Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Telesto - A Distributed Message Passing System

Report Group 32

Dominic Langenegger, Simon Marti
{dominicl,simarti}@student.ethz.ch

Advanced Systems Lab 2013
Systems Group
ETH Zürich

Supervisors:
Markus Pilman
Prof. Dr. Gustavo Alonso

November 15, 2013

Abstract

This document describes *Telesto*, a distributed message passing system built as mandatory course work for the course *Advanced Systems Lab* at ETH Zurich in autumn semester 2013.

After a short introduction, we describe the detailed architecture of the system and share some important implementation aspects before presenting various measurements on the running system that show stability, scalability and performance under different configurations. The measurements are discussed in detail and associated with the architecture in order to identify bottlenecks with respect to multiple performance metrics.

A final conclusion summarizes our experience and learnings out of the whole project.

Contents

Chapter 1

Introduction

For the Advanced Systems Lab course at ETH Zurich, students (in groups of two) have to build their own distributed message passing system following a very openly formulated project description [1] which describes the most important requirements.

The requirements specify the basic architecture of the system and its minimal functionality, but a very large part of the actual design of the system is leaved to the students. This is actually very beneficial for them because for many of them it is the first time they build such a large system from scratch and it is very educational to encounter all possible problems and actually find solutions for them.

While the first part of the project work is about building the actual system, the second part is all about experimentally evaluating the whole system in its key performance characteristics. This includes tasks like running stability and scalability tests but also finding optimal system and workload parameters.

In this second part, the task is not only to measure certain characteristics but also to learn how they arise and why the system behaves like it does.

This report summarizes the entire project work during the design, implementation and evaluation of *Telesto*, the distributed message passing system built for the Advanced Systems Lab course in autumn semester 2013 by Dominic Langenegger and Simon Marti.

1.1 Requirements

For the completeness of this document, we list the most important requirements of the distributed message passing system built (for details consult [1] directly):

- The system consists of three tiers; i.e. a database, one or multiple middlewares and many clients.

- Clients only communicate with one middleware (and not with the database)
- The system lets clients manage queues containing messages by inserting and reading (while deleting) from them
- The whole system is built in *Java* using *PostgreSQL* as underlying database
- No external libraries exception for the *PostgreSQL JDBC* driver are allowed as part of the actual system. Also prohibited is the use of the *Java Persistence API (JPA)* and *Remote Method Invocation (RMI)*.
- The tests of the system should include at least one long running (>2h) trace and many micro benchmarks analyzing separate parts of a messages way through the system.
- Tests have to be repeated often enough to achieve statistical significance for the results

The project description contains various hints on how the system could be built in order to achieve a state of the art implementation that has good performance and is easy to evaluate during the testing phase.

Chapter 2

Goals

The goals for building *Telesto* was clearly producing a fast and very performant system with high scalability and stability. We especially put emphasis on creating an efficient way to communicate between client and the middleware using a lightweight packet format and capable request handling system.

After an initial design phase we came up with a system supporting push based communication and complex caching on middlewares. We realized at the latest when a short *FAQ* [1] was published saying that we should build a pull based system, that our architecture was way too complex and can be replaced by a simpler, more transparent design.

Out of this experience our aim to build a fast system shifted more to a system that is not only fast but also easy to test and understand in its details. Only with this in mind during implementation it is feasible to actually test the system in enough detail to be able to understand and correctly interpret the results.

In the end we clearly wanted to analyze and identify the bottlenecks in *Telesto* with respect to various performance metrics.

[1] Frequently Asked Questions

Chapter 3

Architecture

This chapter explains the basic architecture of *Telesto* explaining how each part of the system works and how they communicate together. Chapter 4 gives a more detailed insight about how the implementation of some important component looks like.

Telesto is a three tier system:

Database
A *PostgreSQL* [1] database storing the persistent state of the system

Middleware
The part that provides many clients simultaneously with services of the message passing system and stores all data in the database. This part can be easily replicated.

Client
Clients that pass and receive messages from the system by talking to one middleware instance.

Figure 3.1 shows a sample architecture diagram. It is important to note, that clients only talk to middlewares and only a middleware has direct access to the database.

3.1 Database

Telesto uses *PostgreSQL* as underlying database. It comes with a lot of features of which only a small subset are actually used by *Telesto*. The main directive for building the database was focusing on a simple and scalable design and using stored procedures to do all database interactions rather than prepared statements. The latter reduces the use of *SQL* in the middleware to an absolute minimum since only function calls have to be passed to the database.

[1] PostgreSQL Website
Available at: http://www.postgresql.org/ [Accessed November 15, 2013]

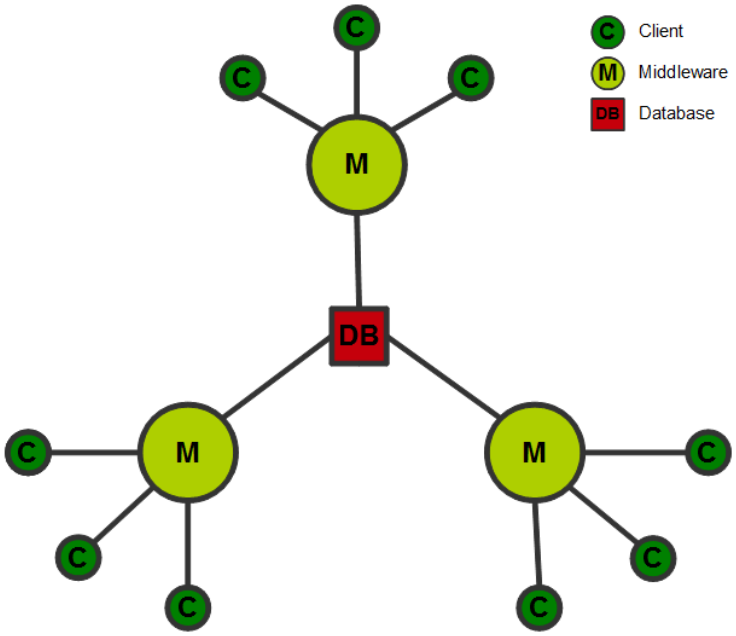

Figure 3.1: Sample architecture diagram of *Telesto* with 3 middlewares each serving 3 clients and one central database.

3.1.1 Entities

In *Telesto* there exist only three different entities:

Client
: A client to the system identified by a unique name and client_id. Clients have a certain mode indicating whether they are only allowed to read messages or also put new ones.

Queue
: A queue that can contain multiple messages and is identified by a unique name and queue_id

Message
: A string message in exactly one queue with a client as sender, a potential receiver, a priority, a content string and, if the message is part of a request response interaction, a context. Additionally a timestamp is stored indicating when the message arrived in the system.

Field name	type	Description
client_id	serial	primary key using sequence
client_name	varchar(255)	unique
client_mode	smallint	

Table 3.1: Table `clients`

Field name	type	Description
queue_id	serial	primary key using sequence
queue_name	varchar(255)	unique

Table 3.2: Table `queues`

Each of this entities can directly be modeled into one database table each as shown in tables 3.1 to 3.3.

In order to create the tables we used *PgAdmin 3*[2] on a local development database and used the backup function to create a dump which could be distributed to our testing environment.

