolders

- folders

The logs for this test can be found in *micro-benchmarks/sequential_requests*.

4.6 Difference in message size



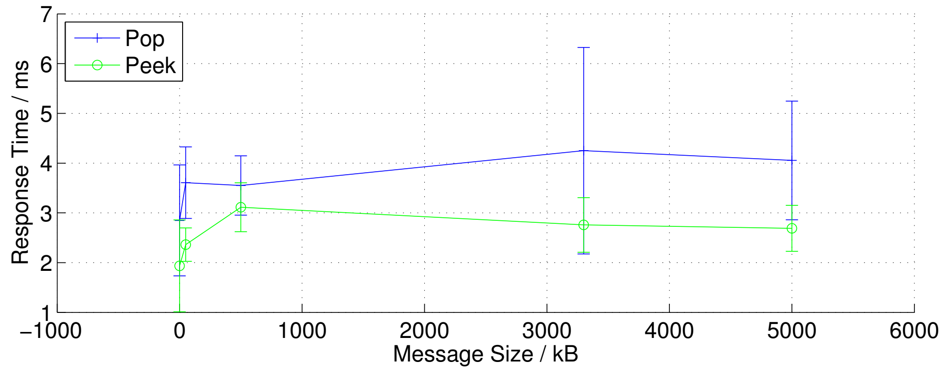


Figure 16: The response time for *send message*, *Peek message*, and *Pop message* as a function of the message size. The peeking and popping curves are also shown again in the lower graph without response time of *Send message* for clarity.

The logs for this test can be found in *micro-benchmarks/message-content-size*.

4.7 Difference in size of dataset

In this test we want to see what the difference of the size of the dataset matters to our system's performance. In this test we're using *Send and Pop Same Client*, varying the number of messages initially stored in the database. From this test we expect that an increase in the size of the data set will increase the think-time of the database, resulting in a lower throughput.

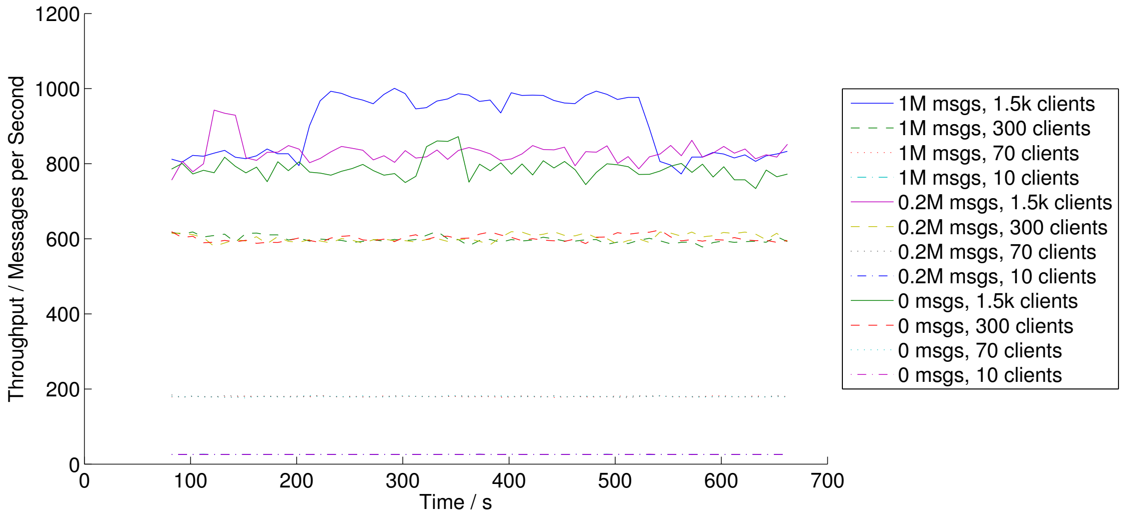


Figure 17: The throughput when running *Standard Test* varying number of clients and number of messages stored in database prior to test start. Run on one middleware having 50 worker threads and 50 database connections.

The raw data from these tests can be found in the log-folder, in the subfolders

- *sendandpopsameclient_diff_clients_msgsize-10b*
- *sendandpopsameclient_diff_clients_msgsize-1mb*
- *sendandpopsameclient_diff_clients_msgsize-10mb*

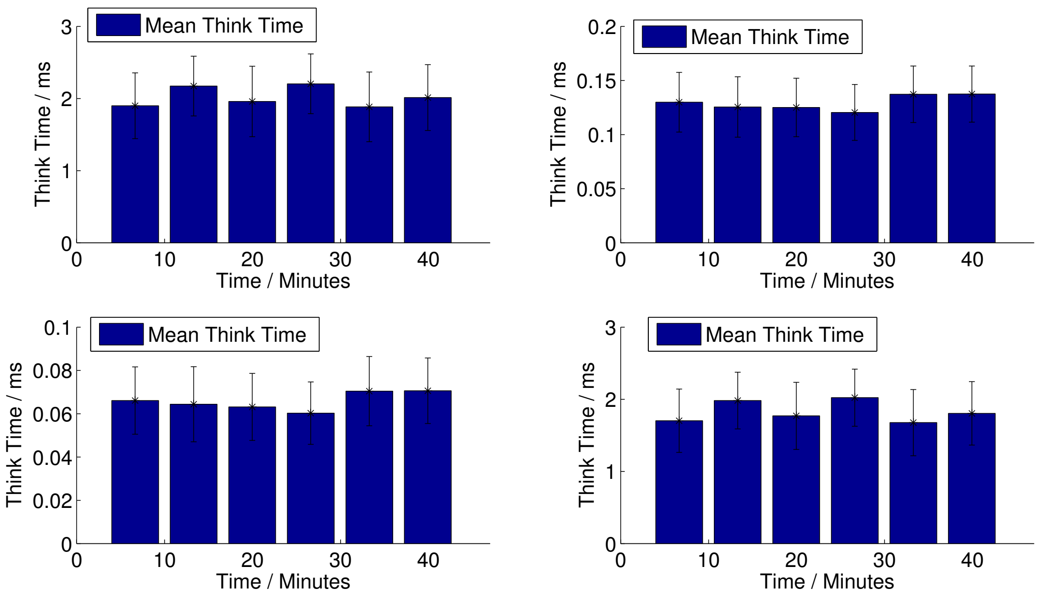


Figure 18: The response time over time when performing sequential peek-requests using one (1) client and 520k messages in the database. The figure shows the total throughput (top left), think time in Socket I/O and job creation (top right), Client Request Worker (bottom left) and Persistence (bottom right)

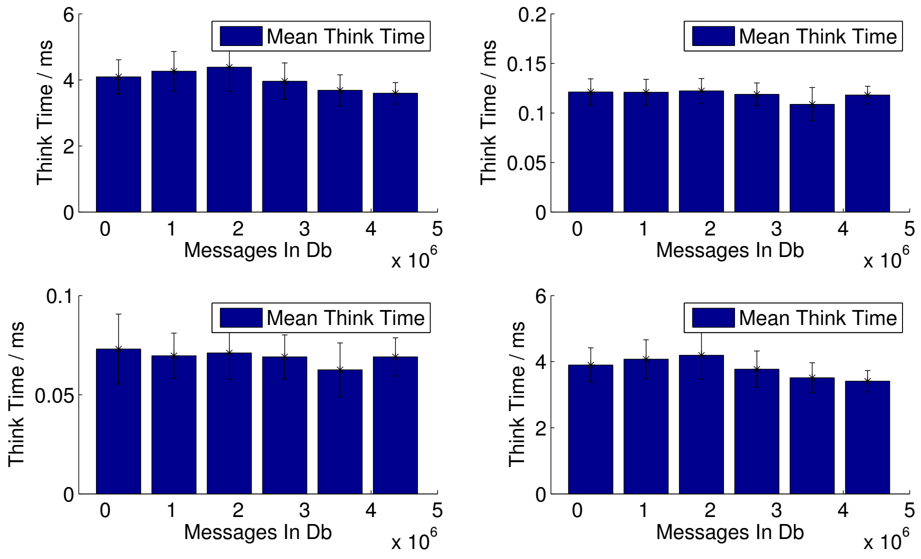


Figure 19: The response time as a function of number of messages in the database when performing sequential pop-requests using one (1) client and initially 520k messages in the database. The figure shows the total throughput (top left), think time in Socket I/O and job creation (top right), Client Request Worker (bottom left) and Persistence (bottom right)

