Advanced Systems Lab (Fall'15) – First Milestone

Name: *Pascal Widmer*

Legi number: 10-*924-439*

Grading

Section	Points
1.1	
1.2	
1.3	
2.1	
2.2	
2.3	
3.1	
3.2	
3.3	
3.4	
3.5	
3.6	
Total	

1. System Description

1.1. Database

For the database PostgreSQL version 9.4 was used running on Windows 10 on an 8-core system with 16GB RAM and an SSD. All SQL statements as well as some performance tests have been tested on Ubuntu 14.04 as well. A master file with all SQL statements concatenated together is executed before the middleware starts up. Besides setting up the database, i.e., dropping and creating tables, custom types, indices and stored procedures it inserts 100 initial clients and queues.

1.1.1. Schema and Indexes

The database consists of three tables. A table called *clients* which keeps track of clients connected to the middleware. A table *messages* and a table *queues* which keep track of existing messages and queues respectively. Clients are identified with a unique and system wide *clientID* which also serves as the primary key of the relation. Even though clients are not authenticated nor can they send requests to join the system the table is still used to keep track of connected clients. All clients are inserted on startup of the middleware. The following gives an overview of the created tables.

```
CREATE TABLE clients (
  client_id SERIAL PRIMARY KEY,
  arrival_time TIMESTAMP DEFAULT NOW()
 );

CREATE TABLE queues (
  queue_id SERIAL PRIMARY KEY,
  creation_time TIMESTAMP DEFAULT NOW(),
  created_by INTEGER NOT NULL,
  CONSTRAINT created_by FOREIGN KEY (created_by)
  REFERENCES clients (client_id)
 );

CREATE TABLE messages(
   message_id serial PRIMARY KEY,
   queue_id integer NOT NULL,
   message text NOT NULL,
   sender_id integer NOT NULL,
   receiver_id integer DEFAULT -1,
   arrival_time TIMESTAMP DEFAULT NOW(),
   CONSTRAINT fk_queue_id FOREIGN KEY (queue_id)
   REFERENCES queues (queue_id)
  ON DELETE CASCADE,
  CONSTRAINT fk_sender_id FOREIGN KEY (sender_id)
  REFERENCES clients (client_id)
);
```

The schema enforces through foreign key constraints that messages have a valid sender, messages belong to an existing queue and queues belong to valid clients. Messages without explicit receiver have a *receiver_id* of -1. This works because receivers are not required to be registered in the system at the time a message for them arrives and as such no foreign key constraint for *receiver_id* exists.

To speed up message lookups the following indices are implemented:

```
CREATE INDEX index_receiverID ON messages(receiver_id);
CREATE INDEX index_senderID ON messages(sender_id);
CREATE INDEX index_queueID_ReceiverID_SenderID_ArrivalTime ON messages(queue_id,
receiver_id, sender_id, arrival_time);
CREATE INDEX index_queueID_ReceiverID_ArrivalTime ON messages(queue_id,
receiver_id, arrival_time);
```

The indices are mainly used in the message lookup procedures *peek_queue*, *peek_queue_with_sender, pop_queue_with_sender, pop_queue* . While indices speed up the lookups for messages they slow down inserts as new tuples have to be inserted in all B-Trees. B-Trees also increase storage requirements. Because the message lookup queries noted above execute multiple SQL queries in sequence to catch all possible error messages making them more efficient is crucial for an average use case, especially when the database gets large. Initially other indices such as on *messages(queue_id,receiver_id)* were implemented, but the index on *messages(queue_id,receiver_id*, *sender_id*, *arrival_time*) can be reused for that purpose. It should also be noted that PostgreSQL version 9.2 and up do not necessarily need to fetch tuples when the index alone answers the query. This partly compensates some of the overhead introduced by having several queries in one function.

1.1.2. Stored Procedures

Stored procedures provide a way to aggregate SQL statements into a single function. This has many advantages. Performance wise the stored procedures allows for more efficient caching on the database side. No lengthy SQL queries need to be sent to the database anymore. In cases where the database runs on a remote server this reduces network traffic. The following stored procedures have been implemented to handle the basic required operations:

```
function create_queue(integer);
function send_message_to_receiver(integer,integer,text,integer);
function send_message_to_any(integer,text,integer);
function query_queues(integer);
function delete_queue(integer,integer);
function peek_queue_with_sender(integer,integer,integer);
function pop_queue(integer,integer);
function pop_queue_with_sender(integer,integer,integer);
function peek_queue(integer,integer);
```

Apart from *query_queues(integer)* all stored procedures are written in *plpgsql*. They return custom types called *error*, *queue*, *message,* which contain error messages, a queue identification and messages respectively. These are in turn parsed by the middleware and an appropriate message object (an instance of the message class) is constructed and forwarded to the corresponding client. This way, apart from special cases, no exceptions are thrown in the middleware, SQL constraints can no longer be violated and there are no *null* results from any query. This comes at the cost of having multiple SQL statements in a single function which lowers throughput and response times.

1.1.3. Design decisions

Because some functions with multiple SQL queries can affect each other a lock on *queue_id* is used to prevent errors. For example, a message on a queue could be read and deleted by *pop_queue()* while *peek_queue()* is running on the same queue. *Peek_queue()* first checks if the queue exists and if a message for the specified receiver is present. Only then it proceeds to read the message at which time it could've already been deleted by *pop_queue()* if the queue was not locked. In addition to the aforementioned constraints the database also returns errors for other cases such as: Request has a clientID not currently in the clients table, empty queues are accessed and non-existent queues are accessed. When clients query a queue for messages and the queue is not empty but has no waiting messages for the client a special type of message called *EmptyMessage* is returned by the middleware.

1.1.1. Performance characteristics

To analyze the throughput and response times between middleware and database a special java class called *MessageGenerator* is used. Its purpose is to empower the middleware to create

database requests without the need for clients. More concretely the MessageGenerator class implements four different methods which create different workloads and thus measure different aspect of the database as well as the network link between middleware and database. The four implemented methods are as follows:

- Method 1 sends 200-character *send* requests to the database.
- Method 2 sends 2000-character *send* requests to the database.
- Method 3 sends *peek* messages to the database.
- Method 4 sends a mixture of 40% *send*, 20% *peek* and 40% *pop* requests.