We deliberately pass on creating foreign key constraints on our tables because *a*) we use table joins for just one operation (i.e. `get_active_queues()`, see section 3.1.3) which can also be handled by an index; *b*) we don't need any actions on update or deletion except for the case when a queue is deleted, which can easily be handled individually; *c*) we are sure that we don't insert inconsistent data because data is never updated [3] and queue existence is checked on insert; and *d*) we support inserting messages for not (yet) existing clients by design because they may only register themself at a later point in time.

[2] pgAdmin Website
Available at: http://www.pgadmin.org/ [Accessed November 15, 2013]

[3] There is actually no support to change queue or client names. This could however be added while still not rendering this argument invalid because only id rather than name attributes would be used as foreign keys

Field name	type	Description
message_id	serial	primary key using sequence
queue_id	integer	
sender_id	integer	
receiver_id	integer	
context	integer	
priority	smallint	between 1 (lowest) and 10
time_of_arrival	timestamp	set to `now()` by default
message	varchar(2000)	the actual message

Table 3.3: Table `messages`

Parameter	affected fields	required	Description
queue_id	queue_id	X	
receiver_id	receiver_id	X	matches if either `null` or own client_id
sender_id	sender_id		
context	context		to identify responses
mode	priority, time_of_arrival	X	one of both used for ordering

Table 3.4: Parameters of a message query

3.1.2 Indexes

The main actions on the database in *Telesto* are inserting messages and removing them (by reading them). Since reading messages supports some parameters (see table 3.4), it is strongly recommended to use appropriate indexes on the affected tables. Additionally to the indexes specifically introduced to optimize the performance of a selection or sorting operation for message finding, there are primary keys indexed on every table which lower the cost of getting entries directly by their id.

Based on the data from table 3.4 we decided only use multi-column indexes for the table `messages` that always include the `receiver_id` as first part and the `queue_id` as second. The `receiver_id` is either an integer value or `null`. In both cases the query executor should be able to use the second part, namely the `queue_id` which is always present. The details of each separate index are listed below:

receiver_id, queue_id, priority
: For a query by priority and without specified sender

receiver_id, queue_id, priority, sender
: For a query by priority with specified sender

receiver_id, queue_id, time_of_arrival
: For a query by time without specified sender

receiver_id, queue_id, time_of_arrival, sender
: For a query by time with specified sender

3.1.3 Stored Procedures

As mentioned above, all database interaction is done using stored procedures[4][5]. For most of the database functions we used the standard *SQL* language syntax rather than the special *PL/pgSQL* language because the simple version serves almost all our requirements and it is often possible to write very easy queries in a very simple way. We however did not test if queries would run faster using *PL/pgSQL* because of the additional options *PostgreSQL* offers for these stored procedures.[6]

Table 3.5 lists all implemented stored procedures in the database of *Telesto*. They very directly resemble the methods supported by our network protocol (see section 3.2) which means there is not much logic required on the middleware in order to execute a query on the database given a request packet.

To simplify the database abstraction in the middleware we tried to produce very consistent return values. All functions either return tables of Queues, Messages, Clients, a set of integers or single integers. (where many are constrained to a single entry) For error handling, unique constraint violations are detected by the middleware and both `put_message` and `put_messages` return the `queue_ids` of the queues successfully inserted to (an id might be missing if the queue did not exist). Like this, errors from the database can be transformed into an appropriate `ErrorPacket` as introduced in the next section.

3.2 Network Protocol

In order to achieve high throughput and low latency, it is essential to have a lightweight communication protocol as a foundation. *Telesto* uses a binary protocol based on TCP to do all the communication between clients and middlewares. Connections to the database are handled by the *PostgreSQL JDBC Driver* [7] which is based on TCP as well but isn't part of *Telesto* itself. This section gives insight about the network protocol introduced by *Telesto* for the communication between clients and middleware.

A middleware offers a certain set of services (i.e methods) to the clients, like

[4] PostgreSQL 9.3 Documentation: SQL Procedural Language
Available at: http://www.postgresql.org/docs/9.3/static/plpgsql.html [Accessed November 15, 2013]

[5] PostgreSQL 9.3 Documentation: CREATE FUNCTION
Available at: http://www.postgresql.org/docs/9.3/static/sql-createfunction.html [Accessed November 15, 2013]

[6] Advantages of Using PL/pgSQL in the official documentation
Available at: http://www.postgresql.org/docs/9.3/static/plpgsql-overview.html#PLPGSQL-ADVANTAGES [Accessed November 15, 2013]

[7] PostgreSQL JDBC Driver
Available at: http://jdbc.postgresql.org/ [Accessed November 15, 2013]

Name	Parameters	Return Value	Description
	Client Manipulation		
request_id	client_name, mode	client_id	create a new client
identify	client_id	Client	identify a client
delete_client	client_id	client_id	delete a client
	Queue Manipulation		
create_queue	queue_name	Queue	creates a new queue
delete_queue	queue_id	queue_id	delete a queue
get_queue_id	queue_name	Queue	get queue by name
get_queue_name	queue_id	Queue	get queue by id
list_queues		array[Queue]	get all queues
get_active_queues	client_id	array[Queue]	get all queues with messages for the given client
get_messages_from_queue	queue_id	array[Message]	get all message in a queue
	Message Manipulation		
put_message	queue_id, sender_id, receiver_id, context, priority, message	queue_id	insert message and return queue
put_messages	array[queue_id], sender_id, receiver_id, context, priority, message	array[queue_id]	insert messages in multiple queues and return queues
read_message_by_priority	queue_id, sender_id, receiver_id	Message	get a message by priority
read_message_by_timestamp	queue_id, sender_id, receiver_id	Message	get a message by timestamp
read_response_message	queue_id, receiver_id, context	Message	get a message by receiver and context

Table 3.5: Parameters of a message query

putting a message in a queue or reading a message from a queue. Every such method is identified by a special `method id`. All method calls and responses are grouped into one *Telesto* packet consisting of four parts:

length
: The length of the entire packet in bytes. This value is sent as a `short` type integer which allows values of up to $32,768$. This limits the packet size, which is fine since the maximum supported message size is 2000 characters and all other fields are a lot smaller. Only the method to read all messages from a queue might (in rare cases) try to serve more data which would then fail.

method id
: A `short` containing the method id in order to identify the service requested and how to interpret the payload.

client packet_id
: An id that is set by the client and repeated by the middleware in the associated response in order to identify which request yielded which response.

payload
: The varying length payload containing all the arguments of the method call or the structured response data.

Besides a packet for each method call, there is one for the according response if applicable and two additional packets named `SuccessPacket` and `ErrorPacket` to indicate a successful call of a method with no return value or an error during execution respectively.

By convention the `packet id` for a response is always higher by one than the according request. A complete list of the currently supported methods and their parameters is shown in table 3.6.

By using this lightweight binary packet format, the overall packet size is only slightly larger than a binary sequence of all input parameters of a method which is certainly a good prerequisite for handling high loads with many requests in short time.

