ASL MILESTONE 1

NICULIN TSCHURR[1]

CONTENTS

ABSTRACT

write abstract

* *tschurrn@student.ethz.ch*

1 INTRODUCTION

For the course Advanced Systems Lab I implemented a message passing system. In the following report, the system as well as design choices are explained. Some of the design decisions have been influenced by the way I originaly intended to test the system. The benefits and drawbacks of these decisions are discussed focussing on possible real world system as well as the performance during the tests that have been done on Amazon servers.

2 IMPLEMENTATION

This section first gives an overview of the system's components and how they work together. Afterwards the single components are discussed in more detail.

2.1 Overall design

The application consists of three major components:

- Database
- Middleware
- Client

Each component may run on a different machine and can be connected to another component via its IP address. There may only be one database, but as many clients and middlewares as wanted. My application requires every client to connect to a single middleware. This is sufficient for my application, since I can assume that the middlewares keep running during the whole time period of every test. All middlewares are connected to the same single database. In order to be able to simulate many clients on a single Amazon server instance a `ClientMaster` instance can create, register and deregister any number of clients. This has been implemented in order to facilitate creating database dumps and running experiments. The clients and middleware are implemented in two different Java projects.

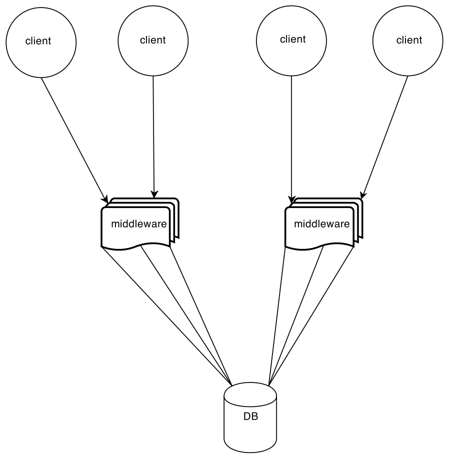


Figure 1: Connections Overview

2.2 Database

This section gives an overview about the database implementation. Performance characteristics are discussed later with the other test results.

2.2.1 *Design*

The database needs to store all communication between the clients. Clients have to write messages into queues. Every message has to be in exactly one queue, but can be addressed to a specific receiver or to everyone. This is shown in the ER-diagramm with two different relations. A Client can decide wether to read or pop a message. In my system when a client reads a message that was specifically for it, the message is always deleted (no read specific receiver messages). I did this in order to shorten the livetime of these messages in the system. I also thought that when a message is for everyone, maybe more than one client may be interested in the message, but if it is for one specific client, the client is responsible for storing the message locally.

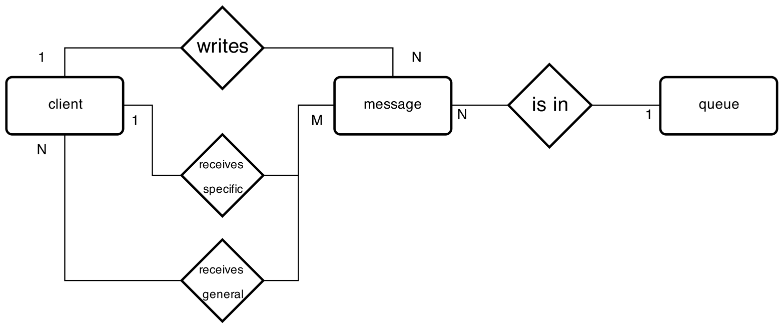


Figure 2: ER-diagramm

The database consists of 4 tables as shown below. I decided to create two tables to store all the information about a message. One table holds the message's content and an identifier, the other table holds all the needed metadata about the messages. By doing so, the metadata table has as many entries as when the content would be stored in the same table, but it needs much less memory. This design leads to many joins during runtime but on the other hand, I have many queries that only need the metadata table and then may read smaller tuples because the content is stored seperately. Overall I expect the cost of the joins to be bigger than what I save with the queries that only need the metadata table. I expect this especially because I added indices to every column in the metadata table. Therefore most queries perform the computationally intensive part only on the indices and then need to read only few tuples. The indices are required because there is a query for each column that needs to filter or sort it. The messages table references the metadata's primary key. This means that the message content has to be deleted before the metadata entry may be deleted. The table `queues` is required because a queue is not represented in the metadata table if it is empty. This may happen because all messages are deleted during runtime or when a new queue is created. By having an extra table where all the queues are stored the clients can easily find out in which queues they may write. The queues are referenced by the metadata table because in my system queues may be deleted only if a queue is empty. When a client tries to delete a non empty queue, it will fail because of the foreign key constraint. Finally the table clients serves several purposes. It is also referenced by the metadata table. It therefore is not possible to write to a non existing client and creating messages which would never be read. The table `clients` is also used for the client's registration. Clients can not only be created but also deregisterd and afterwards re-registered. When a client is created an entry in the table is created. The column flag indicates if the client is currently registered.

