ETH ZURICH | ADVANCED SYSTEMS LABORATORY
AUTUMN SEMESTER 2013

FTAB MESSAGING

A DISTRIBUTED MESSAGING SYSTEM

Report for Milestone 1: System Design & Experimentation

Authored by:

Diego Ballesteros

Jean-Pierre Smith

November 15, 2013

Contents

Contents 1

1. Introduction 4

I. System Design & Implementation 5

2. System Architecture Overview 6
- 2.1. Component Layout . . . 6
- 2.2. Users & Main Functions . . . 6
  - 2.2.1. Transactor . . . 6
  - 2.2.2. Observer . . . 7
- 2.3. System Logic Structure . . . 7

3. Interfaces & Communication 9
- 3.1. Client Interface . . . 9
- 3.2. Middleware Interface . . . 12
  - 3.2.1. Server Interface & Messages . . . 12
  - 3.2.2. Defined Exceptions . . . 15
  - 3.2.3. Communication Protocol . . . 16
- 3.3. Database Interface . . . 23
  - 3.3.1. Database Entities . . . 23
  - 3.3.2. General Database Operations . . . 24
  - 3.3.3. Client Operations . . . 25
  - 3.3.4. Message Operations . . . 26
  - 3.3.5. Queue Operations . . . 30
  - 3.3.6. Database Exceptions . . . 31

4. System Design 32
- 4.1. Clients . . . 32
  - 4.1.1. Client Class . . . 32
  - 4.1.2. ServerRPC . . . 32
  - 4.1.3. ClientShell . . . 32
- 4.2. Middleware . . . 35
  - 4.2.1. ServerManager . . . 35
  - 4.2.2. DBConnectionDispatcher . . . 35
  - 4.2.3. ClientConnection . . . 35
  - 4.2.4. MessagingWorker . . . 35

4.3. Management Console 38
4.4. Database 38
4.4.1. Clients Table : client 38
4.4.2. Queues Table : queue 40
4.4.3. Messages Table : message 40
4.4.4. Messages ↔ Queues Relation : msg_queue_assoc 41

5. Sequence Diagrams **42**
5.1. Client Connection 42
5.2. Queue Management 42
5.3. Sending Messages 42
5.4. Retrieving Messages 42

II. Experiments **48**

6. Introduction **49**
6.1. Definitions 49
6.1.1. System under test 49
6.1.2. Response time (RT) 49
6.1.3. Throughput (TH) 50
6.2. Tools & Procedures 50

7. Initial exploration **52**
7.1. Experimental setup 52
7.2. Experiment 1 - Exploring the system 52
7.2.1. Description 52
7.2.2. Results 53
7.2.3. Analysis 53
7.3. Experiment 2 - Exploring server parameters 56
7.3.1. Description 56
7.3.2. Results 57
7.3.3. Analysis 58
7.4. Experiment 3 - Exploring the database size impact 62
7.4.1. Description 62
7.4.2. Results 62
7.4.3. Analysis 62
7.5. Experiment 4 - Running under less than full load 64
7.5.1. Description 64
7.5.2. Results 65
7.5.3. Analysis 65
7.6. Experiment 5 - Revisiting database size impact 68
7.6.1. Description 68
7.6.2. Results 68
7.6.3. Analysis 70

8. Experiments and Benchmarks **71**
8.1. Experimental setup 71
8.2. Experiment 6 - Varying the load 72
8.2.1. Description 72
8.2.2. Results and Analysis 73
8.3. Experiment 7 - Scaling the server 79
8.3.1. Description 79
8.3.2. Results and Analysis 80
8.4. Experiment 8 - System trace 84
8.4.1. Description 84
8.4.2. Results 85
8.4.3. Analysis 90

9. Conclusions **91**
9.1. Summary of the system performance 91
9.2. Proposed improvements 92

Appendices **94**

Appendix A. Management Console Screenshots **94**

1. Introduction

This report is divided into two parts. The first comprises the details of the system's design and its implementation, most of which relate to the first part of Milestone one, but which have been revisited and revised with the final details of our system. The second comprises of the details of the experiments that were conducted along with the results and analyses of the experiments.

Part I.

System Design & Implementation

2. System Architecture Overview

2.1. Component Layout

FTaB Messaging is a distributed three-tiered messaging system. As a messaging system it facilitates the sending of messages by clients to queues and the retrieval of messages that have been delivered. The major components of the messaging system can be seen in Figure 2.1 and consist of multiple systems each hosting multiple client processes, messaging servers and a database and its corresponding management console.

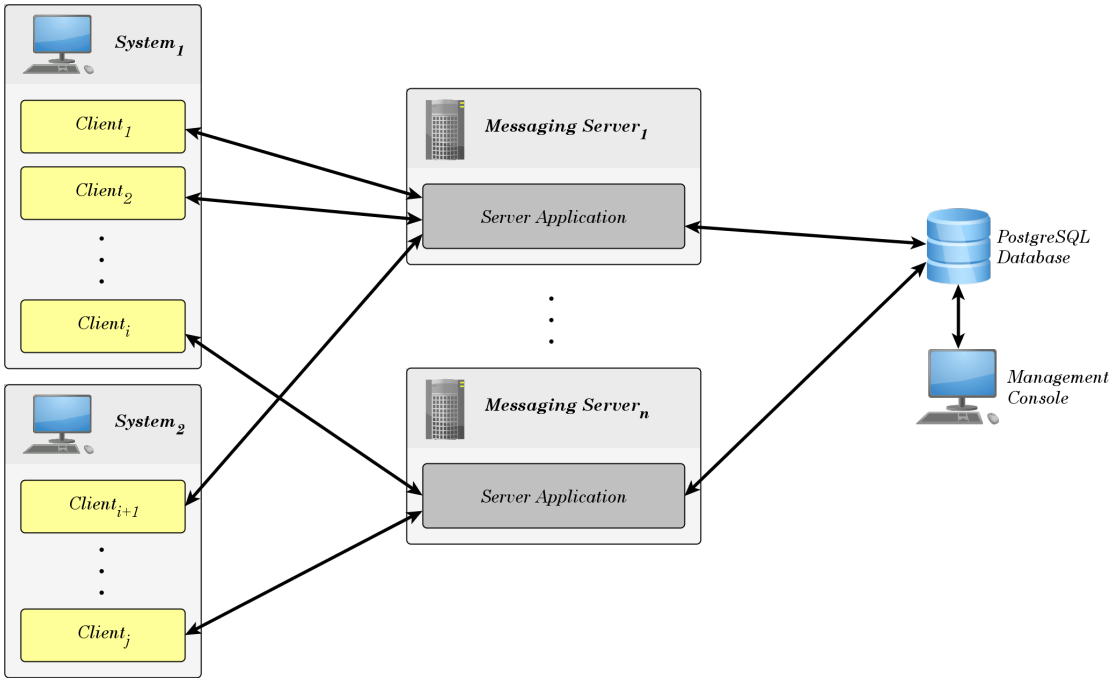

Figure 2.1.: Overview of system components

2.2. Users & Main Functions

The system has two main user types transactor and observer. The system is designed to support many concurrent transactors but only one observer.

2.2.1. Transactor

The transactor operates at the highest tier sending and receiving messages via the client. Transactors are implemented in the system both via a command line interface suitable

for a single user as well as by Jython scripts to facilitate experiments. The main functions that the transactor can initiate at the client are as follows.

- Send messages
  - as one-way or request-response messages
  - to one or more queues
  - with or without a receiver specified.
- Retrieve messages
  - by removing them from the queue
  - by either the earliest time queued or highest priority
  - destined for the transactor or with an unspecified receiver
  - with or without removing it from the queue.
- Manipulate queues
  - by deleting a queue by name
  - by creating a new queue with a unique name

2.2.2. Observer

The observer operates at the lowest tier pulling information from the database via the networked management console. The main functions that the transactor can initiate form the terminal are as follows.

- View queues and their contained messages
- View messages contained within a queue
- View an overview of the system with statistics such as the number of exiting queues and messages.

2.3. System Logic Structure

The system consists three tiers, each tier interacting with the tier directly below it as depicted in Figure 2.2. The presentation tier consists primarily of the client implementation encompassing the client itself and its usage of the server's remote procedure call API and the logging API. This tier communicates with the logic tier below it by means of a custom protocol detailed in subsection 3.2.3. The logic tier comprises primarily of the the server and its components, namely the server network service layer, network logic layer, data access layer and its usage of the logging API. Also present in this tier is the management console which also spans the presentation tier (as an observer may directly interact with it). The logic tier communicates with the data tier comprising of the PostgreSQL database by means of the standard PostgreSQL JDBC driver.

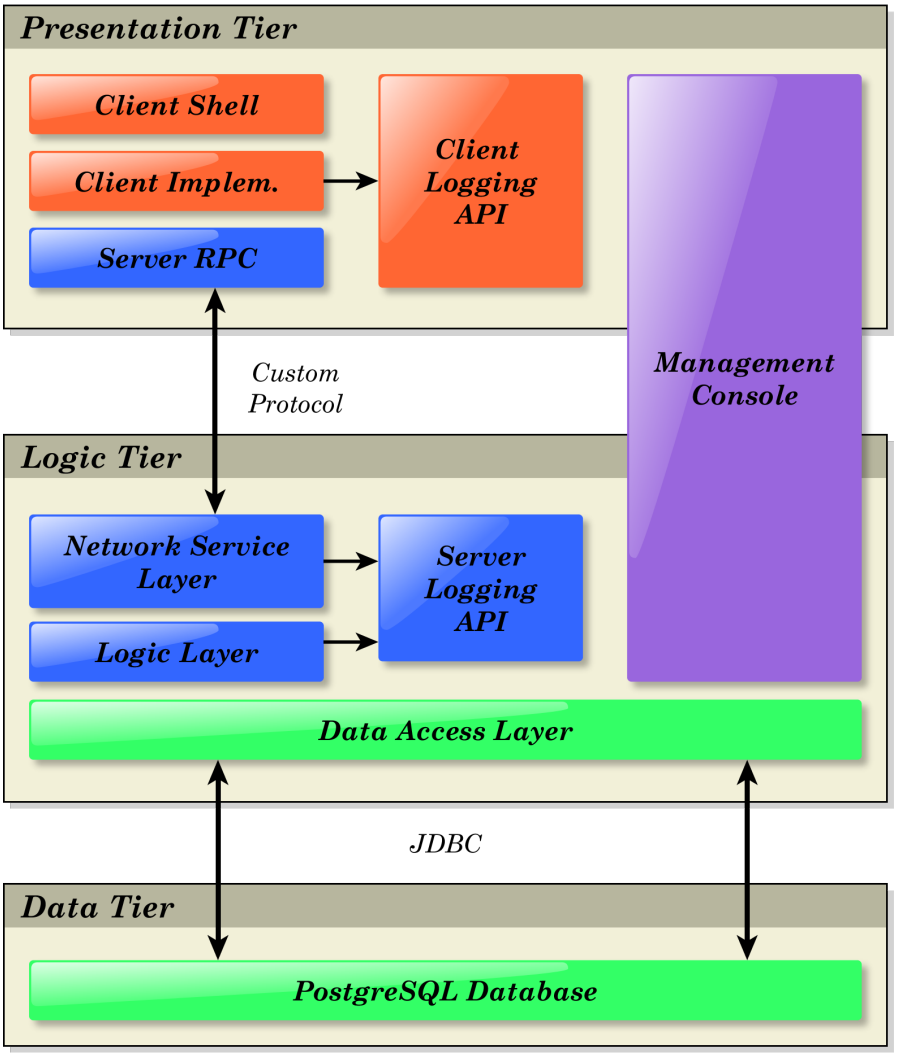

Figure 2.2.: Tiered arrangement of major components

3. Interfaces & Communication

Interaction between the different components of the system can be defined by the interfaces that provide details of the available method calls, the returned objects and exceptions of these calls, and the communication protocols that facilitate these calls. An overview of those used in the system is presented below.