3.3 Middleware

The middleware is the core part of *Telesto* as it serves incoming request from clients in a highly efficient manner. The tasks arising can be split in 4 parts:

1. Handling incoming connections and data

Packet	method_id	payload
Ping	0x01	
Pong	0x02	
Success	0x03	
Error	0x05	error_type, error_string
Client Manipulation		
RegisterClient	0x11	client_name, mode
RegisterClientResponse	0x12	client_id
IdentifyClient	0x13	client_id
IdentifyClientResponse	0x14	mode, client_name
DeleteClient	0x15	client_id
Queue Manipulation		
CreateQueue	0x21	queue_name
CreateQueueResponse	0x22	queue_id
DeleteQueue	0x23	queue_id
GetQueueId	0x25	mode, queue_name
GetQueueIdResponse	0x26	queue_id
GetQueueName	0x27	queue_id
GetQueueNameResponse	0x28	queue_name
GetQueues	0x29	
GetQueuesResponse	0x2a	array[Queue]
GetActiveQueues	0x2b	
GetActiveQueuesResponse	0x2c	array[Queue]
GetMessages	0x2d	queue_id
GetMessagesResponse	0x2e	array[Message]
Message Manipulation		
PutMessage	0x31	Message, array[queue_id]
ReadMessage	0x32	queue_id, sender_id, mode
ReadMessageResponse	0x33	array[Message]
ReadResponse	0x34	queue_id, context

Table 3.6: Supported packets in *Telesto*. By convention an odd `method_id` indicates client to server communication while even values are server to client communication. Queue and Message objects in the payload include all fields stored in the database (see section 3.1). The `mode` in the ReadMessage packet is used to indicate whether the oldest message or the one with the highest priority should be served.

2. Parsing the request packet
3. Executing the according database action
4. Sending back a response

Using asynchronous Java `nio`[8], it is possible to handle a lot of concurrent connections to multiple clients simultaneously in an efficient manner. A single dispatcher thread handles new incoming connections and data by putting the clients into a FIFO queue which is continuously worked off by multiple worker threads. The actual parsing, database action and response sending is done by a worker rather than the dispatcher in order to reduce the load on the dispatcher.

In order to interact with the database, a database connection pool is used with a limited number of connections. Workers can request a connection from this pool, execute their queries and then put the connection back for other workers to use.

Figure 3.2 shows an overview of the three main parts in the middleware; namely the dispatcher, the worker threads and the database connection pool.

It is important to note, that connections to clients are never closed by the middleware (unless on shutdown). This first improves the delay of the system because no new TCP connection establishment is necessary for each request and second it allows to store the client information together with the connection so it is never necessary to send the `client_id` to the middleware again after the initial identification. This is the reason, why every client is first only allowed to request a limited set of services because many of them require identification. These services are namely the client registration and identification, and the pinging system.

3.4 Client

Telesto offers a simple interface for clients that want to use the system. The actual public Application Programming Interface (API) consists of one simple class `TelestoClient` with all the offered functionality. It is as easy as creating a new instance and then start calling functions to actually use *Telesto*.

By design, a client is only allowed to do further actions if he either registered itself as a new client or identified itself using his client id. This means, the first API call has to be to the `connect()` or method supplying either an existing `client_id` or both a new name and the mode of the client. (or `ping()` which is always allowed)

[8] Java Documentation: java.nio
Available at: http://docs.oracle.com/javase/7/docs/api/java/nio/package-summary.html [Accessed November 15, 2013]

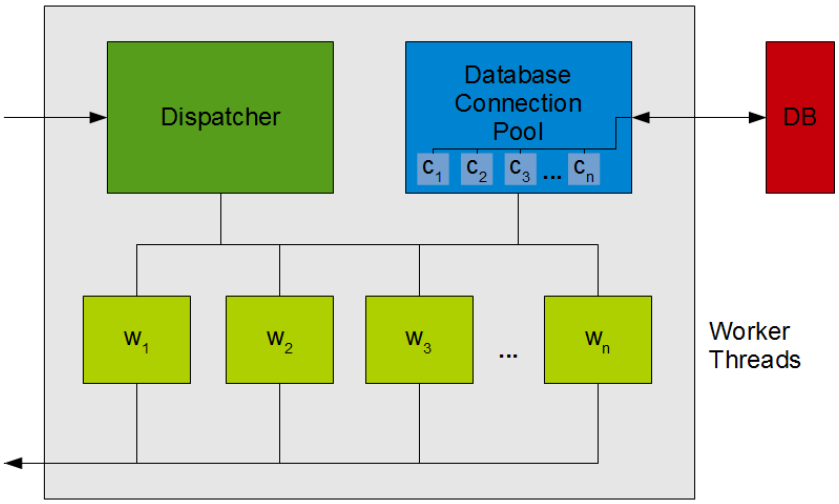

Figure 3.2: Basic setup of a middleware instance including the dispatcher, multiple worker threads and a database connection pool.

The API works in a synchronous way and has some blocking functions that retry an operation with a certain (configurable) delay until successful. An example of this is requesting a message from a queue, which blocks until a message for the client is successfully read.

It would be possible to actually build a client implementation that is asynchronous since the middleware and the used protocol support that feature. However we went without such an implementation as the testing of the system is in many cases much easier when using synchronous clients.

Chapter 4

Implementation

This chapter gives more detailed overview of some decisions made during the implementation phase of *Telesto*. This is a good starting point before getting involved with the code base of the system as it gives the necessary orientation and overview.

The whole code base is organized into six java packages that resemble the different parts of *Telesto*:

ch.ethz.syslab.telesto.client
: The *Telesto* client including the API and multiple specific implementations for different tests in the subpackage `test`.

ch.ethz.syslab.telesto.common
: All the parts that are shared between the middleware and the client implementation with for example model classes for all entities, the configuration and the protocol.

ch.ethz.syslab.telesto.console
: The implementation of the management console displaying all important information of the system.

ch.ethz.syslab.telesto.profile
: Classes specifically used for profiling and benchmark logging. These are used by both the client and the middleware.

ch.ethz.syslab.telesto.server
: The *Telesto* middleware with all database related functionality in the subpackage `db`, the dispatcher and worker thread implementations in the subpackage `network` and the protocol handlers in the subpackage `controller`.

ch.ethz.syslab.telesto.test
: An extensive test suite containing jUnit tests for many different parts of the system.

We chose to develop *Telesto* using *Eclipse (Version 4.3.1 Kepler)*[1] as integrated development environment (IDE) and *git*[2] with a (private) *Github*[3] repository as version control and source code management system.

4.1 Networking

In order to make networking efficient and fast, it is essential that involved parts of the system are never a bottleneck to the entire system performance.

The crucial part of this is optimizing the load on the dispatcher (i.e. an instance of the class `ConnectionHandler`) and limiting the overhead of distributing the tasks over all the workers (i.e. instances of `DataHandler`). This is why *Telesto* uses a `ArrayBlockingQueue<Connection>` to manage incoming requests that can then be processed by worker threads in a first-in-first-out (FIFO) manner. Since this data structure is backed by a simple array, runtime for inserting and removing from the queue always stays $\mathcal{O}(1)$.

The drawback of this implementation is that the queue has a limited size that is initially set and cannot be changed during execution. This however is not a problem because with the synchronous I/O in the clients, where they always wait for a response before requesting another service, the maximum number of connections in the queue should never actually grow larger than the number of clients. Therefore it is sufficient to ensure that the number of clients is always lower than the queue size.

In a scenario where clients would asynchronously send many requests simultaneously or the number of clients is much higher than the queue size, it becomes possible, that the dispatcher becomes blocked when trying to put a connection into the queue because it is already full. If the system is expected to run under such circumstances, then it would be necessary to improve the implementation for this special workload e.g. by dynamically replacing the queue with an other larger one or by just rejecting all incoming requests while the queue is full.