The main idea is to examine the average use-case of the database in terms of request composition and find a relation between the number of database connections and throughput/response time. Method 2 is used to examine how the database responds to a great amount of large inserts and how the database behaves when growing in size. Method 4 should approximately correspond to a typical load distribution from clients while also ensuring that the database stays roughly constant size. Other request types such as *delete_queue*, *create_queue* and *query_queues* are rare compared to reading messages and sending messages. In an average use-case they would also not contribute to much network traffic hence they are not part of this database performance test. Method 3 and Method 4 insert 10'000 random 200-character UTF-8 encoded messages (no special characters) with random receivers into the message table before sending their main load.

The MessageGenerator inserts all requests into a linked blocking queue, hereafter denoted as *request queue*. This queue is accessed by the database threads which send the requests to the database and store the responses in another linked blocking queue called *response queue*. Usually there would be one response queue for each client but for the MessageGenerator there is only one such queue. Because using a queue potentially introduces additional delay measuring the response time is not done by the MessageGenerator but in the database threads themselves. Throughput, on the other hand, is measured by the MessageGenerator. The request queue always has more than one request ready to ensure that database threads never have to wait for new requests. Of course, parallel access of the queue still introduces some negligible wait time.

Figure 1 shows the throughput achieved in relation to the number of database connections while executing each of the mentioned methods. Note that the middleware is running on Windows 10 on a 6-core CPU connected by a 1 GB/s link to a Windows 10 system with 16 GB RAM running PostrgeSQL version 9.4. Because the throughput seemed a lot lower than expected a separate test was performed where the middleware ran on the same system as the database. The result of the second test is displayed in *Figure 2.* Initial tests also showed huge delays of TCP packets which were caused by a common issue with Nagle's algorithm and delayed acknowledgments in combination with small packet sizes. It was solved by setting the *TcpNoDelay* flag on all network sockets. Throughput is measured over two minutes and the first and last two seconds of each run are discarded.

