Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Paštas: The Distributed Messaging System

Advanced Systems Lab, Milestone 1 Report

Martynas Pumputis

November 09, 2014

Advisor: Zsolt István

Department of Computer Science, ETH Zürich

Contents

Chapter 1

Introduction

This is a report for Milestone 1 of Advanced System Lab course project work. The report presents a distributed messaging system `Paštas` (”Post” in Lithuanian language) which serves for a purpose of an extensive performance evaluation and analysis.

The main functionality of the system is to handle a *client* requests. The requests include:

- Creating or deleting account of client.
- Creating or deleting a *queue*.
- Sending a `message` to queue. The message is either broadcast or it has explicitly set receiving client.
- Reading a message from any queue.
- Querying queues for a message from a particular client.
- Searching for queues which contain a message for a client.

1.1 Overview

In Chapter 2 we introduce our storage layer. Followed that, Chapter 3 presents the core of the system. Afterwards, in Chapter 4 we talk about client library. Next, Chapter 5 describes the general design of the system. Later on, Chapter 6 presents the results of experiments we conducted. Finally, in Chapter 7 we draw our conclusions.

Chapter 2

Database

This section describes a storage layer of the messaging system Paštas.

2.1 Schema

The system operates with **user**, **queue** and **message** entities. Each entity has a set of properties and relations, and is stored in a separate table. Schemas of the tables are depicted in Tables 2.1, 2.2 and 2.3.

Field	Type	Constraints	Details
id	SERIAL	PRIMARY KEY	Unique identifier of a user
name	VARCHAR(255)	UNIQUE	User-friendly name of the user
created_at	TIMESTAMP	n/a	Timestamp of event when the user was created

Figure 2.1: users table schema

Field	Type	Constraints	Details
id	SERIAL	PRIMARY KEY	Unique identifier of a queue
name	VARCHAR(255)	UNIQUE	User-friendly name of the queue
created_at	TIMESTAMP	n/a	Timestamp of event when the queue was created

Figure 2.2: queues table schema

Field	Type	Constraints	Details
id	SERIAL	PRIMARY KEY	Unique identifier of a message
queue_id	INTEGER	REFERENCES queues (id) ON DELETE CASCADE	A reference to queue which the message belongs to
send_id	INTEGER	REFERENCES users (id)	The message's sender id
recv_id	INTEGER	NULL REFERENCES users(id)	The message's receiver id. If it is broadcast, then the field is set to NULL
content	VARCHAR (2000)	n/a	Message content
created_at	TIMESTAMP	n/a	Timestamp of event when the message entry was created

Figure 2.3: messages table schema

The presence of explicit references in the schema allows us to detect application logic inconsistencies early, although avoidance of them would have a benefit in performance.

Each entity has an unique identifier which is generated by the database (SERIAL type). It is worth to mention that by default SERIAL does not allow the identifier to wrap around, thus the identifier is monotonically increasing.

2.2 Indexes

Most of the requests to our system results in searching in the database based on multiple parameters. Therefore, the usage of *mult-column* indexes could make some of them faster.

In our database schema design we use the following indexes:

- All primary keys are by default indexed. It includes `queues(id)`, `users(id)`, `messages(id)`.
- `messages(recv_id, queue_id)`. It is used for searching queues which contain a message for the given receiver.
- `messages(queue_id, recv_id, id)`. The index is used when searching for the oldest message in the queue. The message might be of broadcast type (receiver id is `NULL`) or dedicated to a client (receiver id is set).
- `messages(send_id, recv_id, id)`. Used to find a message dedicated to the given client and sent from the given user.

All indexes are of B-Tree type. The other viable solution would be to use Hash type indexes. However, the usage of them is discouraged in concurrent applications.

It is important to node, that the order of columns in index definition is important. Otherwise, the database query execution planner might not be able to use an index, although it contains all necessary fields and might be more faster in that situation.

During the development phase, the indexes were carefully chosen. We ran multiple micro benchmarks to evaluate the best combination for particular queries. The made decisions are based on information gathered from explained execution of queries which were run during the benchmarks.

An interesting finding was that PostgreSQL 9.3 is treating a `NULL` not equally as any other field value. For example, the query to find the oldest broadcast message might do `Bitmap Heap Scan` instead of `Index Scan` and therefore have a lower performance. To overcome it (and PostgreSQL's query planner) we could use `'0'` value instead of `NULL` for broadcast messages, but it would burden the data model. Consequently, we decided not to optimize this particular query.

2.3 Interface

Each of our system's database query is implemented as a stored procedure with PL/pgSQL language. The advantaged of such approach, that each procedure is stored in database and is executed in the same way as PREPARED statement, thus extra overhead for preparing a query is avoided. Also, a query plans of stored procedures in PL/pgSQL language are cached. However, the usefulness of this feature might be debatable, because the database might avoid an opportunity to update an execution plan. As an alternative solution is to use dynamic EXECUTE. Unfortunately, due to time constraints we could not evaluate whether it suits our needs better. Therefore, we excluded it from our implementation.

The supported query procedures are listed in Table 2.4.

Each stored procedure which executes a statement containing a notion of the queue id might raise SQL level exception if such queue does not exist.

The stored procedures are executed in a scope of transaction at database level, thus atomicity is guaranteed.

2.4 Design Decisions

Our initial hypothesis based on previous experience with large-scale systems was that the database will be the bottleneck of our system. For this reason, we put an extra effort to make the database as fast as possible.

One of the consequences of such decision was to not guarantee strong data consistency. We use READ COMMITED transactions isolation level which might return stale records. As a solution, one might use SELECT ... FOR UPDATE statement. In our case, when having multiple clients reading from the same queue, it could result in lock-contention. Also, vacuuming of the database would have a longer delays because of locking. Having that, we would have to pay an extra price in terms of performance. Therefore, the trade-off is that stored procedures might return already deleted messages in a case of two concurrent read requests.

As a next thing, in applications which frequently execute UPDATE and DELETE statements, Dead Rows might cause a lot of performance problems due to memory fragmentation. To avoid that, we could avoid deleting a messages by having a flag which would denote whether the message has been removed. In that case, we would need to use the partial indexes. Trying this approach did not give us any increase in performance. The Dead Rows count decreased and thereby, less vacuuming occurred which resulted in more stable response time. However, quite often, query execution plans con-

tained full table scans when index based access could be faster. Therefore, we decided not to use partial indexes and delete messages instead.

As a part of the course requirements, the messaging system should persist queues, user accounts and messages in `PostgreSQL` database. We chose the latest stable version of the database which is `9.3`. Because of the persistency requirement, the database has configured with enabled `fsync`. Thus, after each `COMMIT`, data is written to disk in a form of `Write-Ahead Log`.

2.5 Performance characteristics

To verify our hypothesis and to get a better insight into performance of our system, we decided to do the database performance benchmarks first. For benchmarks we used a custom system containing of multiple Java threads. Each thread had an open connection to the database and was calling randomly chosen stored procedures of our messaging system. The next call by the thread was issued only after the database has responded to the previous call. Thus we did the closed-system experiment.

After each test run we re-initialized the database, i.e. each test case was running from the clean state. For more detailed description of experimental setup and constraints please refer to later sections.

To evaluate how the database scales out, we run multiple benchmarks. For each benchmark we increased the number of clients. Table 2.5 shows the configuration parameters of the benchmark and Table 2.6 shows probabilities for each request. Due to high message removal probability, the database size was roughly the same during the execution.

Parameter	Value
Common	
Provider	Amazon AWS EC2
Guest OS	Ubuntu 14.04 LTS
Virtualization	PV
Zone	eu-west-1b
Duration of single test case	5 minutes
Client	
Machine	m3.medium
Number of machines	1
Database	
Machine	m3.large
DB	PostgreSQL 9.3

Figure 2.5: Configuration for database scalability test

Parameter	Value
Number of queues	2
insert_message	
Probability for request	0.2
Probability for 200 and 2000 byte message	0.5, 0.5
Probability for broadcast message	0.3
read_message	
Probability for request	0.2
Probability for removing message	0.9
read_message_broadcast	
Probability for request	0.2
Probability for removing message	0.9
query_message	
Probability for request	0.2
get_queues	
Probability for request	0.2

Figure 2.6: Various parameters for each request type

Figure 2.7 shows the throughput when the number of client connection has been increasing, while Figure 2.8 depicts a response time. It can be seen that the database performance peaks when we reach around **10** clients. After

this threshold, the performance starts to drop because the database gets overloaded. Also, it impacted the response time which started to grow more faster after 10 connections.