Table	Columns			Index
Name	Name	Type	Constraints	
metadata	message_id	SERIAL	PRIMARY_KEY	implicit
	timestamp	timestamp	DEFAULT_NOW()	Yes
	queue_id	int	REFERENCES queues(queue_id)	Yes
	sender_id	int	REFERNCES clients(client_id)	Yes
	receiver_id	int	REFERENCES clients(client_id)	Yes
messages	message_id	int	REFERENCES metadata(message_id)	Yes
	payload	varchar(3000)		No
queues	queue_id	SERIAL	PRIMARY_KEY	implicit
clients	client_id	SERIAL	PRIMARY_KEY	implicit
	flag	boolean	DEFAULT TRUE	No

Table 1: Database Tables

2.2.2 *Initialization*

The database is initialazed with the tables and indices as indicated in the table 1. Additionally a client with `client_id` 0 is added. When a message is not intended for a specific client it is sent to
texttclient_id 0. When a client tries to read a message it filters for its own id or 0 as the receiver.

2.2.3 *Queries*

All queries are implemented as `PreparedStatement`. This speeds up query execution. An overview of the queries is given in table 2 on the next page, for the queries themselves please have a look at the class `\code\middleware \src\middleware\Queries.java`. The query `deleteQueue` could have been implemented to first check if the queue is empty and then return if the deletion was successful. Because of the foreign key constraint this is not required, since the query will simply fail if the queue isn't empty.

Name	Arguments	Return Values	Description
`regNewCl`	-	new client_id	a new client is created
`regOldCl`	client_id	-	sets the clients flag to `true`
`deRegCl`	client_id	-	sets the clients flag to `false`
`createNewQueue`	-	queue_id	creates a new queue
`deleteQueue`	queue_id	-	deletes the specified queue (only works if the queue is empty)
`getClientIds`	-	client_id	returns all active clients
`getAllQueues`	-	queue_id	returns all queues
`getQueuesForClient`	client_id	queue_id	get all queues where messages for the specified clients are
`readTopMost`	queue_id, receiver_id	payload	returns the topmost message from the specified queue that may be read by the client, meaning its addressed to it or to everybody
`pop`	queue_id, receiver_id	payload	same as `readTopMost`, but deletes the message before returning it
`popFromOther`	receiver_id, sender_id	payload	returns and deletes the oldest message that the specified sender sent to this specific receiver
`insertMessage`	queue_id, sender_id, receiver_id, content	-	creates a message in the metadata table in the given queue with the specified sender and receiver, the content stored in the `messages` table

Table 2: Queries

2.3 Middleware

This section describes the middleware which communicates between the database and the clients. It focuses on the implementation. The performance characteristics are discussed later. Many clients can be connected to the middleware over a single port and the middleware has several connections to the database. Figure 3 on the following page shows the overall design of the middleware.

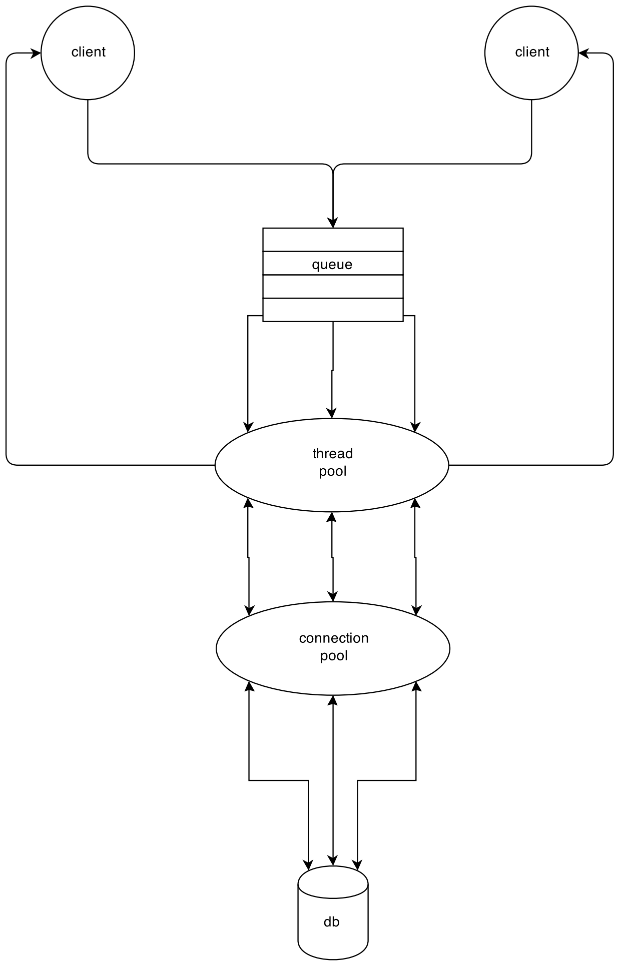


Figure 3: Middleware Design

The class `Controller` initializes the middleware by creating a thread that listens to incomming messages from clients, a thread pool that handles the client's requests, a database connection pool that is used by the handler threads and finally a logger. A client can connect to the middleware with its IP address on port 9000. The middleware runs a thread `RequestAcceptor` which queues all the client requests. The threads that execute the client requests are implemented in a thread pool. Everytime a thread from this pool finishes a request, the `RequestAcceptor` assigns it a new one. The thread handles the client request, which means it queries the database and returns the result to the client. This is implemented in the class`Handler`. This class first reads the clients socket and parses the request. It then attempts to get a new database connection from the connection pool. Over the database connection, the query or update can be executed with the use of prepared statements. When nothing has to be returned to the client the socket is closed even before accessing the database. Otherwise the return message is sent using the client's socket and the socket is closed afterwards.