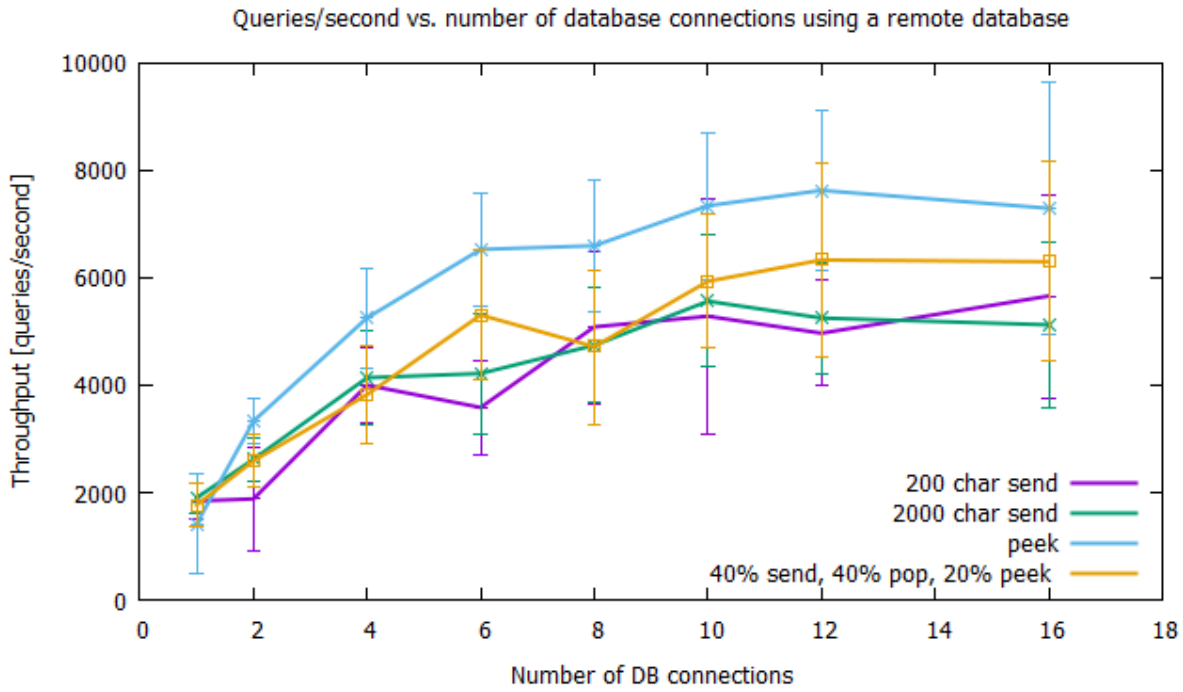

Figure 1

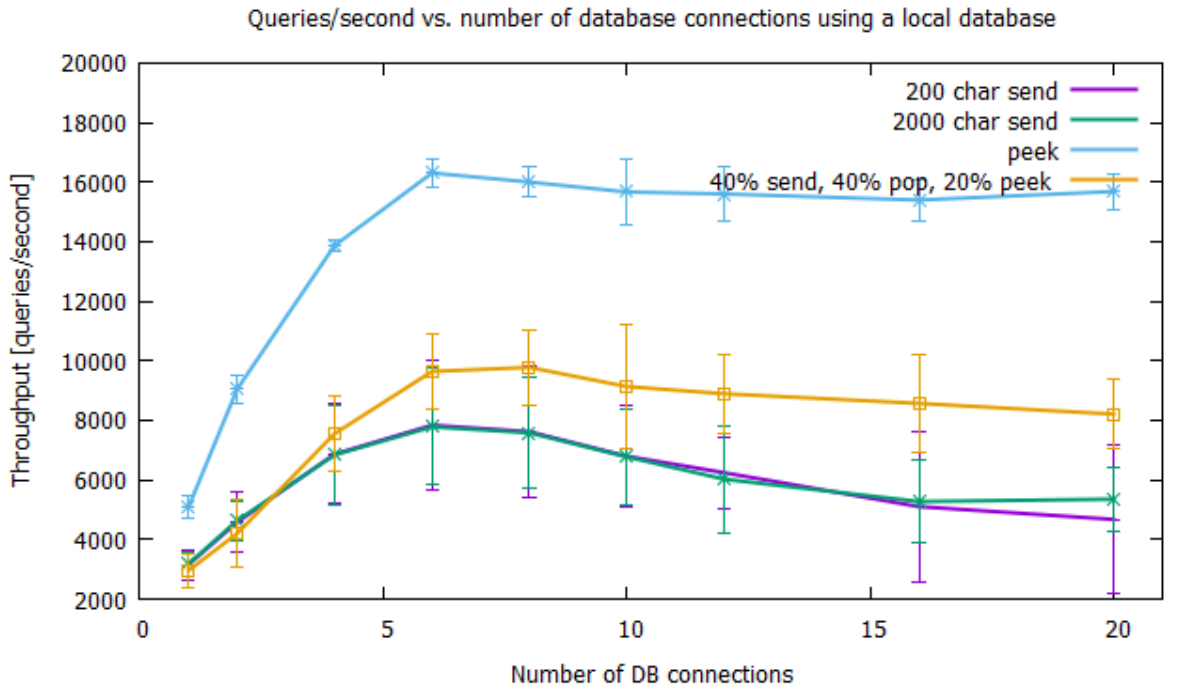

Figure 2

A few things become obvious: A lot of database throughput is lost through the network. This is caused by the RTT and the fact that we always wait for a reply before sending a new request. Secondly, because the database machine spends a lot of time waiting for requests when remotely accessed the optimal amount of connections increases from around 6 to around 12. The message

size proved to be insignificant. A confidence interval of 95% was used. A quick look at the network throughput also verifies that the 100MB/s limit is quickly reached.

In the following test response times and throughput were measured over a period of 2 and 15 minutes respectively. 12 database connections were used as this was shown to be optimal in regards to throughput. *Figure 3* shows the response time in milliseconds. Measurements were done for each request in each of the database threads using the accuracy provided by java's System.nanoTime method. Results were then combined into one-second intervals and the first and last two seconds were discarded.

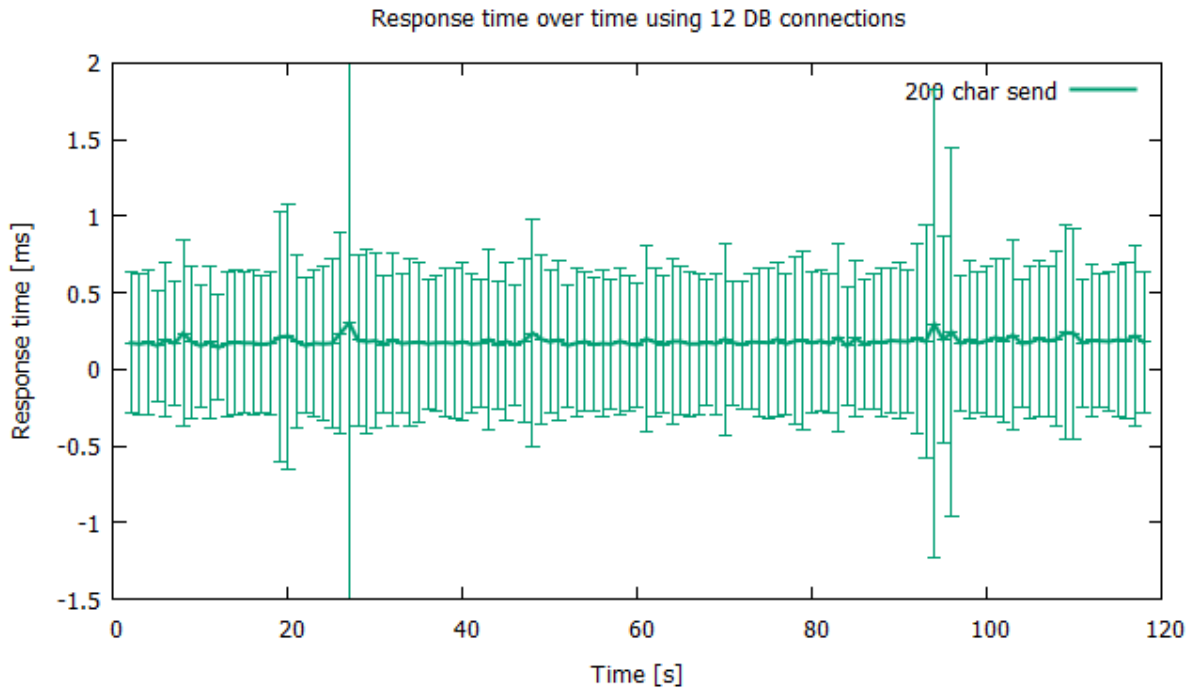

Figure 3

The graph shows very large variance which is partly caused by PostreSQL itself (different indexing time etc.), overhead caused by 12 connections competing for CPU time and IO overhead. The network link increases the average response time from around 0.14 ms to around 0.17 ms, and further increases the perceived variance as can be seen in a separate response time test run. The two response time peaks at around 26 seconds and 95 seconds can be attributed to PostgreSQL flushing dirty pages. The same can be observed in *Figure 4* at relatively regular intervals. Since PostgreSQL needs to update indices for send requests and the messages table continually grows the throughput declines over time.