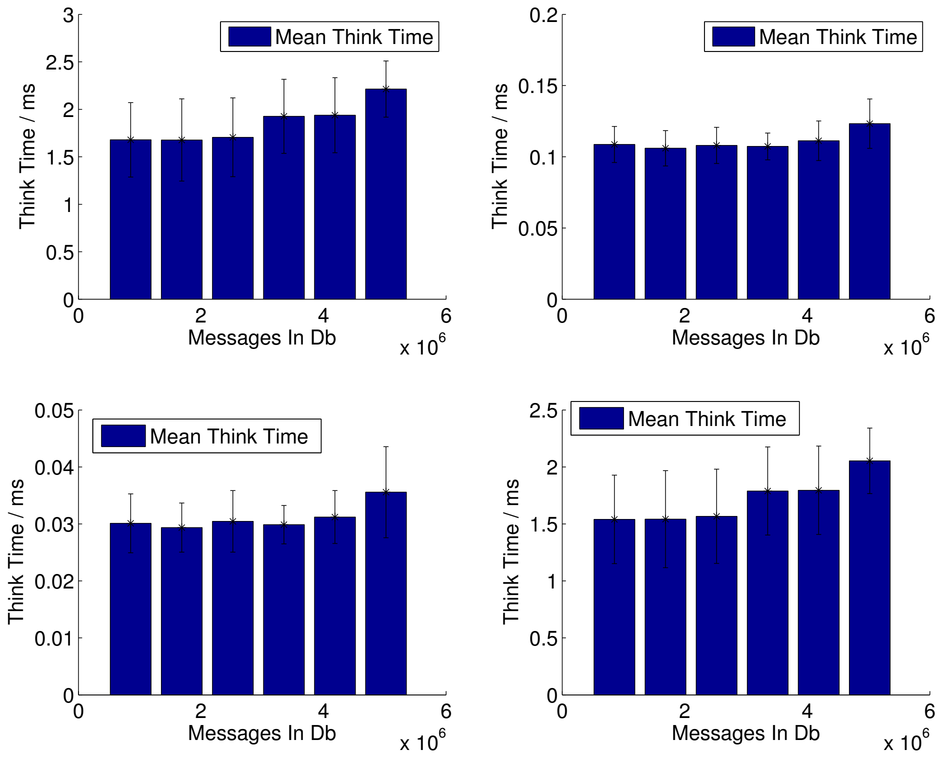


Figure 20: The response time as a function of number of messages in the database when performing sequential *Send message*-requests using one (1) client and initially 20k messages in the database. The figure shows the total throughput (top left), think time in Socket I/O and job creation (top right), Client Request Worker (bottom left) and Persistence (bottom right)

The logs for this test can be found in *micro-benchmarks/sequential_requests*.

4.8 Difference in number of clients

The data needed for this test was gathered during the test described in Section 4.7. On Figure 17, looking at tests with the same initial size of the dataset, we see that an increase in clients gives an increase in throughput in all of the tests.

The logs for this test can be found in *dataset-clients*.

4.9 Difference in frequency of requests

In this test we want to measure the impact on the server when we differ the frequency of requests. We performed 6 *Send and Pop Same Client*-tests, differing only the wait time between requests. The tests were performed using one middleware with 30 database connections and 30 worker threads, and 100 client threads on the client machine.

In this test we expect to see that the throughput increases with the number of requests, possibly reaching a point where it starts becoming unstable.

On the left of Figure 21 we see that the throughput increases as the wait time decreases, which is another way of saying that the throughput increases when the frequency of requests increases. We also see that the standard deviation increases slightly as the throughput decreases

In the top right corner of the graph we see that the average time requests spend queuing in the middleware (waiting to get handed to a worker thread) is higher when the wait time between requests is zero.

In the lower right corner we see that the combined time spent in Client Request Worker and IPersistence is going up when the frequency of requests increases.

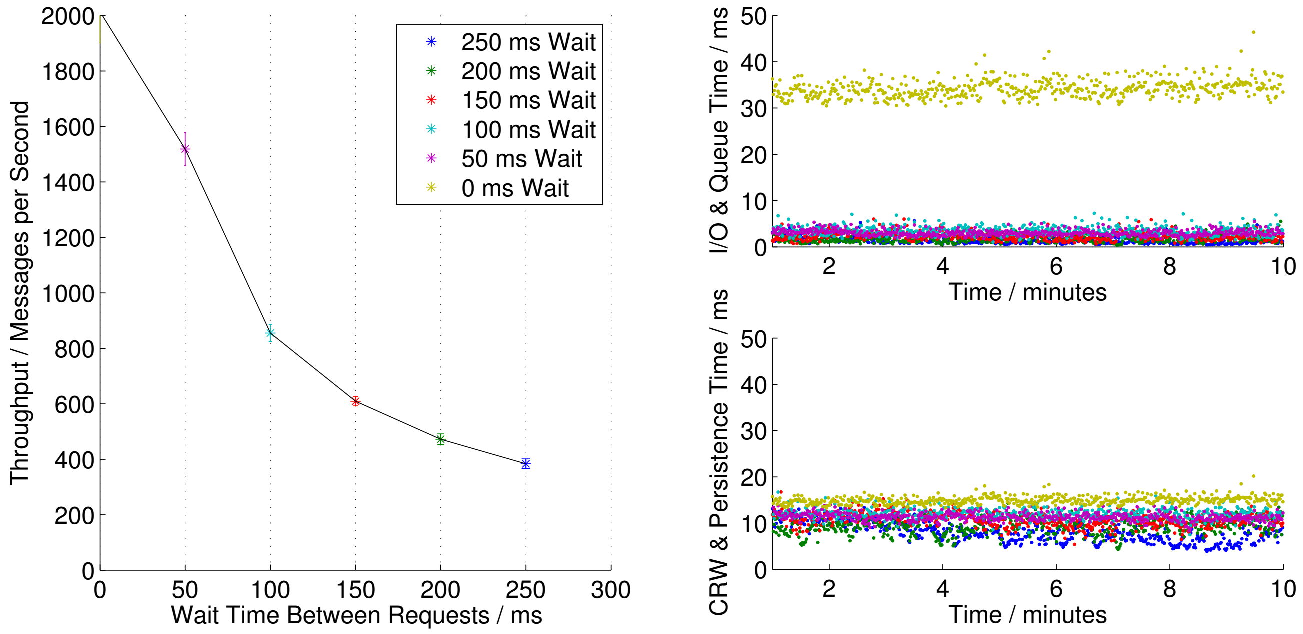


Figure 21: *Send and Pop Same Client* test differing the amount of wait time between requests. On the left side we have plotted the throughput when increasing the wait time. In the top right corner we have plotted the average time of Socket IO & Queue. In the lower right corner we have plotted the sum of *IPersistence* and *Client request worker*

On figure 22 we see that the average response time goes up when we increase the frequency of requests. We also see that the standard deviation of response time decreases when the frequency of requests increases.