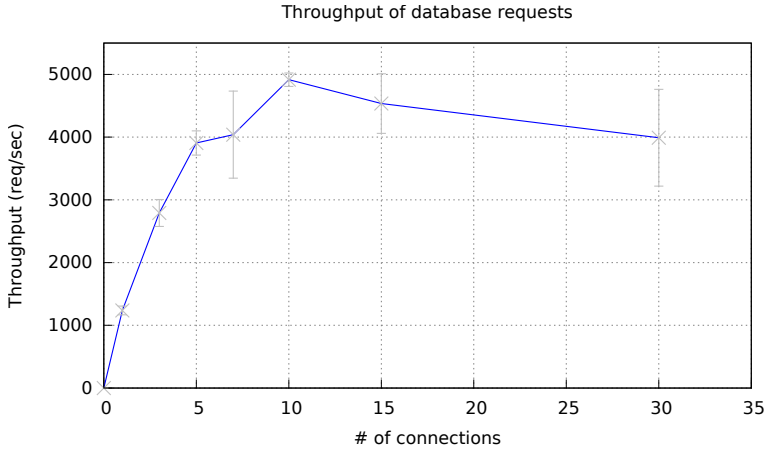

Figure 2.7: Throughput of the database when increasing number clients. Averaged over one minute interval.

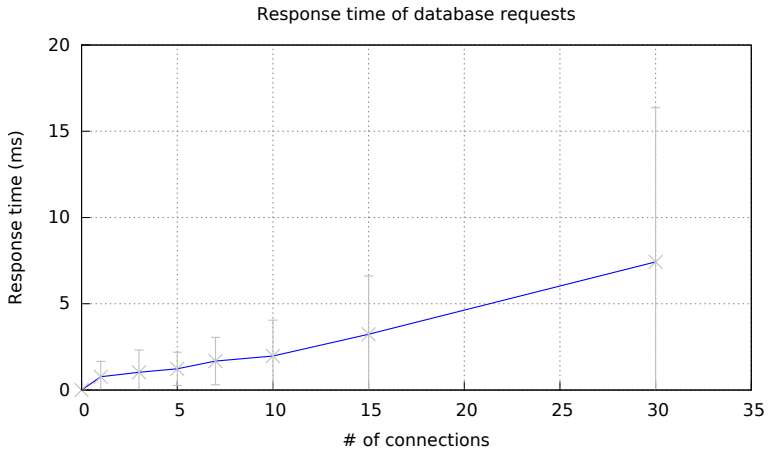

Figure 2.8: Response time of the database when increasing number clients. Averaged over one minute interval.

Procedure	Input	Output	Details
insert_queue	name varchar (255)	q_id integer	Creates a queue
delete_queue	queue_id integer	void	Removes the queue with its messages
get_queue_id	queue_name character varying	q_id integer	Lookup for a queue id
get_queues	client_id int	TABLE (queue_id integer)	Returns queues containing messages for the given client
get_queues	void	TABLE (queue_id integer)	Returns list of queue containing broadcast messages
insert_user	name character varying	u_id integer	Create a new client
insert_message	queue_id int, send_id int, recv_id int, content character varying	m_id int	Inserts a message. In a case of broadcast message, receiver id is NULL
read_message	queue_id int, client_id int, remove boolean	m_id int, send_id int, recv_id int, created_at timestamp, content character varying	Reads the oldest message from the given queue. If remove is set, deletes the message
query_message	client_id int, sender_id int,	m_id int, send_id int, recv_id int, created_at timestamp, content	Searches for a message which is sent by the given client
read_message broadcast	queue_id int, remove boolean	m_id int, send_id int, recv_id int, created_at timestamp, content	Reads the oldest broadcast message from the given queue

Figure 2.4: Stored procedures used by the messaging system

Chapter 3

Middleware

This section describes a middleware - the core of the messaging system. The functionality of the middleware is located inside **ch.ethz.pastas.middleware** package.

3.1 Design

The messaging system consists of a few components. Each component has its own responsibilities. The main idea behind the middleware is to accept incoming requests from clients, process them and respond back with a response message.

Because of the requirements, the middleware has to support multiple concurrent client connections and use the multi-threading inside. Also, the number of threads has to be fixed. Thus, it implies a usage of some queuing mechanism.

As it can be seen from Figure 3.1, the middleware consists of three components:

- `RequestDispatcher` is responsible for accepting client connections and queuing them
- `WorkQueue` is a queue type container which supports concurrent reading is a place where the requests from clients are queued.
- `RequestHandler` threads reads client requests from the queue, executes them by calling the database and responds back to the clients.

3.2 Request Dispatching

RequestDispatcher is accepting incoming TCP connections for a client. Also, its functionality is to read and serialize a request message. Because there

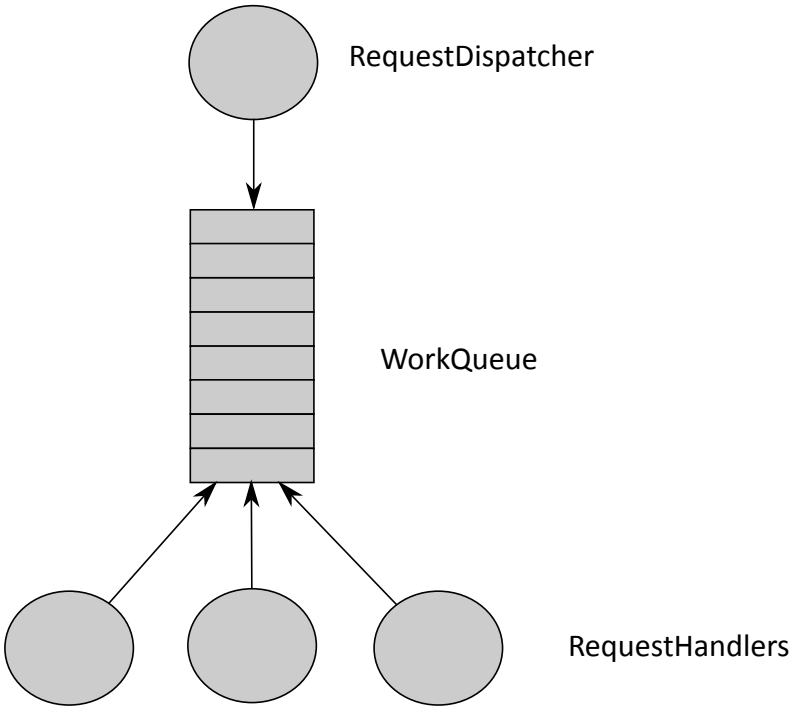

Figure 3.1: The design of the middleware component

is only single RequestDispatcher per one instance of the middleware and TCP is streaming protocol which means that a receiver might not receive the whole packet at once. , we should avoid blocking when waiting and reading a request message. Instead, RequestDispatcher should get a notification when there are an event in a socket channel. To do so, we make all sockets non-blockable and we use Java Non Blocking IO which is based on selectors. The selector is OS dependent, but in our case `epoll` is used.

Due to packets fragmentation, RequestDispatcher has to keep track of partially received packets. It is implemented by attaching the received part to the socket object. Because the request header contains information on packet size, RequestDispatcher is able to detect when the whole request has been received.

After receiving the request, RequestDispatcher deserializes it to Packet, destroys selector associated with socket to the client and finally pushes it to the queue.

It is important to note, that sending back a response to the client is done by RequestHandler.

3.3 Connection Pool

To avoid establishing a new connection to the database when a client request arrives, we use a database connection pool. We chose `PGPoolingDataSource` for implementation, because it fulfils our basic needs to have a pool of the opened connections and to have thread-safe access to such pool.

Because connections are shared and limited resource, therefore it should be used for as short as possible duration and only when it is needed. Thereby, after RequestHandler starts executing a request, it takes the database connection from the pool, executes the stored procedure and returns immediately back the connection to the pool.

