ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Advanced Systems Lab

A Message Passing System Performance Analysis

(Milestone 1)

Departement of Computer Science

Andrin Jenal

November 9, 2014

Contents

1 Introduction

This report is about the system design, implementation and performance analysis of a simple concurrent multi-tier message passing system. The system consists of clients, a middleware and a persistent database storage. Clients and messages have unique system wide identification numbers. All components work in a distributed manner and run independently of each other. Firstly, all basic features and design principles will be introduced and examined. For this purpose multiple different tests, experiments and micro-benchmarks are carried out to determine the behavior of each component. Based on the gained knowledge about the behavior of the individual system components and the importance of the different system parameters the overall system performance will be analyzed. In the end the system behavior will be discussed and explained on the basis of the previous experiments.

2 PostgreSQL Database

Designing the database for such a simple system was straightforward. To keep it as clear as possible only two tables are created. For monitoring the database it was relied on the `pgAdmin` tool provided by Linux. With help of this tool and the log file it is possible to detect failing operations or other problems in the database.

2.1 Design Decisions

The database consists of two tables *mps_message* and *mps_queue* as shown in Figure 1. For the sake of simplicity and since clients only possess a unique number for identification a table for clients is omitted. This number is also visible in the message table. Uniqueness is guaranteed from the client side of the application. On the *mps_message* table a foreign key constrained is imposed to the unique identification number of the queue table, which makes it only possible to write to existing queues. On delete of a queue a cascade function is called. The primary key for the message table is the the unique identification number *m_message_id* of a message. The fields of the *mps_message* table have the following semantics:

- *m_sender_id* determines the client which sent the message
- *m_receiver_id* determines the indented client that receives the message
- *m_queue_id* refers to the queue the message belongs to
- *m_message_body* contains the actual message text
- *m_arrival_time* determines when a message arrived to the database

Some of the queries will be executed many times on a running system. To accelerate the access to desired table columns, different indexes are defined, especially in the case when a query contains a "WHERE" clause. For the message table this is the case for the arrival time, receiver id and queue id. The most commonly used queries need the

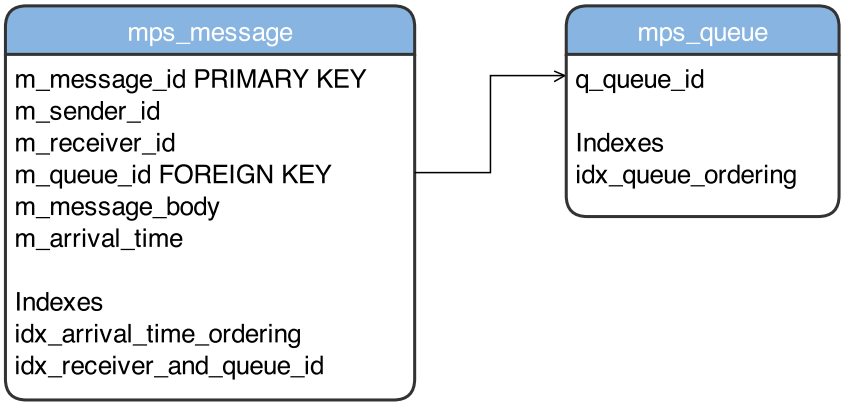


Figure 1: ER diagram of the database schema

information of the arrival time for ordering according to the latest message and the information about the receiver id and queue id. Therefore one index for the *arrival_time* and one index for the *receiver_id* linked with the *queue_id* are imposed.

Another requirement to accelerate the database accesses are stored procedures. This keeps not only the interface between the database and server as simple as possible, but also has a positive effect on performance. In this way it was also possible to use only `java.sql.CallableStatement` as statement type on the application side. The stored procedures on the database side perform the actions of reading, writing and deleting messages and for creating, query and deleting queues.

The way the database is implemented, the performance is a key requirement. It is therefore expected, that only changing additional configurations of the PostgreSQL server will mainly affect performance.

2.2 Database Performance Characteristics

It is assumed that accessing the database and executing queries concurrently will affect the performance significantly, since there is a lot of I/O action contained. Early measurements showed that this assumption is indeed true. One aspect that affects the SQL performance significantly is the size of the messages. Furthermore, the execution time of the queries contributed most to the latency of a request. As the logic of the middleware is quite simple it is expected that analyzing and processing the requests will probably not be the heaviest computation.

2.3 Problems

On later experiments it was revealed that PostgreSQL kept warning about `checkpoint_segments` were too low and at one point the database became very slow. These two observations were the reason that the configuration of PostgreSQL was slightly adapted. First, the `checkpoint_segments` are set to 80 and second, the `shared_buffers` size is set to 2GB.

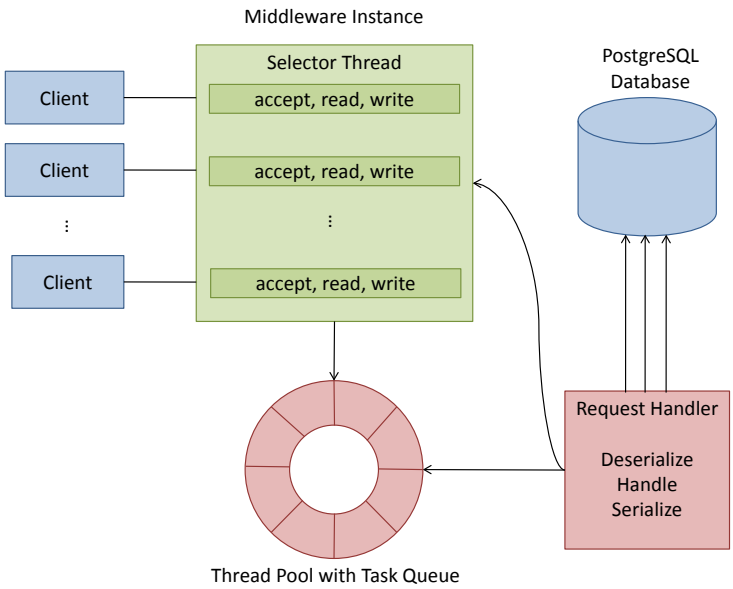


Figure 2: A schematic overview of the middleware interacting with multiple clients and a database

3 Middleware

The place where most of the action happens is the multi-threaded middleware. Requests from clients will be accepted and processed according to their type. All requests attempt to either read, write or update database tables and will always return a response which is sent back to the clients. A schematic overview is depicted in Figure 2. One such middleware component can be instantiated multiple times to increase performance.

3.1 Design Decisions

To guarantee a reliable communication between clients and server, it makes use of TCP. However, establishing TCP connections is very expensive. Therefore it makes sense to keep connections open as long as clients send and receive requests. This is possible using the `java.nio.*` package. In this way the overall performance of opening sockets, handling incoming data and scheduling tasks is improved. The non-blocking thread, a selector, is keeping track of all active connections to the clients and accepts incoming request. By passing the requests to a thread pool the pendant tasks are distributed among the available threads, which deserialize, handle and serialize requests and responses respectively.