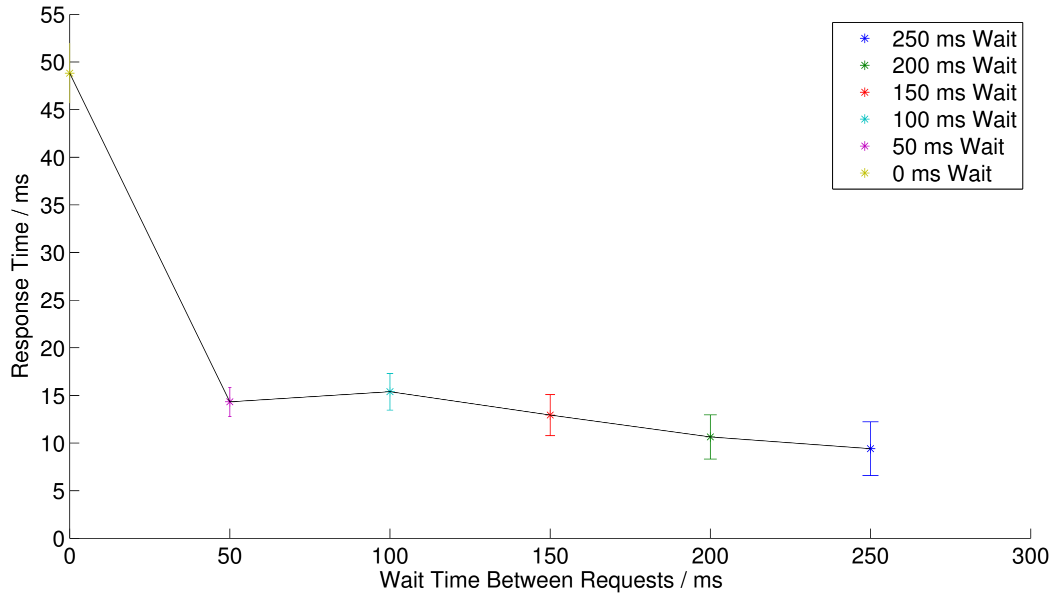


Figure 22: Same test as Figure 21, this time plotting total response time when increasing wait time

The logs for this test can be found in *request-frequency*.

4.10 Difference in data store

In this test we measure the difference between using Postgres and not storing anything at all. We expect to see an an increase in throughput since we don't have to communicate with the database at all and thus don't have to wait for the network or database think time.

Configuration	Mean Push Throughput	Mean Pop Throughput	Mean Response Time
50 Threads	8414.7 ± 66.6	8414.7 ± 66.6	1.30 ± 0.01
5 Threads	8538.7 ± 72.6	8538.7 ± 72.6	1.24 ± 0.01

Table 7: The throughput and response time data and confidence interval when running *SendAndPopSameClient* without any persistence and varying number of worker threads. Using 50 clients.

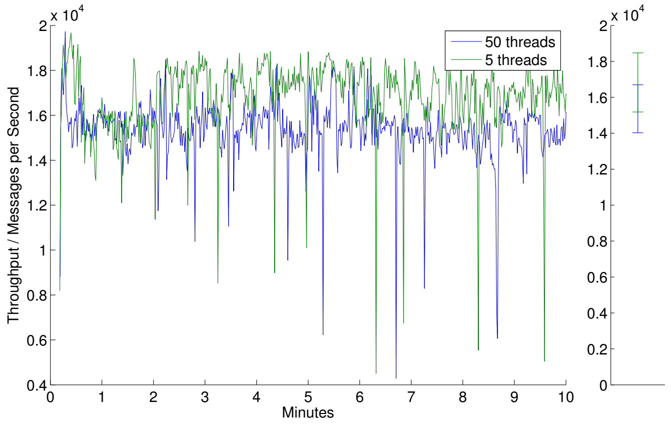


Figure 23: The throughput when running *SendAndPopSameClient* without any persistence and varying number of worker threads. Using 50 clients.

The logs for this test can be found in *no-persistence*.

4.11 Difference in number of middlewares

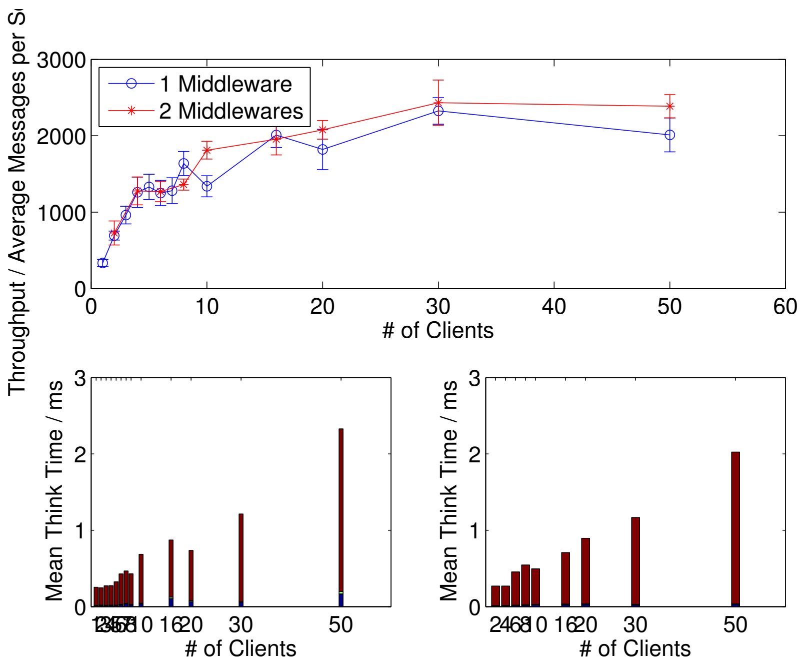


Figure 24: The throughput when running *SendAndPopSameClient* varying number of clients and number of middlewares.

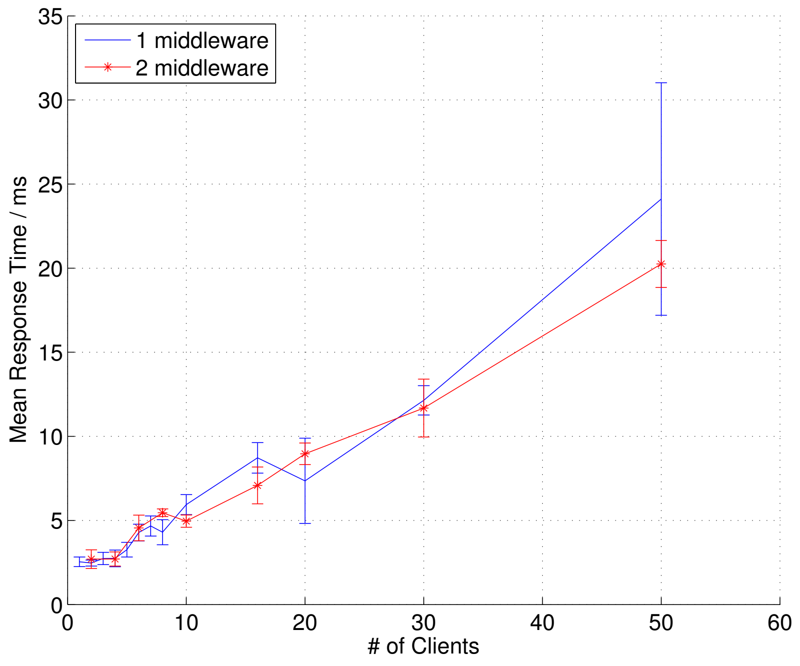


Figure 25: The response time when running *SendAndPopSameClient* varying number of clients and number of middlewares. The figure show response time increasing asymptotically after certain threshold.