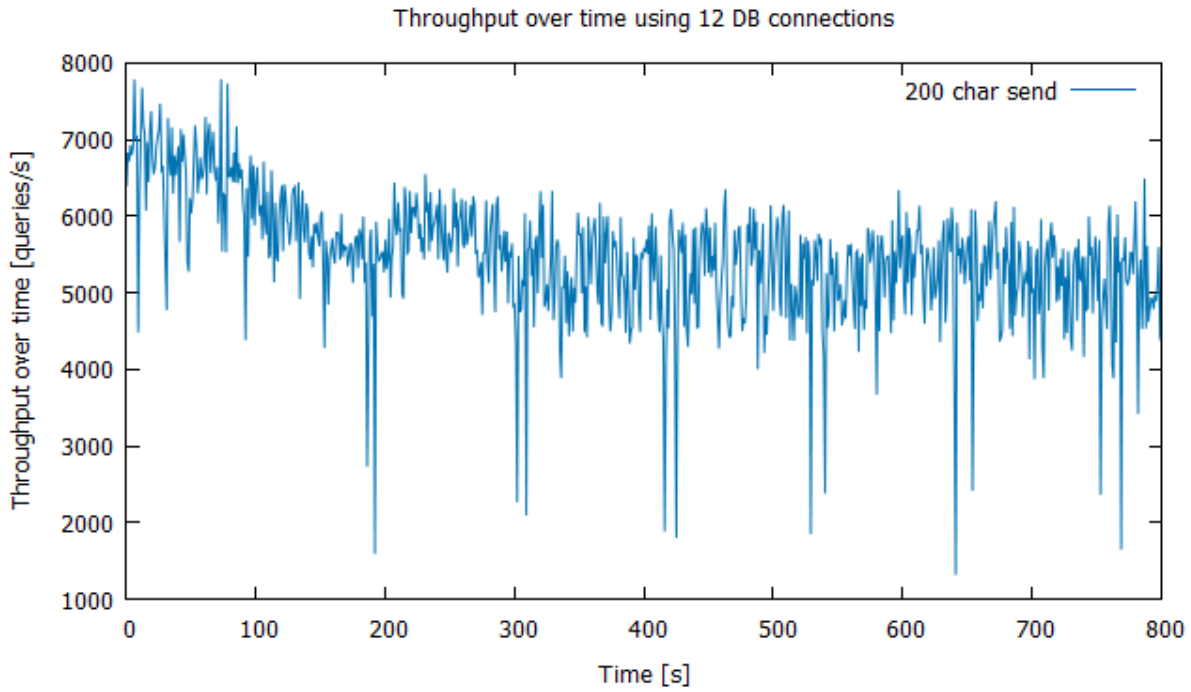

Figure 4

1.2. Middleware

1.2.1. Design overview

The middleware is in charge of communicating with clients as well as the database. In the case of the database a connection pool had to be implemented. For client connections a similar approach was taken. The clients initiate the connection by establishing a connection to the middleware. The middleware listens, accepts connections and passes them on to background threads. Just like database threads the background threads never close the connections. Only clients are able to terminate their sessions by closing the stream. This way overhead of establishing TCP connections is limited.

1.2.2. Interfacing with clients

In my test environment the clients know beforehand where the middleware servers reside, i.e., their IP and ports are passed by command line arguments. When the middleware instances startup they set up the database. This includes inserting the client identification numbers into the clients table and inserting a few initial queue tuples. A middleware server then starts a listening TCP socket and waits for connections. Once a client tries to connect the connection is accepted and passed to a background thread. The server immediately starts listening again. The background thread uses only non-blocking IO. Together with the fact that clients wait for a response from the server before sending a new request it follows that background threads spend most of their time idle. Since modern OS can easily handle thousands of threads so as long as the threads aren't busy this worked out rather well. The background threads themselves are only terminated by interrupts or when the client closes the connection. Interrupts are also used when a client should only run a specified time.

1.2.3. Queuing and Connection pool to database

Each of the background threads that serves a client has access to the global request queue as well as a thread-specific response queue. Database threads read from the request queue, query the database and store their results in the response queue belonging to the respective client that sent the request. This is done by reading the clientID which is part of every message. Since each background thread has only access to the response queue for the client it is serving the response queue there is little contention. Linked blocking queues also neatly implement a wait and notify mechanism which prevent threads from busy-waiting. Response queues either contain 0 or 1 responses, since clients wait for each response. The request queue is implemented as a linked blocking queue and serves as a buffer.

Request messages from clients are put in a *LinkedBlockingQueue* by the background threads. This queue is shared among all clients and messages carry the client identification number to later identify which client sent which request. The database threads continuously check if a new request arrived and if so take it from the queue and start working on it. Once done the database thread ins

1.2.4. Performance characteristics

To check how the middleware performs a special *EchoRequest* message has been crafted. When a client sends an EchoRequest the middleware inserts it to the request queue like any other message. But when a database thread reads the request it immediately answers with an *Echo* message without accessing the database. An Echo message contains the same payload as the EchoRequest.