3.1. Client Interface

This is the interface that is presented to the end user through either a shell or graphical user interface. It consists solely of the client class and its related methods.

org.ftab.client.Client

Clients represent entities that communicate through the messaging system by sending and receiving messages to and from queues.

METHODS		
Connect - *Initiates a connection from this client to a middleware server.*		
Parameter(s):	**String** serverIPv4	The IPv4 string representation or fully qualified domain name of the server the machine the middleware resides on.
	int port	The port on the remote machine that the middleware is listening on.
Returns:	**boolean**	**true** if the connection was successful, **false** otherwise
Throws:	FullServerException	If the server is not accepting any more connections due to full capacity.
	AlreadyOnlineException	If the client is marked in the system as being online.
Disconnect - *Disconnects the client from the server it is connected to.*		
Returns:	**boolean**	**true** if the disconnection was successful, **false** otherwise.
SendMessage - *Sends a message from this client to one or more queues with a designated receiver for the message.*		
Parameter(s):	**String** message	The text of the message to be sent.

	byte priority	The priority of the message, from 1 to 10, with 1 being the lowest and 10 the highest.
	int context	The context of the message
	String receiver	The username of the designated receiver of the message
	java.lang.String[] queues	The names of the queue or queues to which to send the message
Returns:	**boolean**	**true** if the message was sent successfully, **false** otherwise
Throws:	QueueInexistentException	If one or more of the queues specified does not exist in the system
	ClientInexistentException	If the specified receiver does not exist in the system

SendMessage - *Sends a message from this client to one or more queues.*

Parameter(s):	**String** message	The text of the message to be sent.
	byte priority	The priority of the message, from 1 to 10, with 1 being the lowest and 10 the highest.
	int context	The context of the message
	java.lang.String[] queues	The names of the queue or queues to which to send the message
Returns:	**boolean**	**true** if the message was sent successfully, **false** otherwise
Throws:	QueueInexistentException	If one or more of the queues specified does not exist in the system

DeleteQueue - *Requests that the named queue be deleted from the queues in the system.*

Parameter(s):	**String** queueName	The name of the queue to be deleted.
Returns:	**boolean**	**true** if the queue was deleted, **false** otherwise.
Throws:	QueueInexistentException	If the queue to be deleted does not exist in the system.
	QueueNotEmptyException	If the queue to be deleted is not empty

CreateQueue - *Requests that a new queue be created in the the system with the specified name.*

Parameter(s):	**String** queueName	The name to assign the newly created queue.

Returns:	**boolean**	**true** if the queue was created successfully, **false** otherwise.
Throws:	QueueAEException	If a queue with the name specified already exists in the system

GetWaitingQueues - *Gets the names of the queues that have messages waiting for this client.*

Returns:	**Iterable**	An Iterable containing the names of the queues with messages waiting for this client, or null if there are no such queues or the operation did not complete successfully.

ViewMessageFromQueue - *Retrieves the message with the highest priority from a queue and optionally removes the message from the queue.*

Parameter(s):	**String** queueName	The name of the queue from which to retrieve the message
	boolean andRemove	**true** to delete the message after retrieving it, **false** to leave the message on the queue.
Returns:	**Message**	The message with the highest priority in the specified queue
Throws:	QueueInexistentException	If the specified was queue does not exist in the system.

ViewMessageFromQueue - *Retrieves a message from a queue and optionally removes the message from that queue.*

Parameter(s):	**String** queueName	The name of the queue from which to retrieve the message
	boolean andRemove	**true** to delete the message after retrieving it, **false** to leave the message on the queue.
	Order orderedBy	An enumerated value indicating whether to retrieve the message with the earliest time-stamp or with the highest priority.
Returns:	**Message**	A message from the specified queue retrieved by either highest priority or earliest time-stamp.
Throws:	QueueInexistentException	If the specified was queue does not exist in the system.

ViewMessageFromSender - *Retrieves the message with the highest priority from a particular sender and optionally removes the message from the queue it was found on.*

Parameter(s):	**String** senderName	The name of the sender from whom the message is to be.
	boolean andRemove	**true** to delete the message after retrieving it, **false** to leave the message on the queue.
Returns:	**Message**	The message with the highest priority from the sender or null if there was no message available from that sender.
Throws:	ClientInexistentException	If the specified sender does not exist in the system

ViewMessageFromSender - *Retrieves a message from a particular sender and optionally removes the message from the queue it was found on.*

Parameter(s):	**String** senderName	The name of the sender from whom the message is to be
	boolean andRemove	**true** to delete the message after retrieving it, **false** to leave the message on the queue.
	Order orderedBy	An enumerated value indicating whether to retrieve the message with the earliest time-stamp or with the highest priority.
Returns:	**Message**	The message that was selected or null if there was no message available from the sender.
Throws:	ClientInexistentException	If the specified sender does not exist in the system.

3.2. Middleware Interface

This interface connects the client to the messaging system. It consists of the server's API and communication is conducted using the protocol messages defined by the system.

3.2.1. Server Interface & Messages

The server interface is implemented by the ServerRPC class. This class handles the serialization transmission of requests from the client to the server and the deserialization of the associated responses.

org.ftab.client.serverrpc.ServerRPC

Represents a specific middlewae server and provides an interface of the calls that the client can make of the system

METHODS		
Connect - *Connects the provided client to the middleware server represented by this instance.*		
Parameter(s):	**Client** client	The client to connect to this server.
Throws:	FullServerException	If the server is already at full capacity
	AlreadyOnlineException	If the client is already online in the system
Disconnect - *Disconnects the specified client from this server and the system.*		
Parameter(s):	**Client** client	The client to be disconnected
PushMessage - *Pushes a message onto the system.*		
Parameter(s):	**Message** message	The message to be pushed onto the system.
Throws:	QueueInexistentException	If at least one of the queues specified in the message does not exist in the system.
	ClientInexistentException	If the receiver specified in the message does not exist in the system.
DeleteQueue - *Removes an empty queue from the system.*		
Parameter(s):	**String** queueName	The name of the queue to be deleted
Throws:	QueueInexistentException	If the queue to be deleted does not exist in the system
	QueueNotEmptyException	If the queue to be deleted is not empty
CreateQueue - *Creates a new queue in the system with the specified name.*		
Parameter(s):	**String** queueName	The name to be assigned to the newly created queue
Throws:	QueueAEException	If a queue with the specified name already exists in the system
GetWaitingQueues - *Retrieves the names of the queues that have messages waiting the specified client*		
Parameter(s):	**Client** client	The client for whom queues with waiting messages should be fetched

Returns:	**Iterable**	An Iterable containing the names of the queues that have messages waiting

RetrieveMessage - *Retrieves a message from the system according to the specified filter and ordering.*

Parameter(s):	**String** value	The name of the queue or sender to retrieve the message by.
	Filter filter	Whether to retrieve a message on a particular queue or sent by a particular sender
	Order order	Whether to retrieve the message with the highest priority or with the earliest time
	boolean delete	**true** to delete the message on retrieval, **false** otherwise
Returns:	**Message**	The retrieved message object or null if no messages were found that met the criteria.
Throws:	QueueInexistentException	If the filter specified was a queue but the specified name does not exist in the system as a queue
	ClientInexistentException	If the filter specified was a sender but the specified name does not exist in the system as a client.

org.ftab.client.Message

Encapsulates a message that can be passed from client to server or vice-versa.

METHODS

getId - *Gets the identification number of the message in the system.*

Returns:	**long**	The message's identification number or or **-1** if it was not set

getContext - *Gets the context of this message, where **0** represents no context.*

Returns:	**int**	The message's context.

getPriority - *Gets the priority of this message.*

Returns:	**byte**	The message's priority from 1 to 10, 10 being the highest

getContent - *Gets the content of this message.*

Returns:	**String**	This message's content.

getSender - *Gets the username of the sender of this message.*

Returns:	**String**	The username of this message's sender.

getQueues - *Gets the names of the queues this message either is to be sent to or the name of the queue it was retrieved from.*

Returns:	**Iterable**	A list of queue names.

getReceiver - *Gets the designated receiver of this message.*

Returns:	**String**	This message's receiver or null if none was specified.

3.2.2. Defined Exceptions

The following details the exceptions that can be thrown by calls to the middleware interface.

org.ftab.client.exceptions.AlreadyOnlineException

The exception that is thrown by the server when a client that is currently recorded as being online attempts to log in.

org.ftab.client.exceptions.ClientInexistentException

The exception that is thrown by the server when a username specified as an argument in a server call does not exist in the system.

org.ftab.client.exceptions.FullServerException

The exception that is thrown when the server is refusing to accept a connection due to being at its full capacity.

org.ftab.client.exceptions.QueueAEException

The exception that is thrown when a queue in the system already has the name specified to be used to create a new queue.

org.ftab.client.exceptions.QueueInexistentException

The exception that is thrown by the server when a queue name specified in a call does not exist in the system.

org.ftab.client.exceptions.QueueNotEmptyException

The exception that is thrown when an attempt is made to delete a queue that is not empty.

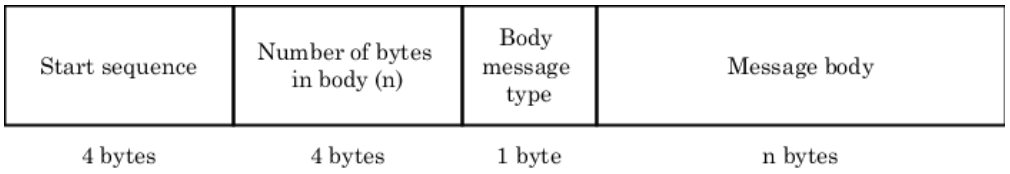

Figure 3.1.: Structure of a ProtocolMessage

3.2.3. Communication Protocol

The communication between the clients and middleware will be accomplished using plain TCP sockets as implemented in the standard Java libraries. In order to accomplish the different functionality requirements for the system, we designed the following protocol and message structure for the different scenarios.

It is important to note that this message protocol is rigid and application-specific, this is because we are striving towards performance by using a smaller custom message format than more verbose and flexible formats such as XML or JSON.