The thread pool is an `ExecutorService` with fixed size and uses a standard implementation of a `BlockingQueue` of infinite size, according to the API specifications. This means if there are too many incoming tasks that should be handled, the queue will eventually blow up and never be processed. On the other hand, the selector thread should not be too busy with accepting requests and passing them on. For this reason, the selector thread should scale well with a increasing number of clients.

Regarding the communication between the middleware and database it makes sense

to use a connection pool. This way connections to the database never get physically closed, but will just be passed on to the next thread that needs to query the database. The number of connections in the pool is fixed and equals the number of threads in the thread pool. This is because all client requests require database access, therefore each thread needs a free connection from the connection pool. The pooled connection is realized with `org.postgresql.jdbc3.Jdbc3PoolingDataSource`.

3.2 Middleware Performance Characteristics

The complexity of the middleware is kept as low as possible. It is no need of having difficult logic to handle incoming requests. It is therefore expected that the processing of incoming requests will be fast and consequently the processing time low. Since stored procedures were used in the underlying database it is not even necessary to set up complex queries. However, it is assumed that each middleware instance will have its capacity limits where a certain throughput maximum is reached. In early experiments (Section 7.1) it was shown that the processing time did contribute far less to the latency than the query execution time, which was expected.

The middleware is the core component of the messaging system and it is built in an easy extensible manner. Performance and simplicity are key features that have been in the focus while building the middleware. It is expected that, while a single middleware instance already performs well, multiple running instances in parallel will even scale better.

4 Clients

Mainly for performance analysis and testing clients interact with the middleware. A client can perform the following requests: creating and deleting queues, sending, receiving or deleting messages from particular receiver or from a particular queue and query for queues or messages. As a sanity check, all basic functionalities of all requests have been extensively tested on correctness using the JUnit test suite (these tests will also be uploaded to the svn repository). This included also testing exception paths and their behavior. Correct behavior of the randomized clients is therefore guaranteed.

To facilitate the experiments and deployment, a client itself runs multiple threads, where each thread simulates a client independently of the others. The communication with the middleware is done by serializing a simple class `mps.request.Request.java` manually. First, the `java.io.Serializable` approach was used, but it turned out that this is an unnecessary overhead. To reduce the experimental complexity it is assumed that the system under test (SUT) is a closed system. Therefore, a client waits after each request for a response (`mps.request.Response.java`). Unless, the server did not crash it is guaranteed that each request will be answered. Responses are deserialized on the client side.

It is also possible to run multiple physical client instances where each instance simulates multiple virtual clients. Thus, clients scale very well, as long as a single instance is not overloading the network with requests. To describe the performance of clients as detailed as possible, clients are restricted to send a certain amount of messages

per seconds. Which is also the unit of determining the length in time of an experiment. To guarantee correctness in the number of sent messages per second the thinking time of each virtual client is also considered. The whole workload is generated by the clients and the maximum workload is defined the following: $maximum\ workload = number\_of\_clients * number\_of\_requests\_per\_second$

4.1 Experimental Configurations

The default behavior of clients is defined in the `mps.config.Config.java` class. These properties will be overwritten by configuration properties used for a particular experiment. As a general guideline for this application a client will produce as much data as he deletes to keep the average load of the system as stable as possible. This property may vary on certain experiments where the limit of components is tested. Concretely, mostly it is implemented that a client sends messages 66% of the time and receives responses 33% of the time, if not otherwise stated. It is to be mentioned that receiving responses also means, that if the request was successfully received, the specific message gets deleted from the database. Previously it was mentioned, that a client sends a certain amount of requests per second. To guarantee this behavior the sleep time in between sending requests takes into account the thinking time and is defined as: `sleepTime = (1000 / numRequestsToSendPerSecond) - thinkingTime`. Furthermore, though all request types have been extensively tested, for the purpose of performance analysis in this project only two requests types will be considered. Namely these are the two requests: sending messages and receiving messages from a certain queue, where the recipient is specified or not.

5 Overall Design

The system is built in a manner that it easy to vary the different components in size and behavior. For all experiments there is one client instance that simulates multiple virtual clients and only one physical database. The number of instances of the middleware changes depending on the type of benchmark, as stated in each experiment description. For each test run there is one specific `config.property` file which defines the experiment configuration. In case this file is not provided the system falls into the default behavior which is implemented in `mps.config.Config.java`. As mentioned above for all experiments the load of the system should not increase rapidly, therefore the balance of random requests is about 66% sending and 33% receiving and deleting messages.

5.1 Experiment Setup

To handle the setup, several scripts will be executed. For deploying the system to the different Amazon instances the following scripts are used: `copy_client_to_remote, copy_server_to_remote, update_config_on_remote`. After successfully running an experiment, the benchmarks log files will be retrieved from the server with `get_remote_logs`. All the further processing of the gained data is handled locally. First the cleaning up of the data is done with `process_data` or `timing_analysis` and then the final plots are mainly achieved with `plot_throughput` and `plot_response_time` which rely on `R` and

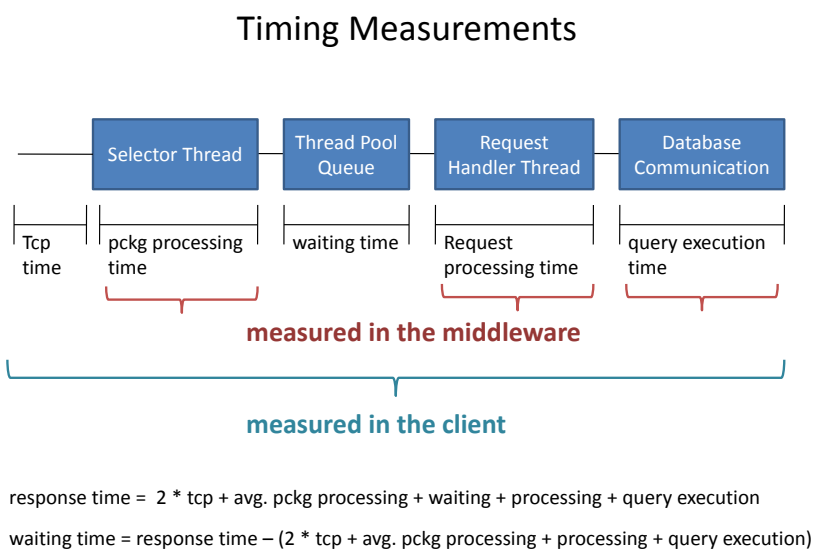


Figure 3: A schematic overview of the different timing measurements from the client's and the middlware's perspective.

`gnuplot`. For all experiments the warm-up and cool-down phases are always cut off. This amount depends on the length of an experiment and is about 1 to 5 minutes. All the experiments are conducted in such a way that they reached a 95% level of confidence for an interval 5% around the mean. This is usually already the case after running the experiment a few minutes. For the plots the mean of 60 seconds of the mean for one second is calculated. All the scripts can be found in the `scripts/` folder.

Before each experiment can start, the database is initially restored from a backup image. This backup contains a prepopulated message table with 1000 random messages, a queue table with 100 queues, all indexes and the stored procedures. The reason to use a non-empty database is, to avoid too many database misses in the beginning of the experiment. The nice thing about indexes and stored procedure is, that they are automatically included in the backup as well. For almost all experiments some fixed parameters are used, like the message size and number of available queues.