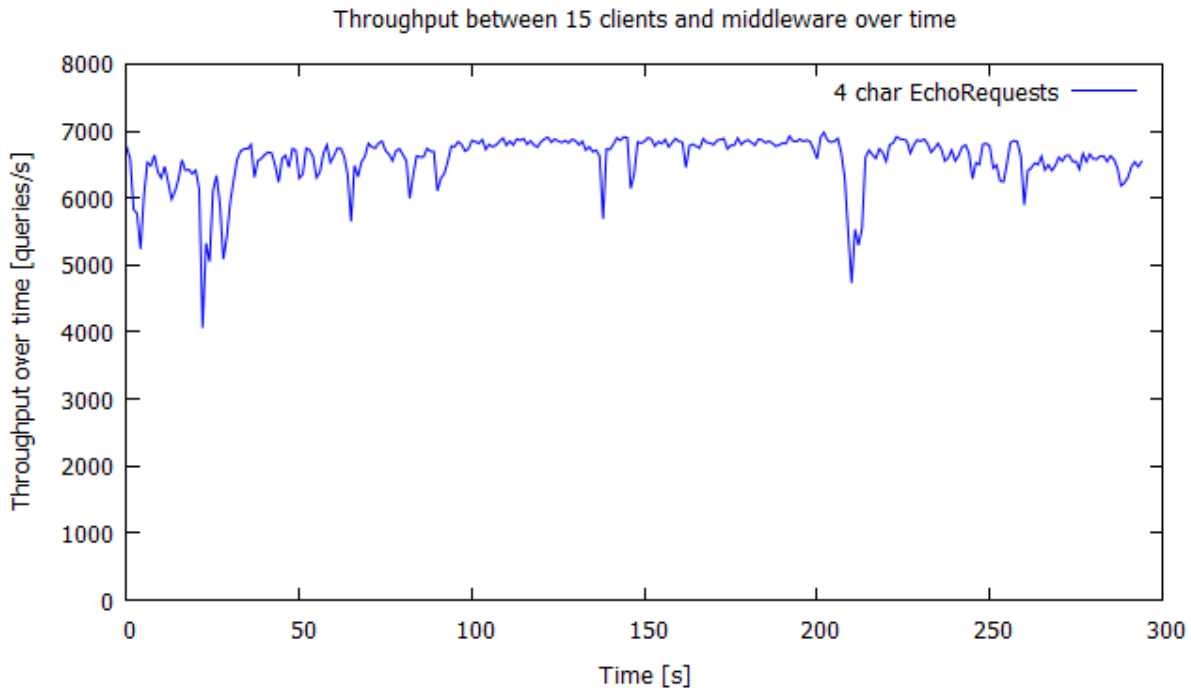

Figure 5

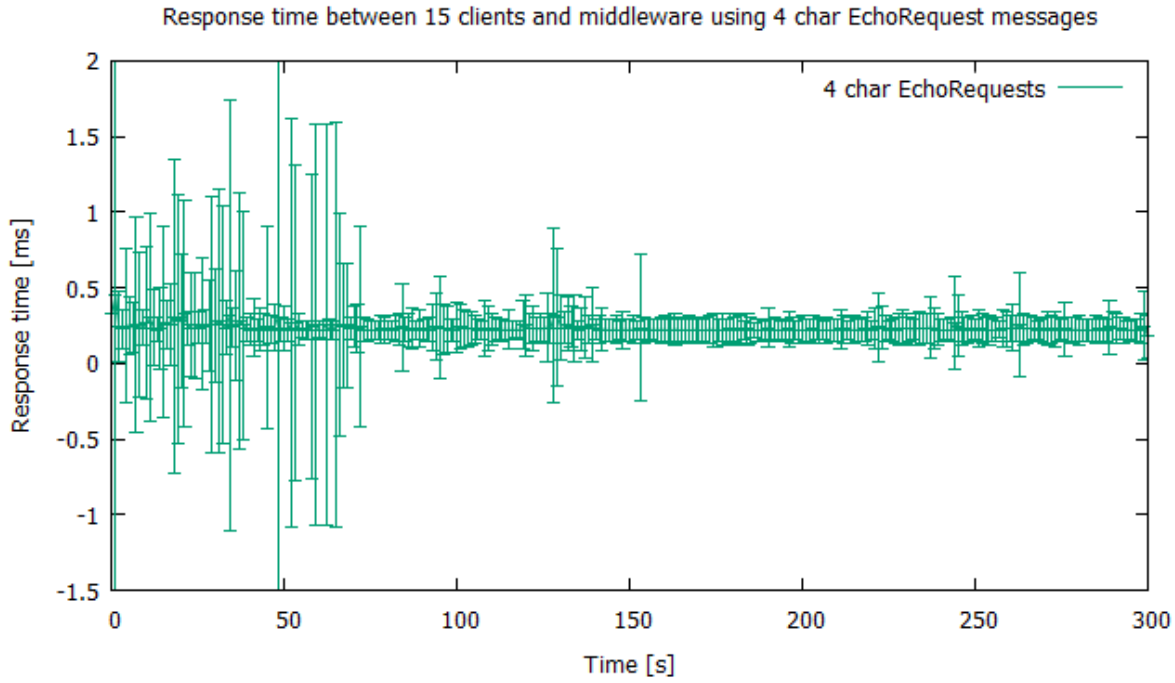

Figure 6

In Figure 5 the throughput is plotted over time sending a 4 character request. In the second graph the mean response time is barely visible but what it better shows is the variance in response times which is quiet high but stabilized stabilize after a while.

1.3. Clients

1.3.1. Design and interface

Clients use blocking IO to communicate with the middleware. Each client is equipped with a send method which sends requests to the middleware using a TCP java socket and waits for a response. Internally both request and response messages are represented as objects of their payload specific classes. For example, there is a class called *CreateQueue*, one called *PopQueue* and so on. The client can differentiate between instances of each class by a unique integer identifier which is always the first thing sent over the wire. Secondly the message size is sent following the message payload if needed.

1.3.2. Instrumentation

Several predefined workloads have been implemented for the purpose of performance testing. This includes a method that sends only send messages, a method sending only peek requests, one which used a mixed load of 40% send, 40% pop and 20% peek requests, another mixed load of 50% send, 45% peek 2%queryQueus, 3% pop messages. Another method sinds only EchoRequest messages, their payload is immediately returned by the database threads as Echo messages. They are used for throughput/response time measurements between clients/middleware. Similar to the middleware a client has the capability to log both errors and events, events consisting mainly of sent requests and received replies. In addition, there is a logger implemented that measures throughput and one that measures response times. Throughput is averaged every second and written to the log file. With only so little writes to the file there is no interference with the actual throughput measured. Response measurements are made by calculating the difference between System.nanoTime before sending/after receiving the message. It also includes the time needed to decode the stream and create a new client-local representation of the message.

1.3.3. Workloads and deployment

A client execution is started with the command line and a handful of arguments are passed specifying, among other things, the middleware's IP/port the predefined workload to be executed, a client identification number and how long client should run. For example:

```
java -jar c.jar 192.168.0.15 6013 15 1 60000 0 0 1 0 4)
```

starts a client connecting to the middleware running on 192.168.0.15 and port 6013. 15 is the number of total clients connected to the middleware and it is only needed so the client knows who else is registered in the system and can read its messages. The remainder are arguments specifying the clientID, how many milliseconds the client should run, which of the 4 logs are enabled and what method should be executed (what should be sent). Multiple clients are started using batch files. When multiple middlewares are used they run on different IPs or ports.

1.3.4. Sanity checks

To check whether requests receive appropriate responses an exhaustive Junit test set with 30 tests and over 60 assertions is implemented. The test requires that 100 clients and 100 queues are inserted in a local database before the test is started. The test proceeds to send every possible message type with every possible argument and checks whether the response is of the right message type and contains the correct payload. For example a request PopQueue(101) is sent where 101 represents the queueID. The test checks whether the returned message is of type Error and if the field errorCode of the Error object is equal to "ERROR_NO_SUCH_QUEUE". All tests are passed. The Junit test is delivered in a standalone unittest.jar file. For it to run a local database called ASL running on port 5432 with user postgres and password 'wurst' must exist. It also needs the tables etc. to exist.

2. Experimental Setup

2.1. System Configurations

For the database PostgreSQL version 9.4 was used running on Windows 10 on an 8-core system with 16GB RAM and an SSD. Two 6-core machines running Windows 10 were used to run middleware instances and clients. The machines are connected through a switch. All machines are capable of 1GB/s network transmission but the switch used to connect all machines does not achieve more than 100Mbit/s/port. A laptop running Ubuntu 14.04 has been used to check if things run on a Unix system.

2.2. Configuration and Deployment mechanisms

The system consists of three separate projects. The middleware (mw.jar), the client (c.jar) and a Junit test (unittest.jar). All of them can be executed from the command line, the middleware and client can also be executed using the predefined .bat scripts instead. The middleware and Junit tests require the database to be setup in advance while the client requires a running middleware. The database is called *ASL* and accessed with user *postgres* and password *wurst*, it was created manually. The client table needs at least as many entries as clients are started afterwards. The queues table does not necessarily need any queues, but the Junit tests as well as the predefined tests expect 100 queues with queueIDs from 1 to 100. Queues require a clientID of the client that created them but since the field is not used it can be set to arbitrary values. Setting up the database tables is done by letting the .bat (or shell script) execute a file called AllinOne.sql. It contains all necessary commands to drop all tables, stored procedures and types and to create new tables, types, indices, and functions. In essence it contains the individual SQL files

concatenated together for convenience. In the end it calls one of the functions, namely *insert_starting_values(_nof_clients INTEGER),* which inserts 100 clients and queues.