As written above, the database connection pool is initialized by the `Controller`, which means that as many connections are created as stated in the run configurations and then the prepared statements are created for every connection in the class `MyConnection`. I decided to create one database connection for every `Handler` thread in the middleware. Therefore every `Handler` thread receives a database connection almost instantly. This means that when a client request is taken out of the queue and given to a `Handler` thread, the parsing takes very little time and the request gets executed on the database almost immediately. This means that every request has to wait in the queue and then again for the database to respond. If I had implemented the system with more `Handler` threads than database connections I might have a small performance increase because the database connections would not be unused during the time the `Handler` parses the request. Since the parsing is done very fast compared to the database execution time I decided to implement the design that is easier to test for.

I expected that the middleware performs the best if the number of `Handler` threads equals the number of CPU-cores, or twice as much, when using hyperthreading. But because there is only one database but potentially many middlewares and many more clients I expect the database to be the limiting factor. If this is the case I expect the system to perform the best if the number of database connection equals the number of CPU-cores of the database server. The test results for this hypothesis are discussed in the section 3.3 on page 12.

2.3.1 *Logging*

The middleware logs every request it processes. Each log entry consists of:

- request type
- timestamp
- time in queue
- database execution time
- time to send answer to the client

In order to minimize file access 500 log entries are written into a buffer before written to the log file. The database execution time can show when the database reaches a maximum workload. The time it takes to send the answer to the client can be used to analize the network.

In order to run the middleware, use `ant run -Dargs " <dbIP> <#dbConnections> <ownIP> <testVersion> "`, where the last argument two arguments can be arbitrary.

2.4 Clients

The client is the most complex component in this system. The client needs to be able to fill the database with messages of a certain size and create queues in order to get database dumps which can be used for testing. The client also needs to implement an interface to the middleware. In order to test the system, the client needs to be able to run over a long period of time without altering the number of messages in the database. I implemented different types of clients which can all be configured by the `ClientMaster`. The `ClientMaster` can create, register, deregister and start differint kinds of client instances.

2.4.1 *ClientBase*

This class provides the basic funcionality. It creates a socket for every request and connects to the middleware. This class defines a function for every type of request the client should be able to perform. Because the requests are serialized as `Strings` I implemented a class `Message` which can be used to create a request and then consistently serialize the request. When the client writes a message, a random content is created. There is a method which can create messages with random size in a specified interval. Queries can fail, e.g. when two clients try to pop the same message. One client succeeds and deletes the message while the other is still reading. The second client will fail. In a real world system the client would probably retry a couple of times and then give up after some time. A problem with this strategy arises when trying to test the system for throughput. It might be that there was only one message in the queue, now the other client has to wait for a long time. The client might even stuck if the queue gets deleted in the meantime. In order to get more consistent measurments for the tests, the client simply registers a failed pop request. At a later point the `ClientWorker` deals with the failed requests as discussed below. A frequently failed query is when a client tries to delete a non-empty queue. The client gets a SQLException, which has to be catched. When catched, the function returns that the delete was unsuccessful. It is important that a `rollback()` is executed on the connection, otherwise many of the following queries will fail. For every request the `ClientBase` calls the logger.

2.4.2 *FillerClient*

The FillerClient creates a specified number of queues, then waits for some seconds, retrieves all queues in the system and writes a specified amount of messages into the database. After every message the `FillerClient` looks if new queues were created. The `FillerClient` extends the `ClientWorker` because it should fill the database with the same kind of messages that are created during the experiments. This means that the percentage of messages for a specific receiver and the percentage of messages for everybody should be the same in the initial database dump and during the whole experiment.

2.4.3 *ClientWorker*

The `ClientWorker` achieves a mixture of requests for the experiments. It is implemented as a child class of `ClientBase` A fixed percentage of requests are read requests. The pop and write requests are executed with changing frequencies. This is because write requests have a very high probability to succeed, while about 2% of pop requests fail. These pop requests are counted by the client. If more writes have been executed than successful pops, the `ClientWorker` increases the probability of pops and vice versa. This keeps the number of messages in the database constant over a long period of time. This will be verified in section 3.3 on page 12. It is important to mention that the `ClientWorker` updates the list of queues with messages waiting and clients available periodically. This has been done in order to reduce the number of failed request due to missing messages in queues. But as it turned out this effect can be seen very clear in the experiments and may have an influence on the response time for different intervals. This will be discussed in more detail in the section 4.1 on page 15. Because the `ClientWorker` is started with a predefined database, it should register with the id of a client that created the database content. The id needed for registration is calculated in the `ClientMaster`.