Messages sent between the client process and the server are serialized as ProtocolMessages. All protocol messages extend the ProtocolMessage class. To serialize a protocol message the static **toBytes(ProtocolMessage)** method of the message is called passing the message to be serialized. This method serializes the message with the structure shown in Figure 3.1 where the message body is determined by the implementation of the sub-classed protocol message and the message type is the MessageType of that protocol message. To deserialize a message, the 8 starting bytes are read and passed to the static **getBodySize(ByteBuffer)** method which then returns the amount of bytes that need to be read for the message body. These bytes are passed to the **fromBytes(ByteBuffer)** which deserializes and returns the message.

org.ftab.communication.ProtocolMessage

Base class for all messages sent from the ServerRPC to the remote server. This class also handles the the conversion from bytes to an instance of one of its subclasses. Supported subclasses are:

- org.ftab.requests.ConnectionRequest
- org.ftab.requests.GetQueuesRequest
- org.ftab.requests.QueueModificationRequest
- org.ftab.requests.RetrieveMessageRequest
- org.ftab.requests.SendMessageRequest
- org.ftab.responses.GetQueuesResponse
- org.ftab.responses.RequestResponse
- org.ftab.responses.RetrieveMessageResponse

METHODS
getMessageType - *Returns the message type of this message.*

Returns:	**ProtocolMessage.MessageType**	The MessageType enumerated value corresponding to this message.

getBodySize - *Returns the size of the body to be read based on the information in the header segment.*

Parameter(s):	**ByteBuffer** header	A sequence of HEADER_SIZE bytes beginning with the byte array defined by START_MESSAGE.
Returns:	**int**	The number of bytes to be read to construct the corresponding message.

toBytes - *Performs serialization on the supplied protocol message.*

Parameter(s):	**ProtocolMessage** message	The ProtocolMessage or subclass to be serialized.
Returns:	**ByteBuffer**	A ByteBuffer containing the protocol message with its limit set to the end of the message and its position set to zero.

toBytes - *Serializes this ProtocolMessage excluding its header information.*

Returns:	**ByteBuffer**	A ByteBuffer containing this message with its limit set to the end of the message and its position set to zero.

fromBytes - *Parses a ByteBuffer to its containing ProtocolMessage.*

Parameter(s):	**ByteBuffer** input	The ByteBuffer to be read for the message.
Returns:	**ProtocolMessage**	The ProtocolMessage corresponding to the contents in the buffer.

org.ftab.communication.ProtocolMessage.MessageType

Enumeration defining the type of messages that a ProtocolMessage can be.

FIELDS

ProtocolMessage.MessageType CONNECTION_REQUEST

Marks that the message body contains a message requesting a connection or disconnection.

ProtocolMessage.MessageType QUEUE_MODIFICATION

Marks that the message body contains a message requesting a queue modification action.

ProtocolMessage.MessageType SEND_MESSAGE

Marks that the message body contains a message requesting to send a message.

ProtocolMessage.MessageType RETRIEVE_MESSAGE

Marks that the message body contains a message requesting to retrieve a message.

ProtocolMessage.MessageType RETRIEVE_QUEUES

Marks that the message body contains a message requesting to retrieve queues with messages waiting.

ProtocolMessage.MessageType REQUEST_RESPONSE

Marks that the message body contains a generic request response.

ProtocolMessage.MessageType RETURNED_MESSAGES

Marks that the message body contains a the requested message.

ProtocolMessage.MessageType RETURNED_QUEUE

Marks that the message body contains the requested list of queues.

METHODS

getByteValue - *Converts the enumeration constant to its byte value.*

Returns:	**byte**	The byte value associated with the enumeration.

fromByte - *Converts a byte value to an enumeration constant*

Parameter(s):	**byte** b	The byte value to convert
Returns:	**ProtocolMessage.MessageType**	The enumerated constant corresponding to the supplied byte value.

org.ftab.communication.requests.SendMessageRequest

Represents a request from to send a message to a queue or group of queues.

METHODS

getReceiver - *Gets the designated receiver of the message*

Returns:	**String**	The name of the receiver of the message or null if the receiver was not specified.

hasReceiver - *Gets a boolean value that indicates whether a receiver was specified for the message.*

Returns:	**boolean**	**true** if the message has a receiver, **false** otherwise

getMessage - *Retrieves the content of the message to be sent*

Returns:	**String**	A string containing the content of the message.

getPriority - *Gets the priority of this message.*

Returns:	**byte**	A value from 1 to 10, 10 being the highest, indicating the priority of the message.

getContext - *Gets the context of the message*

Returns:	**int**	An integer value indicating the context of the message.

getQueueList - *Gets the queues that the message should be sent to.*

Returns:	**Iterable**	An iterable containing the names of the queues to put the message upon.

org.ftab.communication.requests.RetrieveMessageRequest

Encapsulates a request to retrieve a message from the system.

METHODS

getFilterType - *Gets a value indicating whether this request is for a message from a queue or sender*

Returns:	**Filter**	A value of Filter.QUEUE to get a message from a queue and Filter.SENDER to get the message sent by a particular sender.

getOrderBy - *Gets a value indicating whether to retrieve the message based on priority or the message with the earliest time.*

Returns:	**Order**	A value of Order.Priority to retrieve the message by priority or Order.TIMESTAMP to retrieve the message by time stamp.

getFilterValue - *A string that represents the queue or sender from which to retrieve the message.*

Returns:	**String**	A string value that either represents a queue or sender's name.

isPopMessage - *Whether to pop the message after retrieving it.*

Returns:	**boolean**	**true** to pop the message from the queue, **false** otherwise.

org.ftab.communication.requests.QueueModificationRequest

Encapsulates a request to create or delete a queue in the system.

METHODS		
getQueueName - *Gets the queue name of the queue on which the action is being performed.*		
Returns:	**String**	The name of the queue to be deleted or to assign the created queue.
isDelete - *Gets a boolean value indicating whether this request is to create or delete the named queue.*		
Returns:	**boolean**	**true** to delete the named queue, **false** to create a queue with the supplied name.

org.ftab.communication.requests.GetQueuesRequest

Encapsulates a request for the names of the queues containing messages for the calling client.

org.ftab.communication.requests.ConnectionRequest

Encapsulates a client's connection or disconnection request.

METHODS		
getUsername - *Gets the username of the client requesting the connection.*		
Returns:	**String**	The username of the client requesting a connection or null if the request is a disconnection.
isConnection - *Gets whether this connection request corresponds to a connection or a disconnection of the client from the server.*		
Returns:	**boolean**	**true** if this request encapsulates a connection request, **false** if this request is a disconnection request.

org.ftab.communication.responses.RetrieveMessageResponse

Encapsulates the system's response to a org.ftab.communication.requests.RetrieveMessageRequest. It contains details of the retrieved message.

METHODS		
getMessageId - *Get the identification number of the message in the system.*		
Returns:	**long**	The message identification number.
getMessageContent - *Gets the content of the message.*		

Returns:	**String**	The text contained within the message.

getSender - *Gets the username of the sender of the message.*

Returns:	**String**	The sender's username in the system.

getReceiver - *Get the designated receiver of the message.*

Returns:	**String**	The username of the receiver of the message.

hasReceiver - *Indicates whether the message had the receiver specified.*

Returns:	**boolean**	**true** if the receiver field is not null, **false** otherwise.

getQueue - *Get the name of the queue that the message is or was contained in.*

Returns:	**String**	The name of the containing queue.

getPriority - *Gets the priority of the message.*

Returns:	**int**	The message's priority from 1 to 10, 10 being the highest.

getContext - *Get the messages context value.*

Returns:	**int**	The integer context of the message.

org.ftab.communication.responses.RequestResponse

Encapsulates a general response message with flags for operation success or failure. Also contains any details of the failure such as system specific exceptions.

METHODS

getStatus - *Gets the enumerated value pertaining to the status of the response.*

Returns:	**RequestResponse.Status**	The response status.

getDescription - *Gets the description of the response, if any.*

Returns:	**String**	The response description.

org.ftab.communication.responses.RequestResponse.Status

Enum documenting the possible statuses a RequestResponse message can contain. Included are statuses for success, exception, full server, no applicable queues, user already online, queue not empty, no client, no message, queue already exits, queue does not exist.

FIELDS

RequestResponse.Status SUCCESS

Marks that the call generating this request was a success.

RequestResponse.Status EXCEPTION

Marks that the call generating this request failed for a unexpected reason.

RequestResponse.Status FULL_SERVER

Marks that the connection call failed due to the server being at full capcacity

RequestResponse.Status NO_QUEUE

Marks that a call requesting queues with messages waiting resulted in no queues.

RequestResponse.Status USER_ONLINE

Marks that the call requesting to login a user failed due to the user already being online.

RequestResponse.Status QUEUE_NOT_EMPTY

Marks that the call requesting the deletion of a queue failed due to the queue not being empty.

RequestResponse.Status NO_CLIENT

Marks that in a call involving a client, the specified client does not exist in the system.

RequestResponse.Status NO_MESSAGE

Marks that for a call to retrieve a message, no messages were found.

RequestResponse.Status QUEUE_EXISTS

Marks that the call to create a queue failed due to the queue already existing.

RequestResponse.Status QUEUE_NOT_EXISTS

Marks that in a call involving a queue, the specified queue does not exist in the system.

METHODS

org.ftab.communication.responses.GetQueuesResponse

Encapsulates a response to a org.ftab.communication.requests.GetQueuesRequest containing the queue names that had messages waiting for the client.

METHODS

getQueues - *Get an Iterable container of the names of the queues with messages waiting for the client.*

Returns: **Iterable** An Iterable over the queue names.

3.3. Database Interface

This interface comprises a series of data access objects (DAOs) which define the method calls for interacting with the database, and the associated entity objects and exceptions, for querying the database from the middleware or the console. These calls are communicated to the database using the PostgreSQL JDBC driver. While many of the calls to the DAOs throw SQL exceptions, they are not listed here for the sake of brevity.

3.3.1. Database Entities

The following classes represent the entities stored in the database.

org.ftab.database.Client

Class that represents a client, i.e. a component in the first tier of the messaging system. This contains the information about the client that is stored in the database.

METHODS		
getClientId - *Get the client's id in the database*		
Returns:	**int**	client's id in the database
getClientUsername - *Get the client's username*		
Returns:	**String**	client's username
isClientOnline - *Indicates if the user is currently online*		
Returns:	**boolean**	true if the user is online, false otherwise

org.ftab.database.Message

Class representing a message in the system with information extracted from the database, suitable for encapsulating results from queries on the message table.

METHODS		
getId - *Getter for the id field.*		
Returns:	**long**	message's id.
getContext - *Getter for the context field.*		
Returns:	**int**	message's context.
getPriority - *Getter for the priority field.*		
Returns:	**short**	message's priority.
getContent - *Getter for the content field.*		

Returns:	**String**	message's content.

getSender - *Getter for the sender field.*

Returns:	**String**	message's sender.

getCreateTime - *Getter for the create time field.*

Returns:	**int**	message's creation timestamp.

getQueueName - *Getter for the queue name.*

Returns:	**String**	message's queue name.

getQueueId - *Getter for the queue id.*

Returns:	**long**	message's queue id.

getReceiver - *Getter for the receiver field.*

Returns:	**String**	message's receiver.

org.ftab.database.Queue

Class that represents a queue in the database and includes minimal information about it, namely the unique name and the current number of messages in it.

METHODS

getName - *Retrieve the name of the queue.*

Returns:	**String**	the queue name.

getMessageCount - *Retrieve the number of messages in the queue.*

Returns:	**long**	the message count.

3.3.2. General Database Operations

These DAO are responsible for the operations of creating and destroying the database.

org.ftab.database.Create

DAO for creating the database schema for the system

METHODS

execute - *Create the requested tables in the database.*

Parameter(s):	**boolean** client	indicates if the client table should be created.
	boolean queue	indicates if the queue table should be created.

	boolean msg	indicates if the message table should be created, creating the message table implies creating the client and queue table.
	Connection conn	database connection.

org.ftab.database.Destroy

DAO for dropping the system's table from the database. It is meant only for tests and should be used carefully.

METHODS

execute - *Drop the requested tables in the database.*

Parameter(s):	**boolean** client	indicates if the client table should be dropped.
	boolean queue	indicates if the queue table should be dropped.
	boolean msg	indicates if the message table should be dropped.
	Connection conn	database connection.