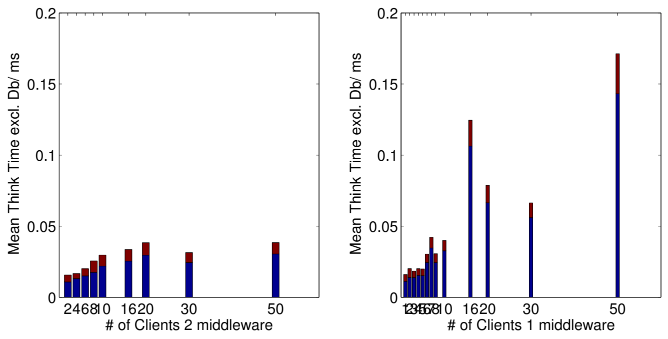


Figure 26: The mean think time excluding the persisting component for a request when varying number of clients and middleware.

Clients	**Mean Throughput** (messages per second)
1	$343.1 \pm [-48.7, 52.4]$
2	$693.0 \pm [-88.4, 68.1]$
3	$961.0 \pm [-176.4, 172.4]$
4	$1260.3 \pm [-314.3, 223.1]$
5	$1330.4 \pm [-214.4, 249.6]$
6	$1250.2 \pm [-162.2, 249.8]$
7	$1280.8 \pm [-194.8, 306.0]$
8	$1636.2 \pm [-239.6, 195.8]$
10	$1338.1 \pm [-267.5, 179.4]$
16	$2009.1 \pm [-260.6, 166.9]$
20	$1820.6 \pm [-542.8, 261.4]$
30	$2324.8 \pm [-270.2, 247.2]$
40	$2010.3 \pm [-576.7, 226.4]$

Table 8: Mean Throughput when using 1 middleware with 50 worker threads and 50 database connections with the clients using the *Send And Pop Same Client* configuration.

Clients	**Mean Throughput** (messages per second)
2	$727.6 \pm [-203.6, 196.3]$
4	$12780 \pm [-310.0, 230.0]$
6	$12682 \pm [-228.2, 155.8]$
8	$13605 \pm [-116.5, 115.5]$
10	$18108 \pm [-172.8, 165.2]$
16	$19544 \pm [-490.8, 231.8]$
20	$20772 \pm [-212.5, 180.8]$
30	$24324 \pm [-542.4, 409.6]$
50	$23862 \pm [-237.0, 183.8]$

Table 9: Mean Throughput when using 2 middleware with 50 worker threads and 50 database connections with the clients using the *Send And Pop Same Client* configuration.

The logs for this test can be found in *middleware-scaleout*

4.12 1 vs 10 middlewares

In this test with *StandardTest* 1000 one-way clients and 500 two-way clients were used and the difference in performance when going form 1 middleware to 10 middlewares were measured.

The logs for this test can be found in *1-vs-10-middleware*.

Configuration	Mean Push Throughput	Mean Pop Throughput	Mean Response Time
10 Middlewares	440.75 ± 0.099	2035.53 ± 1.37	11.55 ± 0.11
1 Middleware	324.84 ± 3.20	2081.93 ± 16.05	44.55 ± 0.49

Table 10: The difference in performance when using 1 and 10 middleware. It is clear that when increasing the middleware the response time decreases, while the total throughput only increases marginally.

4.13 Worker-threads vs Clients

In this test the Standard Test was run for 10 minutes with the purpose of finding the limit on where contention in the middleware would appear. The configuration that was used in this test was 1 middleware, varying the number of clients and the number of worker threads and database connections. This can be seen as a 2^k test even though number of clients, number of database connections and number of worker threads are both varied due to the result which was that the ratio should be 1:1 between the number of worker threads and number of database connections in the system for best performance. The results are presented in Table 11.

Configuration	Mean Push Throughput	Mean Pop Throughput	Mean Response Time
100/100 + 500/250	$335.95 \pm [-62.20, 70.30]$	$2173.2 \pm [-242.2, 265.0]$	$113.16 \pm [-32.18, 36.47]$
100/100 + 300/125	$147.39 \pm [-26.39, 28.95]$	$1613.72 \pm [-115.92, 69.28]$	$15.79 \pm [-6.6, 12.21]$
100/100 + 100/50	$163.22 \pm [-9.22, 8.78]$	$690.85 \pm [-16.85, 12.85]$	$3.90 \pm [-0.87, 1.71]$
30/30 + 500/250	$325.95 \pm [-45.95, 51.05]$	$1945.82 \pm [-165.72, 161.18]$	$147.39 \pm [-26.39, 28.95]$
30/30 + 300/125	$376.53 \pm [-43.63, 29.47]$	$376.53 \pm [-43.63, 29.47]$	$13.14 \pm [-7.14, 13.73]$
30/30 + 100/50	$160.36 \pm [-12.36, 2.83]$	$638.58 \pm [-26.58, 19.12]$	$5.40 \pm [-2.63, 2.83]$

Table 11: Testing for number of clients and number of database connections and worker threads. The configuration that was used in this test was 1 middleware and the test was Standard Test. The notation in the configuration-columns is so that 1/2 + 3/4 means 1 database connection, 2 worker threads, 3 one-way clients and 4 two-way clients.

The logs for this test can be found in *threads-vs-clients*.

4.14 Difference in database connections and worker threads

In this test we perform 4 *Send and Pop Same Client*-tests, differing the amount of worker threads and database connections. As explained in Section 1.2 we expect to see that the throughput of the middleware is bounded by the minimum of the number of worker threads and the number database connections.

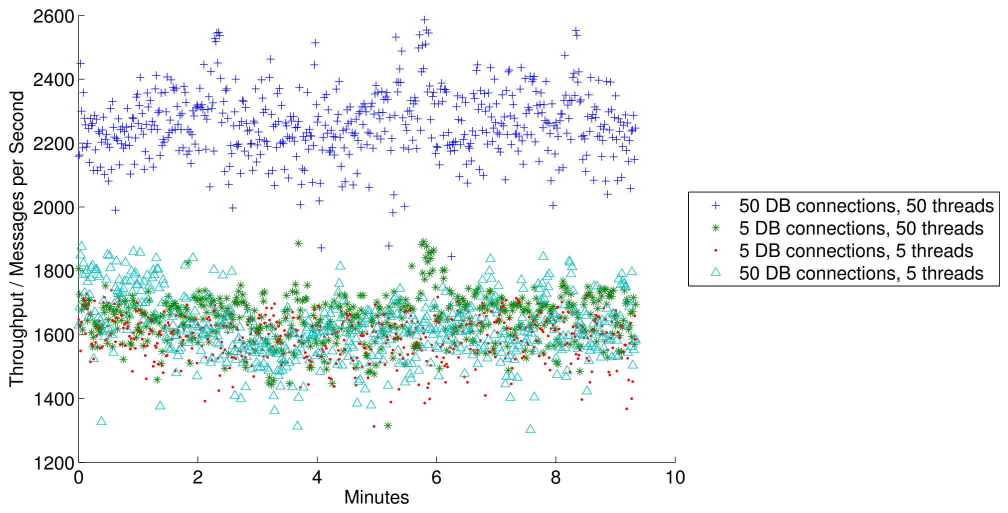


Figure 27: The combined throughput of *Send Message*- and *Pop message* requests of the system while performing *Send and Pop Same Client*, differing the number of database connections and worker threads on the middleware, performed on a single middleware.