However in the settings where *Telesto* was tested in this was not critical and the chosen implementation proofed to be fast enough to never become a bottleneck to the whole system.

[1] Eclipse website
Available at: http://eclipse.org/ [Accessed November 15, 2013]

[2] Official git website
Available at: http://git-scm.com/ [Accessed November 15, 2013]

[3] Github website
Available at: https://github.com/ [Accessed November 15, 2013]

4.1.1 Packet parsing

A special part of *Telesto* is its binary packet format (see section 3.2) which requires special handling of the incoming data for every method supported. For this purpose, the package `ch.ethz.syslab.telesto.common.protocol`, contains one class for every packet with the following components (as specified in the abstract class `Packet`):

1. All the parameters that are part of the packet as fields
2. The method `emit()` to write the fields to a `ByteBuffer`
3. The method `parse()` to build a packet instance from a `ByteBuffer`

Because it is a rather ungrateful task to write about 30 packet classes and according handlers that share many lines of code, we built an automated *Python*[4] script to generate all packet and packet handling classes using *jinja2*[5] templates and a small configuration specifying what packets exist with what fields.

The code for this tasks is available under `tools/protocol/` with the packet specification in the source file `messages.py`. The initial effort to build this small tool was very much worth it because it makes actually changing the protocol or the methods and implementation of a class very easy because all handler and packet classes can be regenerated using very little effort.

4.1.2 Packet handling

Each worker parses incoming data using the static method `create(ByteBuffer)` in the abstract `Packet` class which creates an instance of the right packet class using the method id at the very beginning of the packet data and lets this instance's `parse(ByteBuffer)` method handle the parsing of the individual fields.

The built packet instance containing all information that was sent over the network is then handled by a `ProtocolHandler` that contains a method `handle()` that is overloaded to take every existing packet as input and returning a packet as response to be sent back to the client. This class is actually also generated by the above mentioned automated python script to contain all the necessary methods.

For the overloading to correctly work with the dynamic typing and instantiation of the packet classes, it is necessary to use the visitor pattern on the packet classes to let them call the handle method for themself.

[4] Python Website
Available at: http://python.org/ [Accessed November 15, 2013]

[5] Jinja2 Documentation
Available at: http://jinja.pocoo.org/docs/ [Accessed November 15, 2013]

*Telesto*contains two different implementations for the `ProtocolHandler`:

ServerAuthenticationProtocolHandler
: The protocol handler for all the packets that are allowed to send before authentication. An instance of this class is switched out by one of `ServerProtocolHandler` as soon as authentication is completed.

ServerProtocolHandler
: The actual protocol handler that handles all packets that are sent to a middleware instance. Each `handle()` methods contains the necessary logic to query the database and build a response for the client.

In order to handle packets that should not be sent to the server (i.e. response packets), the abstract `ProtocolHandler` just throws an appropriate exception that is converted into an `ErrorPacket` upon catching in the worker to notify the client of his misbehaving.

4.2 Database

As explained in section 3.1, *Telesto* uses stored procedures for all database interaction. Using the *PostgreSQL JDBC* driver, it is rather easy to make calls to such a procedure but a lot of code lines goes into error handling, statement generation (i.e. setting parameters) and reading out the response data and build entity instances.

This is why we built a heavy abstraction around the *JDBC* driver that is able to share most of the code and takes care of statement generation, error handling and directly returns appropriate entity objects (or lists) to be used in the response packets. Using this abstraction layer, it is possible to initiate all database action using a single line of code in the `ProtocolHandler` implementation.

As seen in section 3.1.3, there are only five different return value types for all procedures:

1. Table of Clients
2. Table of Queues
3. Table of Messages
4. Set of Integers
5. Single Integer

Therefore our abstraction contains an `enum` in the package `db.procedure` for each of the first three where all available stored procedure are enumerated

according to their return value. Procedures returning integers are included in the `enum` that best categorizes them.

By offering a simple method on the `Database` class (which does all database related work) for each of the above stated types, it is possible to directly return the according instances and control (to some extent) that only procedures really producing that output type can be called. The signature of such a function looks like this:

```
public List<Message> callMessageProcedure(MessageProcedure proc,
                     Object...  arguments)
```

Each `enum` value contains the necessary information to assign the right types to the arguments, the return type and the name of the stored procedure in the database.

4.2.1 Connection Pool

For the implementation of the database connection pool, *Telesto* completely relies on the *PostgreSQL JDBC* driver which offers a complete implementation in the `PGPoolingDataSource`[6] class.

4.3 Client

Table 4.1 shows a brief overview of the offered functionality by the *Telesto* client API. A more detailed description of each method is available inside the class `ch.ethz.syslab.telesto.client.TelestoClient` as *javadoc*[7].

The networking of the client is mostly shared with the one in the middleware but does not need any special `ProtocolHandler` because it doesn't directly use any content of the response packets. Those are directly handled in the `TelestoClient` where the returned packet is casted to the right response packet before the information is extracted and returned to the user calling the API function.

[6] PGPoolingDataSource in the PostgreSQL JDBC driver documentation
Available at: http://jdbc.postgresql.org/documentation/publicapi/org/postgresql/ds/PGPoolingDataSource.html [Accessed November 15, 2013]

[7] Oracle: How to Write Doc Comments for the Javadoc Tool
Available at: http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html [Accessed November 15, 2013]

Method	Parameters	Return Value	Description
	Setup		
ping		round trip time	ping the middleware
connect	clientName, clientMode	Client	connect to the middleware as new client
connect	clientId	Client	connect to the middleware as existing client
	Queues		
createQueue	queueName	Queue	create a new queue
deleteQueue	queueId		delete a queue
getQueueByName	queueName	Queue	get a queue by its name
getQueueById	queueId	Queue	get a queue by its id
getQueues		List<Queue>	get all queues
getActiveQueues		List<Queue>	get all queues with messages for this client
readMessages	queueId	List<Message>	get all messages from a queue
	Messages		
putMessage	Message		insert a new message
putMessages	Message, queueId[]		insert a new message into multiple queues
sendRequestResponseMessage	Message	Message	send request and retrieve response
retrieveMessage	queueId	Message	get message from queue by priority
retrieveMessage	queueId, readMode	Message	get message from queue by the indicated readMode
retrieveMessage	queueId, senderId, readMode	Message	get message from specific sender from queue by the indicated readMode
retrieveMessage	queueId, senderId, readMode	Message	get message from specific sender from queue by the indicated readMode

Table 4.1: Public methods on the `TelestoClient` class. The class is also fully documented using *javadoc* in order to allow for easy usage.

4.4 Error Handling

Other than just handling errors within a single application, *Telesto* has to handle errors that occur in an other part of the system appropriately. This is the case if something goes wrong in the middleware which makes it necessary to inform the client that his request could not be executed. In order to do this, there exists a special `ErrorPacket` (as introduced in section 3.2) that contains an error number and an error message.

Possible error codes are stored in the enum `ErrorType` and are listed in table 4.2 together with a short description.