2.4.4 *ClientMaster*

The `ClientMaster` represents an Amazon server instance. Its possible to configure the `ClientMaster` to create `FillerClient` or `ClientWorker` instances. The `ClientMaster` creates a random name in order to create unique log files. All `ClientBase` instances use their master for logging. As described above, a `ClientWorker` needs an existing id. Therefore one can configure the `ClientMaster` to create a specified amount of clients with id starting from the given offset.

In order to run the `ClientMaster`, two different configurations are possible. Fill the database with the help of `FillerClient` instances: `ant run -Dargs “<middlewareIP> <#ofClients> <offset> <verionNumber> <minMessageSize> <maxMessageSize> true <#ofQueuesclient> <#messagesclient>”`, where the offset may be arbitrary, as well as the verionNumber.

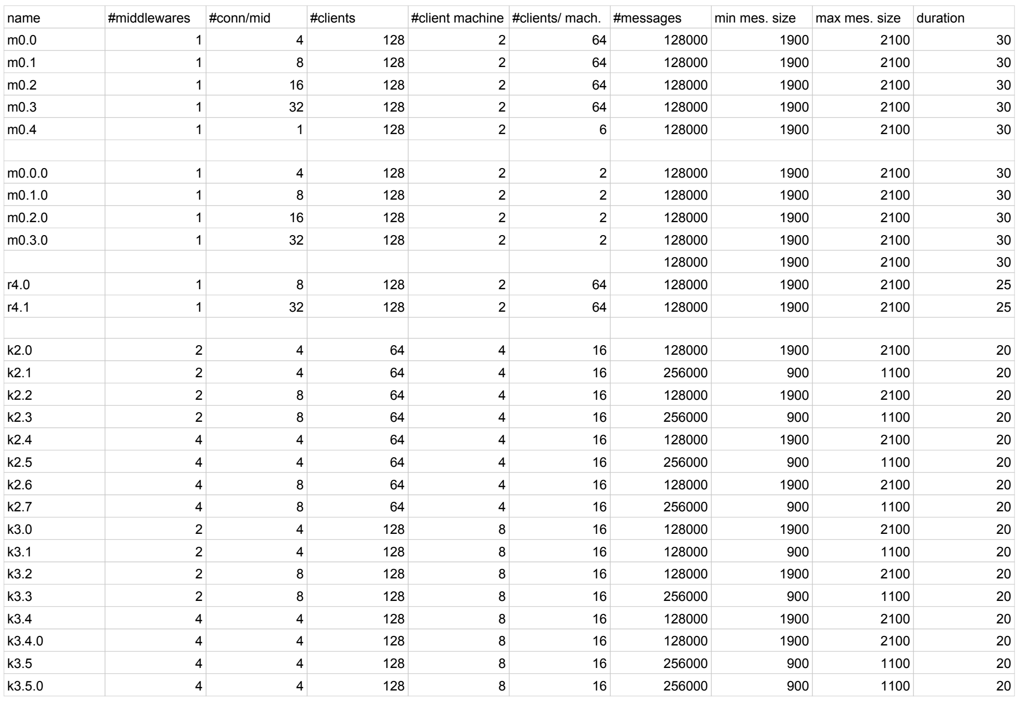
Start an experiment on an already existing database: `ant run -Dargs “<middlewareIP> <#ofClients> <offset> <verionNumber> <minMessageSize> <maxMessageSize>”`, here again, the versionNumber may be arbitrary, but the offset has to be set correctly if multiple instances are started.

2.4.5 *Logging*

The logging is done for every server instance rather than per client. This facilitates the interpretation of the log files. Therefore the logging is implemented at the level of the `ClientMaster`. All clients call the `ClientMaster`’s logger method in order to log every request. There are two different kinds of logs. One is very similar to the logging in the middleware. For every request, the request type, a timestamp and the execution time are stored. The logs are again written into a buffer and written to file every 250 requests. The second type of logging is done periodically. A thread is invoked every second to collect statistics. Because the thread is not invoked exactly every second, the data gets normalized. Here I measure how often every request type has been executed in the last second. I also log the successes and pop fails. Because I calculate the successes and failes only for a subset of the requests, they don’t equal the total amount of requests.

3 TESTING

For the Testing I run the database and all middlewares on c3.2xlarge instances. To simulate the clients I took c3.xlarge instances and run several clients on the same instance. In the following I will refere to the experiments by there name. The exact configurations for every experiment are shown in table 4 on the following page.