3.4 Connections to Clients

The clients are running not in the same Java Virtual Machine as the middleware and for this reason they should be able to connect to the middleware via network. For the transport layer we chose to use TCP and IPv4 for the internet layer.

3.4.1 Request Types

After a client has established the TCP connection to the middleware, it is able to send a request. A request should be serialized into to `ByteBuffer` before being sent over the network. To do so, we created `Packet` class which is able to serialize a request which inherits the class.

A request is serialized into ByteBuffer which consists of two parts:

- `Header`. 4 bytes header denotes the size of the packet.
- `Payload`. The rest of the packet is payload. The payload is Java object serialized via `Serializable` and converted to the `ByteBuffer`.

For each request type we have a request and a response class. Both of them inherit the Packet class. Table 3.2 shows all supported types which can be found in **ch.ethz.pastas.common.protocol** package.

3.4.2 Error Handling

Any error while processing a client request is returned to the client by serializing it into `ErrorResponse`. The response should contain an error code. The comprehensive error codes list is shown in Table 3.3.

Request	Response
CreateQueueRequest	CreateQueueResponse
DeleteQueueRequest	DeleteQueueResponse
SelectQueueRequest	SelectQueueResponse
InsertMessageRequest	InsertMessageResponse
GetMessageRequest	GetMessageResponse
GetBroadcastMessageRequest	GetBroadcastMessageResponse
FindMessageBySenderIdRequest	FindMessageBySenderIdResponse
GetQueuesRequest	GetQueuesResponse
PingRequest	PingResponse

Figure 3.2: Supported request and response classes.

Name	Details
`INTERNAL_ERROR`	Internal system error
`QUEUE_NAME_EXISTS`	A queue with such name already exists.
`QUEUE_NOT_FOUND`	Queue not found.
`QUEUE_NOT_EMPTY`	Queue is not empty. Happens when deleting the queue.
`NO_PERMISSION_TO_READ`	Receiver does not have permissions to read the message.
`QUEUE_IS_EMPTY`	Queue is empty.
`NO_QUEUES_FOUND`	No queues found containing any message to the given receiver.
`UNKNOWN_REQUEST_TYPE`	Unknown client request type.
`NO_MESSAGE_FOUND`	Cannot find any message in the given queue.

Figure 3.3: Error codes used in ErrorResponse.

3.5 Queueing Mechanism

Because the middleware should support such cases when there are more clients than available RequestHandlers, each client request should be placed in a queue and handled by non-busy RequestHandler. For this purpose, we use `ExecutorService` with `FixedThreadPool` from Java standard library. The pool is using `LinkedBlockingQueue` internally to store requests. An read access to the queue is blocking the worker thread when the queue is empty. The blocking results in making thread to sleep which allows OS scheduler to execute the other threads. When a new item is placed to the queue and queue is empty, all blocked threads are signaled and they all compete on the newly placed item.

FixedTheadPool creates a predefined number of threads. Thus, we avoid forking a new thread per request which could have quite high costs in performance.

3.6 Design Decisions

Because of our hypothesis stating that the database might be the main bottleneck of the system, we decided to offload RequestHandlers as much as possible that they could fully utilize the database. For this reason, we make RequestDispatcher to read and parse requests instead of putting this functionality inside RequestHandler.

Because of obvious lock-contention on the queue, we were considering using multiple queues. But due to increased complexity when analysing the system, we decided to ditch such approach. Also, utilizing the queues equally would require some additional effort. For example, work-stealing could be implemented.

For simplicity reasons we decided to use Serializable for serialization of the request payload. Serializable comes with its costs and it is possible to make the serialization more efficient. However, we tested the costs of serialization in terms of memory and time and we concluded that it is constant per message type and this cost can be neglected.

One would think about improving the throughput by using UDP instead of TCP, but it would require additional work for handling timeouts on client side and making explicit acknowledgments for requests. Thus, we excluded it from our consideration list.

Our hypothesis for the middleware is that if it contains any bottleneck, it can be ruled out by increasing number of its instances. We base our hypothesis on the fact that it follows shared-nothing architecture pattern, i.e. there is nothing shared among middleware if we do not consider the database.

3.7 Performance characteristics

In order to evaluate the middleware we implemented dummy PingRequest which does not involve usage of the database, thereby we could test the performance of the middleware only. The request takes the path in the middleware as a regular request, only the database is avoided. The PingRequest sends a small payload of the size 1 byte to the middleware and waits for the response which contains the same 1 byte.

Table 3.4 shows the test parameters. We run two experiments. The first one included the middleware which ran on single machine, while the second

run two instances of the middleware on separate machines. In both cases the total number of the worker threads was kept the same.

Parameter	Value
Common	
Provider	Amazon AWS EC2
Guest OS	Ubuntu 14.04 LTS
Virtualization	PV
Zone	eu-west-1b
Duration of single test case	5 minutes
Client	
Machine	m3.medium
Number of machines	1
Middleware	
Machine	m3.medium
Number of worker threads in total	20

Figure 3.4: Configuration for the middleware scalability test

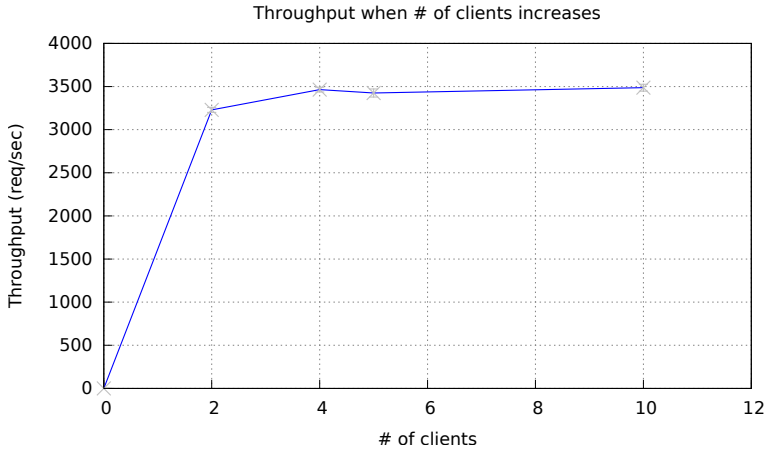

Figure 3.5: Throughput of the middleware running on one node when increasing number clients. Averaged over one minute interval.

As we can see from Figures 3.6 and 3.8 the response time increases linearly although the throughput stays the same. It is expected behaviour when the workload increases and the system reached the maximal performance, i.e. throughput stays roughly the same. Also, the response time decreased

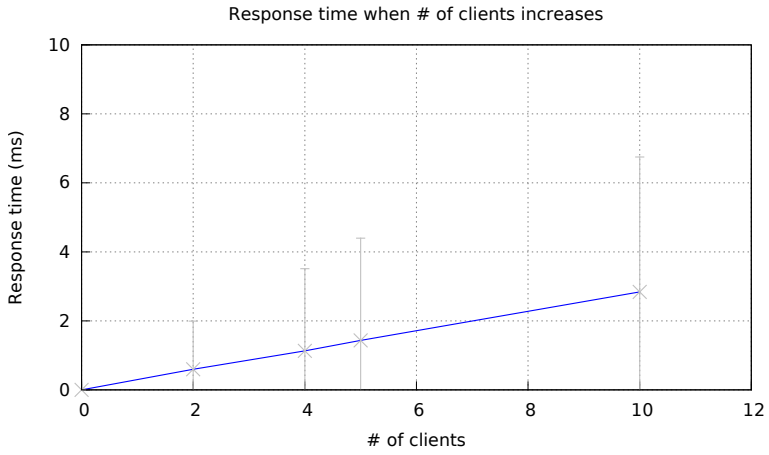

Figure 3.6: Response time of the middleware running on one node when increasing number of clients. Averaged over one minute interval.