ErrorType	**Description**
Unexpected	
INTERNAL_ERROR	A typically unexpected error
IO_ERROR	An error during network interaction
Client Error	
UNEXPECTED_PACKET	A packet that could not be handled was received
UNIQUE_CONSTRAINT	A unique constraint was violated
QUEUE_NAME_NOT_UNIQUE	The specified `queue_name` is not unique
CLIENT_NAME_NOT_UNIQUE	The specified `client_name` is not unique
CLIENT_NOT_EXISTING	The specified client does not exist
QUEUE_NOT_EXISTING	The specified queue does not exist
REQUIRED_PARAMETER_MISSING	A required parameter in a packet is missing
CLIENT_MODE_PERMISSION_VIOLATION	The `ClientMode` prohibits the use of the called service
Client Notification	
NO_ACTIVE_QUEUES_EXISTING	No queues contains messages for the client
NO_QUEUES_EXISTING	There exist absolutely no queues
NO_MESSAGES_IN_QUEUE	There are no messages in the specified queue
NO_MESSAGES_RETRIEVED	A receive query did not yield any messages

Table 4.2: Error codes in the system. Specified in the class `ErrorType`.

Additionally two special exceptions were introduced to handle errors within the system in a simple way:

1. `ProcessingException` on the client
2. `PacketProcessingException` on the middleware

Both exceptions allow to specify an `ErrorType` and can include nested exceptions.

When a `PacketProcessingException` is thrown in the middleware, it is automatically caught and sent to the client in an `ErrorPacket` where it is again

processed into a `ProcessingException` which is handled accordingly or thrown to the user.

4.4.1 Logging

Telesto does not only have a solid error and exception system but also uses logging in many places to allow for deep investigation of possible problems. For this purpose we introduced the statically accessed `Log` class that operates as a facade to `java.util.logging`[8] . The use of the built in *Java* logging interface, allows to configure logging parameters in a very convenient and powerful way.

4.5 Configuration

To configure client and middleware instances, *Telesto* uses both a configuration file and command line parameters. The following two sections briefly describe how both work.

4.5.1 Configuration File

Both the client and the middleware use the same global `config.properties` file to read the configuration for the current run. If no such file is present in the execution directory, then a default version is extracted from within the compiled `.jar` file.

This configuration file is a simple *Java Properties* file[9] containing parameters like the middleware address or database connection information. Details about these parameters and how they were varied over multiple tests, are explained in section 5.3.

4.5.2 Command Line Parameters

Even if the code base is organized in a way that would allow building multiple `.jar` files for client and middleware, we decided to just build one and use command line parameters in order to determine which part to start. This makes it more convenient to do tests because only one `.jar` file has to be distributed over many machines.

[8] Java Documentation: java.util.logging
Available at: http://docs.oracle.com/javase/7/docs/api/java/util/logging/package-summary.html [Accessed November 15, 2013]

[9] Java Documentation: java.util.Properties
Available at: http://docs.oracle.com/javase/7/docs/api/java/util/Properties.html [Accessed November 15, 2013]

The first parameter always determines which tier should be started. The values “MW” and “CL” stand for middleware and client respectively. Only the client takes some additional arguments to control the following parameters:

name The `client_name` to be used (registers new client)

id The `client_id` (identifies as existing client)

test An identifier for the test to run (see section section 5.2)

These additional arguments are always supplied as key-value pairs separated by a space character. Of the name and the client id only one has to be specified.

Chapter 5

Evaluation and Analysis

This chapter gives an overview of the experiments conducted with *Telesto* and their results and implications.

5.1 Setup

We conducted all experiments on the *Amazon Elastic Compute Cloud*[1] (EC2). One large general purpose instance was used for each the database, the middleware and the clients. The experiments were controlled by a fourth instance which was allowed shell access on all of the other ones. All instances were placed on the same sub-network within *Amazon*'s network in order to minimize network latency.

The experiments were controlled by a single configuration file which was automatically distributed by the controlling instance to the clients and the middleware for each test, using a `bash` script. Given a number of these configuration files the controlling instance was able to run multiple experiments completely autonomous.

The clients and the middleware are built to measure the duration of each phase a request goes through and write the results into a log file which is collected by the controlling instance and can then be processed.

5.2 Client Types

The system was tested using three different kind of clients to simulate a practical workload that includes multiple different applications of a message passing system:

[1] Amazon EC2 Website
Available at: http://aws.amazon.com/ec2/ [Accessed November 15, 2013]

One Way
These clients send a message containing a counter value to a random other client of the same type and then wait for a message for them. Upon receiving a message they increase the counter and again send it to a random other client. This was implemented using a single queue for all one way clients.

Request Response Pair
These clients act in pairs as client and server. They basically send messages back and forth between them both increasing a counter which is part of the message text. All these clients again use only one single queue.

Request Response Pool
Contrary to the clients acting in pairs, these clients act in two equally sized pools of clients and servers. Clients make a request which is answered by any client of the server pool. All requests and responses were stored in the same queue.

All tests described below were conducted using the distribution shown in table 5.1.

Client Type	Fraction
One Way	5/9
Request Response Pair	2/9
Request Response Pool	2/9

Table 5.1: Client type distribution for all tests

5.3 Parameters

All experiments were run with the same basic parameters, save the ones being experimented on. Before each experiment the database was emptied to get consistent results. The default parameters are listed in table 5.2.

Parameter	Value
Running time	10 minutes
Client	
Number of instances	90
Retry delay	100ms
Network buffer size	32KB
Middleware	
Number of instances	1
Network buffer size	32KB
Worker pool size	25
Database pool size	25

Table 5.2: Default parameters for experiments

5.4 Metrics

Middleware response time
Time the client waits for a response after sending a request to the server.

Packet parsing duration
Time the middleware needs to parse a request from a client.

Database response time
Time the middleware waits for a response from the database after making a request.

Packet emitting duration
Time the middleware needs to construct and send a response to the client.

Middleware worker downtime
Time a worker instance of the middleware waits before receiving a client request to handle.

Packet throughput
Number of network packets sent by the middleware.

Message throughput
Number of messages successfully received by a client.

5.5 Experiments

The experiments were generally run for 10 or 5 minutes to make sure the relative confidence interval for a certainty of 95% is alwasy lower than 5%. We never experienced messages or packets being dropped during any of the conducted experiments.

5.5.1 Stability

To measure the stability of our system we ran it with the default configuration for 2 hours without interruption.

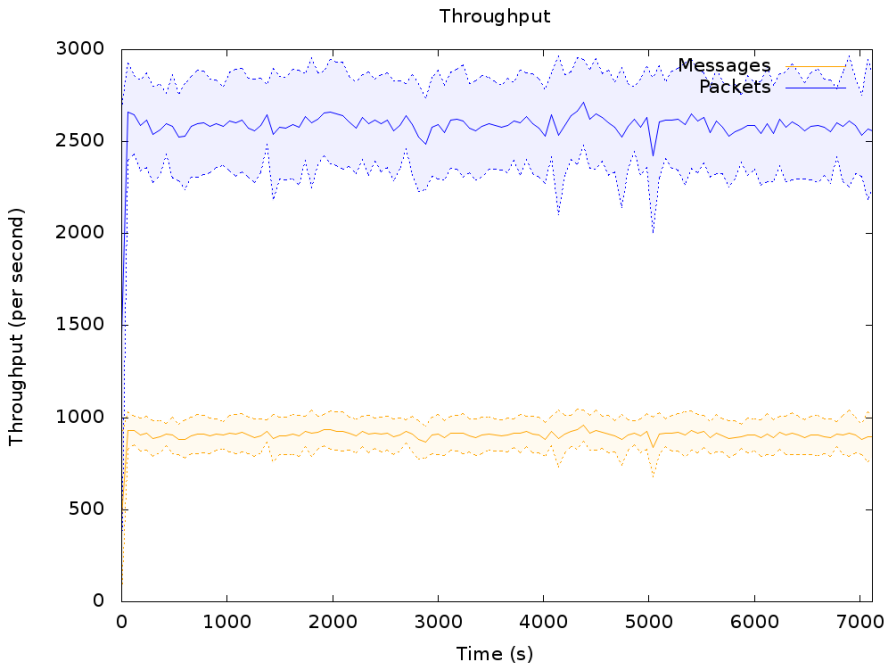

Figure 5.1: Throughput measured every second and averaged over an interval of one minute.