name	#middlewares	#conn/mid	#clients	#client machine	#clients/ mach.	#messages	min mes. size	max mes. size	duration
m0.0	1	4	128	2	64	128000	1900	2100	30
m0.1	1	8	128	2	64	128000	1900	2100	30
m0.2	1	16	128	2	64	128000	1900	2100	30
m0.3	1	32	128	2	64	128000	1900	2100	30
m0.4	1	1	128	2	6	128000	1900	2100	30
m0.0.0	1	4	128	2	2	128000	1900	2100	30
m0.1.0	1	8	128	2	2	128000	1900	2100	30
m0.2.0	1	16	128	2	2	128000	1900	2100	30
m0.3.0	1	32	128	2	2	128000	1900	2100	30
						128000	1900	2100	30
r4.0	1	8	128	2	64	128000	1900	2100	25
r4.1	1	32	128	2	64	128000	1900	2100	25
k2.0	2	4	64	4	16	128000	1900	2100	20
k2.1	2	4	64	4	16	256000	900	1100	20
k2.2	2	8	64	4	16	128000	1900	2100	20
k2.3	2	8	64	4	16	256000	900	1100	20
k2.4	4	4	64	4	16	128000	1900	2100	20
k2.5	4	4	64	4	16	256000	900	1100	20
k2.6	4	8	64	4	16	128000	1900	2100	20
k2.7	4	8	64	4	16	256000	900	1100	20
k3.0	2	4	128	8	16	128000	1900	2100	20
k3.1	2	4	128	8	16	128000	900	1100	20
k3.2	2	8	128	8	16	128000	1900	2100	20
k3.3	2	8	128	8	16	256000	900	1100	20
k3.4	4	4	128	8	16	128000	1900	2100	20
k3.4.0	4	4	128	8	16	128000	1900	2100	20
k3.5	4	4	128	8	16	256000	900	1100	20
k3.5.0	4	4	128	8	16	256000	900	1100	20

Figure 4: Experiments Overview

3.1 Stability

In this section I will show that my system runs stable. This is shown with the same experiments as the middleware testing has been done. For this section I evaluated 8 experiments, the experiments are called m0.*, where the experiments m0.0 to m0.3 describe the first run, the experiments m0.0.0 to m0.3.0 describe the second run. The two sets of experiments have been conducted on two consecutive days, but with a time difference of 6 hours.

3.1.1 *Database*

First I show that the number of messages in the database is constant over a longer period of time. This is a condition in order to get meaningful test results. This can be seen in figure 5 on the next page, where the number of messages is shown over 28 minutes from 4 different experiments. The graph shows that the number of messages fluctuates a little, but on average is always around the initial size.

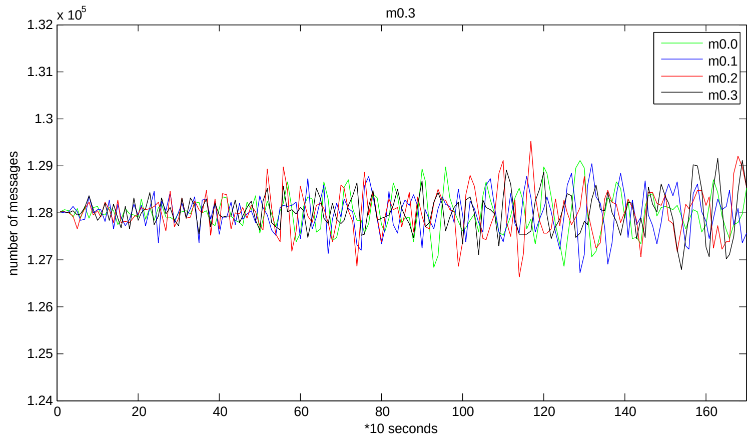


Figure 5: Number of Messages in Database

Experiment	Average	StD
m0.0	127990	401.1
m0.1	127990	408.8
m0.2	128000	449.6
m0.3	128000	402.4

Table 3: Number Of Messages in Database

3.1.2 *Amazon Servers*

The Amazon servers produced very similar results for the different runs of the experiments. The clients throughput has been almost identical for all the experiments. Every subplot shows two experiments with identical configurations conducted at different times. skalen unterschiedlich

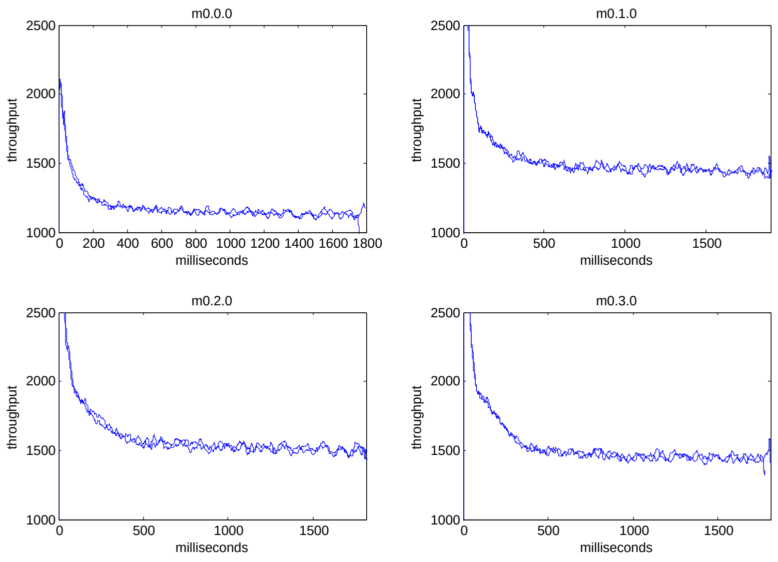


Figure 6: Client Throughput

3.2 Throughput and Response Time

As can be seen in figure 6 on the preceding page, the throughput is very constant after a warm-up phase. I therefore did all the other experiments only over 20 minutes instead of 30. When comparing the experiments I looked at the interval from 7 to 18 minutes, since during this interval the system's performance is very stable, therefore all calculation of averages are on this interval if not stated otherwise. The table 4 shows the similar averages for the two runs of every experiments shown in figure 6 on the preceding page.