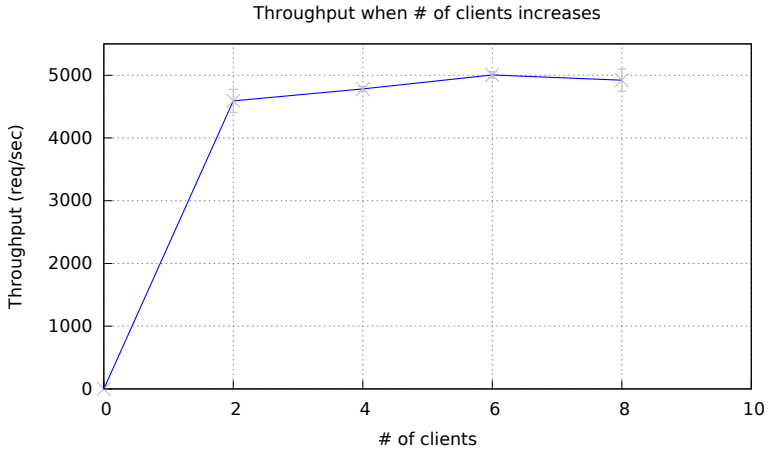

Figure 3.7: Throughput of the middleware running on two nodes when increasing number of clients. Averaged over one minute interval.

almost twice when running the two nodes which should result in increased throughput.

An interesting behaviour can be seen in Figures 3.5 and 3.7. Doubling the number of middleware nodes increased the system performance. Also, in a case of two nodes, the system got utilized (**6** clients) with almost double number of clients compared to the case when there was only one middleware node (between **2** and **4** clients).

Such performance doubling when increasing number of middlewares gives us a hint about the bottleneck residing inside the middleware. Our assump-

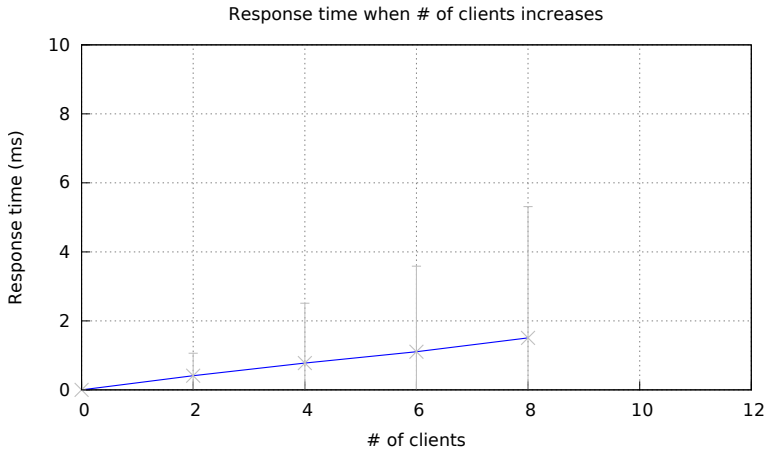

Figure 3.8: Response time of the middleware running on two nodes when increasing number clients. Averaged over one minute interval.

tion is that RequestDispatcher is the bottleneck, because its only one instance is running per node and it is responsible for reading the packets before they can be processed. With two nodes, we doubled the number of dispatchers and improved performance. Further investigation with `VisualVM` sampler proves our assumption, i.e. the largest fraction of the time JVM spends running RequestDispatcher thread.

Also, from the benchmarks above, we can conclude that the client node did not reached its maximal performance. Therefore, for benchmarks presented in the report, we considered using only a single machine for clients.

Chapter 4

Clients

This section presents the third building component of our messaging system - clients.

4.1 Design

We designed the client as a library which makes our system accessible from any Java thread. The client library is located inside **ch.ethz.pastas.client** package.

By creating an instance of `Client` class, we establish TCP connection to the given middleware. An endpoint of the middleware is specified with a pair of hostname and port which the middleware is listening on. The Client object is able to connect only to one middleware node.

4.2 API

Client class implements the messaging system API. The API is shown in Listings 4.1 4.2 4.3 4.4.

```
/**
 * Send a ping request to the connected middleware.
 *
 * @param payload           any string which will be sent
                            back with response
 * @throws ClientException
 */
public void ping(String payload) throws ClientException
```

Figure 4.1: Instrumentation API

```
/**
 * Creates a queue.
 * @param queueName     string denoting queue name
 * @return              newly created queue id
 * @throws ClientException
 */
public int createQueue(String queueName) throws ClientException
/**
 * Removes a queue.
 *
 * @param queueId       queue id
 * @throws ClientException
 */
public void deleteQueue(int queueId) throws ClientException
/**
 * Returns queue id for the given name. I.e. it does lookup.
 *
 * @param queueName     a name of the queue
 * @return              queue id if found
 * @throws ClientException
 */
public int selectQueue(String queueName) throws ClientException
```

Figure 4.2: Queue manipulation API

Any API function might raise a `ClientException` which contains error code described in Table 3.3.

4.3 Workloads

To generate a workload needed for the system evaluation we run a client node. The client node consists of JVM which is running multiple client threads. Each thread is running only one instance of Client class.

Because clients do not share anything besides CPU time and IO we can increase the workload by either adding more threads to the runtime or by adding more client nodes. To avoid contention on the OS resources we can run the client nodes on different machines. Therefore, we can assume that generated workload by clients scales linearly with increasing number of machines running client threads.

During the benchmarks, each client opens a single separate connection to a chosen middleware and does not close it until the end of benchmarks. During each iteration, client might send any request with any parameters. The

```
/**
 * Sends message.
 *
 * @param queueId       queue id
 * @param receiverId    receiver id | 0 - if message is of
                                broadcast type
 * @param msg           message payload string
 * @return              message id
 * @throws ClientException
 */
public int sendMessage(int queueId, int receiverId, String msg)
    throws ClientException
```

Figure 4.3: Sending message API

probability of each request and parameters is up to configuration. Basically, the client might:

- Send a broadcast message to a queue.
- Send a message to a queue for a receiver.
- Read broadcast message for a queue.
- Read a message from a queue.
- Lookup queues containing a message for the client.
- Find a message from a sender.

Each client API call logs the parameters and return value of the call. Also, the duration of the call is logged. These two are helpful for instrumentation of clients during benchmarks and testing.

4.4 Testing Clients

To test correctness of client behaviour including our system, we use the following:

- Analyzing logs produced by client. Each exception is logged.
- From the logs we can deduce how many requests to the database have been made. This allows to check results consistency.
- Simple testing of the entire system was done during development phase.

```
/**
 * Requests a message from the given queue.
 *
 * @param queueId       queue id
 * @param removeMsg     denotes whether to delete
                        a received message
 * @param retry         denotes whether auto-retry should
                        be done if there is no message for
                        the receiver
 * @return              Message object which contains
                        all message attributes
 * @throws ClientException
 */
public Message getMessage(int queueId, boolean removeMsg,
        boolean retry) throws ClientException
/**
 * Requests a broadcast message.
 *
 * @param queueId           queue id
 * @param removeMsg         whether to delete a message
 * @param retry             whether to retry
 * @return                  Message object
 * @throws ClientException
 */
public Message getBroadcastMessage(int queueId,
    boolean removeMsg, boolean retry) throws ClientException
/**
 * Lookup a message from a particular sender.
 *
 * @param senderId          sender id
 * @return                  Message object
 * @throws ClientException
 */
public Message findMessageBySenderId(int senderId)
    throws ClientException
/**
 * Return a list of queues which contains a message
 * readable by the client.
 *
 * @return      list of queue ids
 * @throws ClientException
 */
public ArrayList<Integer> getQueues() throws ClientException
```

Figure 4.4: Receiving message API

Chapter 5

Design of the System

This section present the overall design of our system. Also, it describes how the benchmarking of the system is done.

5.1 Complete System

The system is organized in multiple tiers. The complete picture is shown in Figure 5.1.

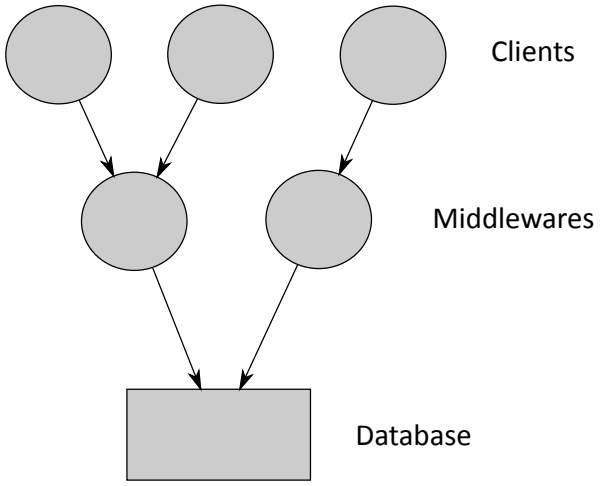

Figure 5.1: The design of the messaging system

The system is built in such way that adding any component on new machine is just a matter of configuration. Therefore, we can easily test the system on different machines with different parameters.