Figure 5.1 shows the message and packet throughput as measured by the clients over time. The throughput was measured every second and averaged over the interval of one minute in the shown figure.

It is clearly visible that our system is fairly stable over such a long period of time. The small variation seen especially in the packet throughput most likely originated in the unpredictable network environment of the *Amazon Cloud*. It is further visible that the packet throughput is roughly 3 times as big as the message throughput, meaning that only around one sixth of message requests by

clients returned no messages.

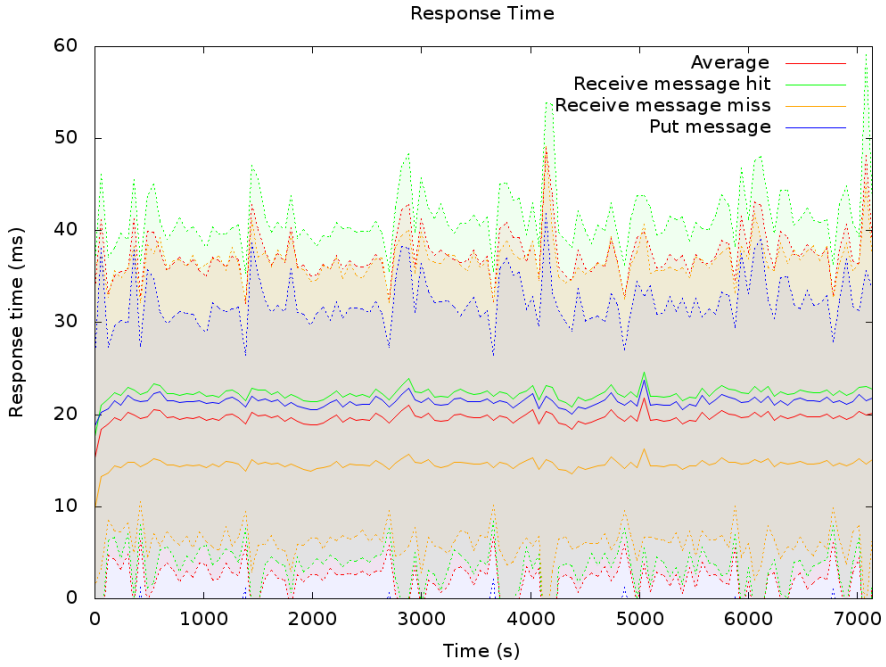

Figure 5.2: Middleware response time for different request types, averaged over an interval of one minute.

Figure 5.2 shows the middleware response time as measured by the clients over time. The response time is averaged over the interval of one minute for each of the shown curves.

The averaged middleware response time seems fairly stable over long periods of time, the individual measurements however fluctuate quite heavily as can be seen by the relatively large standard deviation. This effect is not very concerning as the absolute value of the standard deviation (generally less than 25ms) is quite small and can easily be caused by interference of other tasks run on the same physical machine.

The different types of requests take roughly the same time to complete, message request misses being slightly faster than the rest. This is caused directly by the way the database operates, it neither has to write or read any complex data as the necessary information is already available in the index.

Figure 5.3 shows the distribution of the observed middleware response time during the two hour long experiment. As expected it roughly follows a normal distribution with no values being higher than one second.

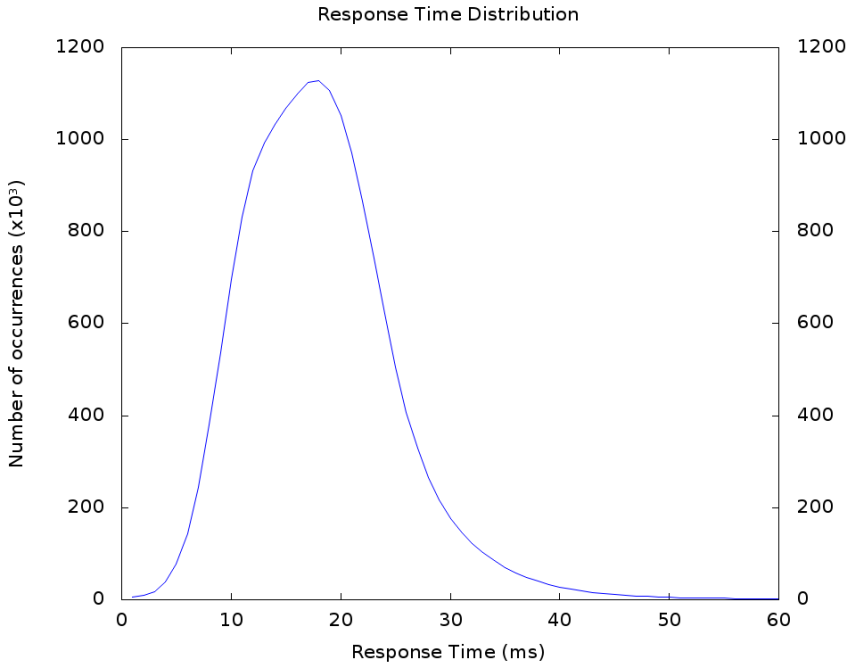

Figure 5.3: Middleware response time distribution measured in chunks of 1 millisecond.

5.5.2 Benchmarks

Phase	Time	Relative
Waiting for client	79 µs	0.82%
Parsing request	14 µs	0.15%
Waiting for database	9.574 µs	98.80%
Responding	22 µs	0.23%

Table 5.3: Time spent on various tasks by middleware workers

Table 5.3 shows the time spent by the middleware on various tasks while handline a request. It is clearly visible that most of the time, over 98%, is spent waiting for the database. From figure 5.2 we can also conclude that around half of the time a request takes is spent by the middleware and the other half is needed for networking.

5.5.3 Scalability

We conducted various experiments to figure out the optimal parameters for the system and see how well it scales with higher load.