Experiment	Average	StD
m0.0	1151	112
m0.0.0	1141	101
m0.1	1475	143
m0.1.0	1477	141
m0.2	1537	140
m0.2.0	1548	139
m0.3	1473	129
m0.3.0	1477	142

Table 4: Client Average Throughput

The performance drop at the beginning may have different causes. One of the causes is that the indices corrupt over time. This causes the indices and the database to grow and get slower. To show the index corruption has an effect on the throughput I conducted experiments r4.0 and r4.1. In these experiments I rebuilt the index after 15 minutes. Once with `reindex` and once with `vacuum full` which rebuilts the database completely from the current snapshot. On the left hand side of figure 7 one can see the database size and the effect the two operations had. On the rigt side the througput is shown. Rebuilding the database and the index clearly have a positive shortterm effect on the performance.

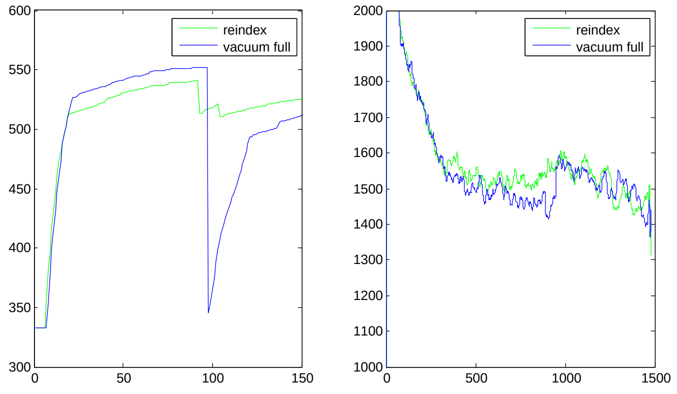


Figure 7: Reindex & Vacuum Full

3.3 Middleware and Database in Depth

In this section I have a closer look at the system components. I therefore use the experiments m0.0.0 to m0.3.0. The idea is to show the different response times of the requests. I also compare the middleware's performance depending on how many threads and therefore database connections the middleware has.

Figure 8 shows that the middleware performs the poorest with 4 connections and almost identical with 8, 16 and 32 connections. This result can be expected as long as the database is not already the bottleneck with 4 connections. It can be expected because the middleware runs on a server with 8 CPU-cores and will have only half its power with 4 threads, compared to 8 or more.

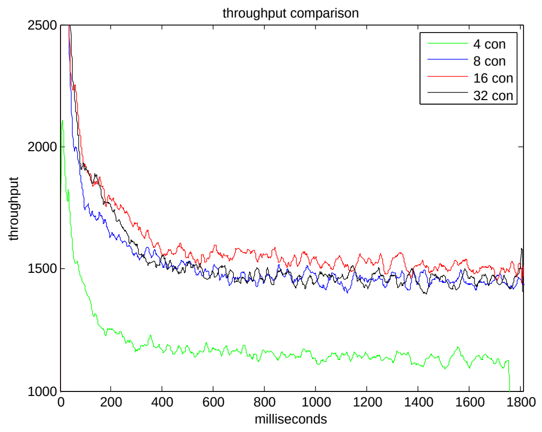


Figure 8: Throughput Comparison

3.4 System performance

To evaluate the overall systems performance and to find the configuration with the highest throughput as well as to understand the relation between different factors I did a 2k-testing. The exact cofigurations of the 16 tests, called k can be found in the table 9 on page 17. I selected 4 factors that I did the 2k-testing with. These factors are:

- number of clients
- number of middlewares
- number of connections per middleware
- number of messages in the database

I decided to run 16 clients on every client server instance. I also decided to scale the number of queues in the database with the number of messages. Therefore the average number of messages in a queue is equal in all experiments. I decided to do this because when a client asks for a message in a queue, all messages in this queue have to be sorted according to the timestamp. This will take the database a long time and I wanted to test if the number of messages has an effect on the execution time and not the number of queues. I also decreased the message size for the dataset with more messages.I did this in order to reduce the effect of the database size on the performance. Unfortunatelly I lost the statistic files of two client machine in experiment k2.3. I therefore investigated the differences of the client machines in a single experiment. Figure 10 on the following page shows the 4 client machines for a single experiment in every subplot. Even in the start-up phase the different client machines behave almost equaly. I therefore conclude that doubling the data from the two client machines still gives valid results. Another indicator for the very similar performance of the client machines is given in section 4.1 on page 15, where I show that periodic tasks are done by all clients at almost the same time.