3.3.3. Client Operations

The DAOs responsible for client related operations such as creating clients, logging the client in and out, as well as retrieving client information are detailed below.

org.ftab.database.client.RetrieveClients

DAO for retrieving a snapshot of the client table in the database.

METHODS

execute - *Retrieve information about all the clients in the database.*

Parameter(s):	**Connection** conn	database connection.
Returns:	**ArrayList**	list of UserClient objects with the information.

org.ftab.database.client.FetchClient

DAO for retrieving an user from the database.

METHODS

execute - *Retrieve a client given its username*

Parameter(s):	**String** username	username of the client.
	Connection conn	connection to the database.

Returns:	**Client**	object with the information about the client in the database if such exists, otherwise it is null.

org.ftab.database.client.CreateClient

DAO for creating a client record in the database.

METHODS

execute - *Insert a new client with the given username and online status.*

Parameter(s):	**String** username	desired username for the client.
	boolean online	online status of the client.
	Connection conn	database connection.
Returns:	**int**	the id of the newly created client.

org.ftab.database.client.ChangeClientStatus

DAO for changing the online status of a client.

METHODS

execute - *Change the online status of the client given its username.*

Parameter(s):	**String** username	client's username.
	boolean status	online status.
	Connection conn	database connection.

execute - *Change the online status of the client given its id.*

Parameter(s):	**int** id	client's unique id.
	boolean status	online status.
	Connection conn	database connection.

3.3.4. Message Operations

The DAOs responsible for message related operations such as creating, deleting and retrieving messages are detailed below.

org.ftab.database.message.RetrieveMessage

DAO for peeking at messages in the database, in every realization of this DAO at most one message is returned.

METHODS

execute - *Retrieve a message according to the specified criteria. If no message is found then null is returned. Exceptions are thrown when the parameters are not valid, e.g. the given sender doesn't exist.*

Parameter(s):	**int** receiver	client that is retrieving the message.
	String argument	either sender or queue that will be used as criteria for retrieving the message.
	boolean prioFirst	indicates if the top priority message should be retrieved, otherwise it will be the newest one. true indicates by priority.
	boolean byQueue	indicates if argument is a queue or a sender. **true** indicates queue.
	Connection conn	database connection.
Returns:	**Message**	top message found, or null if there is not any.

org.ftab.database.message.GetAllMessages

DAO to retrieve all messages in the database, sorted by priority. The information provided with the message is the same as defined in the Message class.

METHODS

execute - *Execute the query to retrieve all messages and return a list with all the Message objects.*

Parameter(s):	**Connection** conn	database connection.
Returns:	**ArrayList**	list with all the messages in the database.

org.ftab.database.message.DeleteMessage

DAO to pop a message from a queue given the message and queue name.

METHODS

execute - *Delete a message from a queue and subsequently delete the message if it is not in anymore queues.*

Parameter(s):	**long** messageId	id of the message to delete, expected to be a valid id.
	String queueName	name of the queue where the message is present, expected to be a valid name.
	Connection conn	database connection.

org.ftab.database.message.CreateMessage

DAO for creating messages in the database, it provides different overloaded execute methods depending on the type of message being sent. Checking restrictions such as maximum size of the inputs is expected to be done by the caller of this class.

METHODS

execute - *Create a message with a specific receiver and put it in multiple queues as specified in the input list.*

Parameter(s):	**int** sender	id of the client that sends the message, assumed to be valid.
	String receiver	username of the client that should receive the message, may not exist in the database.
	java.lang.Iterable queues	list of queues where to send the message.
	int context	context of the message.
	short priority	priority of the message, from 1 to 10.
	String message	content of the message, assumed to be less than 2k characters.
	Connection conn	database connection.
Throws:	InexistentQueueException	if one of the queue names in the list doesn't exist.
	InexistentClientException	if the receiver doesn't exist.
	CreateMessageException	if the message was not created but no error was triggered.

execute - *Create a message without specific receiver and put it in multiple queues as specified in the input list.*

Parameter(s):	**int** sender	id of the client that sends the message, assumed to be valid.
	java.lang.Iterable queues	list of queues where to send the message.
	int context	context of the message.
	short priority	priority of the message, from 1 to 10.
	String message	content of the message, assumed to be less than 2k characters.
	Connection conn	database connection.

Throws:	InexistentQueueException	if one of the queue names in the list doesn't exist.
	CreateMessageException	if the message was not created but no error was triggered.

execute - *Create a message with a specific receiver and put it in a single queue.*

Parameter(s):	**int** sender	id of the client that sends the message, assumed to be valid.
	String receiver	username of the client that should receive the message, may not exist in the database.
	String queue	queue where to send the message, it may not exist.
	short context	context of the message.
	short priority	priority of the message, from 1 to 10.
	String message	content of the message, assumed to be less than 2k characters.
	Connection conn	database connection.
Throws:	InexistentQueueException	if one of the queue names in the list doesn't exist.
	InexistentClientException	if the receiver doesn't exist.
	CreateMessageException	if the message was not created but no error was triggered.

execute - *Create a message without specific receiver and put it in a single queue.*

Parameter(s):	**int** sender	id of the client that sends the message, assumed to be valid.
	String queue	queue where to send the message, it may not exist.
	short context	context of the message.
	short priority	priority of the message, from 1 to 10.
	String message	content of the message, assumed to be less than 2k characters.
	Connection conn	database connection.
Throws:	InexistentQueueException	if one of the queue names in the list doesn't exist.
	CreateMessageException	if the message was not created but no error was triggered.

3.3.5. Queue Operations

The DAOs responsible for queue related operations such as creating, deleting and retrieving queues are detailed below.

org.ftab.database.queue.RetrieveQueues

DAO to retrieve information about all queues in the system for the management console.

METHODS		
execute - *Retrieves all queues in the system.*		
Parameter(s):	**Connection** conn	the database connection.
Returns:	**ArrayList**	An ArrayList of all the queues currently in the system.

org.ftab.database.queue.GetQueuesWithMessages

DAO to retrieve the list of queues where there are messages addressed to a specific client, i.e. their receiver field is not null and points to the given client id.

METHODS		
execute - *Retrieve the list of queues with messages for the given receiver.*		
Parameter(s):	**int** receiverId	id of the client that should receive the messages.
	Connection conn	database connection.
Returns:	**ArrayList**	list with the queues with messages waiting for the client.

org.ftab.database.queue.DeleteQueue

DAO for deleting a queue record in the database.

METHODS		
execute - *Delete a queue given its name, it only deletes non-empty queues.*		
Parameter(s):	**String** queueName	name of the queue.
	Connection conn	database connection.
Throws:	InexistentQueueException	if there is no queue with the given name.
	QueueNotEmptyException	if the queue is not empty.

org.ftab.database.queue.CreateQueue

DAO for creating a queue record in the database.

METHODS

execute - *Create the queue with the given name.*

Parameter(s):	**String** queueName	desired name for the queue.
	Connection conn	database connection.
Throws:	QueueAlreadyExistsException	if there is already a queue with the given name.

3.3.6. Database Exceptions

The following details the exceptions that can be thrown when interacting with a database through a DAO.

org.ftab.database.exceptions.QueueNotEmptyException

Exception that identifies an scenario where a client attempted to delete a non-empty queue.

org.ftab.database.exceptions.QueueAlreadyExistsException

Exception class that identifies the scenario where a queue already exists in the database and someone attempted to create it.

org.ftab.database.exceptions.InexistentQueueException

Exception class that identifies scenarios where a non-existing queue was referenced, e.g. when trying to delete a queue that is not present in the database.

org.ftab.database.exceptions.InexistentClientException

Exception class that identifies scenarios where a inexistent client is referenced in a database operation, e.g. when trying to create a message with a receiver that doesn't exist.

org.ftab.database.exceptions.CreateMessageException

Class to describe scenarios where the message creation failed for unknown reasons. This is a very rare exception and must be treated carefully.

4. System Design

4.1. Clients

The client is the predominant entity in the presentation tier, comprising of the client class, the server remote procedure implementation and the client's logging API. The relationship of these classes can be seen in the class diagram depicted in Figure 4.2 with the interfaces and method parameters presented in greater detail in chapter 3.

4.1.1. Client Class

The Client class is the interface for the transactor to the system. All the necessary functionality of a client is encapsulated by the implementation of the client class which is either called directly such as during experimentation or via the created command line user interface, ClientShell. Exception handling and client side logging is also conducted by this class. Further information about the client class can be found in its documentation in section 3.1.

4.1.2. ServerRPC

This class forms the client's interface with the messaging system. A ServerRPC instance can be considered a client-side representative of a single server within the system that the client can connect to, disconnect from, and make other requests of. Methods called on the ServerRPC class by the client class are serialized into an instance of a subclass of the ProtocolMessage class and sent over the network to the physical server than this instance represents. Likewise, responses, errors and exceptions as ProtocolMessages are received from the physical server by this class, deserialized and returned to the client or thrown. Further information about the ServerRPC class, its associated exceptions, ProtocolMessage and its subclasses can be found in section 3.2.

4.1.3. ClientShell

The client shell can be run using the command

```
# java -jar FTaB-client.jar
```

Passing the boolean argument of `false` to the java application

```
# java -jar FTaB-client.jar false
```

causes the client to propagate exceptions and their stack traces out to the command line, thus providing more detailed debugging information. The user interface itself is an option driven command line interface, presenting only those options to the user that are currently available, and as such is intuitive to use. An screenshot of the interface can be seen in Figure 4.1.

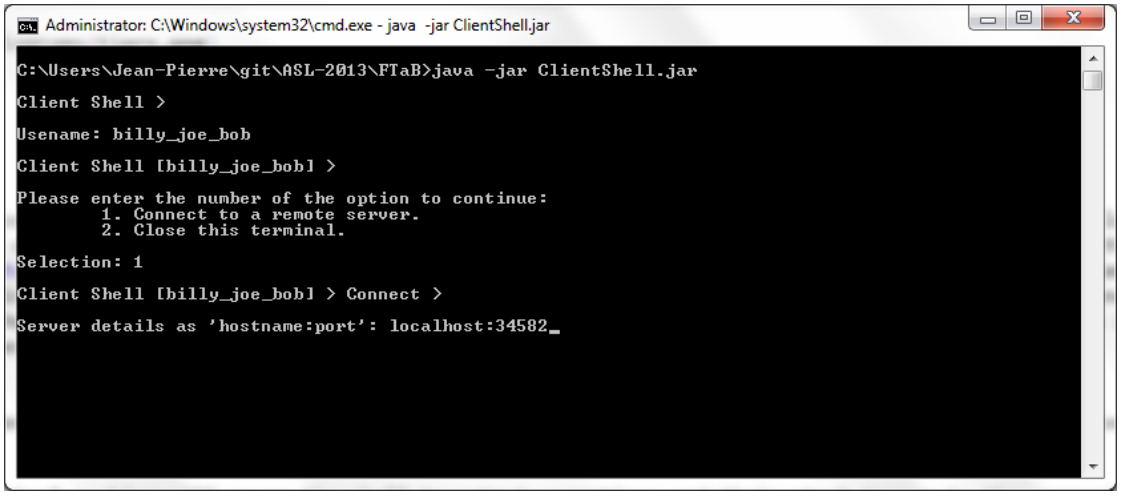

Figure 4.1.: Screenshot of the client shell.