Worker Thread Count

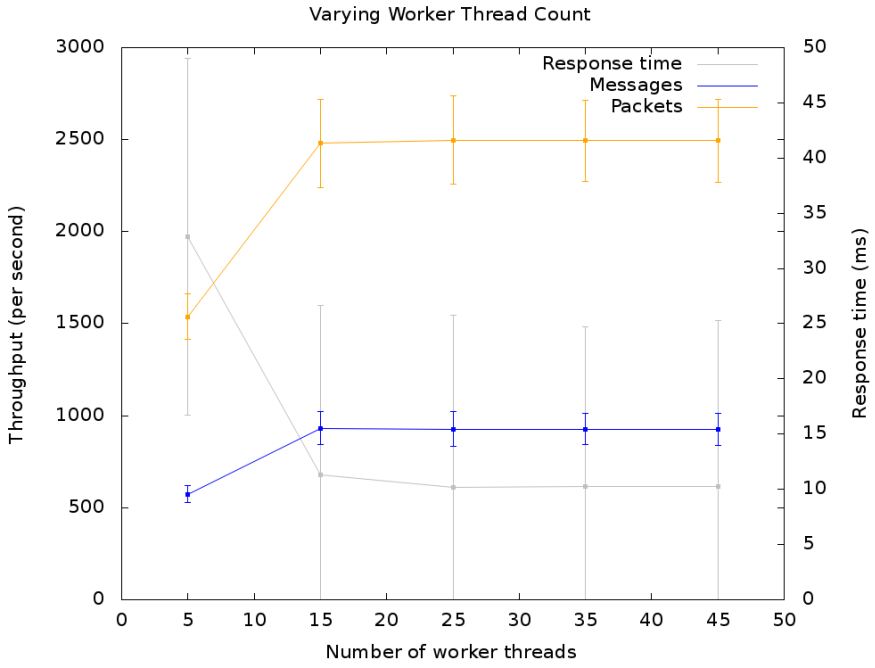

Figure 5.4: Throughput and response time for varying number of worker instances

Figure 5.4 shows the system with a varying number of worker threads but the same number of database connections (25). It can be seen that increasing the worker thread count beyond the number of database connections clearly doesn't increase throughput, it can even be lowered and the system performs equally well.

Database Connection Count

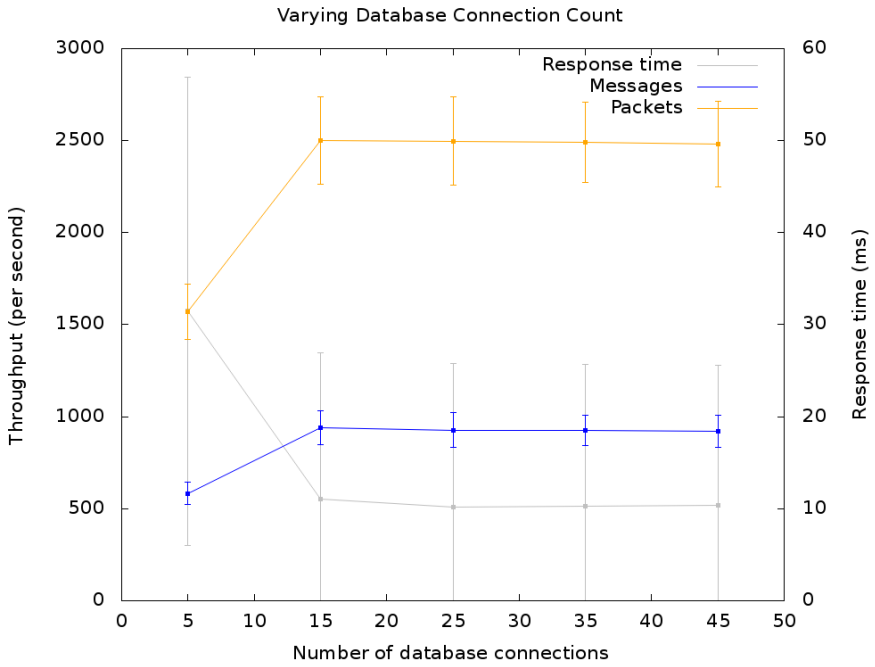

Figure 5.5: Throughput and response time for varying number of database connections

Figure 5.5 shows the performance of the system with various numbers of database connections and a static number of worker threads (25). It can again be seen that the optimal performance is already reached with only 15 connections. The fact that the performance doesn't increase until the number of worker threads is reached shows that the database can in fact not process requests any faster than our middleware is able to send over 15 connections.

Combined Database Connection and Worker Thread count

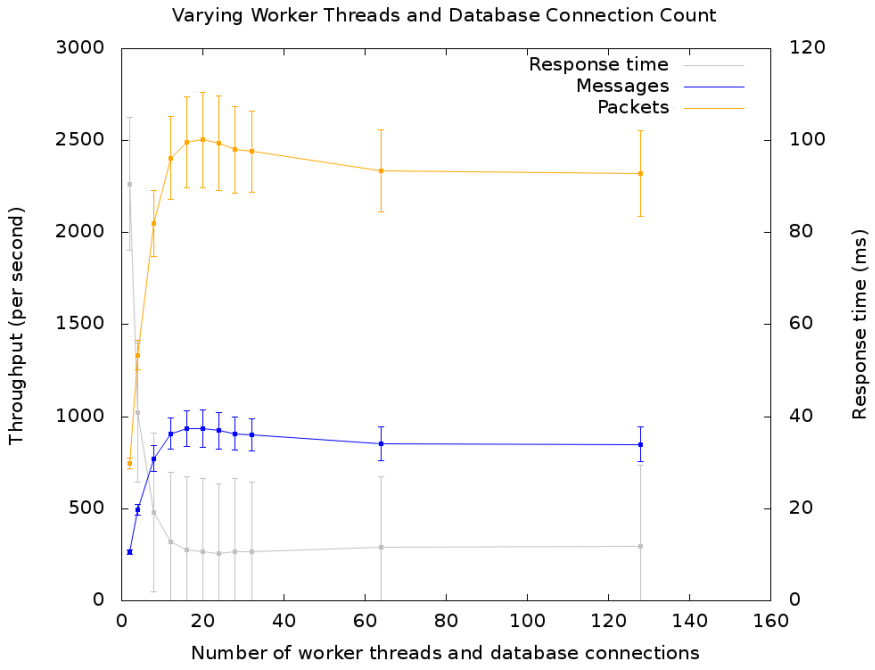

Figure 5.6: Throughput and response time for varying number of worker instances and database connections

Since worker thread and database connection count seem to be linked we increased them together as shown in figure 5.6. Again, the best performance is achieved around a value of 20 and it decreases slowly as more workers and connections are added. The steep slope to the left of the optimal value is the range where the database still has free resources and the middleware can't make requests fast enough.

Client Retry Delay

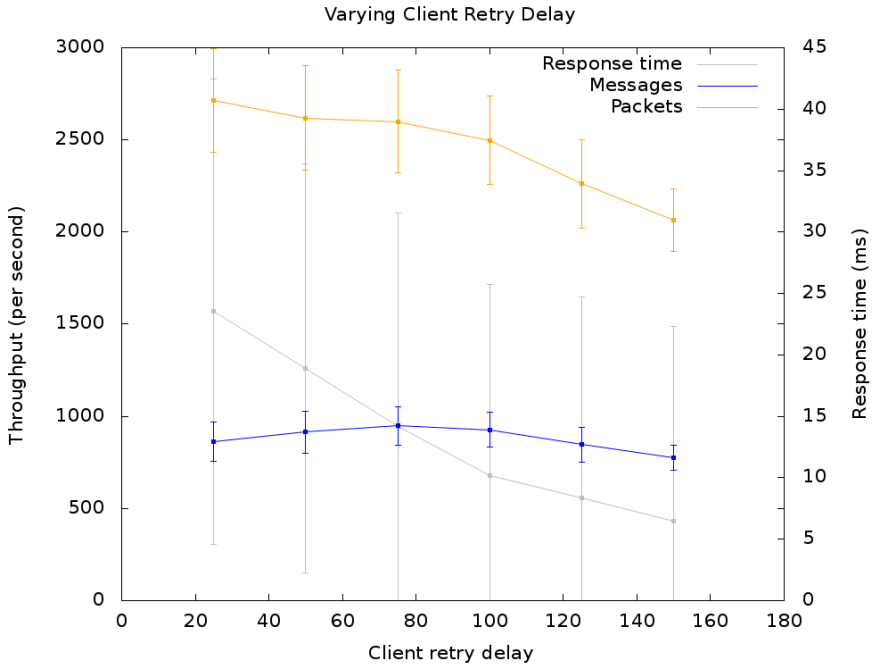

Figure 5.7: Throughput and response time for varying client retry delay

In this experiment we varied the amount of time a client waits before retrying to retrieve a message. As seen in figure 5.7 this parameter doesn't have a very huge impact on the message throughput of the system, it does however decreases the overall packet count (and therefore throughput) because less packets are wasted when querying for nonexisting messages. A higher retry delay also causes the average response time to decrease as the middleware can process the lower number of packets much faster. The ideal value in regard to message throughput appears to be between 80 and 100 ms.