All the following pre-examination, microbenchmarks and experiments are conducted on the Amazon machines. For the database component a machine of the type *m3.2xlarge* was chosen and the client and the server instance are running each on a *m3.xlarge* machine.

5.2 Logging System Numbers

To exactly know what is going on in the system and therefore understand how the final system behaves, it was a core desire to have meaningful numbers. In this project it was tried to obtain an exact trace of each individual message. Therefore, logging happens especially at two places. One is the client that records all messages that he sends and receives, and assigns unique client and request to them. The other is the middleware instance which keeps track of all requests that pass through it (see Figure 3). In the

middleware two different times are measured - *processing_time* and *sql_execution_time*. The *processing_time* includes the deserialization, serialization and the sql execution time, but not the waiting time in the task queue of the thread pool. The *sql_execution_time* measures the time it takes to execute a single SQL query. To compute the waiting time in the task queue one has to know the round trip time and the network time. With all this information, which is available thanks to the clients logging the waiting time is: $waiting\_time = latency - (2 * network\_time + processing\_time)$

6 System Stability

It was decided to increase the complexity of the experiments ongoing. With this approach it was possible to figure out how individual components behave, how they are limited and how they interact. For stability testing of the system first some trace experiments were run. All the traces run on the Amazon cluster with the specified machines. The 60 mininutes trace (Figure 4) is a pre-examination of the system and shows some undesired behavior. After 40 minutes there occurs some strong oscillations, which is most probably because of thrashing. However, the throughput remains constant over time, which is already an indication that everything works well, even regarding the fact that the size of the database is increasing.

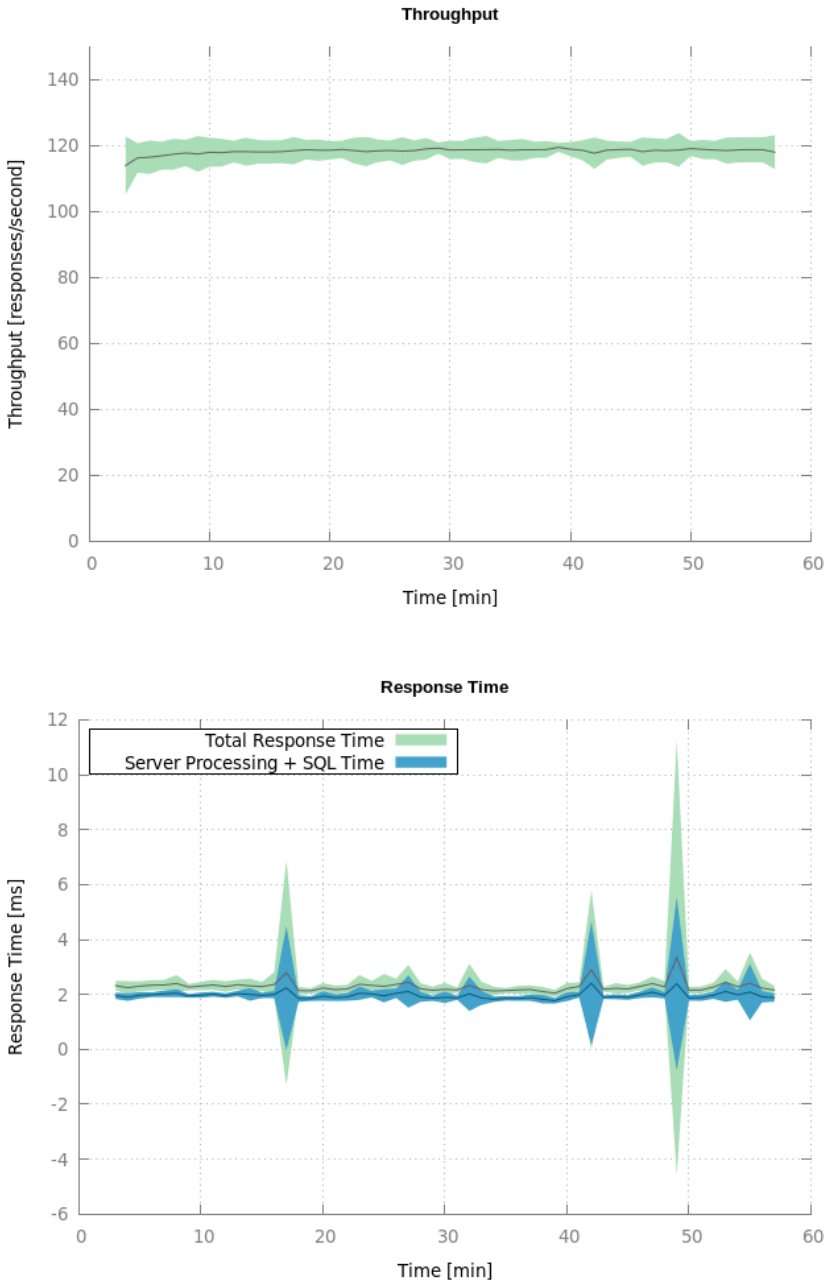


Figure 4: Amazon trace run of 60 minutes: *Number of clients* = 60, *requests per seconds* = 2, *number of instances* = 1, *number of threads in thread pool* = 20

After the gained knowledge of the first traces, the PosgreSQL configuration was slightly adapted. The size of the `shared_buffers` was increased to 2GB. With this configuration a 120 minute trace was conducted, which is depicted in Figure 5. As one can see, this trace is already much more stable, even though the load was increased. For this experiment 60 clients send simultaneously 10 requests per second, which is visible in the averaged throughput graph. Response time and throughput stay constant during the whole experiment.