On Figure 27 we see that the result seems to be matching our expectations; the throughput is higher when both the number of database connections and worker threads is 50, but stays lower when either the number of database connections or worker threads is 5. Also we notice that the standard deviation of our results seem to increase as the maximum of the number of database connections and worker threads increases. The raw data from these tests can be found in the log-folder, in the subfolders

- *sendandpopsameclient_diff_worker-threads_db-connections*

4.15 Network Latency

By performing 500 000 requests to the middleware and measuring the response time on the middleware and also on the client, we can calculate the network latency caused by the network by subtracting the mean response time of the client with the mean reasponse time on the middleware. The distribution of the response times can be seen in Figure 28 and the measures difference can be seen in Table 12.

MRT[2] on Client (ms)	MRT on Middleware (ms)	Mean Network Latency (ms)
$4.358 \pm [-0.724, 0.915]$	$3.957 \pm [-0.652, 0.696]$	0.402

Table 12: Network Latency approximated by subtracting the mean response time on the server with the mean response time on the client.

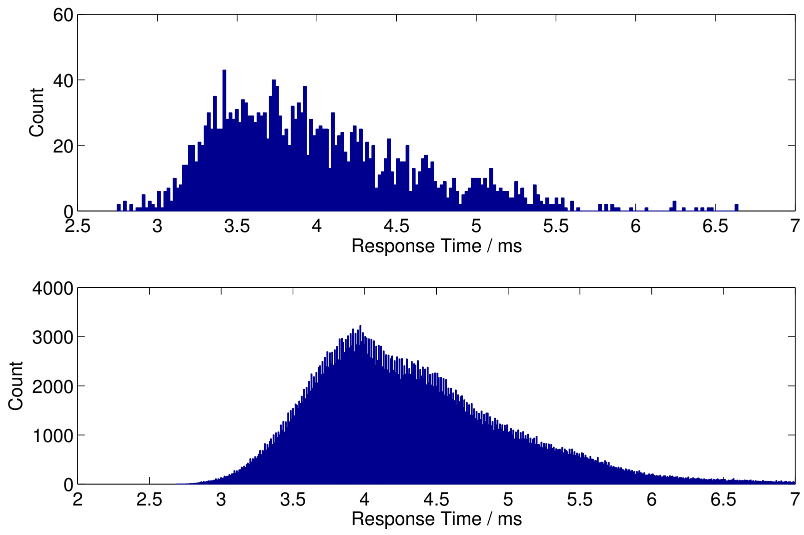


Figure 28: The distribution of the response times on the middleware (top) and client (bottom). Sampled using 500 000 peek (as to not to vary number of messages in the system) requests to the middleware using one (1) client and more than 500,000 messages in the database.

The logs for this test can be found in *micro-benchmarks/sequential_requests*.

4.16 Neighbour Noise

As discussed in Section 4, all our deployments live on a cloud hosting platform and thus are subject to neighbour noise. To demonstrate that the difference in computing power avaiable to our system varies over time an identical test were run on the evening (around 20:00) on day, and once again in the morning the next day (around 10:00). The difference in throughput and response time is presented in Figure 29 and 30.

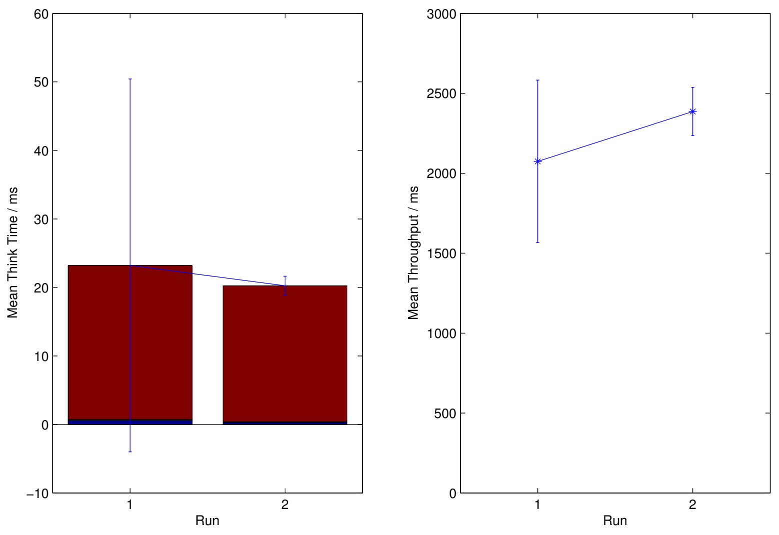


Figure 29: The mean think time and mean throughput for the same test run at two different times under the day using exactly the same configuration. Run 1 was run at the evening around 20:00 and Run 2 was run in the morning around 10:00.

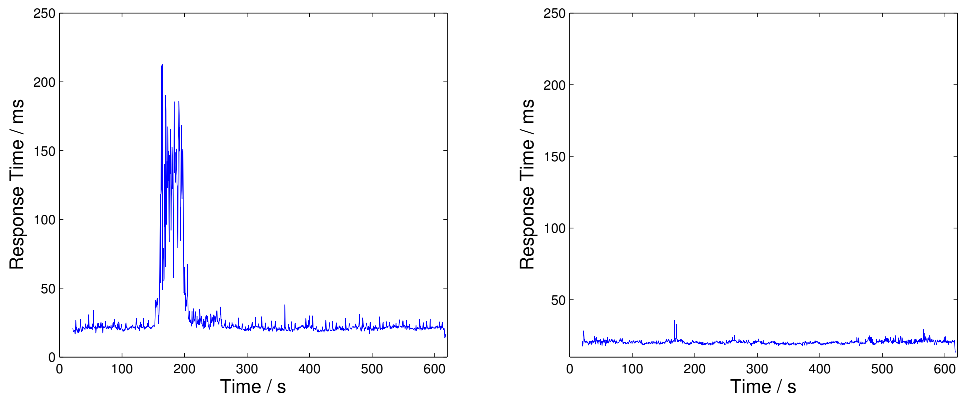


Figure 30: The response time over time for the same test run at two different times under the day using exactly the same configuration. The left figure shows the test that was run at the evening around 20:00 and the right was run in the morning around 10:00.

The logs for this test cane be found in *same-test-differente-times*

5 Analysis

In this section we interpret and compare the results shown in Section 4 above, in order to figure out what the different performance metrics of our system are.

5.1 Time spent in each part of the system

In order to determine how much time is spent in each part of our system, we look at the results we measured during code profiling (Figure 6 and 7) and the data we have on the bottom of Figure 24.

The results of our code profiling show that 83-87% of our codes' execution time is spent waiting for replies from the database. It is important to note that this time *includes the network delay* between the middleware and the database, in both directions.
The results from the code profiling are backed up with our own logging data, where we have also measured how much time is spent in different parts of our system. On Figure 24 we also have evidence that the server spends most of its time waiting for a response from the database, waiting in the IPersistence class of our implementation. Assuming that the VisualVM tool is correct this confirms that our measuring techniques employed are valid.