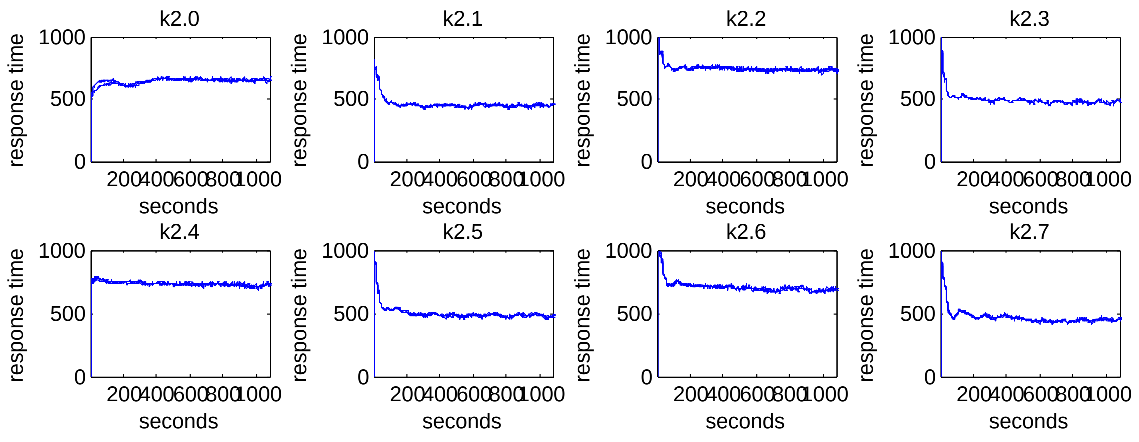


Figure 10: Client Machine Comparison

The calculation of the different factors can be seen in 9 on the following page. I didn't include the standard deviation for the response time. It only makes sense to include it for a single type of operation, but not for all. When a client writes to the database, the socket is close as soon as the request is given to a `Handler` thread. Therefore a write operation measures only the time in the queue, while requests with a return value measure the whole roundtrip time. Because of the very different execution times of the different requests, the standard deviation over all requests is almost as big as the average value.

The number of middlewares has little influence on the system's performance. If the number of middlewares has little influence on the system's performance than the Database has to be the bottleneck. In this case the number of connections cannot have an influence on the overall performance. This is because the database has 8 CPU-cores and should therefore perform best with 8 or 16 (hyperthreading) threads. Conducting the experiment with 8 connections per middleware would have given a performance increase in case the middleware was the bottleneck with only 4 threads, as can be seen in figure 8 on page 13.

The number of messages has a huge impact on the system. If the database has more messages it is much slower. As already explained, this is not because the queues are longer and it takes longer to sort the messages. The effect is mainly because the metadata table is twice as big for the setup with more messages. For every request the whole table, meaning one or more indices have to be queried. Since these indices are also twice as big the performance is worse.

The number of clients has a huge impact on the systems performance. More clients reduces the throughput and increases the response time. With twice as many clients but the same database performance it can be expected that the responstime doubles, since the throughput stays the same. But I am not sure why it has such a huge influence on the throughput. A possible explenation could be that more clients increase the probability of cache misses. With fewer clients maybe the queues for most clients could be kept in memory.

My system performs the best in experiments k2.2 and k2.4. It is obvious from that those two perform very similarly since the only difference between the experiments is the number of middlewares. As shown above the number of middlewares has little influence on my system. In both experiments the throughput reaches about 2900 requests/second, with a response time of 29 milliseconds.

4 POSSIBLE PROJECT CHANGES

In this section I briefly describe what I would change or do additionally if I redid the project. Some of the system components would be changed, as well as the experiment setup.

4.1 System Changes

My `ClientWorker` periodically checks for all queues, all clients and all queues where messages for this client exist. I underestimated the robustness of Amazons servers and thought that after a short period of time, the different client machines will do this update evenly distributed over the time. As it turned out, this was never the case. In all 2k-tests, a clear clustering of these updates can be seen. If I could implement the application again I would certainly do this update at random time intervalls. An example of this clustering is given for experiment k3.7 in figure 11 on page 16

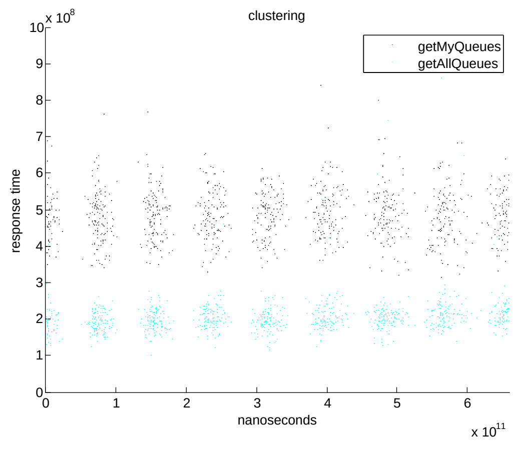
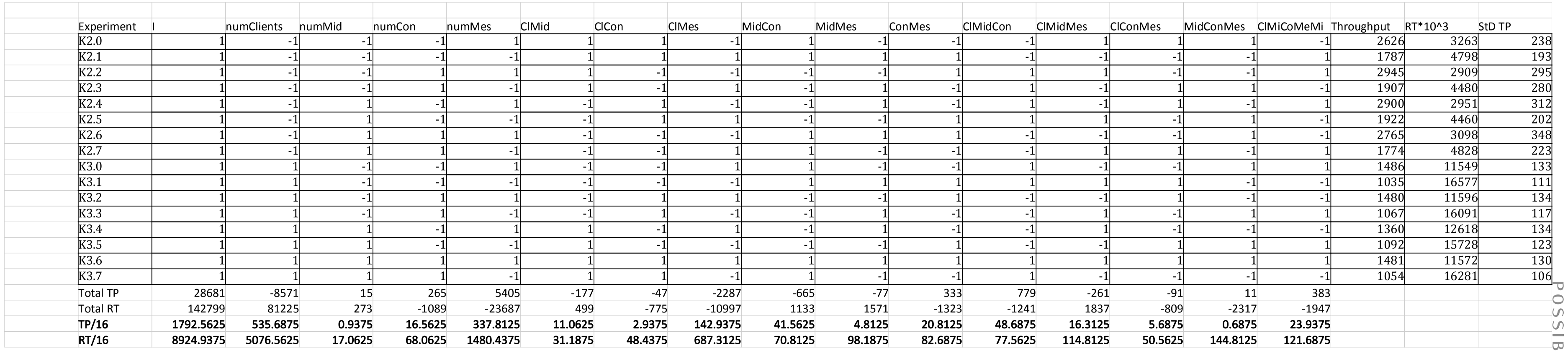