5.1.1 Configuration

The system is configured via Java Property files. The system configurable parameters include:

- Number of clients per machine.
- List of middleware ip addresses.
- Worker threads count per machine.
- Database connection pool size per machine.
- Database connection details.

Besides that, it is possible to configure such benchmark parameters as:

- Probability for each request type.
- Number of queues used during a benchmark.
- Probability for message size.
- Probability that receiver / sender will request queue which belongs to them, i.e. queue id is equal to the client id.
- Probability for reading or sending broadcast message.
- Probability for deleting a message after it has been read.

5.1.2 Code Organization

The code of the system is organized in the following packages:

- **ch.ethz.pastas.client**. Client library.
- **ch.ethz.pastas.middleware**. The middleware.
- **ch.ethz.pastas.common**. The shared libraries between client and middleware.
- **ch.ethz.pastas.bench**. Benchmarking code.

5.1.3 Logging

For the system instrumentation and benchmarking purpose, the logging is used extensively in the system. For logger implementation we use `Logger` from the Java standard library. Additionally, we introduced a new log level `BENCH` which is used for benchmarks and has the highest severity. Thus, we can avoid the noise when running benchmarks.

To log duration of some executions, we introduced helper class `BenchLog`. The object provides a stopwatch functionality, thereby it allows to measure

duration of some predefined event. The user object has to issue start and finish commands. After the finish, BenchLog will log the duration of the event. In our system we log the duration of all client-level API calls and also the duration of the database calls. The extensive list of the benchmarking events can be found in `BenchEvents` class.

5.2 System Deployments

Considering the number of benchmarks which we conduct, the need for automatic deployments is obvious and thus we created a few shell scripts for this purpose.

First of all, we automated launching of virtual machines on Amazon AWS EC2 infrastructure. We use `awscli` to spawn a new machine and it is done by **setup/aws/create-ec2.sh** script. The script not only creates the machine, but also installs all necessary dependencies depending on machine type. There are three types of VM:

- Database machine.
- Client machine.
- Middleware machine

Next, with **setup/aws/upload-files.sh** we upload all necessary files to run the benchmarks. The files are sent to the database machine. To communicate with machines we use `ssh` and `scp` clients. All machines share the same key pair used for SSH authentication. To make sure that any script keeps running after we close the SSH session, we start the script with `nohup` utility.

Further on, the database machine starts benchmarks by using **setup/aws/run-experiments.sh** script. The script uploads the necessary files including benchmark configuration property file to client and the middleware machines. The benchmark scenarios are defined by benchmark scenario file named after **runs.txt** which denotes what configuration should be used for benchmark, how many and what kind of machines should be used, duration of experiment. Afterwards, the benchmarks are started and after the time needed for the experiment the client and middleware instances are stopped. Finally, the database machine gathers all logs for those machines and parses them for plotting.

The logs are parsed and stored in `SQLite3` database by using **setup/plot/create-db.py** script. Before writing to the database, the parser script removes the records of the first and the last minute to avoid outliers which are due to warm-up and cool-down phase. Afterwards, confidence intervals are checked of the parsed data. If the confidence interval check fails, then we re-run the benchmark.

Chapter 6

Experiments

This section presents all experiments which were conducted in order to evaluate performance and behaviour of the messaging system.

6.1 Setup

All experiments were done on Amazon AWS EC2 virtual machines. Table 6.1 summarizes the parameters of used machines. During the phase of experimentation we encountered quite a few problems with Amazon infrastructure, including CPU and IO scheduling. We experienced that running the long experiments on machines which were used before for experimentation might result in unexpected increases of latency and consequently - drops in throughput. It make an hypothesis that such behaviour is due to Amazon AWS SLA policy, but we cannot prove the hypothesis, because Amazon AWS Support did not respond to our issue. Luckily, creating new machines instead re-using the old ones, solved the stability problems. Also, switching between data-center zones helped. But Amazon data-centers differs in hardware, therefore the final benchmarks were done in the same DC.

Additionally, we ran a few experiments on Digital Ocean virtual infrastructure. To conclude, the system behaved more stable. Unfortunately, due to Digital Ocean billing policy we could not run the whole set of benchmarks for sake of comparing it to Amazon AWS.

All experiments presented in the report are of closed-system type. All the measurements reached **95%** level confidence for an interval **5%** around the mean. Most of the experiments were run between 5 and 6 minutes.

The experiments presented below used the same configuration of clients if it is not stated. It is due to be able compare results of different experiments. Also, the configuration ensured that database size stays roughly the same

Parameter	Value
Provider	Amazon AWS EC2
Guest OS	Ubuntu 14.04 LTS
Virtualization	PV
Zone	eu-west-1b
DB	PostgreSQL 9.3
Client machine type	m3.medium
Middleware machines type	m3.medium
Database machine type	m3.large

Figure 6.1: Infrastructure parameters for all benchmarks

during an experiment. Table 6.2 shows the client configuration of each experiment.

Parameter	Value
Number of queues	2
Number of client nodes	1
Number of client nodes	1
insert_message	
Probability for request	0.2
Probability for 200 and 2000 byte message	0.5, 0.5
Probability for broadcast message	0.1
read_message	
Probability for request	0.6
Probability for removing message	0.9
read_message_broadcast	
Probability for request	0.2
Probability for removing message	0.9

Figure 6.2: Client-side configuration parameters for running benchmarks

Before running an each case of experiment, we restarted each JVM instances including the database, also we re-created the schema of the database. As a next thing, we populated the database with data before creating stored procedures. It is because of the fact that stored procedures execution plan is cached and having some data in the beginning might give useful hints for query execution planner.

6.2 Stability

To evaluate the stability of our system, we run two experiments. Each experiment lasted for **60** minutes. In both experiments we ran **10** clients and the total number of worker threads including database connection pool size was equal to **10**. In the first experiment the middleware was running on a single machine, while the second experiment increased the number of middleware machines by two.

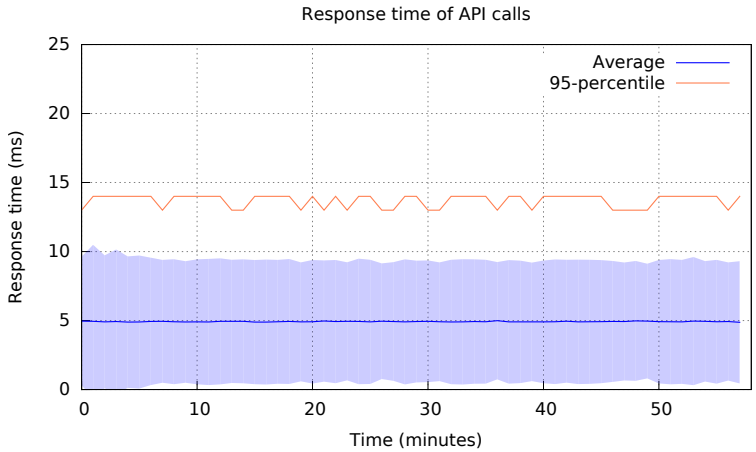

Figure 6.3: Response time when running the middleware on a single node. Averaged over one minute interval.

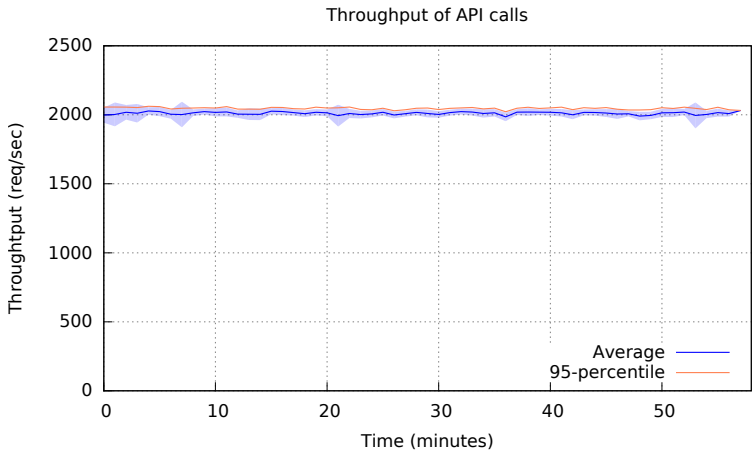

Figure 6.4: Throughput of API requests when running the middleware on a single node. Averaged over one minute interval.