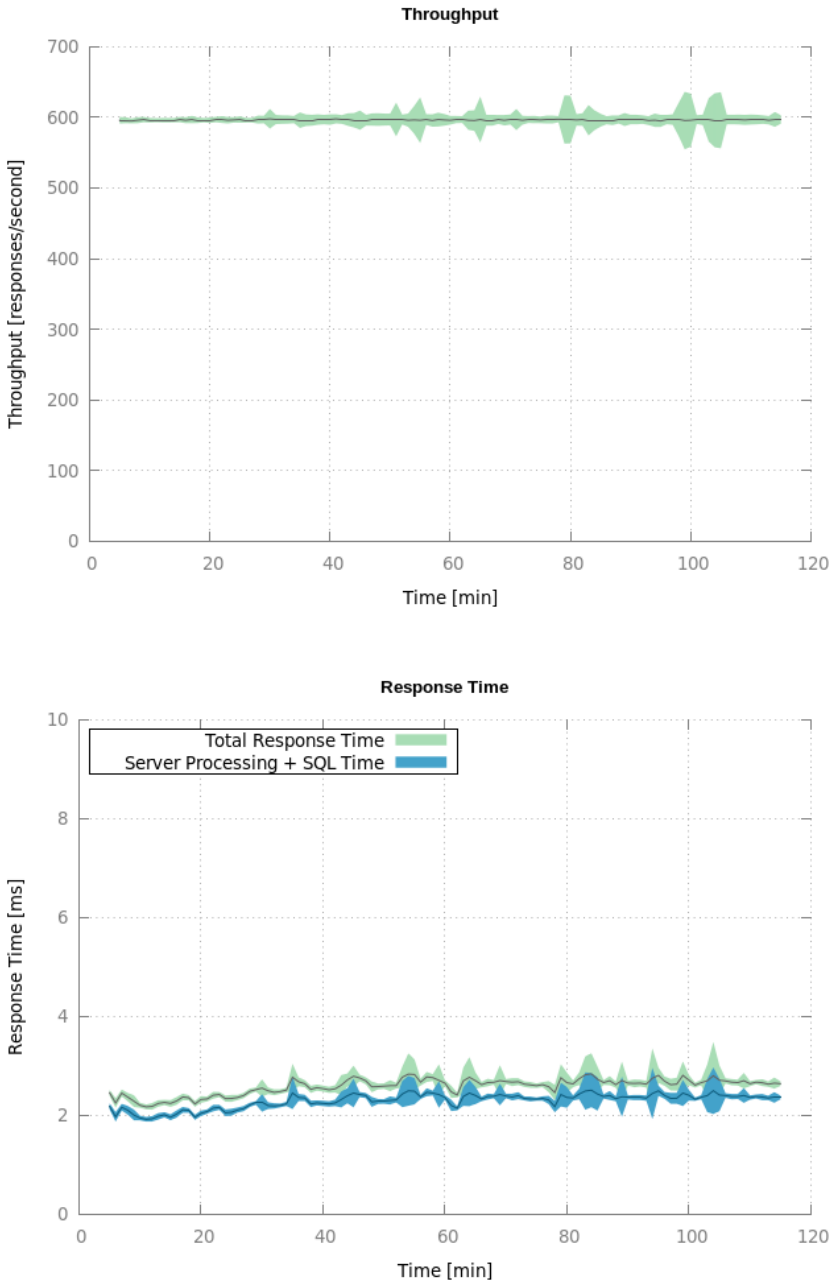


Figure 5: Amazon trace run of 120 minutes: *Number of clients* = 60, *requests per seconds* = 10, *number of instances* = 1, *number of threads in thread pool* = 50

7 System Performance

With the previous trace experiments it was shown that the system behaves stable under well-defined load. In this section the interference of different system parameters and limits of the system will be examined and possible bottlenecks will be searched. But first some microbenchmarks are performed to gain knowledge about how individual requests behave. Later on the systems limit will be extensively tested.

7.1 Microbenchmarks

To begin with the analysis, single request types where examined in the first place. In the pie charts (see Figure 6) sending and receiving requests with different message size are compared to each other. It is obvious that they all have similar percentages for the timing for the different parts of the system. Since these requests are the most complex ones and also the ones that are used for the remaining benchmarks, the repetition of the same experiment is omitted for the other request types. For the microbenchmark of the sending request the database was initially emptied and for the receiving microbenchmark the database was filled with 1000 random messages to avoid failing queries. The action was performed by one client thread, keeping all parameters in the middleware constant.

7.2 2k Factorial Design

The systems behavior is influenced by many factors and a lot of different parameters. For this project, the most important factors are: *The number of clients*, *the number of middleware instances*, *the number of threads*, *the number of pooled connection to the database*, *the number of requests sent per second*, *the message size* and *the number of messages in the database*. Considering all factors for an experimental performance analysis is impossible. Therefore, focusing on the middleware, only the two primary factors *the number of middleware instances* and *the number of threads* are chosen for the 2^{kr} factorial design (according to R. Jain [1]), with a replication of three.

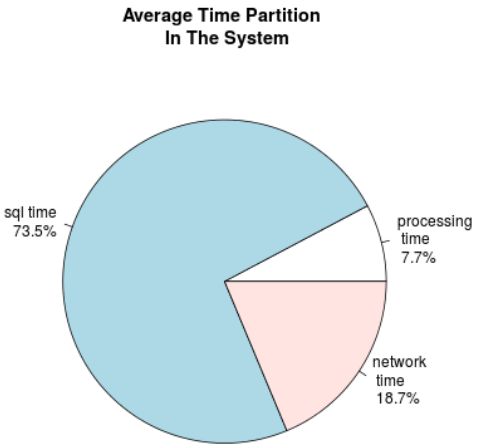
Number Of Middleware Instances	Number Of Threads
1	20
2	60

I	A	B	AB	y	$\bar{y}_{mean}$
1	-1	-1	1	(584.73, 581.95, 580.53)	582.40
1	1	-1	-1	(587.28, 588.51, 578.30)	584.70
1	-1	1	-1	(579.56, 579.89, 586.82)	582.09
1	1	1	1	(560.35, 563.38, 578.81)	567.51
2316.72	-12.27	-17.50	-16.87		Total
579.18	-3.06	-4.37	-4.21		Total/4

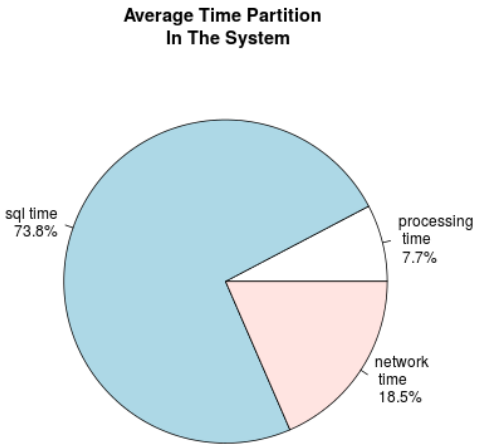
SSE	SSA	SSB	SSAB	SST
300.84	113.05	229.76	213.46	857.13

Percentage of Variation			
35.09%	13.19%	26.80%	24.90%

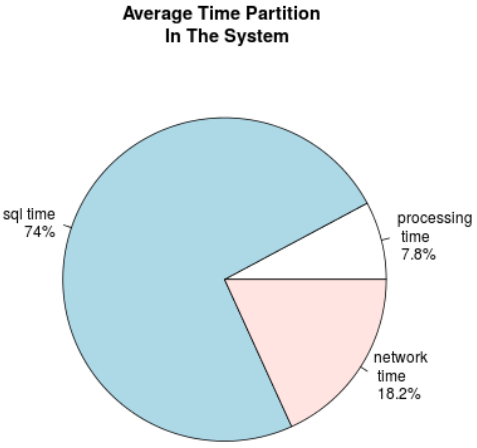
The response variables, are the throughput per minute measured for the 4 different levels. In the last row of the top table all the q's are listed (q_0, q_A, q_B, q_{AB}). It stands out that all the values are negative, which is not a good indication for the systems behavior, since it is assumed that the system performance scales with increasing middlewares and number of threads in each middleware. In the lower table the percentage of variations are listed. From the values, it is clear, that no factor is of major importance (which is



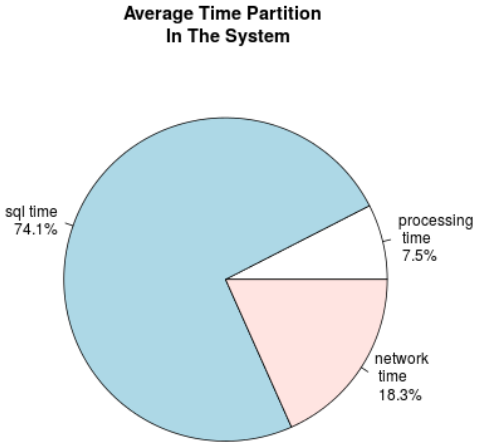
(a) Timing partition for sending small messages.