Figure 11: Cluster Pattern

4.2 Experiment Changes

I explicitly didn't include the number of messages in a queue as a parameter in my tests. After seeing different behaviors of my system I wonder how the sorting of messages according to a timestamp impacts the database performance. The Postgres configuration also might have a huge impact on the system's performance. I only tuned a few memory and cache parameters. It would be interesting to test for the effect of these parameters. Especially the option of automatic reindexing or vacuuming would be interisting. This was not an option for this project since it could have given fluctuating throughputs. Nevertheless I think these are options that are useful in a real world system, espacially if the system runs for an extended period of time.



Experiment	I	numClients	numMid	numCon	numMes	ClMid	ClCon	ClMes	MidCon	MidMes	ConMes	ClMidCon	ClMidMes	ClConMes	MidConMes	ClMiCoMeMi	Throughput	RT*10^3	StD TP
K2.0	1	-1	-1	-1	1	1	1	-1	1	-1	-1	-1	1	1	1	-1	2626	3263	238
K2.1	1	-1	-1	-1	-1	1	1	1	1	1	1	-1	-1	-1	-1	1	1787	4798	193
K2.2	1	-1	-1	1	1	1	-1	-1	-1	-1	1	1	1	-1	-1	1	2945	2909	295
K2.3	1	-1	-1	1	-1	1	-1	1	-1	1	-1	1	-1	1	1	-1	1907	4480	280
K2.4	1	-1	1	-1	1	-1	1	-1	-1	1	-1	1	-1	1	-1	1	2900	2951	312
K2.5	1	-1	1	-1	-1	-1	1	1	-1	-1	1	1	1	-1	1	-1	1922	4460	202
K2.6	1	-1	1	1	1	-1	-1	-1	1	1	1	-1	-1	-1	1	-1	2765	3098	348
K2.7	1	-1	1	1	-1	-1	-1	1	1	-1	-1	-1	1	1	-1	1	1774	4828	223
K3.0	1	1	-1	-1	1	-1	-1	1	1	-1	-1	1	-1	-1	1	1	1486	11549	133
K3.1	1	1	-1	-1	-1	-1	-1	-1	1	1	1	1	1	1	-1	-1	1035	16577	111
K3.2	1	1	-1	1	1	-1	1	1	-1	-1	1	-1	-1	1	-1	-1	1480	11596	134
K3.3	1	1	-1	1	-1	-1	1	-1	-1	1	-1	-1	1	-1	1	1	1067	16091	117
K3.4	1	1	1	-1	1	1	-1	1	-1	1	-1	-1	1	-1	-1	-1	1360	12618	134
K3.5	1	1	1	-1	-1	1	-1	-1	-1	-1	1	-1	-1	1	1	1	1092	15728	123
K3.6	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1481	11572	130
K3.7	1	1	1	1	-1	1	1	-1	1	-1	-1	1	-1	-1	-1	-1	1054	16281	106
Total TP	28681	-8571	15	265	5405	-177	-47	-2287	-665	-77	333	779	-261	-91	11	383			
Total RT	142799	81225	273	-1089	-23687	499	-775	-10997	1133	1571	-1323	-1241	1837	-809	-2317	-1947			
TP/16	**1792.5625**	**535.6875**	**0.9375**	**16.5625**	**337.8125**	**11.0625**	**2.9375**	**142.9375**	**41.5625**	**4.8125**	**20.8125**	**48.6875**	**16.3125**	**5.6875**	**0.6875**	**23.9375**			
RT/16	**8924.9375**	**5076.5625**	**17.0625**	**68.0625**	**1480.4375**	**31.1875**	**48.4375**	**687.3125**	**70.8125**	**98.1875**	**82.6875**	**77.5625**	**114.8125**	**50.5625**	**144.8125**	**121.6875**			

Figure 9: 2k-Tests