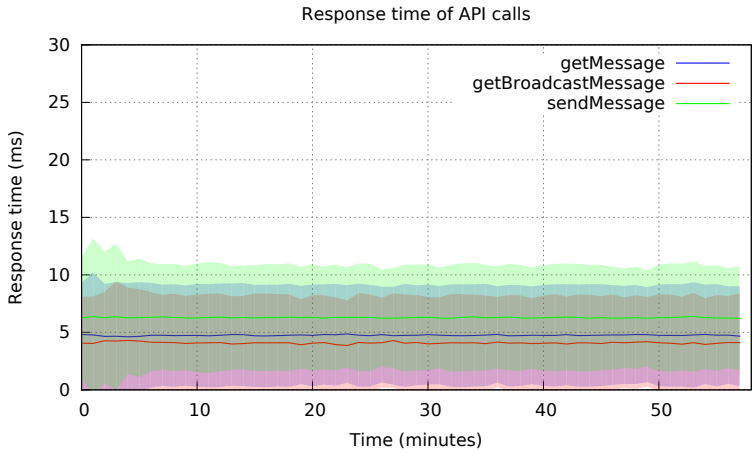

Figure 6.5: Response time of separate API calls when running the middleware on a single node. Averaged over one minute interval.

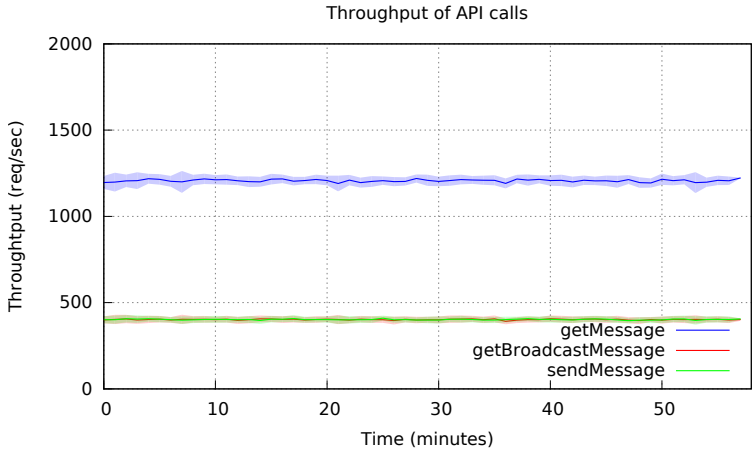

Figure 6.6: Throughput of separate API requests when running the middleware on a single node. Averaged over one minute interval.

As it can be seen from Figures 6.3, 6.4, 6.5, 6.6 the system behaved stable during long period of time when running the middleware on a single node.

Adding additional node for the middleware, as it can be seen from Figures 6.7, 6.8, 6.9, 6.10 introduced small fluctuations in the throughput. It is because adding a new node reduced impact of RequestDispatcher bottleneck and the database was fully utilized. The most of fluctuations might be caused by the database vacuuming activities due to frequent `DELETE` operation. Small spikes in latency can be correlated to decrease in throughput.

In both cases, `getMessage` has the highest throughput due to high probabil-

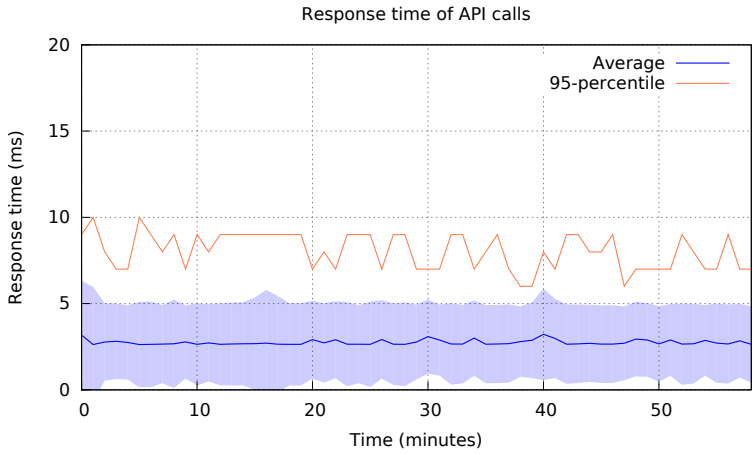

Figure 6.7: Response time when running the middleware on two separate nodes. Averaged over one minute interval.

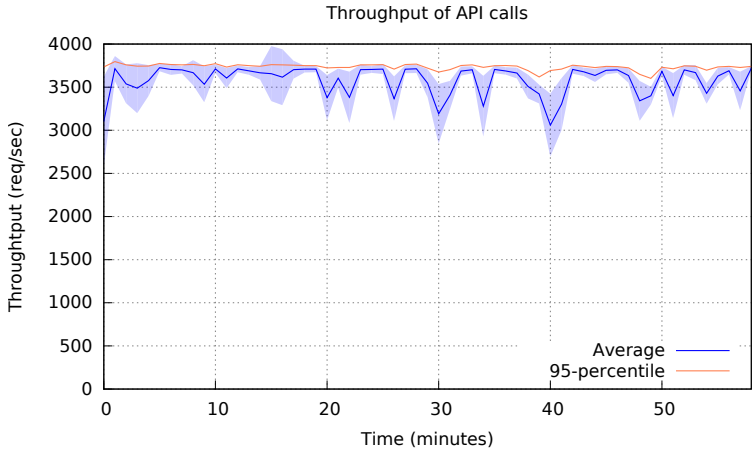

Figure 6.8: Throughput of API requests when running the middleware on two separate nodes. Averaged over one minute interval.

ity of this operation probability as it was shown in Table 6.1.

6.3 Scalability

To evaluate the scalability of our system, we run series of experiments. Each set of experiment was increasing only one factor. The other two factor values were set to **20**. The factors include:

- Number of clients

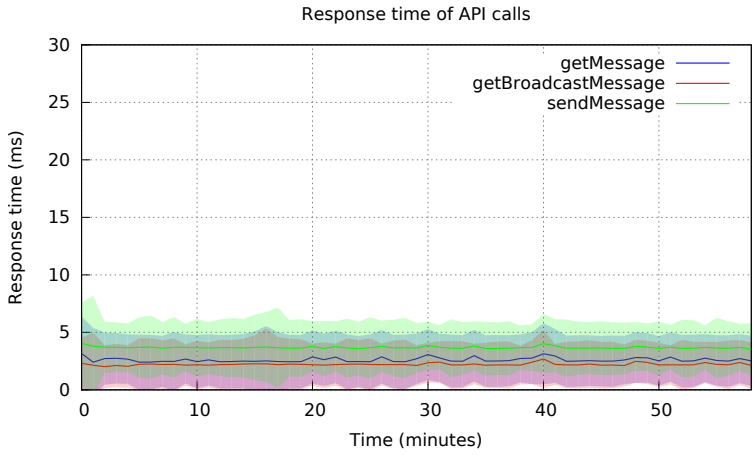

Figure 6.9: Response time of separate API calls when running the middleware on two separate nodes. Averaged over one minute interval.

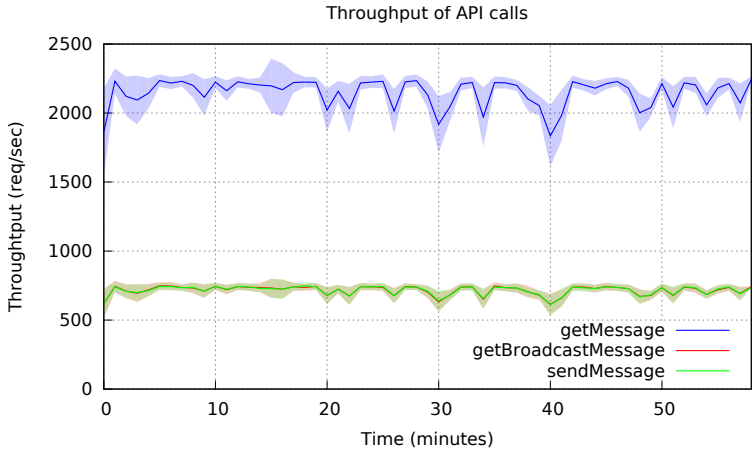

Figure 6.10: Throughput of separate API requests when running the middleware on two separate nodes. Averaged over one minute interval.