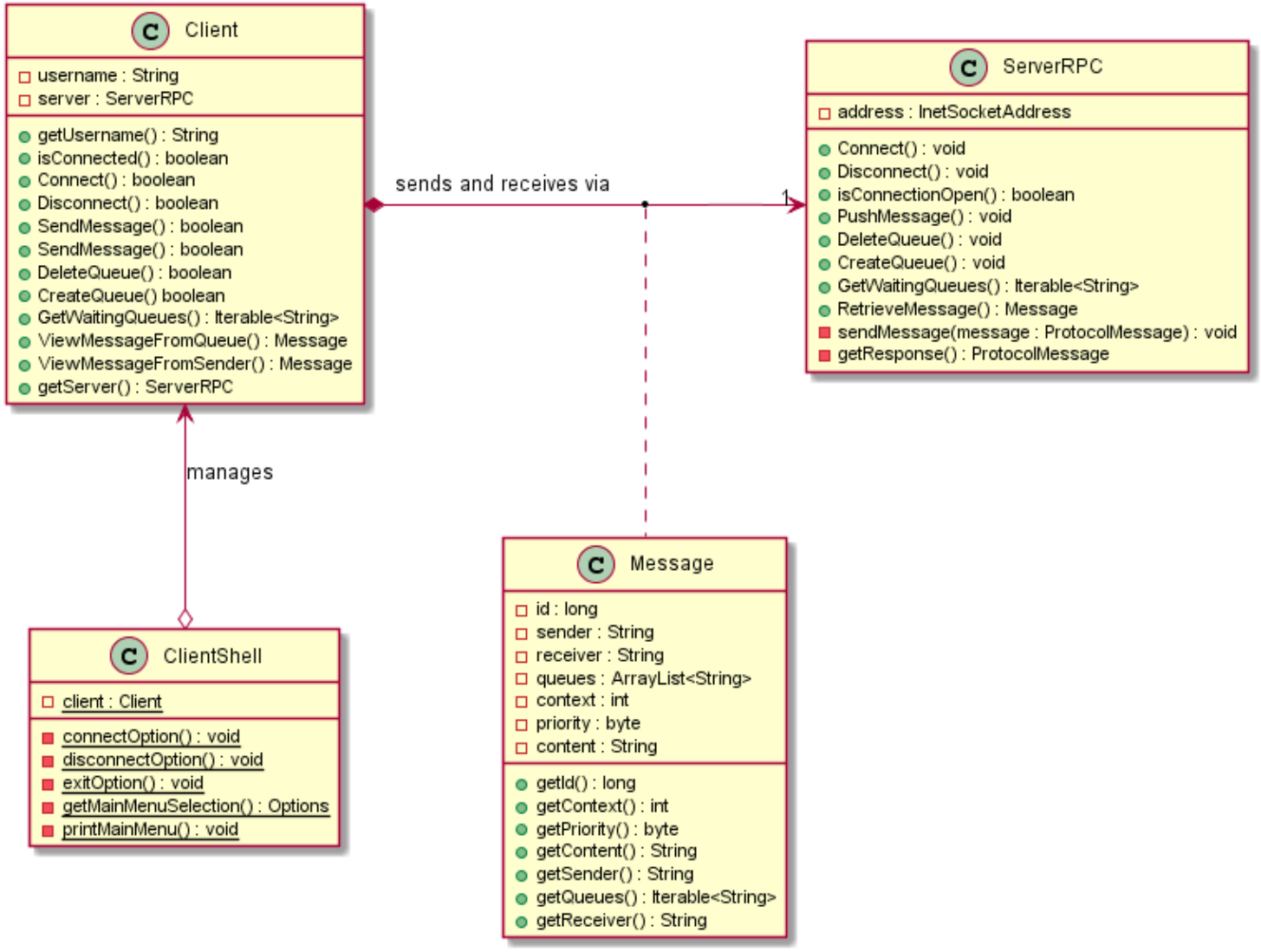

Figure 4.2.: Client UML2 class diagram. Parameters for public methods were can be viewed in chapter 3

4.2. Middleware

The middleware instances are the primary components in the logic tier of the system. Each middleware instance acts as a logically independent unit in the system, accepting connections from clients and handling their requests thereafter. Figure 4.4 shows the main classes that comprise the middleware implementation.

4.2.1. ServerManager

The ServerManager component is the base component of the server, created and ran by the ServerFactory[1] class. A ServerManager instance uses the java.nio package along with multiple MessagingWorkers each run on its own thread from a thread pool maintained by a java.util.concurrent.ExecutorService, to handle its client load. The ServerManager's main responsibilities are to accept incoming connections (refusing them if the server is at its predefined capacity), create new MessagingWorkers as necessary and to delegate clients on connection to a MessagingWorker.

4.2.2. DBConnectionDispatcher

A wrapper class for database connection pool implemented in the PostgresSQL JDBC driver. Its sole responsibility is to manage the number of connections in the connection pool as per the configuration file and to delegate connections on request.

4.2.3. ClientConnection

The role of the ClientConnection component is best explained before addressing MessagingWorkers as its name belies its function. ClientConnections store both a reference to the socket channel that connects the server to the remote client as well as a reference to the DBConnectionDispatcher used by the server. The ClientConnection is thus responsible for deserializing the requests that arrive on its channel, retrieving the required information from the database, serializing the response and sending it back over the channel to its remote client, all from its calling thread.

4.2.4. MessagingWorker

Each MessagingWorker is responsible for a set of ClientConnections assigned to it by its ServerManager. When a channel is flagged to be read or written to by a java.nio.channels.Selector the worker retrieves the ClientConnection associated with the channel and has it evaluate and execute the request. This however leads to an interesting feature of our system better seen in Figure 4.3. As each MessagingWorker is shared among multiple client connections only a single ClientConnection within that MessagingWorker (shown in green) can hold a database connection or be otherwise active at any given time while the others (shown in red) are potentially blocked. Thus spawns the question as to efficient ratios of MessagingWorkers to ClientConnections.

[1] Not shown in Figure 4.4 as its sole purpose is reading configurations from XML files

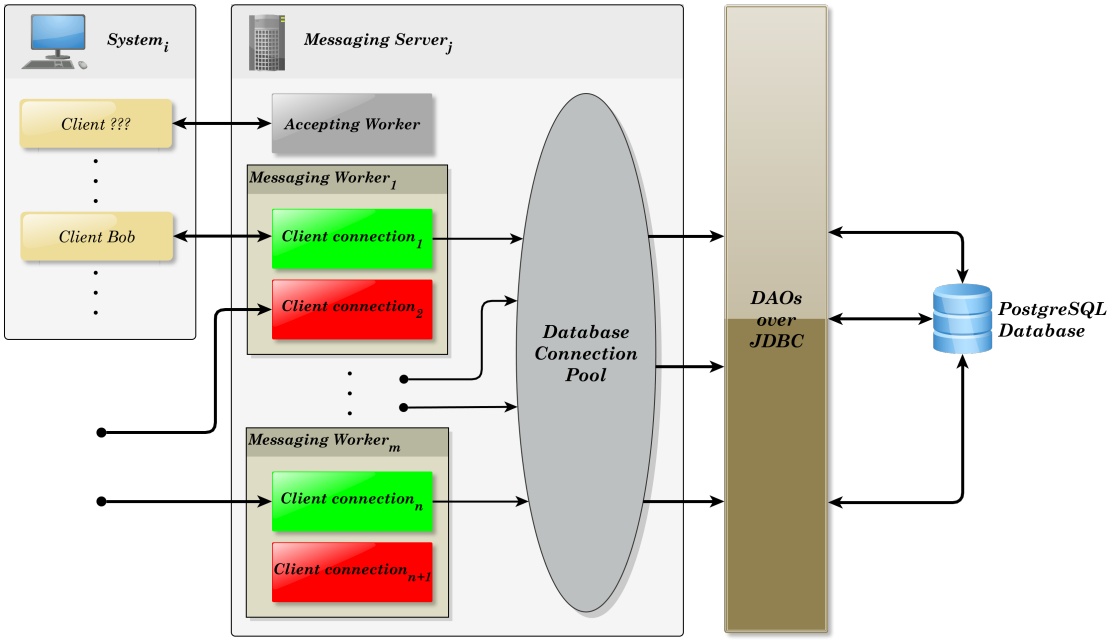

Figure 4.3.: Logical structure within the server.

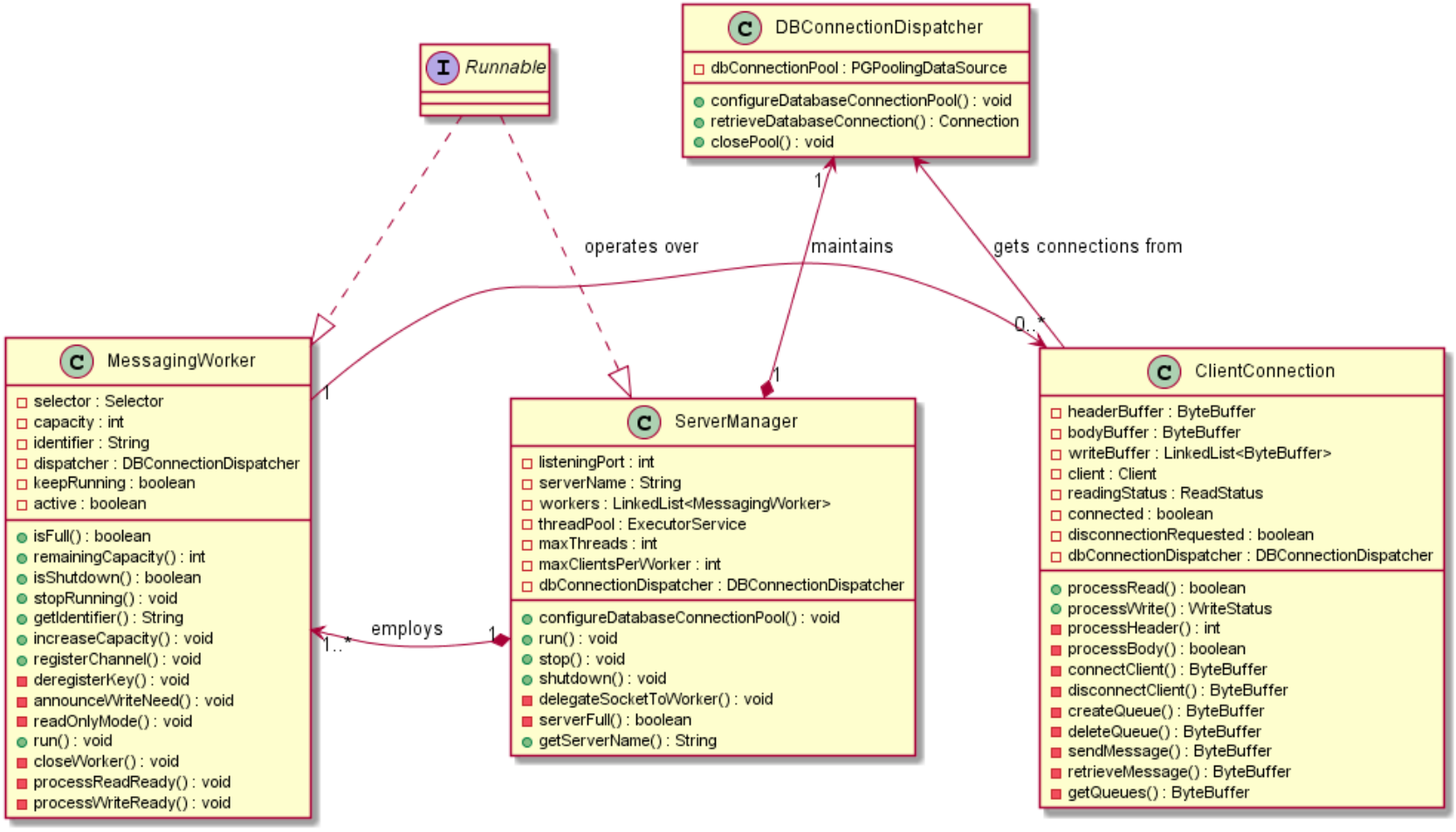

Figure 4.4.: Server UML2 class diagram.