This data all points to the database or the network being the bottleneck in these particular tests. We contribute this to choosing bad indexes for our data, having a poorly configured database, or having a slow network. The slow network is though very unlikely, since from what we can deduce from Section 4.15 the network latency is very small (around 0.5ms) and we observe think times in the persistence component bounded by around 1.5 from below, see Figure 11. We also get a much higher throughput when not storing any messages at all (i.e. the *I Fooled You* persistence implementation), as seen in Figure 26, where we have a throughput of around 17k messages per second and a response time of around 1.3 ms.

5.2 Size of messages

We can see from Figure 16 that the size of a message only affects the time to perform *Send message. Pop message* and *Peek message* seem unaffected by the size of the message. This is attributed to the fact that the database has to push data to disk (whether it's the write-ahead-log or the actual data) in order to promise durability, whereas it's possible not to hit disk when reading data from memory. We draw the conclusion that our system handles big payloads well.

5.3 Number of middlewares

In Figure 24 it is shown that when we increase the number of clients across one and two middleware, they hit the throughput threshold at the number of clients. We can also see in Figure 26 that while the throughput is the same for the system and the response time is increasing with the same amount (Figure 25) the response time excluding the persistence component is increasing more for each additional client when running 1 middleware relative to distributing the clients to 2 middlewares. This shows that while the system on the middleware is less stressed, the think time in the database, which is the bottleneck in the whole system, is increasing for both configurations (1 vs 2 middleware). One can also see a tendency to the system with two middlewares having a slightly shorter response time overall – which is expected (Figure 25). We can also see that the throughput will flatten out after a certain threshold, and that this threshold is the same as for when the response time start to behave asymptotically.

5.4 Number of clients

When increasing the number of clients (which perform the *Standard Test*, i.e. is interacting with the system) to a middleware we see that the throughput increases, see Figure 11. This is expected behavior, as is the observation that the system has an upper limit on the throughput and thus the number of clients connected the middleware. When this limit is hit, adding more clients to the middleware will not lead to an increased throughput, just increase response time.

5.5 Stability of the system

In our long trace test we saw that our system behaved in a reasonably stable manner over the four hour test.

When comparing the throughput and the response times in Table 11 for 750 clients, 100 worker threads and 750 clients, 30 worker threads, we see that the throughput stays about the same and that the response time goes up. From this we can see that the increase in clients means that queues are building up in the middleware. Even when we have queues in the middleware, we see that the system behaves in a stable manner.

When we compare the throughput from when 150 client and 100 worker threads were used, and 750 clients and 100 worker threads we see that the throughput is increasing when the number of clients increases. We see that the confidence interval (again, see numbers in Table 11) increases heavily. This increase in the confidence interval shows that we are pushing the system a lot more with 750 clients than we do with 150 clients. The narrow confidence interval we have with 150 clients shows that we aren't pushing the system to the point of making it unstable, but that we are on our way to this point when we use 750 clients.

5.6 Frequency of requests

As noted in Section 4.9 we see on Figure 21 that the throughput of our system goes up when the frequency of requests increases. This seems reasonable, and points to the fact that our system can handle more load than the test produces when the wait time is relatively high. What seemed to be counter-intuitive to us in the beginning was that the average response time (as shown on Figure 22) is much higher when the wait time is lower, even though the throughput is much higher at the same time. After plotting the top right part of Figure 21, we realized that this must be because there are many requests queued, waiting to be served. This gives a higher throughput since the server always has a new request to process and at the same time gives a higher average response time since there are requests waiting to be served in the queue.

As we also noted in Section 4.9 the standard deviation of response time becomes higher as the average response time becomes lower, and the frequency of messages decreases. What's believed to be happening at this point is that the requests are so infrequent that there is (almost) no queuing in the middleware, causing requests to be processed immediately. When the average response time becomes as low as 10 milliseconds, it is seems very reasonable to us that the standard deviation increases, since things such as network i/o will play a big part in the result.

Finally we note that our system seems to be performing best (in this particular setup) when the wait time between requests is around 50-60 milliseconds, an average of 1500-1600 requests per second (when the distribution of requests is half *Pop message* and half *Send message*), since this is the point where the throughput is relatively high while the average response time hasn't begun increasing drastically yet.

5.7 Bounded throughput caused by middleware configuration

On Figure 27 we saw that, as per our expectations, the throughput of the middleware (assuming a high enough frequency of requests) is bounded by the minimum of the number of database connections and the number of worker threads. This is a very intuitive result since each worker thread will use exactly one database connection per (well-formed[3]) request. This means that if we want to keep the utilization of both worker threads and database connections high, we should keep the ratio between them 1, as other ratios would leave one of the resources idle.
This result is backed up by the results of our long traces where, in Table 11 and 11, we saw that increasing the amount of database connections and worker threads also resulted in a higher standard deviation in throughput.

[3]I.e. not an invalid request or a request that is not a handshake request.

In both of the mentioned tests we also see that the standard deviation of the results seems to be increasing when the amount of worker threads and database connections is increased. We attribute this to the fact that we give the database more work to do simultaneously, causing a higher response time for the database.

If we look at the measured think-time of the database during the long traces, we see that this confirms our suspicion; the think-time of the database increases when we increase the amount of worker threads and database threads.

5.8 Micro benchmarks

The response time patterns for the pop request showed in Figure 13 is a strange pattern that seems to be caused by the decreasing the number of messages in the database cause by the pop calls themselves. This is concluded since the pattern does now show up when only peeking and not thus now varying the number of messages in the database. It seems that some external factor might be causing this, for instance we have Postgres' automatic VACUUM activated. It could be that the trend we see decreases because the number of messages in the database decrease and that the VACUUM operation therefore takes less time.

In section 4.3.1 we see that the average response time of *Pop message* is around twice that of *Peek message*. This is explained by the fact that we, as described in 1.2, in our implementation of *Pop message* in the middleware, make a call to *Peek message* and if a message is returned we make a call to delete the this message from the database. This means that we send two requests to the database for all *Pop message* requests that finds a message requested by the client.

We also see that the response time for popping is decreasing when the number of messages in the database is decreasing (see Figure 19), and we see the inverted pattern when pushing messages, i.e. the response time for pushing is increasing when the number of messages in the database is increasing (see Figure 20). When peeking, the response time stays constant withing the standard deviation (see Figure 18).

6 Conclusion

In Section 4.14 we found that the performance of our middleware depends, among other things, on the ratio between the amount of worker threads and database connections. In this section we found that the ratio should be 1 if we want to utilize all of our database connections and worker threads simultaneously. This was a very intuitive result to us, and was exactly what we expected to see.

Not only does the ratio between worker threads and database connections matter, so does the *number* of worker threads and database connections. We can see this when having a saturated system with 30 worker threads and 30 db connections, and then increasing number of worker threads to 100 and db connections to 100 also. Here we can see (Table 11) that the throughput increase is not statistically significant (stays within standard deviation) but the response time is decreased.

In our analysis in Section 5.1 we found that found that during execution of our middleware, most of the time is spent waiting in the persistence component or, more specifically, waiting for responses from the database. This result is consistent with the test results from the other tests we made that put a significant load on the system. This implies that, in our current setup, the database is the bottleneck. We do not blame this on the database implementation, but rather on ourselves, for choosing bad indexes and/or configuring the database poorly w.r.t. the dataset we have and the type of queries our middleware performs.
If we had had more time to test our system we would try to change the indexes and rerun our tests to see how much performance we could gain from a more appropriate configuration of the database. This is also further confirmed by the scaling up test (see Section 4.11) where increasing the number of middlewares did not increase the throughput. This is expected when the database is the bottleneck since this is a shared resource between the middlewares, and when this is the limiting device for throughput then it does not matter how many middlewares are used.