The jar files as well as log files are deployed/gathered using a cloud service provider and peer to peer file transfer.

2.3. Logging and Benchmarking mechanisms

Three different logger classes are implemented. An error logger, an event logger and a third logger called deferred logger which uses an instance of the java's StringBuilder class internally. Contrary to the two other logger classes it can buffer and flushes on command. This can be helpful if a hard disk is used for IO because of the high latency. Each logger can be enabled or disabled using command line arguments. In general, the logs are immediately flushed when a new measurement is made. For performance tests the event logs are turned off via command line arguments to ensure correct measurements. The error logger can directly log exceptions and it prepends a timestamp to every line. All logs are written to the user's home directory in a subfolder called logs. The folder is further subdivided in DBThreads, ServerThread, ClientServiceThreads and CLientThreads. Note that some of the error logs contain an "IO Exception" or "Interrupt Exception". This is not an actual error; most threads are terminated by interrupt timers (Interrupt Exception) while background threads communicating with clients close when the client closes the socket (IO Exception).

3. Evaluation

3.1. System Stability

For performing the system stability test I used a composition of 2 x 15 clients connected to 2 separate middleware systems which use send requests with a 10-character payload. The reason behind choosing a 10-character payload is the fact that the network drastically reduces throughput when packet sizes increase. Of three machines one runs the 30 clients, one runs 2 middleware systems and one runs the database. The test is run for a period of 30 minutes. Each client generates a response, throughput, event and error log file. The middleware system uses 6 database connections each, totaling the optimal connection count of 12 found in the database performance analysis. The database is setup with 100 initial queues and clients. The database immediately grows in size because more send messages are sent than pop messages. A throughput lower than the optimal throughput using 12 database connections is expected, along with an increase in response time caused by additional network load and context switching. The throughput and response time are expected to decline over time. Of course, the system is expected not to crash.