(b) Timing partition for receiving small messages.



(c) Timing partition for sending large messages.



(d) Timing partition for receiving large messages.

Figure 6: Amount of time request spends in different components of the system.

already obvious, when observing the mean trend of the measured throughputs). However, it stands out that, the factor *Number Of Threads* clearly is twice as relevant as the factor *Number Of Middleware Instances*. It might be, that for a better analysis the level discrepancy should have been larger. However, during the analysis of the 2^k factorial design a bug was revealed in the implementation which explains the observed behavior (see Figure 7). The middleware instances are sharing a common connection pool to the database, which considerably slows down the access to the database.

7.3 Scale-Up of Middleware

A first in depth analysis of the middleware is, how well it scales with multiple instances. It is expected that the throughput increases linearly with the number of instances and the response time decreases linearly. However, as mentioned above a bug was found in the code while performing these benchmarks. This led to the result that increasing the number of middleware instances actually decreased throughput. To fix the bug, two major changes have been made. First, each middleware instance is operating now on its own connection pool and second, the transaction isolation level of `PostgreSQL` has been set to *repeatable read* to guarantee correctness. The latter change has the negative fact that it slows the database access. As depicted (in Figure 7 "Scaling Middleware"), the SQL execution time decreases with increasing instances, but the processing time increases linearly. This leads to the assumption that the database cannot perform more queries per second due to concurrency or there is another problem in the middlware component. Interesting to observe is, that the maximum CPU utilization peek was never over 30% in the scale benchmark (see Figure 8), which indicates, that the server is not fully busy yet. This implies also that the throughput (Figure 7 "Throughput") is not increasing linearly.

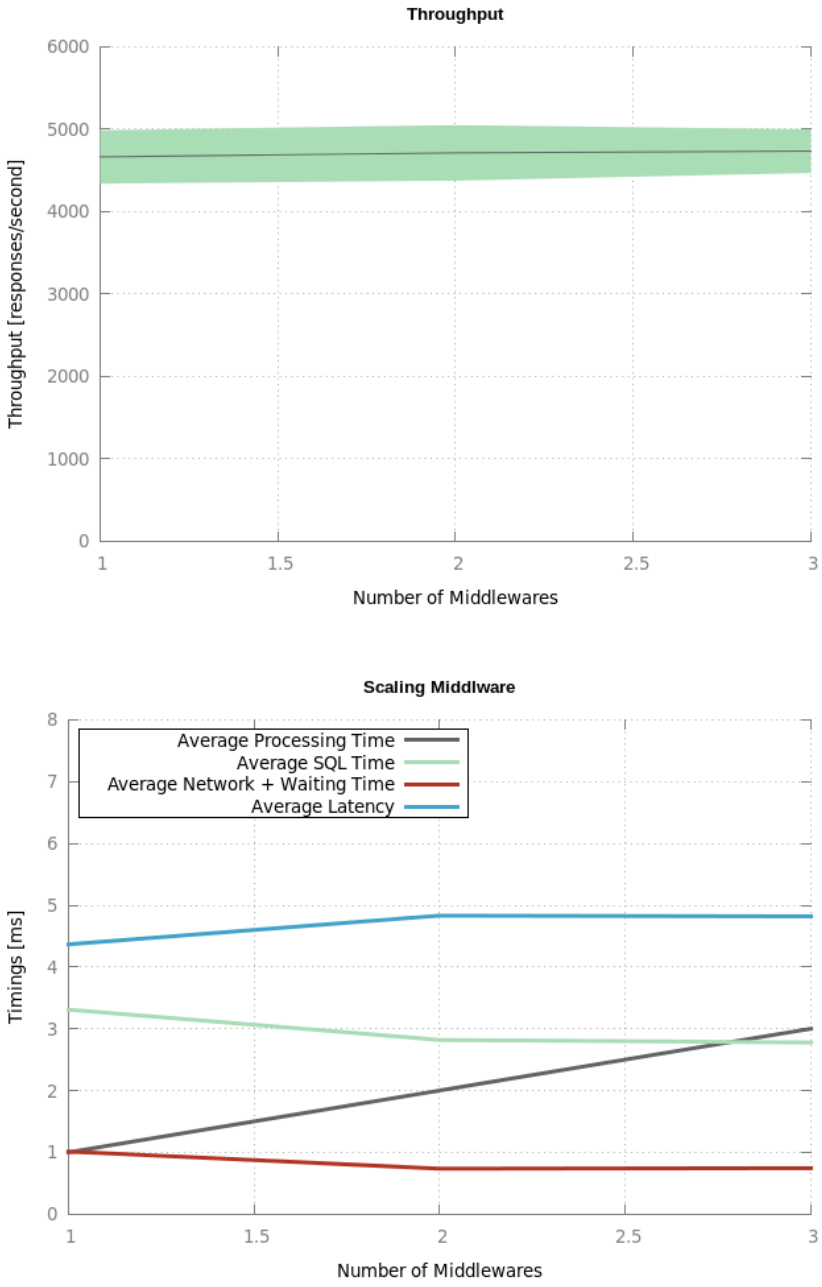


Figure 7: Amazon run - scaling middlewares experiment: *Number of clients* = 100, *requests per seconds* = 50, *number of instances* = 1-3, *number of threads in thread pool* = 50, *duration = 600 seconds*

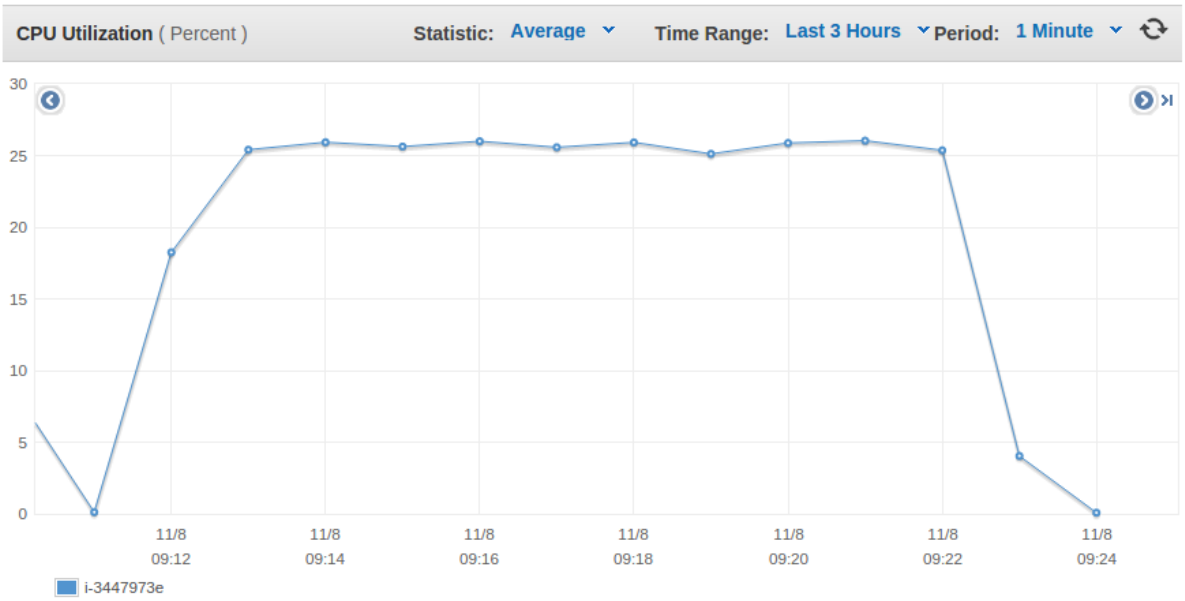


Figure 8: CPU Utilization of the Amazon Server Machine: Running a benchmark with three middleware instances.