The maximum throughput is around 2500 ± 300 requests per second for the *Standard Test* (see Table 11, and around 2000 ± 210 requests per second for *Send and Pop Same Client* (see Table 8 and Table 9).
We found that we got the best performance with the *Send and Pop Same client* test when there are around 1500 requests per second, since this is where the throughput is the highest while the response time is still as low as 15 milliseconds (See Figure 21 and 21).

A Message protocol and API

A.1 Exceptions

If a request cannot be fulfilled by the server, the server will respond with an exception. Exceptions have the following format:

FAIL <type> <id>

Where <type> is either QUEUE, CLIENT, MESSAGE, or UNKNOWN, and where <id> is the id of the queue, client or message that 'failed'. For instance, if a client tries to send a message to the queue with id 5 and if that queue doesn't exist, the server will respond with "FAIL QUEUE 5".

A.2 Handshake

The client should send HELLO when first connecting to the server. The server should respond with OK if it accepts the client, otherwise the server should respond with something else. The client must disconnect if it receives anything other than OK from the server.

A.3 Send Message

MSG,ReceiverId,SenderId,QueueId,Priority,Context,Content

Response
The server should respond with OK if the message was inserted into the queue successfully, otherwise it should respond with FAIL.

Remarks
To send a message to multiple queues, separate the QueueIds with a semicolon, e.g. 1;2;3 to send the message to queues 1, 2 and 3.

A.4 Message Response

MSG,SenderId,Context,MessageId,Content

This is here to save space in the definitions below. This is how the server should return a message upon request from the client.

A.5 Peek Queue

PEEKQ,ReceiverId,QueueId,OrderByTimestampInsteadPriority

Response
The server should respond with the message formatted as in section A.4 if there is a message in the queue. Otherwise the response should be MSG0.

Remarks
OrderByTimestampInsteadPriority is either 1 or 0.

A.6 Peek Queue with Specific Sender

PEEKS,ReceiverId,QueueId,SenderId,OrderByTimestampInsteadPriority

Response
The server should respond with the message formatted as in section A.4 if there is a message in the queue. Otherwise the response should be MSG0.

Remarks
OrderByTimestampInsteadPriority is either 1 or 0.

A.7 Pop Queue

POPQ,ReceiverId,QueueId,OrderByTimestampInsteadPriority

Response
The server should respond with the message formatted as in section A.4 if there is a message in the queue. Otherwise the response should be MSG0.

Remarks
OrderByTimestampInsteadPriority is either 1 or 0.

A.8 Pop Queue with Specific Sender

POPS,ReceiverId,QueueId,SenderId,OrderByTimestampInsteadPriority

Response
The server should respond with the message formatted as in section A.4 if there is a message in the queue. Otherwise the response should be MSG0.

Remarks
OrderByTimestampInsteadPriority is either 1 or 0.

A.9 Create Queue

CREATEQUEUE,NameOfQueue

Response
The server should respond with the id (long) of the queue, if a queue with the same name exists the server should respond with FAIL.

A.10 Remove Queue

REMOVEQUEUE,QueueId

Response
The server should respond with OK or FAIL.

B Log Files

In our tests we logged both at the client and at the client side. This section goes through how to parse the log files.

B.1 Middleware Logs

The middleware log on a set interval which can be configured in a configuration file which is read once at startup by the middleware, and each time this interval is hit, this is added to the log:

```
[milliseconds since middleware start], [count
  of push requests], [count of pop/peek
 requests], [total number of requests
 received], [total time in Socket IO], [
 total time in CRW], [total time in
 persistence]
```

Note: All logged data except the first column is reset for each logging event. So the count of push requests is the count of push requests since $t - \Delta t$ where Δt is the logging interval. Also note that Socket IO includes the amount of time the request spends in waiting in for a free worker thread.

B.2 Client Logs

The log files used in the tests described in this test have the follow formatting, and is logged every time a request finishes:

```
[milliseconds since client start], [duration
 of request]
```

C Sample Distributions

Here are some figures over the distributions of some of the samples collected during this project. Not all of these have a normal distribution and those that doesn't have the corresponding notation in the report is not assumed to be normally distributed, they are in the appendix because it is also interesting to see the distribution of the asymmetrical distributions. The confidence interval for the (assumed) normally distributed samples is done with the MATLAB function `normfit` which returns the confidence interval. For the asymmetrical / non-normally distributed samples the MATLAB function `quantile` is used to calculate the 5th and 95th quantile, then the mean of the samples is subtracted.

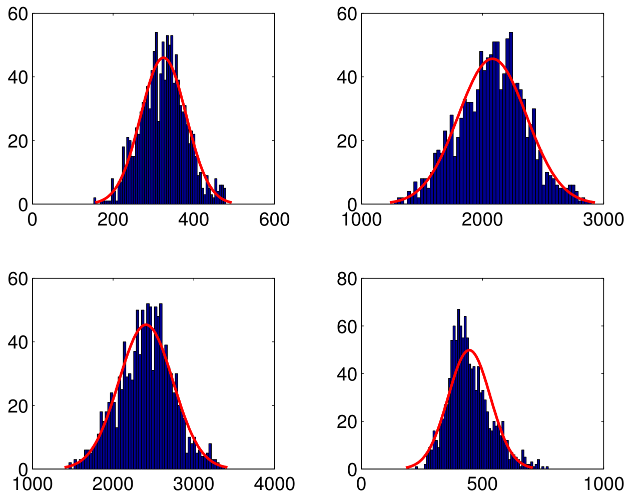


Figure 31: The distribution for push throughput (top left), pop throughput (top right), total throughput (bottom left) and response time for 50 SendAndPop-SameClient clients using 50 db-connections and 50 worker threads using one (1) middleware.

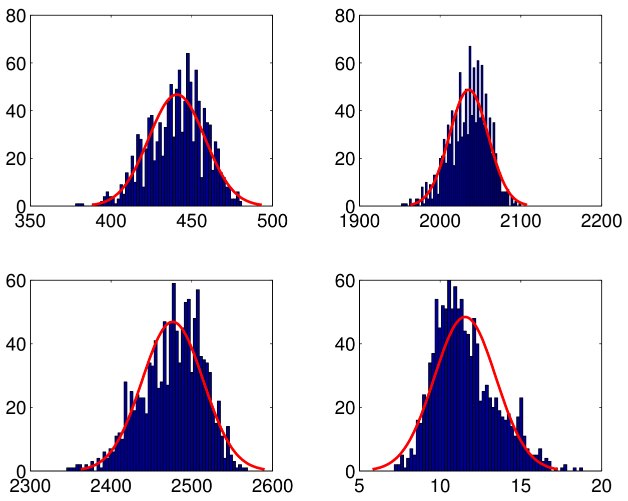


Figure 32: The distribution for push throughput (top left), pop throughput (top right), total throughput (bottom left) and response time for 50 SendAndPop-SameClient clients using 50 db-connections and 50 worker threads using two (2) middleware.