Client Count

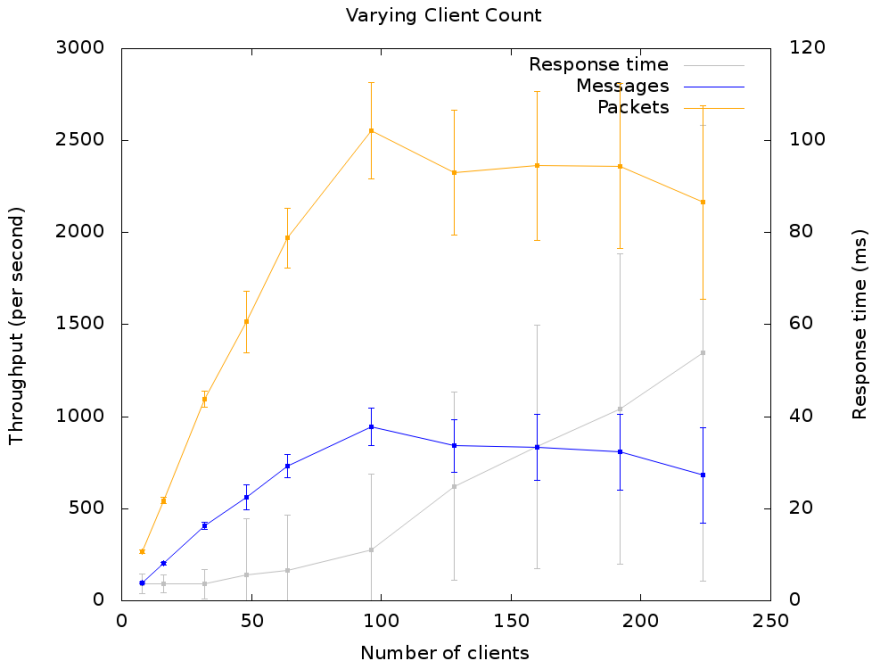

Figure 5.8: Throughput and response time for varying number of clients

When running the system with an increased number of clients we noticed a small but clear drop in message throughput as can be seen in the right half of figure 5.8. When using less clients than the usual 90 the system was underloaded which also resulted in much lower throughput of both messages and network packets. The response time steadily rises with the growing number of clients as expected. It's also noticeable how the system behaves much more predictably while underloaded than overloaded as can be seen by the very small standard deviation for low numbers of clients.

Middleware Count

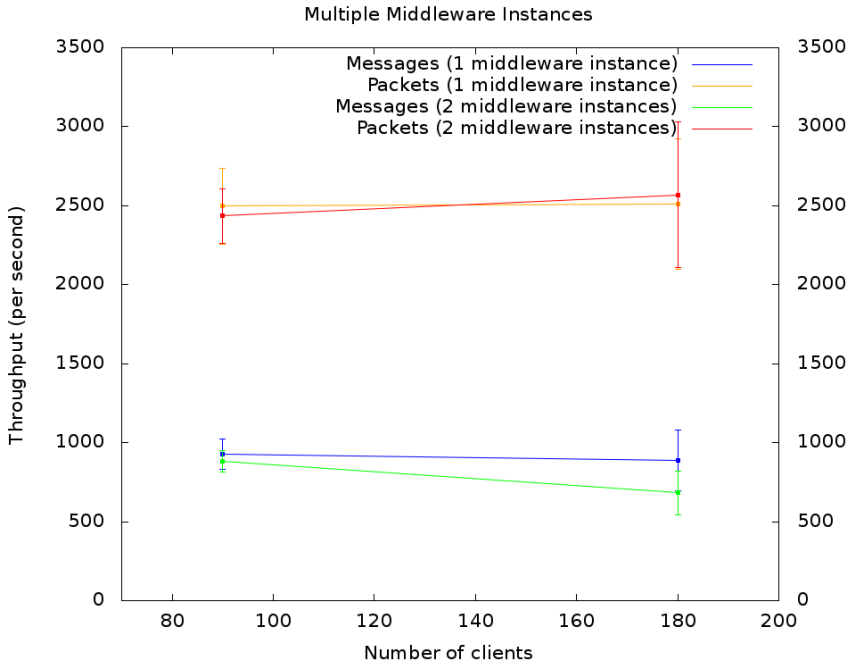

Figure 5.9: Throughput and response time for varying number of clients with both one and two middleware instances

We conducted most of our experiments on a single middleware instance as it was already evident that not the middleware but the database was limiting the system performance. The two measurements shown in figure 5.9 proof that adding more middleware instances does indeed not improve performance. The second middleware was run on the same *EC2* instance as the first one, but we payed close attention to the usage of memory, CPU and the network to make sure they wouldn't interfere with each other.

5.6 System Characteristics

Our system generally performed very well and met our expectations. A possibly limiting factory is the single thread handling new connections and incoming data. Even though this proofed to be an effective architecture in our experiments, it could certainly limit the network throughput in situations with a much higher load. The worker threads also share a single request queue which is used extensively but seems to handle almost any number of workers and requests very well. The queue is backed by a fixed array which is slightly faster than a linked queue or similar implementation and should not limit the overall speed of the

system. Parsing and emitting packets is very cheap as can be seen in table 5.3 and the binary network protocol also minimizes the amount of data sent over the network. Overall wi think our system could handle a much higher load than id did in any of our experiments as we have seen in tests run without a database.

Chapter 6

Conclusion

As discussed in chapter 5, the major bottleneck in the performance of *Telesto* is the database. It is important to note, that we did no experiments to optimize the database configuration. Most likely a more optimal configuration of the database would have improved the overall performance of the system but when looking at the relative time consumption for tasks by middleware workers (table 5.3) it is very unlikely that an other part would actually become the new bottleneck.

When looking at the completed *Telesto* implementation we are confident to have built a scalable and well performing distributed message passing system that fulfils the given requirements. The number of tests and traces with different metrics we executed, is sufficient to allow for reasonable conclusions and optimization of the most important system and workload parameters.

In the end we learned a lot about testing and benchmarking an entire system in a profound way and drawing comprehensible conclusions. Although both of us already had experience in building a large system from scratch, we got a lot of interesting insights in how it could look like when building and especially evaluating a system in the industry, which might not even be that far away...

Bibliography

[1] Alonso, G.: Advanced Systems Lab, Project and Course Description. Systems Group, ETH Zurich (Fall Semester 2013)