- Number of worker threads
- The database pool size

We ran each set of experiments when middleware was running on one, two and four machines.

6.3.1 Workers count

The impact of varying workers count can be seen in set of Figures 6.14. In all cases the performance maxed out when we did not reach **20** workers.

Considering the fact, that there were **20** active clients, the queueing of their request did not have any effect in performance.

Also, doubling the size of middleware nodes almost doubled the performance and reduced the response time. But there is no differences in performance, when the number of middleware nodes reached four. The maximal throughput in a case of single node was reached with **4** workers, while with more than one node **8** workers peaked the throughput. We can conclude that with such worker configuration we reached the bottleneck which is the database.

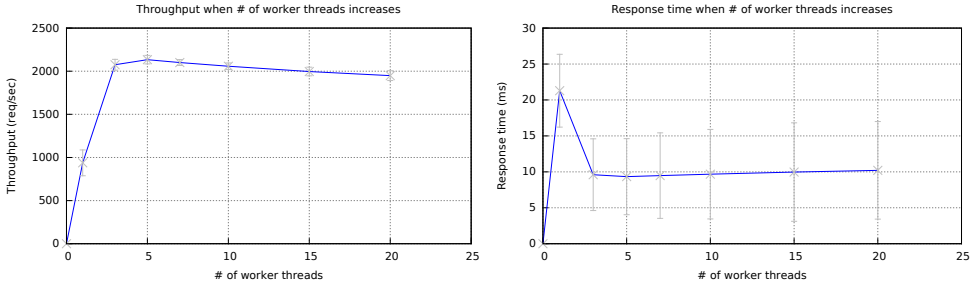

Figure 6.11: The middleware is running on a single node

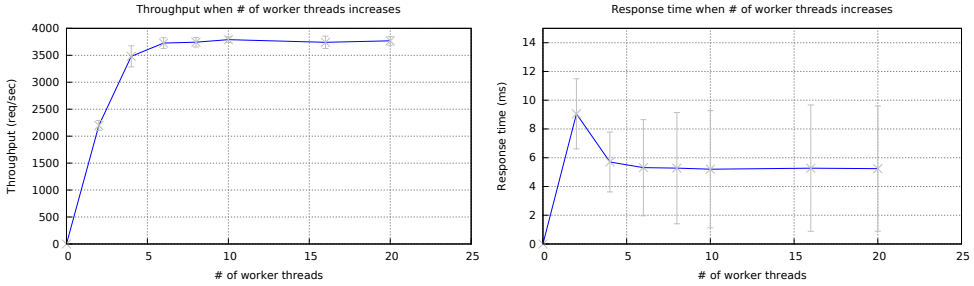

Figure 6.12: The middleware is running on two nodes

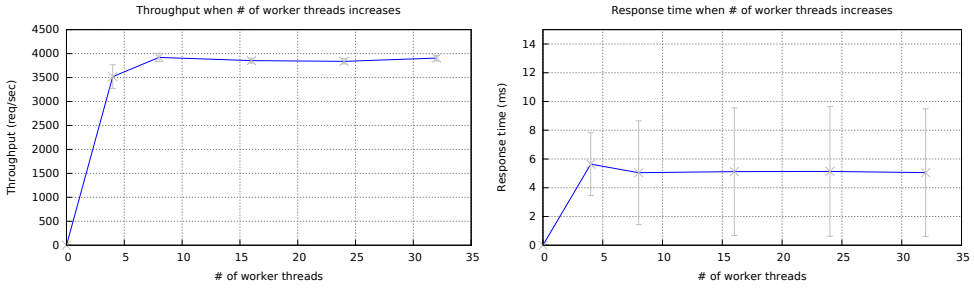

Figure 6.13: The middleware is running on four nodes

Figure 6.14: Response time and throughput of API requests when the number of worker threads increases.

6.3.2 Database connection pool size

Increasing database connection pool size has similar effects as in the workers case. Figure 6.18 shows such trend.

There are spikes in response time in each case's beginning of the plot. These spikes are due to the fact that in the beginning the database is not utilized and the bottleneck is the number of resources. In this case, the database pool size is the limiting factor.

In a case of single node, the maximal throughput was reached with the pool size of **4** and with more nodes it was around **8**.

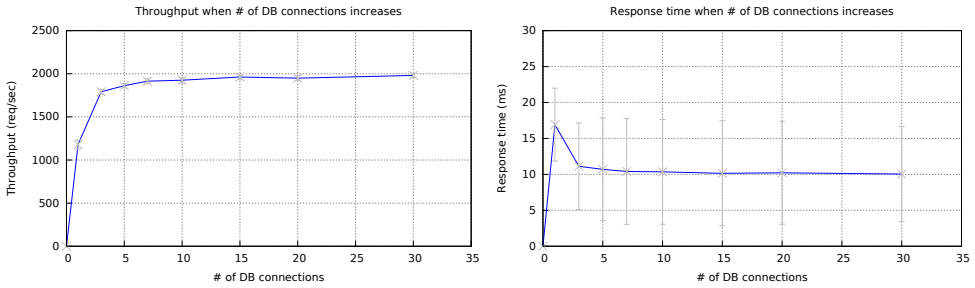

Figure 6.15: The middleware is running on a single node

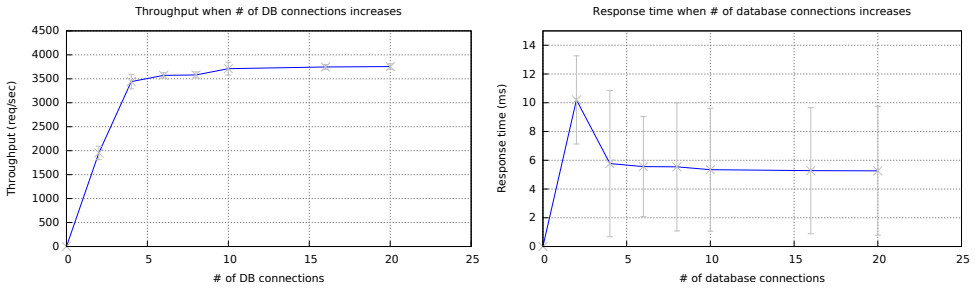

Figure 6.16: The middleware is running on two nodes

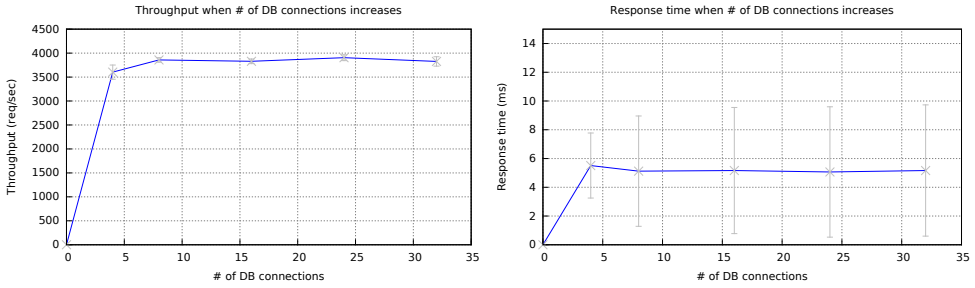

Figure 6.17: The middleware is running on four nodes

Figure 6.18: Response time and throughput of API requests when the number of the database connection pool increases.

6.3.3 Clients Count

The increasing count of clients has different effect on the response time as it can be seen in Figure 6.22. The response time is increasing linearly because the requests get queued.

The throughput has the similar pattern as in the benchmarks above. With single node it peaked with less clients than with more than one node. Also, the throughput has doubled when node count increased from one to two.

