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.

TODO final findings

Contents

Chapter 1

Introduction

Chapter 2

Goals

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 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]

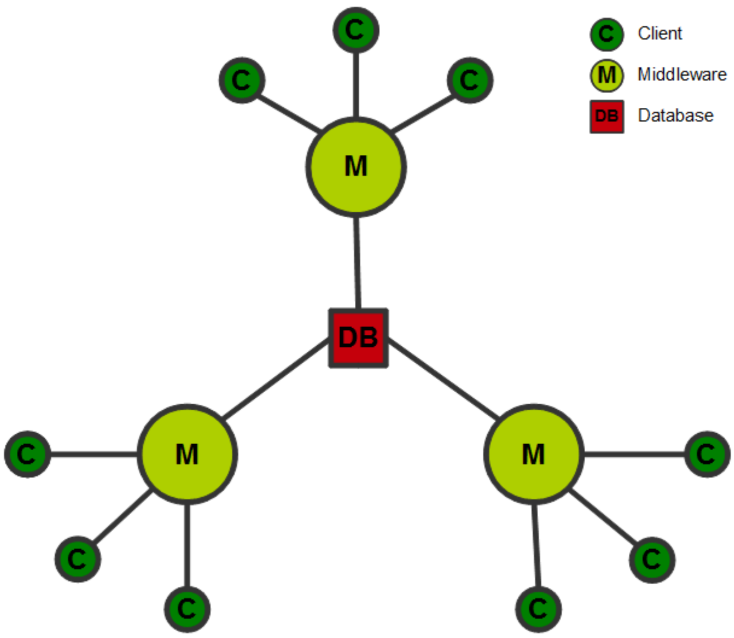

Figure 3.1: Sample architecture diagram of *Telesto* with 3 middlewares each serving 3 clients and one central database.

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.

Figure TODO add protocol figure shows the basic structure of such a packet.

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
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]

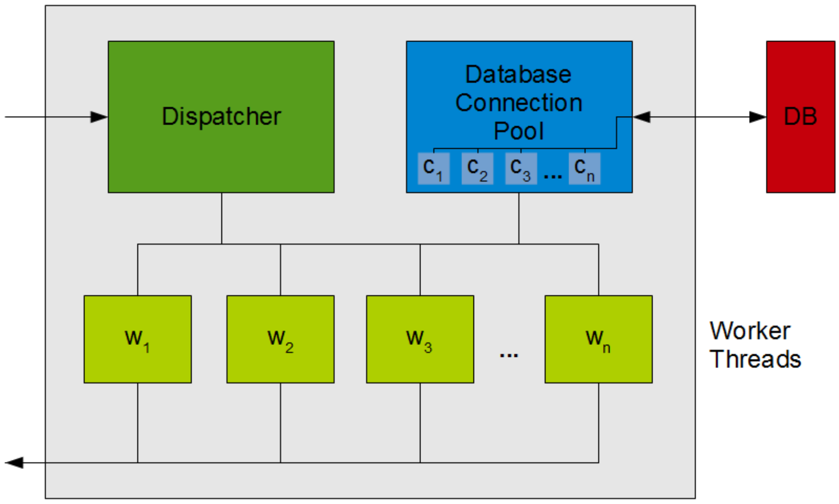

Figure 3.2: Basic setup of a middleware instance including the dispatcher, multiple worker threads and a database connection pool.

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].

4.4 Error Handling

4.5 Configuration

4.6 Profiling

[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.

Chapter 5

Evaluation and Analysis

5.1 Setup

5.2 Parameters

5.3 Metrics

5.4 Tests

5.4.1 Scalability

5.4.2 Stability

jukefox-explore-album-list-1.png

Figure 5.1: The album list containing suggested albums.

Figure 5.2: The jukefox music streaming view.

Chapter 6

Future Work

6.1 Possible Improvements

Chapter 7

Conclusion

Appendix A

Appendix Chapter

A.1 Database Structure