7.4 Limits Of The System

Apart from the 2^k factorial design additionally some single factor experiments have been performed to find the limits of the individual components. Unless not explicitly stated the message size is fixed to 200 characters and the number of queues is fixed to 100 for all the subsequent benchmarks.

7.4.1 Limit Of A Client

The maximum throughput of a single client is found by increasing the number of requests a client sends per second. It is assumed that the throughput increases until a maximum has been reached. The response time by contrast should stay constant. The experiment is run with only a single client and increasing number of requests per second every 10 seconds. As it is shown (in Figure 9) the assumptions are correct. The maximum throughput is reached at approximately 520 requests per second. This also corresponds to the constant response time of approximately 1.9ms. It holds that $1/1.9ms \approx 520 \frac{request}{second}$.

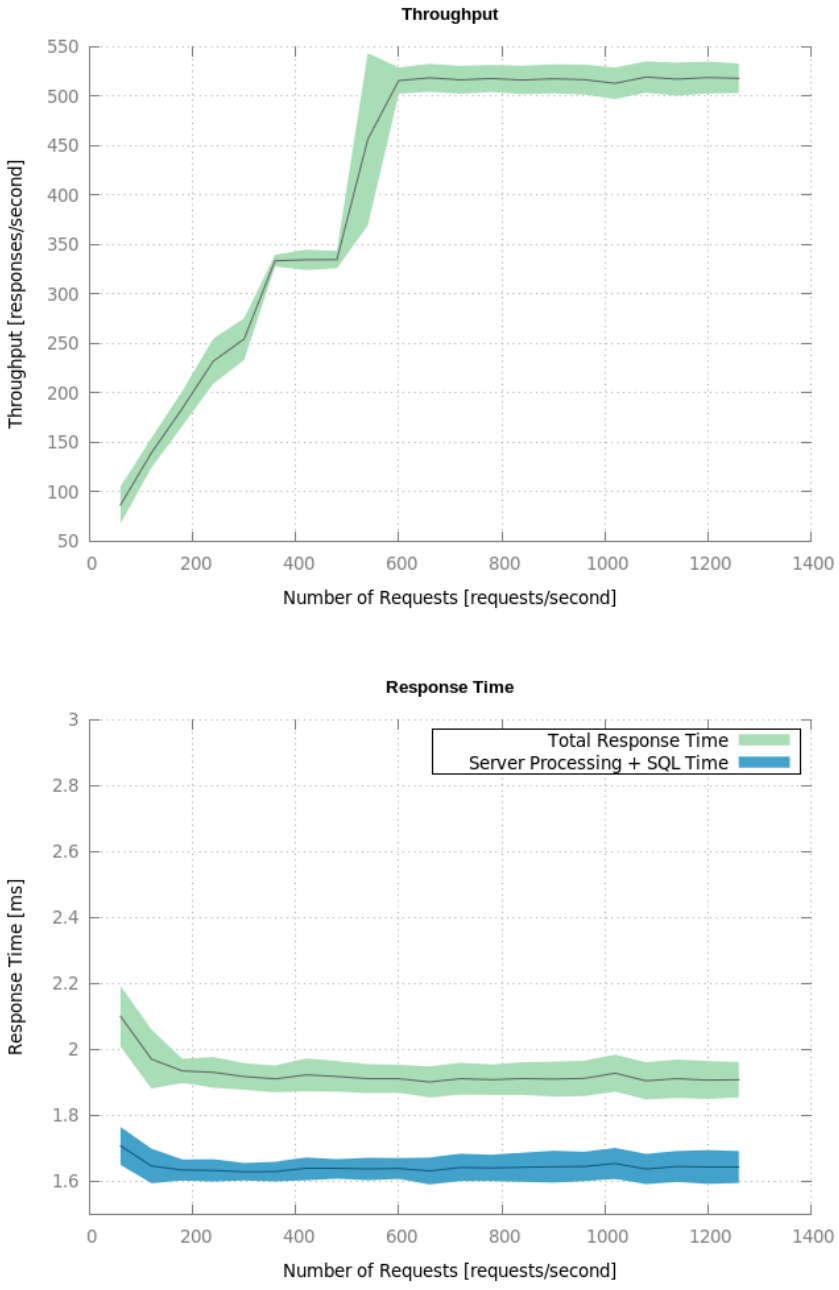


Figure 9: Amazon run - increasing requests per second benchmark: *Number of clients* = 1, *number of instances* = 1, *number of threads in thread pool* = 50

7.4.2 Limit Of The Middlware

Another interesting limitation is how many clients a single middleware instance can handle. For this experiment the number of clients sending requests is increasing, but the number of requests per second is fixed. The expected behavior is that a middleware node can handle up to a certain amount of clients and then the behavior will be unstable. At this point the *Selector* thread is overloaded and cannot handle all requests from the new and the old clients simultaneously. Therefore the response time will increase with the

number of clients connected to the server. As shown below (in Figure 10) this is exactly the case. After 350 clients connect, each sending 20 requests per second the middleware reaches its limit of handling all the requests. This is also visible observing the response time. A nice behavior is that the SQL time does not increase, since the database is decoupled from the server. However, the processing time drastically rises over 10 times the usual response time.