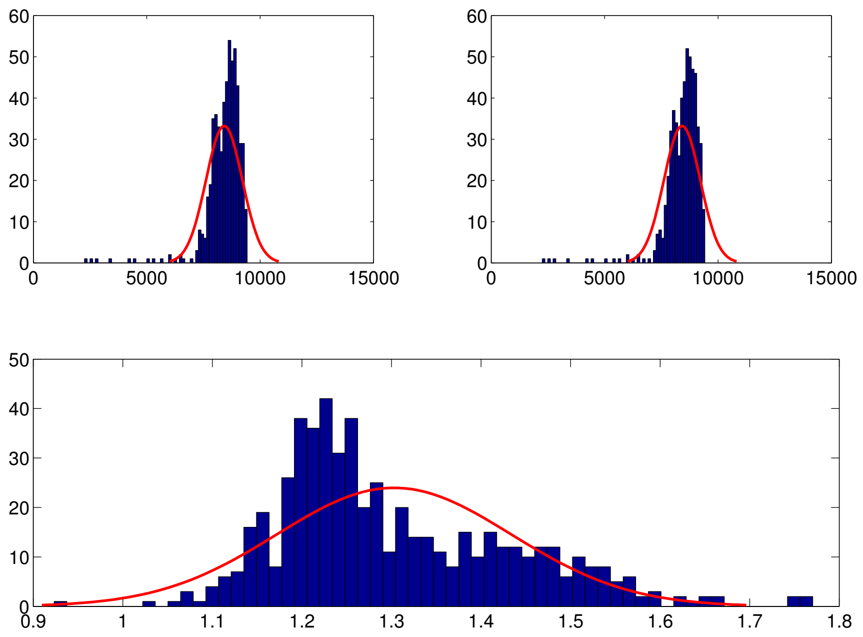


Figure 33: The distribution for push throughput (top left), pop throughput (top right), response time (botto) for 50 SendAndPopSameClient clients and 50 worker threads using one (1) middleware with *I fooled you*-peristence.

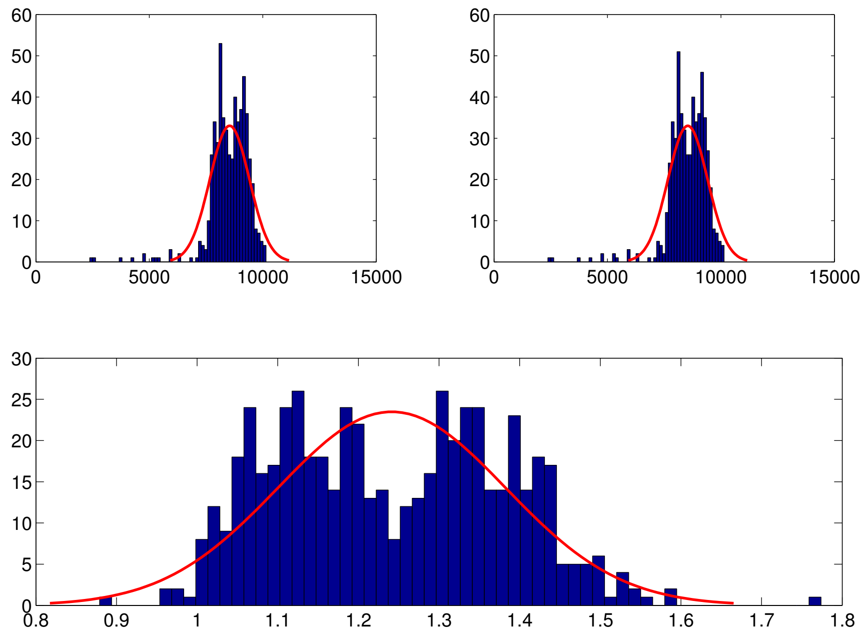


Figure 34: The distribution for push throughput (top left), pop throughput (top right), response time (botto) for 50 SendAndPopSameClient clients and 5 worker threads using one (1) middleware with *I fooled you*-peristence.

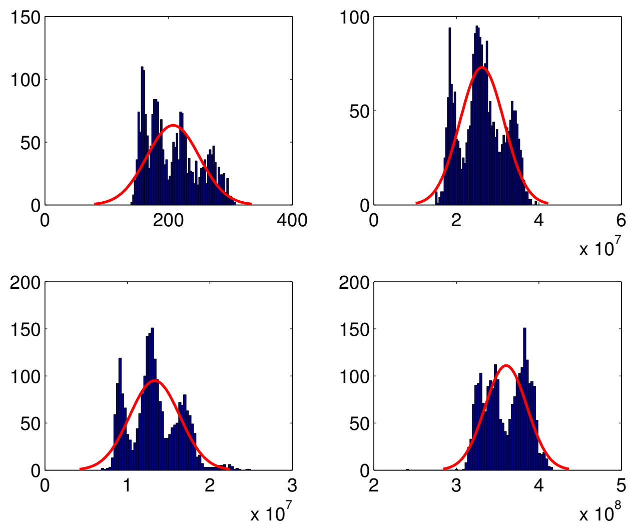


Figure 35: The distribution for peek throughput (top left), Socket I/O and Job Creation response time (top right), Client Request Worker response time (bottom left) and Persistence response time (bottom right) for micro benchmarking peek requests.

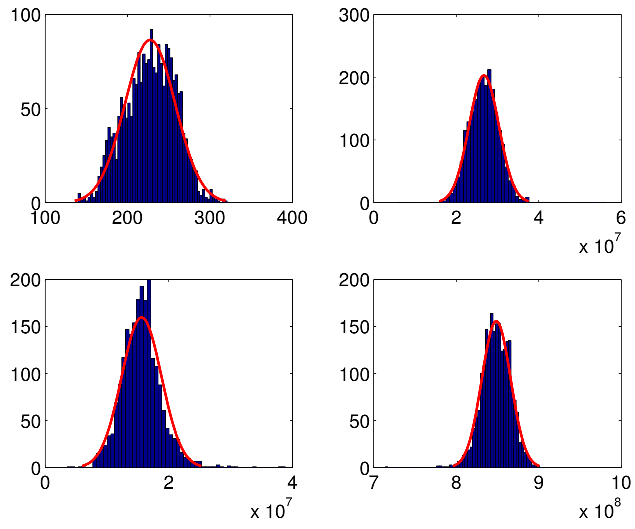


Figure 36: The distribution for peek throughput (top left), Socket I/O and Job Creation response time (top right), Client Request Worker response time (bottom left) and Persistence response time (bottom right) for micro benchmarking pop requests.

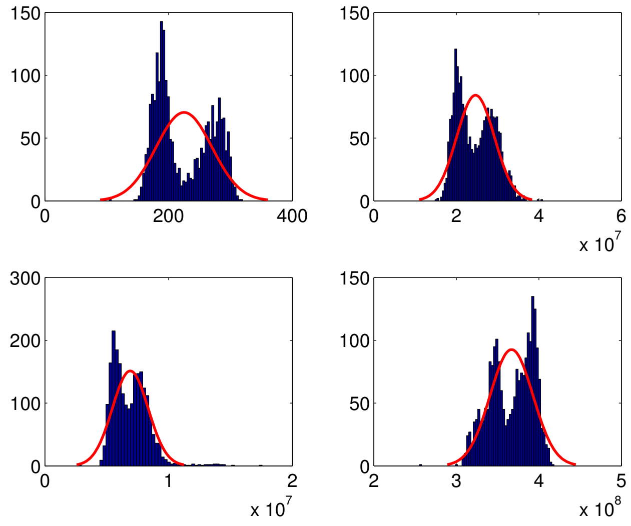


Figure 37: The distribution for peek throughput (top left), Socket I/O and Job Creation response time (top right), Client Request Worker response time (bottom left) and Persistence response time (bottom right) for micro benchmarking peek requests.