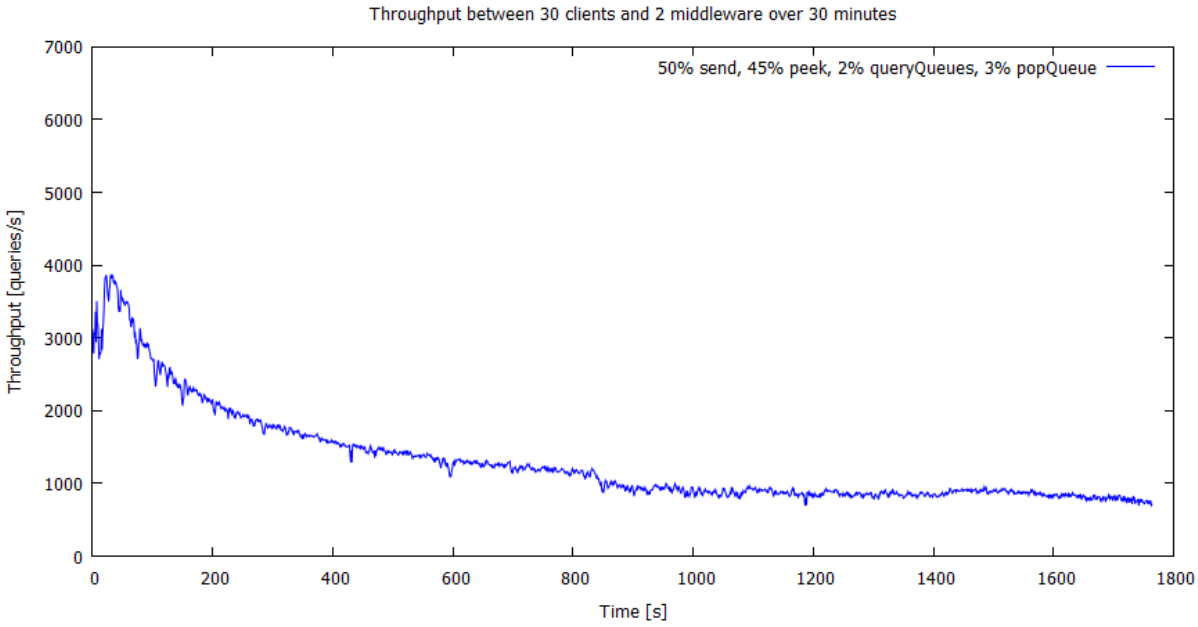

Figure 7

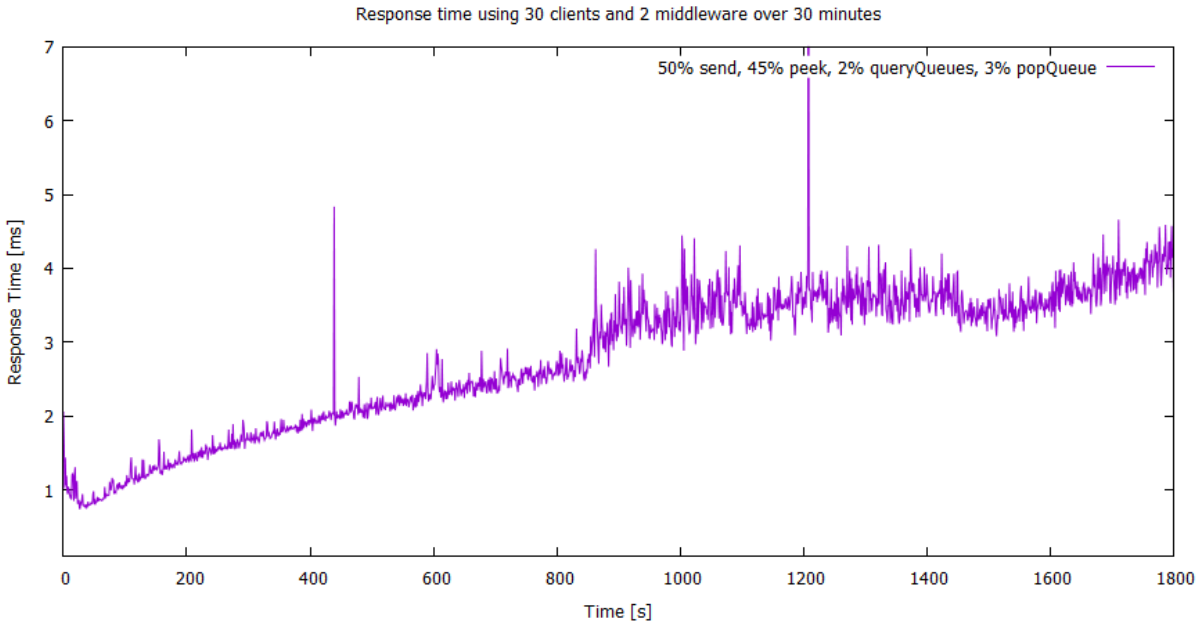

Figure 8

Figure 7 and Figure 8 show the Throughput and Response time respectively. The high peaks in the response time graph could be that PostgreSQL flushes some data. Other than that it might also be caused by dropped packets that get resent after a while. The full logs including Events/Errors/Response/Throughput from the two middleware as well as of all 30 clients can be found in RawLogs/SystemStability30Min. The data used for plotting can be found (including 95% confidence intervals) in ProcessedLogs/ThroughputClientMW30MinMixedLoad.txt and ProcessedLogs/ResponseClientDB12Cons30Clients30MinStability.txt.

1.1. System Throughput

The maximum throughput depends not only on the optimal configuration but also on the type of load the system is subjected to. As in similar tests I assume a typical workload consists of 50%

send, 45% peek, 2% queryQueues and 3% pop requests. The send messages have a receiver of -1, i.e., anyone can read the message. Because of the infrastructure used in my tests the limiting factor is the network transmission. As such the maximum throughput in terms of queries per second could be achieved by finding the smallest request in terms of bytes. Since the workload is already defined we are left with optimizing the number of clients used. The optimal number of database connections has already been found to be around 12, albeit the parameters are not completely independent it is assumed to be constant here. The maximum throughput is approximately 4500 queries/second using 12 DB connections and 10 clients.

1.2. System Scalability

In the system scalability test I focused on testing the middleware and client code in particular. The network has already been identified as rather big bottleneck, and if it wasn't for the network the PostgreSQL server would presumably limit the throughput. Here we test this hypothesis by running a middleware instance and with a variable amount of clients on a local machine. Only 4 character EchoRequest messages are sent. Later message sizes could be increased and the systems memory access speed and the total required memory might come into play. But initially we are more interested in how many threads interact, especially in regards to the request queue which is accessed by the database as well as threads handling clients. We are mostly left with threads competing for CPU time and the queue, the network as well as the database is removed from the equation.

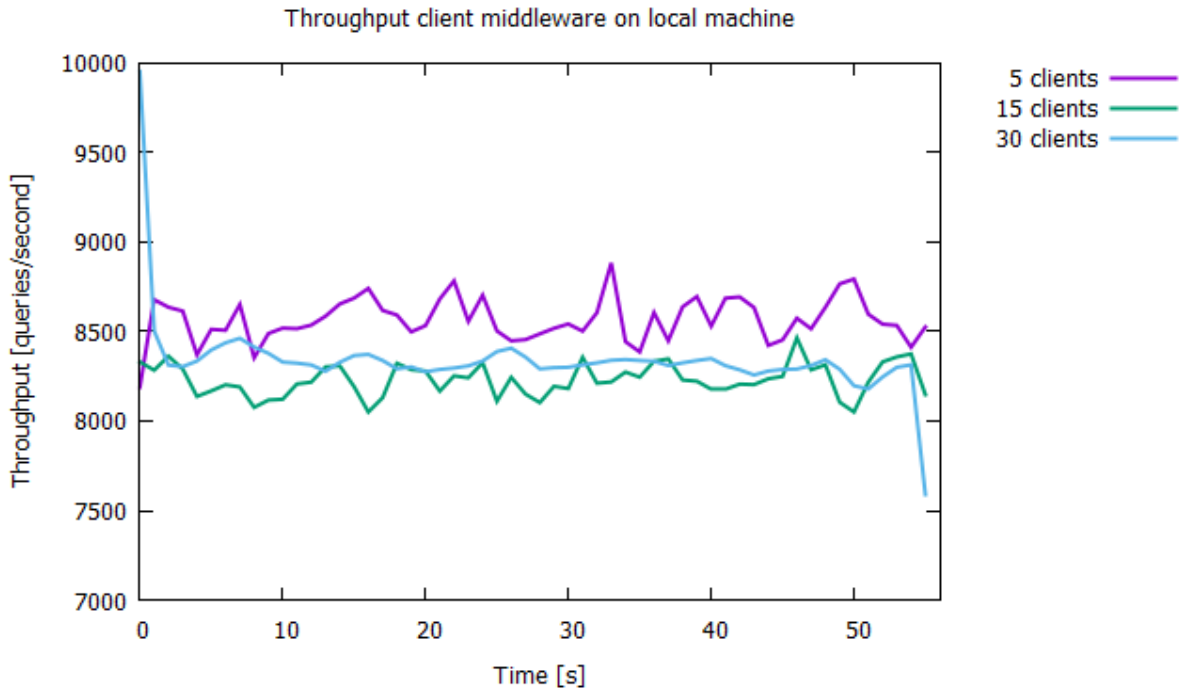

Figure 9

The graph shows that the throughput without networking and database still has high variance with an average of around 8250 queries/s for 15 clients. Increasing the amount of clients further has minimal effect on throughput. The test was executed using 4-character echo requests on a 8-core system, the middleware runs next to the clients and 12 DB connection threads work on the request queue. Logs can be found in logs/SystemScalability/

1.3. Response Time Variations

For finding the relation between response times and number of clients the number of clients was varied between 1 and 100. A send message of 4 character was sent to the database. All of the clients are running on one system connecting to a single middleware instance.