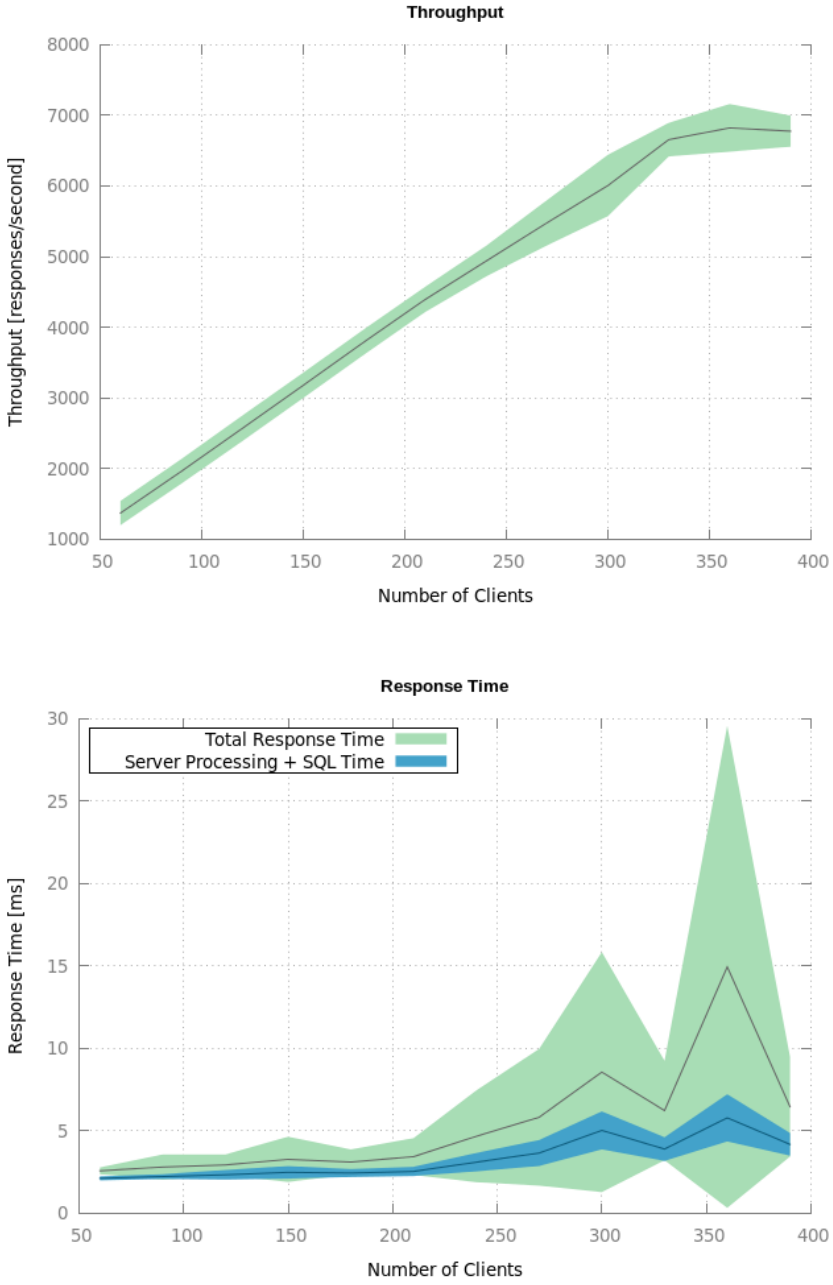


Figure 10: Amazon run - increasing number of clients: *requests per seconds* = 20, *number of instances* = 1, *number of threads in thread pool* = 50

7.4.3 Limit Of The Database

The designed system is capable of sending and receiving messages with different lengths. Focusing on the database it is worth seeing, where the limits of the database are. While the number of clients and the requests sent per second is kept constant, the message size varies. It is assumed that the systems throughput will decrease rapidly once the messages get large and therefore the response time will increase significantly. This turned out to be the case. It is shown (in Figure 11) that the throughput decreases, but not as steep as expected. The response time increases mainly because of the SQL execution time, which increases significantly. This is especially apparent when the message size exceeds 4000 characters. The SQL execution time explodes to almost 1 second.

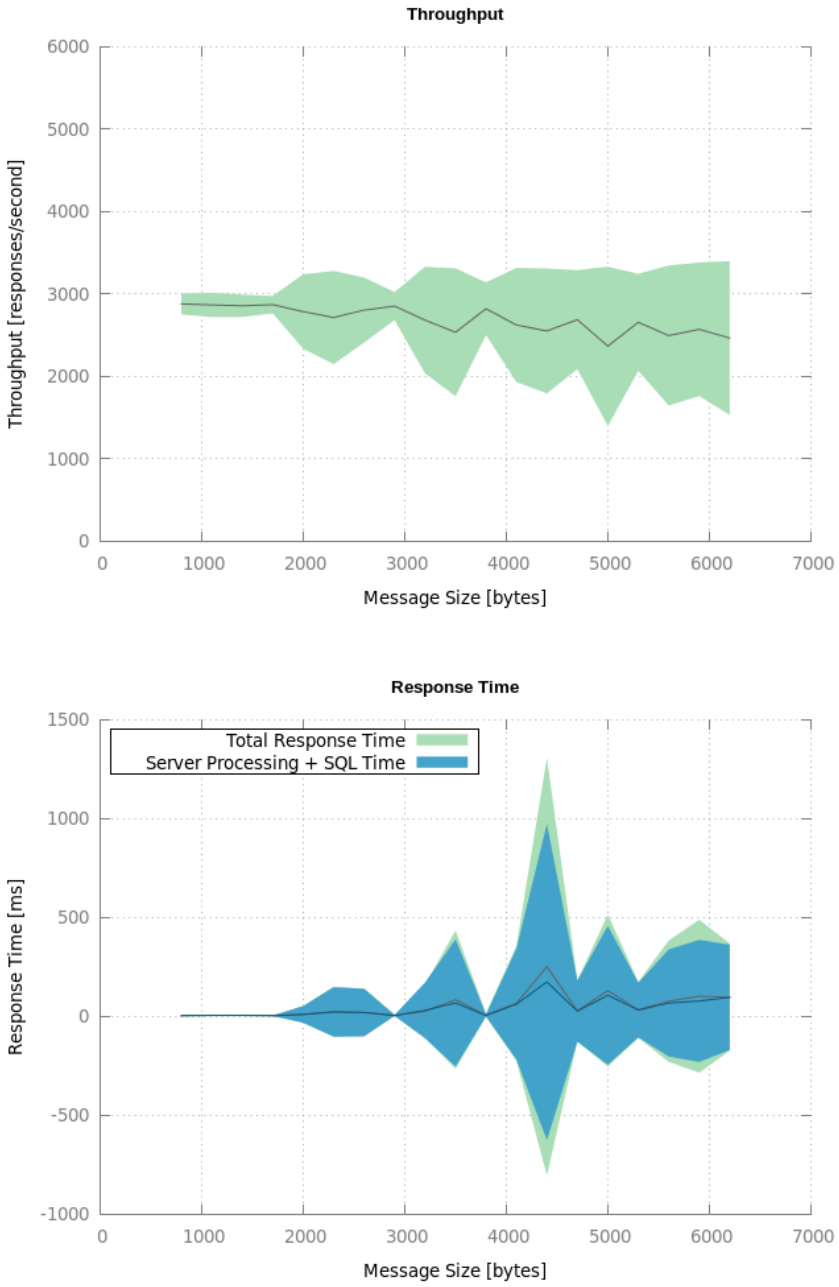


Figure 11: Amazon run - increasing the message size: *Number of clients* = 60, *requests per seconds* = 200, *number of instances* = 1, *duration* = 1200 seconds

7.4.4 Scaling The Thread Pool

Having again a closer look to the middleware, it is interesting to see how changing the size of the thread pool on the server side affects the system. For this examination, a constant amount of clients is used. Each client tries to send a maximum amount of requests per second. By increasing the number of threads in the pool, the number of pooled connections to the database is also increased (linearly). It is expected, that initially the few threads cannot handle all requests, but by increasing the size of the thread pool, the

throughput will increase and the response time will decrease. This is also what happens, when running the experiment as depicted in Figure 12. The throughput increases until there is no benefit of having more threads available. The little drop after having 90 threads available is also because of the 90 opened connections to the database, what slows down the database access. The response time decreases as expected and again it is important that this doesn't affect the SQL time, which is the case here. The maximum throughput of the system is therefore 13000 with 90 threads in the thread pool.