4.3. Management Console

The management console is a Java Swing GUI component spanning both the presentation and logic tiers of the system. It facilitates the observer accessing and viewing information about the state of the messages, queues and clients in the system.

On launching the console the observer is presented with a settings dialog to enter the server address, database name, username and password to connect to the PostgreSQL server and database (Figure A.1). Beyond this dialog one has access to the details screens of clients (Figure A.2), queues (Figure A.3) and messages(Figure A.4). These screens provide detailed information about the content of each item in the system as well as basic statistical information such as the the quantity of an entity in the system. In the case of the queues and clients, the messages in a particular queue and the messages in the system from or to a client can also be viewed. The currency of the data in each screen is managed by a manual refresh button to avoid the added load that constantly polling the database would create.

The management console, while implemented as a requirement for the project, was not used frequently as the same and more detailed information could gathered through running queries from pgAdmin III or the command line. It was also felt that time would be better spent on the implementation of the major components of system and so the console did not grow beyond its current structure.

4.4. Database

The traditional view of each entity being represented as a single table was utilized in the design of the schema, and as such it consists of four tables that contain the information about clients, queues, messages and a relation table linking queues to the messages they contain. An overview of the database schema is presented in Figure 4.5.

4.4.1. Clients Table : client

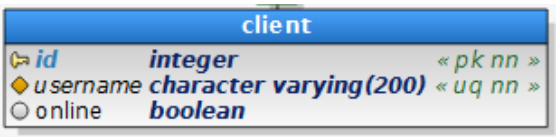

Figure 4.6.: Schema of the client table.

From the schema displayed in Figure 4.6 one can see that for the purpose of our system there is not much information needed about the client. Required however is an unique identifier called “username”. Additionally, in order to have an schema which allows for a more extensible software (e.g. providing the functionality of changing the username of a client), we don’t use this username as primary key but rather use a surrogate key called “id”. Finally, to keep a deterministic behavior in the whole system we decided to keep track of which clients are online in order to avoid having more than one client component using the same username and reading/inserting from the same set of messages, this is stored in the “online” column.

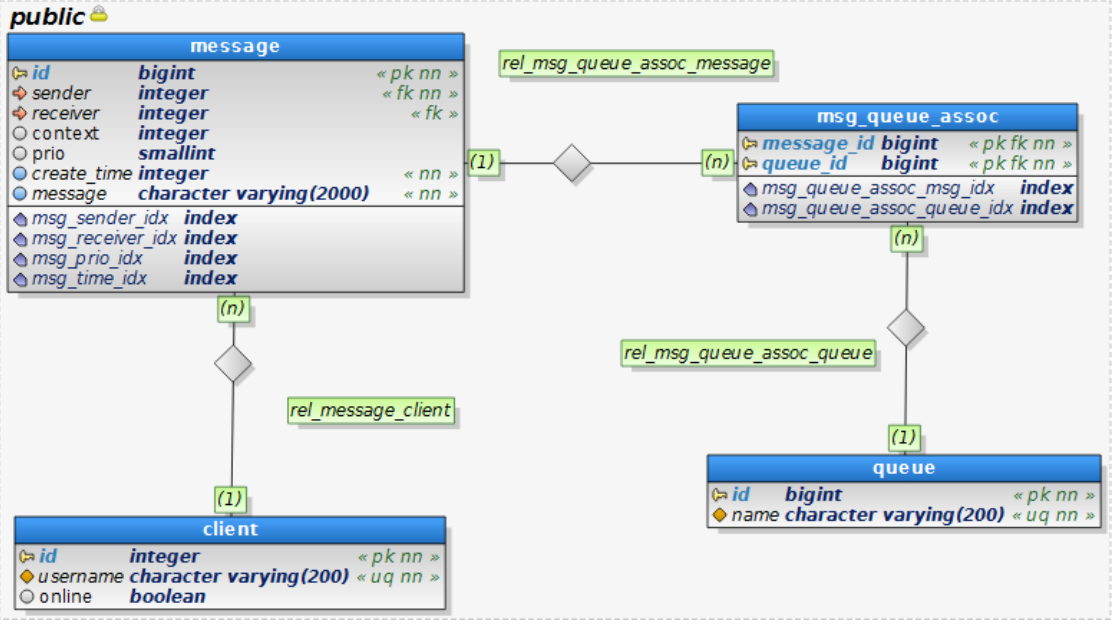

Figure 4.5.: Diagram of the database schema

4.4.2. Queues Table : queue

Figure 4.7.: Schema of the queue table.

The schema for the queue table is shown in Figure 4.7. In order allow clients to easily identify queues we provide a “name” column which will be unique in the system, additionally we use a surrogate key (“id”) as primary key, the design rationale for this is the same as in the client table, to have greater extensibility.

4.4.3. Messages Table : message

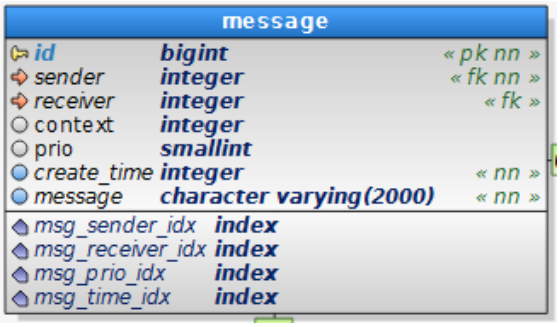

Figure 4.8.: Schema of the messages table.

Figure 4.8 shows the schema implemented for messages table. The requirements for the system specifies for certain information to be kept about each message, this is contained in the message table in the following columns:

- id: Serves as a the primary key for the messages.
- message: The contents of the message.
- sender: The client who created the message.
- receiver: Messages may be addressed to an specific client.
- context: Indicates the context of a message
- prio: Priority of the message (1 to 10).
- create_time: Time of creation of the message in the system.

Foreign Keys

Both the sender and receiver columns are foreign keys on the client table to maintain data consistency.

Indexes

As messages can be accessed by sender (when the client requests a message from a particular sender), receiver (when retrieving messages for a particular client) these columns are indexed. Also as messages are retrieved they are retrieved as either the highest priority message with the earliest arrival time or the earliest arrived message with the highest priority, in order to improve the efficiency of such queries two multi-column indexes were created to accommodate retrieval of this type.

4.4.4. Messages ↔ Queues Relation : msg_queue_assoc

Figure 4.9.: Schema of the association table for messages to queues.

The functional requirement of sending a single message to multiple queues creates a many-to-many relationship in our schema, in order to handle this we opted to create a junction table as this is the preferred practice in database design, even though it may come with a cost in performance since operations between messages and queues will require an extra join.

Foreign Keys

As a relation table both the message_id and queue_id are foreign keys into the message and queue tables respecitvely.

Indexes

Given the size of the table resulting from such a relation (at least row for each message sent), to facilitate efficiently filtering the messages contained in a particular queue, indexes were added on both the message_id and queue_id columns.

5. Sequence Diagrams

This chapter contains a number of sequence diagrams depicting how the system handles various requests from the client under differing conditions.

5.1. Client Connection

In order to use the messaging system a client must first connect to a middleware. The non-error sequence of events for the connection of a client is displayed in Figure 5.1, additionally an user must disconnect at the end of his session. However, in case of dropped connections the server will take care of marking the user as disconnected in the database.

5.2. Queue Management

Clients can create and delete queues, the non-error sequence of events for the queue creation/deletion is presented in Figure 5.2.

5.3. Sending Messages

The diagram in Figure 5.3 describes the events in the scenario of a client trying to send a message to a queue, or multiple queues, with and without specific receiver.

5.4. Retrieving Messages

The two scenarios for reading messages from the queues are displayed in Figure 5.4 and Figure 5.5.

Additionally, an client can ask for queues where there are messages addressed for it. This scenario is described in Figure 5.6