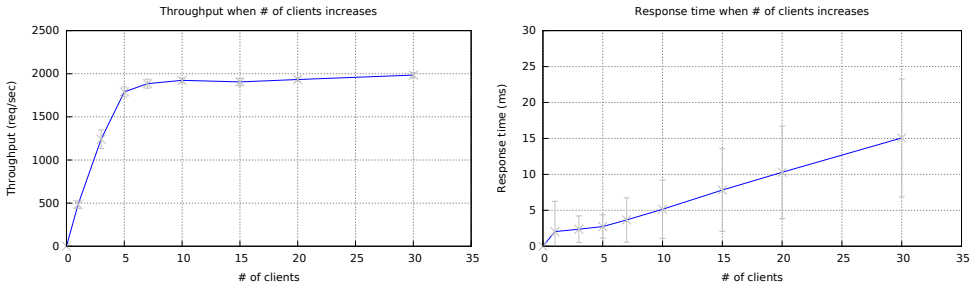

Figure 6.19: The middleware is running on a single node

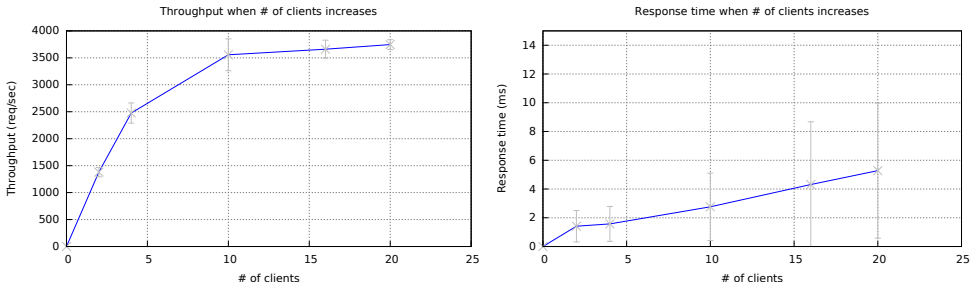

Figure 6.20: The middleware is running on two nodes

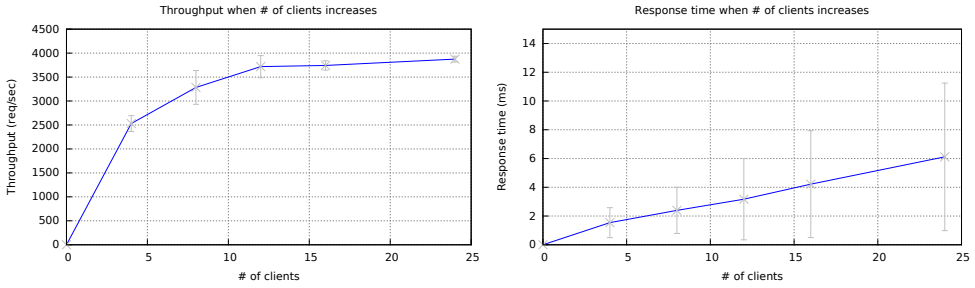

Figure 6.21: The middleware is running on four nodes

Figure 6.22: Response time and throughput of API requests when the number of the clients increases.

6.3.4 Message Size

The impact of message size is shown in Figure 6.23. For the experiment we were increasing the message size, while clients count, the databae pool size and workers count was the same and equal to **10**. Also, to support larger than **2000** messages, we changed `mesages(content)` type from `VARCHAR(2000)` to `TEXT`. Such change does not have impact on performance.

The trend can be seen from the plots - the throughput decreases with the message size. After **2000** PostgreSQL should start `TOAST` messages, therefore

throughput decreases slightly faster after that point.

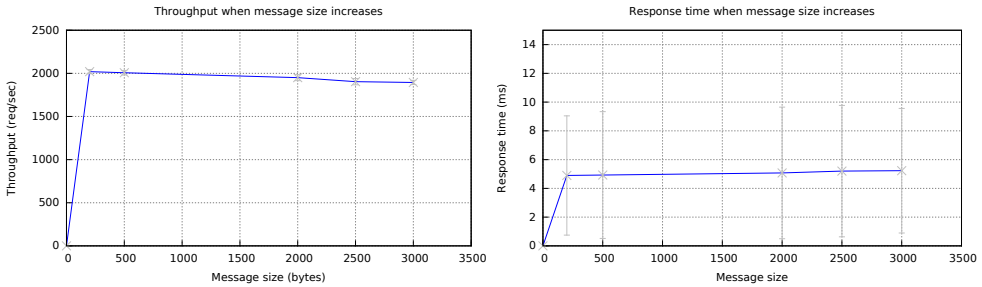

Figure 6.23: Response time and throughput of API requests when the size of message increases.

6.3.5 Maximal throughput

As we can see from experiments described in Sections 6.3.1, 6.3.2 and 6.3.3, we can reach the maximal throughput with different configurations. One of the possible is **2** middleware nodes, **10** clients, worker threads and database connections. Such numbers can be proved by Figure 2.7. For this case the throughput is close to **4000** requests per second and the response time is almost **6** ms. The throughput is limited due to the database bottleneck.

6.3.6 Response time breakdown

To show the correlation between the database and API calls, we plotted the response time of `getMessage` call together with response time of `read_message` stored procedure. The data was taken from measurement presented in Figure 6.7. As it can be seen in Figure 6.24, all the spikes in API call response time correlates to the database request response time. It might be either database activities such as vacuuming or Amazon resource scheduling policy. Considering the fact that database is running on different machines, the first assumption is more likely to be true.

Also, the difference between both calls is roughly the same which is good indicator that behaviour of our system is stable.

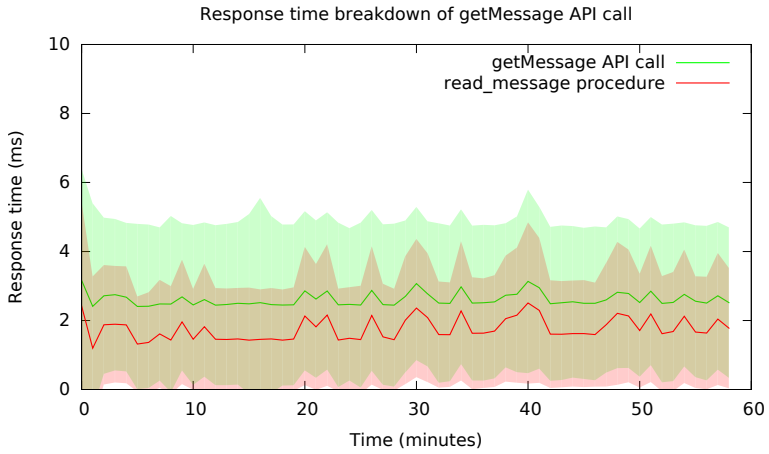

Figure 6.24: Response time of `getMessage` together with response time of the `read_message` stored procedure. Averaged over one minute interval.

Chapter 7

Conclusions

First of all, we built the fully functioning distributed messaging system. While building the system, we learned to back each design decision by micro benchmarks. Such systematic approach helped us not only to understand the system, but also to predict the performance of the system.

The messaging system has two bottlenecks. The first is single instance of `RequestDispatcher` per node. Fortunately, this bottleneck can be eliminated by increasing the number of middleware nodes. As it was expected from the beginning, the second bottleneck is the database. Unfortunately, not so much of vertical scaling can be done in order to increase the system's throughput. Horizontal scaling, i.e. adding more database nodes would improve the throughput. However, such scaling was out of scope of this course. Although, it would allow us to make interesting performance evaluations.

Next, during the experimentation phase we had a lot of problems with Amazon AWS. Eliminating them early would give us a reasonable amount of time for more interesting benchmarks. For example, measuring the behaviour of the system when the database size increases.

Finally, building or evaluating a larger scale system brings us something new. Having an opportunity to build the same system from scratch, we would introduce another level of worker threads inside the middleware. The threads would be responsible for reading requests. Thus, the bottleneck of single middleware node could be eliminated without adding additional nodes. Also, we would consider using system monitoring tools such as `Zabbix` or `Nagios` to be able to make better correlations between system events and spikes in measurements.

To conclude, the milestone was really demanding in terms of time, but fun factor of the project overcome the frustrations during sleepless nights.