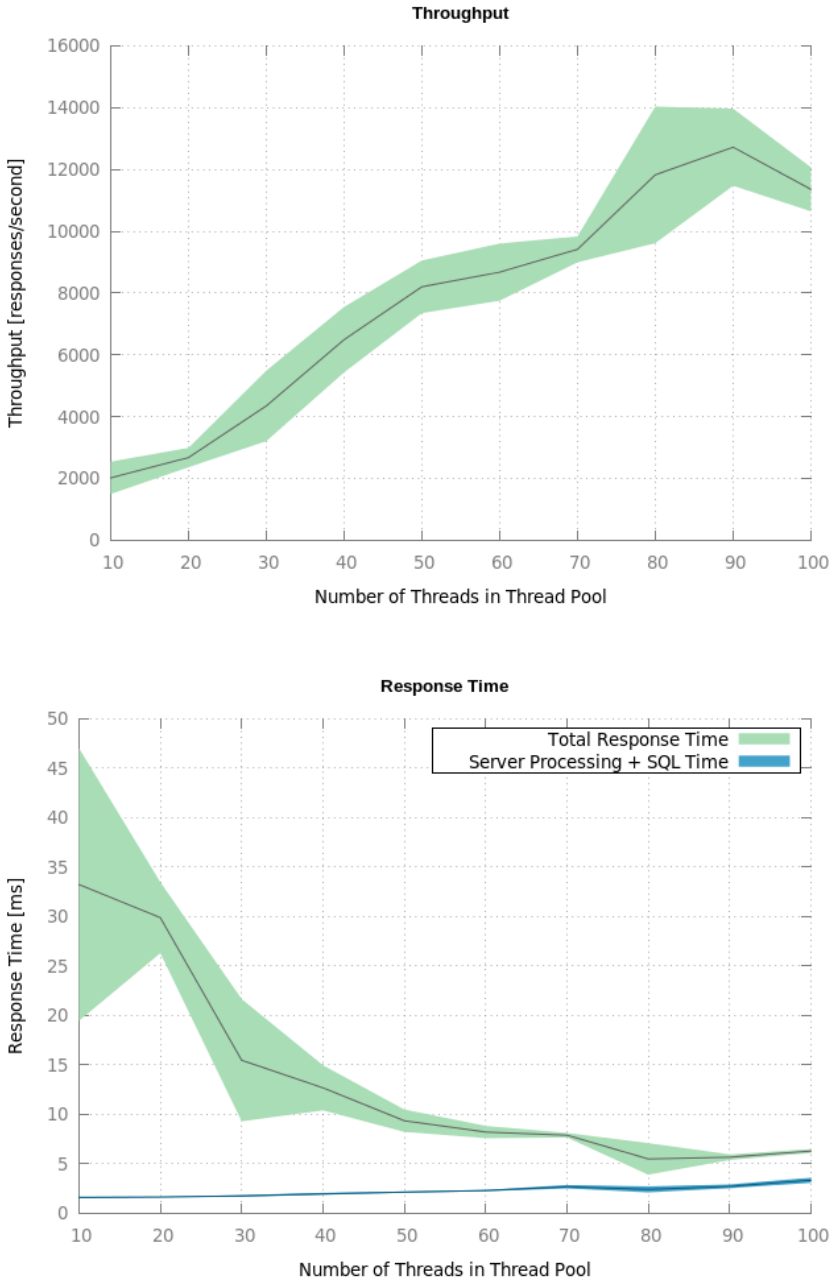


Figure 12: Amazon run - increasing number of threads: *Number of clients* = 80, *requests per seconds* = 200, *number of instances* = 1

8 Analysis Of The Performance Numbers

Thanks to the many experiments run it also possible to state, that the system runs stable and predictable under a well-defined load (Figure 5). The limit benchmarks show the maximum capacities of clients, middlware and database respectively. All these limitations occur at some points as expected. However, as shown in Figure 9 the system can handle an almost infinite number of requests with constant clients, which will be queued in the task queue. The limitation in this case is that, if the requests per seconds increases dramatically, the task queue will not be processed anymore. Furthermore, some secondary factors are found by the limitation benchmarks. Namely, *the number of requests per second*, *the number of clients* and *the message size*. These factors all affect the performance, but it highly depends on the requirement how the system should perform when changing those values.

Concerning scalability in the middleware, the increasing number of threads in the middleware (shown in Figure 12) scale very well. The performance is increased significantly when having more threads in the thread pool - up to a certain number. Also, increasing the number of clients (depicted in Figure 10) is performing well, which are all handled by a single "Selector" thread. However, increasing the number of middlewares (Figure 7) did not show the expected effect. Although, each middleware instance has its own task queue and connection pool to the database, the performance did not increase. It just stayed constant.

8.1 Bottlnecks

From the 2^k factorial design and the above experiments, it is not immediately clear where the bottleneck of the designed system is. It was shown, that the system did not improve performance, when adding more middlwares and hence adding more independent connections to the database. This is most probably due to the isolation level. Changing different PostgreSQL configurations did not lead to the expected improved result. Although, the SQL execution time is decreasing it seems that the number of executed SQL queries concurrently is limited. It was shown (Figure 10) that a single *Selector* thread could handle approximately 350 clients at once. Additionally, the middlware scaled well, when more threads in the thread pool where added (see Figure 9). Based on this insight and the fact, that the CPU of the PostgreSQL server is using much more of its capacity (over 40%) compared to the client and middleware instances, it is assumed that getting a free pooled connection and execute a query concurrently takes its time. Therefore, the *proccesing_time* increases significantly as shown in Figure 7.

This might be improved, by redesigning the way the middleware communicates with the database. An optimal configuration and balance of connections to the database and running threads in the middleware has to be found. The database itself using stored procedures and indexes cannot be significantly improved anymore. The schema is already simplified as well. Regarding the middleware, the serialization and deseralization are already as simple as possible. And the way how the *Selector* thread handles connecting clients and their requests is also as optimized as possible according to the API specifications. It might be worth thinking about multiple *Selector* threads doing the client handling. Referring to the implementation of the `mps.middleware.ConnectionManagement.java`,

it is only twice a `synchronization` necessary, when sending responses back to the clients. There, a `ConcurrentHashMap` is used. In both cases these `synchronization` is not affecting the processing time significantly. Another issue that has to be considered is the concurrent access of the log file. Nevertheless, this was neither a relevant problem for this project.

A major change, which has to be considered if the system were to design anew, is an even more precise tracking of the messages and hence their timings. To fully identify bottlenecks in each component of the system, each processing step has to be analyzed in detail. Consequently, the system would be designed around these specific components to measure the performance more precise.

References

[1] R. Jain. The art of computer systems performance analysis: Techniques for experimental design, measurement, simulation, and modeling. *Book*, 1991.