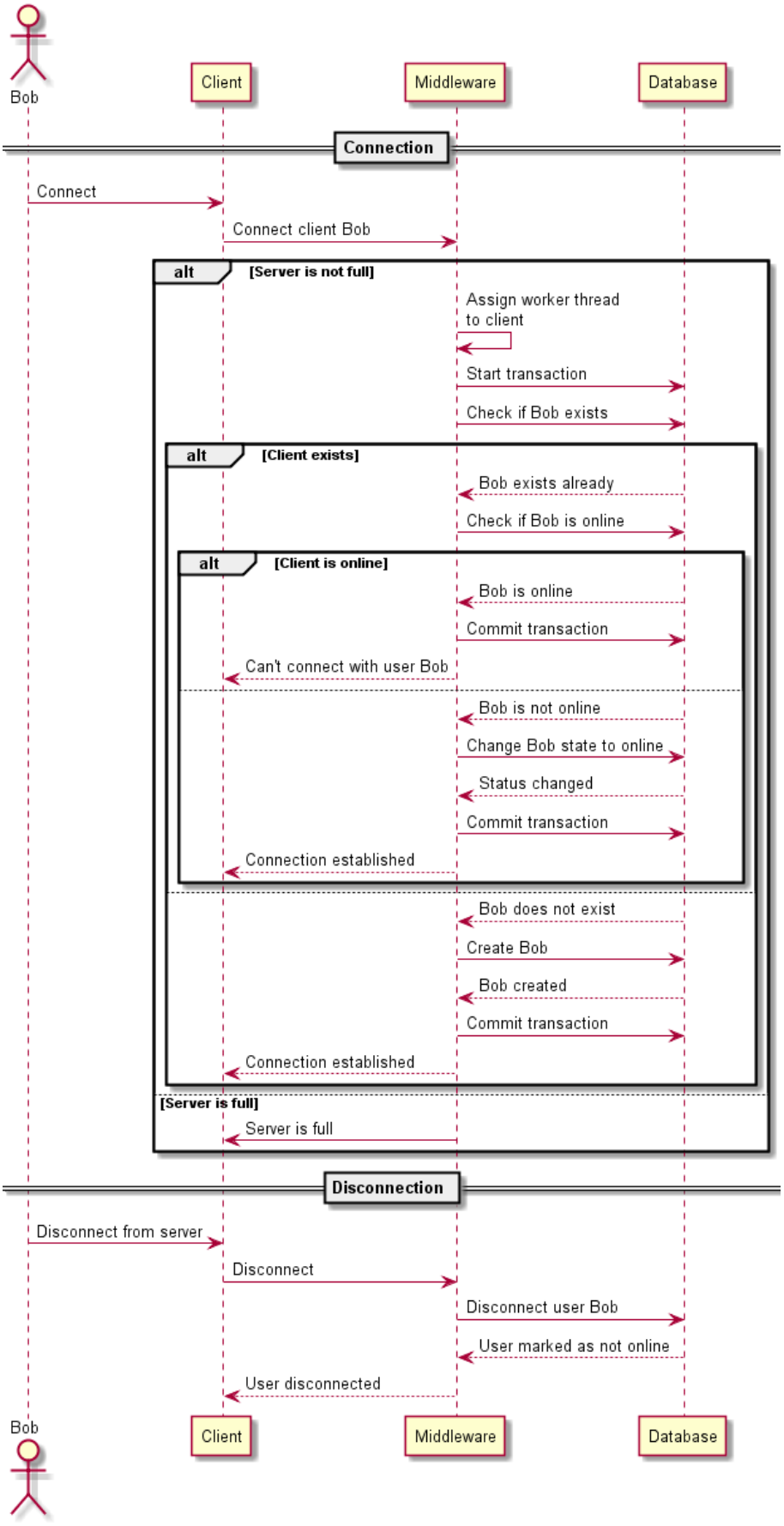

Figure 5.1.: Client connection/disconnection

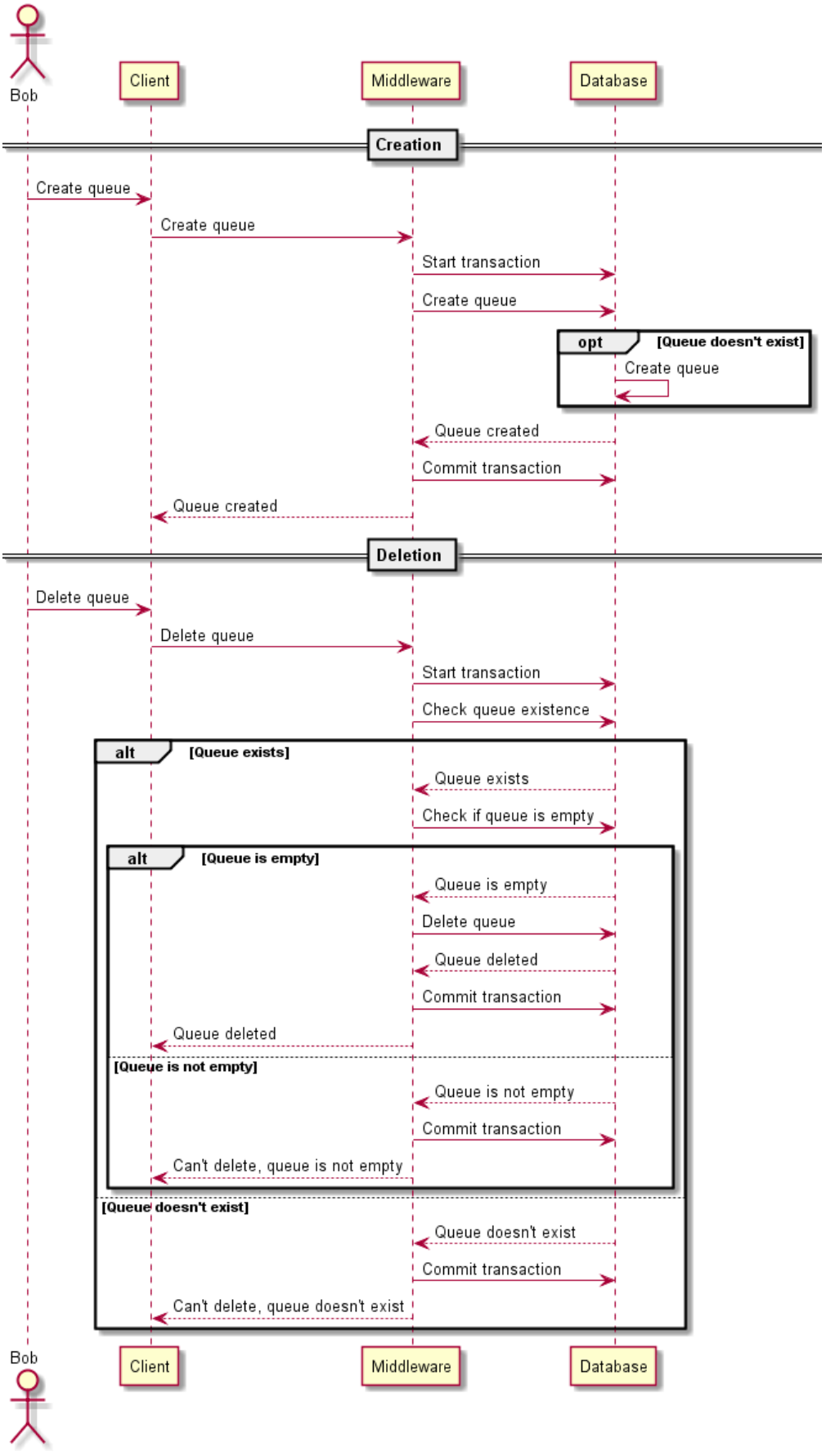

Figure 5.2.: Queue management

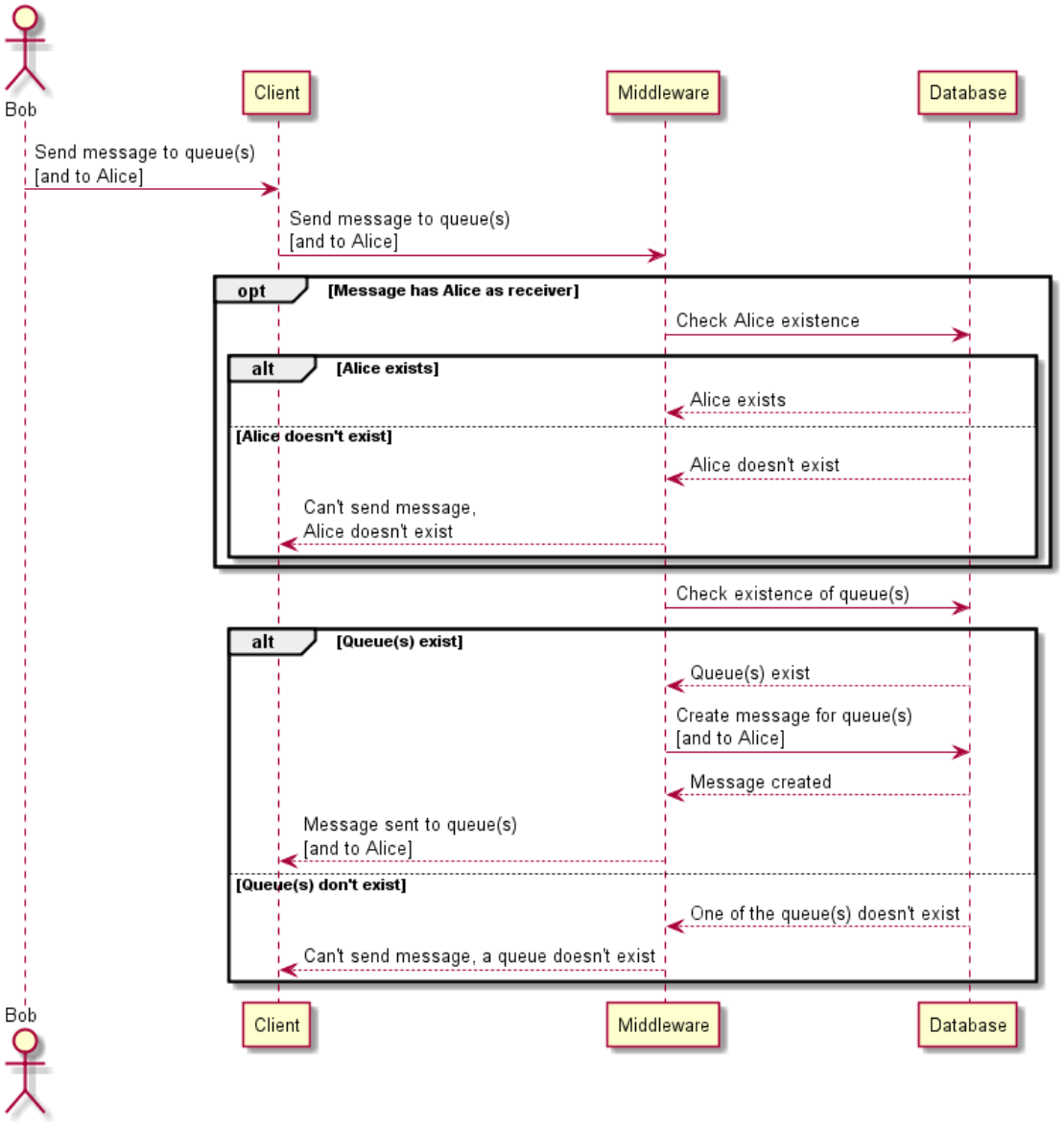

Figure 5.3.: Sending a message

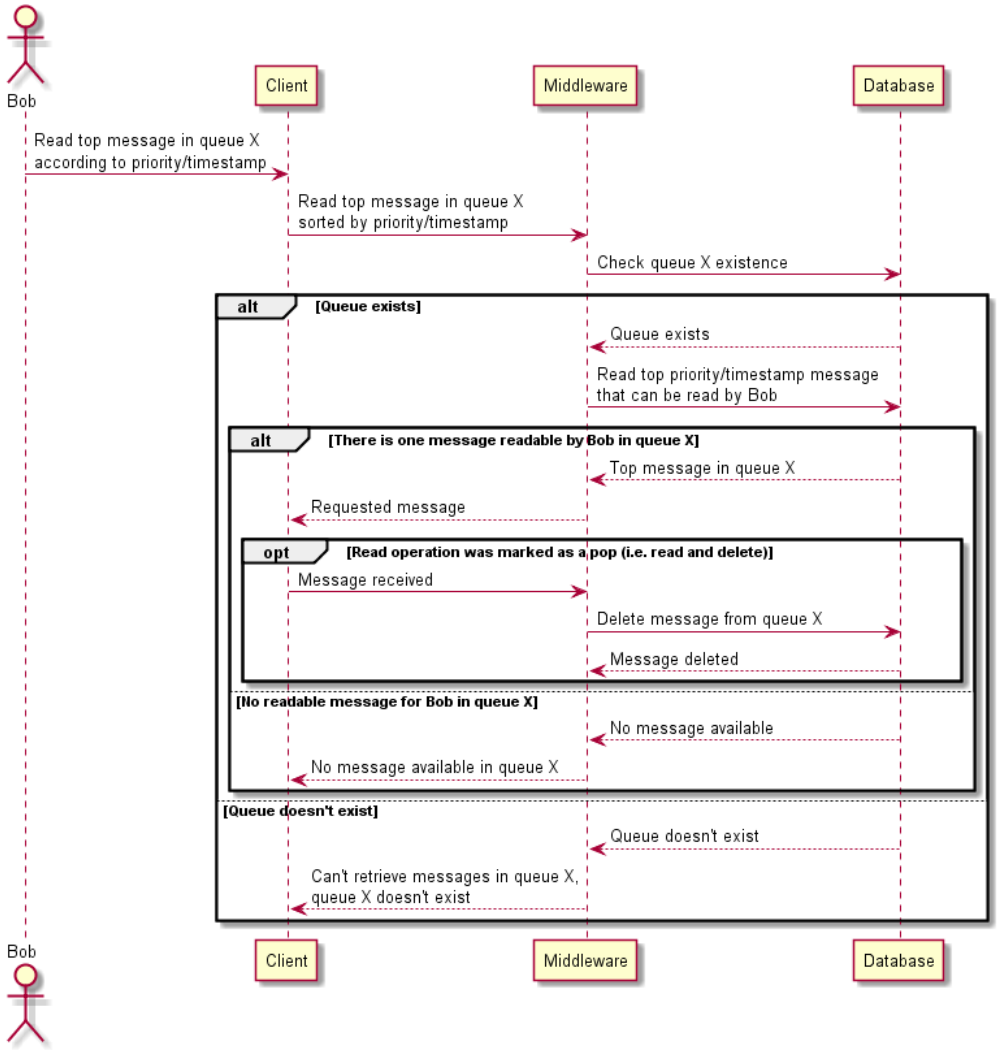

Figure 5.4.: Reading a message from a specific queue

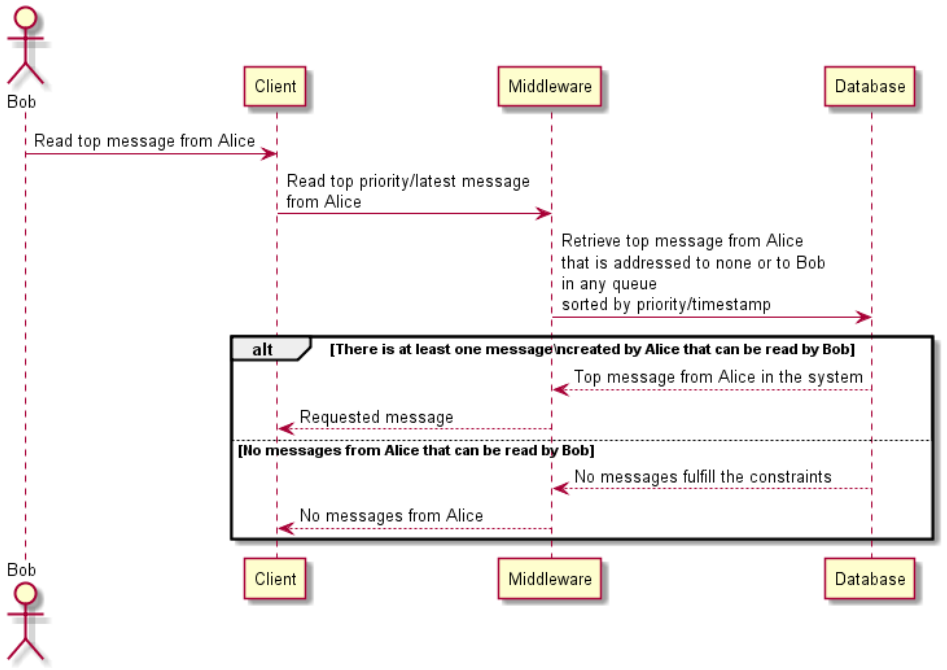

Figure 5.5.: Reading a message from a specific sender

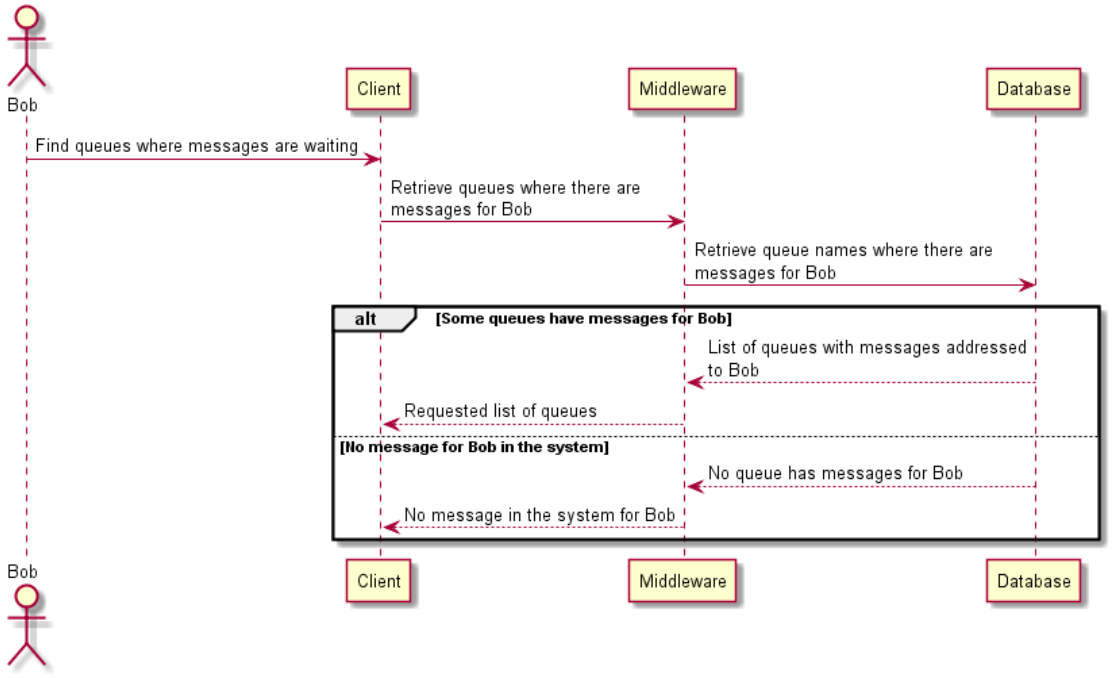

Figure 5.6.: Retrieve queues with messages for a client

Part II.

Experiments

6. Introduction

In part II the different experiments performed on the system and their results are presented with the subsequent analysis, chapter 6 describes the overall testing environment and procedures.

The remainder chapters are organized as follows, in chapter 7 a series of experiments that were performed in an initial version of the system under test is presented, based on the information provided by these initial results the server implementation was improved and final experiments which include a trace of the full system followed, these final experiments are described in chapter 8. Note that all experiments are presented in the same chronological order in which they were performed and the analysis is presented in a way that illustrates the evolution of the authors' understanding of the system.

6.1. Definitions

The following definitions apply to the different experiments described in the subsequent chapters:

6.1.1. System under test

In this case the system under test corresponds to the middleware and database components of our system, this is treated as a black box against which the metrics will be calculated. In this document, the terms server and system are used interchangeably to refer to the system under test.

Note that in all the deployments of the system, both the middleware and database are isolated from the client at a hardware level. Each middleware instance and the database instance run in their own dedicated (virtual) machines, while the client threads share a separate set the (virtual) machines.

6.1.2. Response time (RT)

The response time is calculated as the time that a request takes from its inception in the client software until a response, indicating either success or failure, is received from the server. This measures the responsiveness of the system under test from the client's point of view. This response time can be divided in different components.

Network travel time of the request Time it takes for the request message to reach the server instance.

Waiting time in server Time it takes for the request message to be read and acknowledged by the server instance.

Database time Time it takes for the database query(s) to be processed.

Waiting time in server before send Time it takes until the response message is sent by the server instance.

Network travel time of the response Time it takes for the response message to reach the client after being sent.

During the experiments it is not possible to measure all of these intervals though, due to limitations imposed by the design. It is possible to measure the database time by sensing the time before and after a query is processed in the server, however the network travel time and waiting time for a request/response to be read/sent in the server can only be measured as a single interval. This is because the system can not know if a request/response exists in the queue until the worker thread actually reads or sends it, due to the round-robin system implemented in the thread workers as described in part part:design. Also, for the experiments a symmetric time distribution in travel and waiting time for request and responses is assumed. This is because the expectation is that the waiting time in the server is much more significant than the network travel time and it can be seen from the implementation that this is symmetric in average, later during the analysis this assumption will be evaluated. During the experiments, the response time is calculated as average in intervals of 2 minutes, this window length provides a clearer visualization of the data which is generally taken over intervals of 30 minutes.

6.1.3. Throughput (TH)

The throughput is calculated as the number of operations in a given interval in the whole system, i.e. the sum of all operations performed by each of the clients per second. This is calculated in the same interval as the response time, i.e. 2 minutes.

6.2. Tools & Procedures

For the experiments, a series of scripts were developed in bash and python which allow the automation of the deployment, start, shutdown and data collection operations. The client was not used through its CLI[1] for the experiments but instead the scripts were run using Jython[2] which allows running python code inside a JVM[3]. In this fashion, the fast development capabilities of python code were integrated with the project's java code, this does not introduce additional overhead because the python code is compiled into java classes before running it with the usual JVM.