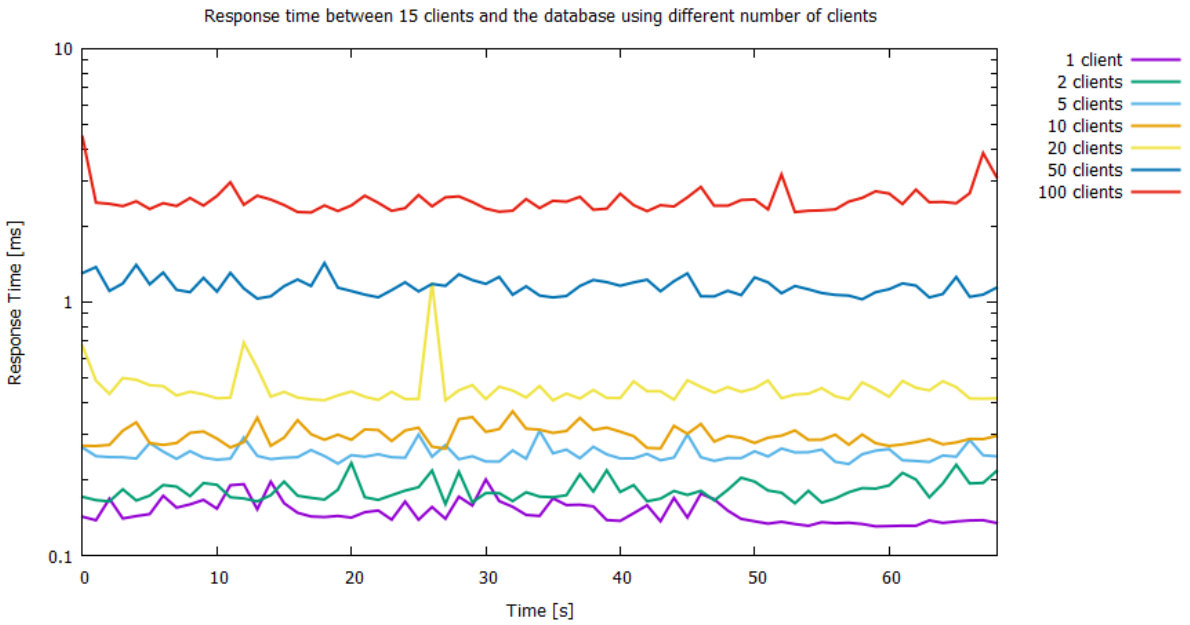

Figure 10

The graph shows a clear picture, when the number of clients increase the response times drastically decrease (note the logarithmic axis). What is not shown in the graph are the confidence intervals although they are calculated. Plotting them proved difficult because the variance is extremely high. For 100 clients the 95% confidence moves between 0 and 5ms while a lower number of clients also lowers the variance accordingly. The increase in response time with a higher client count is explained by the network itself. The link is shared between all clients so clients wait longer to get access to the medium.

Similarly, an increase in message size increases the response times. This is to be expected because sending a larger packet over the network takes trivially longer and congestion naturally increases response time variance even higher. Messages of 2000 characters, which corresponds to more than 2000 bytes with overhead and UTF-8 encoding, yield a 95% confidence interval of -20ms to 70ms with an average response time of 25ms.

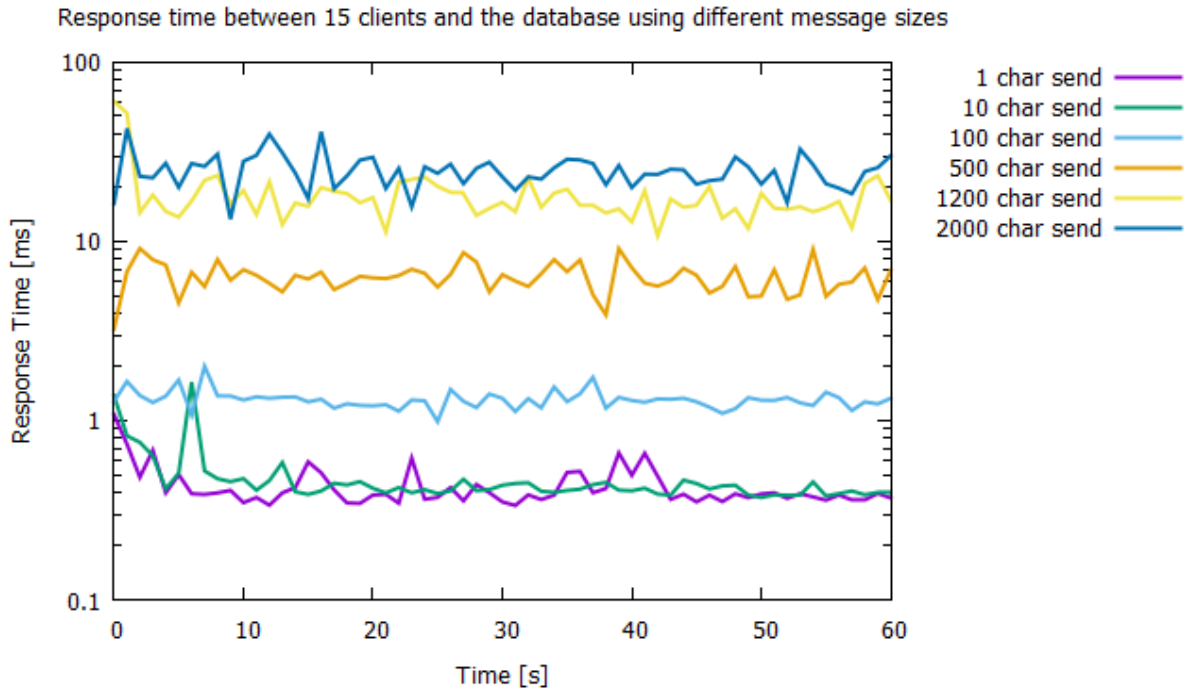

Figure 11

1.4. 2^k Experiment

1.5. Conclusion

The system performed reasonably well and I would not change the overall design. Having a separate thread for each client that keeps the connection open proved very efficient, in the current configuration one middleware can easily handle all clients by itself. In the future the background threads serving the clients could easily be remodeled to handle messages asynchronously. This can be achieved by using a special message identifier sent by the client with which the client can later. However a few things could be optimized. Using a linked blocking queue for response messages might not be the best choice. Considering that the size is constant and only two threads access it there might be simpler and/or more efficient data structures. In regards to PostgreSQL I expect there to be some settings to be optimized further increasing response times and/or throughput. Of course the settings depend on what whole system is used for eventually, e.g., how frequent are inserts compared to reads and how large are individual messages. Some implemented stored procedures might not need all implemented SQL queries and could be optimized as well.

Because in the end it turned out the network itself caused the greatest performance hit in the setup I used, it would certainly help if network packets were smaller. However, this is hard to achieve since TCP carries a lot of overhead by itself. Using UDP might be an option, especially in combination with asynchronous messages where not every packet needs an immediate answer or acknowledgment.