The initial experiments were performed in the dryad cluster during the dedicated time slots in order to have a controlled environment where external factors would not interfere with the system's performance therefore allowing a better understanding of the system's characteristics. Afterwards, for the final characterization of the system, the experiments

[1] Command Line Interface
[2] http://www.jython.org/
[3] Java Virtual Machine

were moved to the EC2 service provided by Amazon[4]. This allows testing in a real production environment and adds more flexibility in terms of hardware and scalability.

The graphs presented in the following chapters were obtained from the generated data logs using matplotlib[5] and numpy[6].

[4]http://aws.amazon.com/ec2/
[5]http://matplotlib.org/
[6]http://www.numpy.org/

7. Initial exploration

7.1. Experimental setup

For the following experiments, the dryad machines were used. In all of the experiments one machine was dedicated for the PostgreSQL database, one or two machines were used for running the middleware instances and one machine was used for running the client threads.

The releases used for these experiments were:

- PostgreSQL 9.3.1
- Java 1.6.0
- Jython 2.5.3

All the experiments presented in the following sections were performed during the dedicated time slots and therefore no external processes were running during the experiments.

One important note about this phase of experimentation is that instrumentation in the server side was not yet implemented and therefore the response times can't be broken down in the components listed in 6.1.2, only the client's point of view can be sensed, i.e. the overall response time and throughput.

In preparation for each of the experiments, the database is populated only with a list of 100 empty queues, the clients records are created dynamically as each client connects to a server.

7.2. Experiment 1 - Exploring the system

7.2.1. Description

The purpose of this initial experiment was to get a feeling of how the system behaves over time, in preparation for the required 2 hour trace this experiment tests part of the required implementation and provides a integration test on the complete messaging system.

For this experiment, a client that behaves as follows was implemented:

- The client runs continuously until signaled to stop, the running time is controlled by a separate thread that signals all client threads when the experiment is over.

Table 7.1.: Setup of the server for experiment No. 1

Worker threads	25
Clients per thread	2
Database connections	25

- At the beginning of the execution the client selects a random queue from a predefined list and puts in it a message with random English text of size equal to 2000 characters as the content, the priority is set to 5 for all messages, this message is addressed to a random client.

- Afterwards, the client checks for queues with messages addressed to him and when one is found it pops the available message with highest priority from that queue. Since all messages have defined receivers, this pop operation will only select messages addressed to him. This message is then sent to another random queue and a random receiver.

This type of client is also known as Larry in the system and will be used again in the final trace.

This experiment was run with a single server with the setup from table 7.1, the total number of clients was 50 which is equal to the max capacity of the server. The running time was set to 2 hours and the selection of queues used was the first 20 queues of the 100 available in the database, this allows for some overlapping between the messages since there are more clients than queues.

7.2.2. Results

Figure 7.1 shows the response time for sending a message, and figure 7.2 shows the response time for reading a message from a given queue. Figure 7.3 shows the response times for finding a queue with messages for the client, the times are differentiated between responses when a queues was found or not. Finally, 7.4 shows the throughput for sends and reads during the experiment.

From these plots is easy to see that the behavior of the system is stationary, after exclusion of the warm up and cool down intervals. Given this property, the mean and standard deviation for the metrics plotted were calculated using all points in each experiment in the interval from 200 seconds to 7000 seconds which excludes the warm up and cool down periods. Additionally, the 99.9% confidence intervals were calculated for each of the mean values, this was done using a t-student distribution with the sample mean and variance, the degrees of freedom were around 2 million for each metric. These statistics are summarized in table 7.2

7.2.3. Analysis

The results obtained from this initial experiments are encouraging and show a very stable system. Given that each client only send messages after popping one, the database

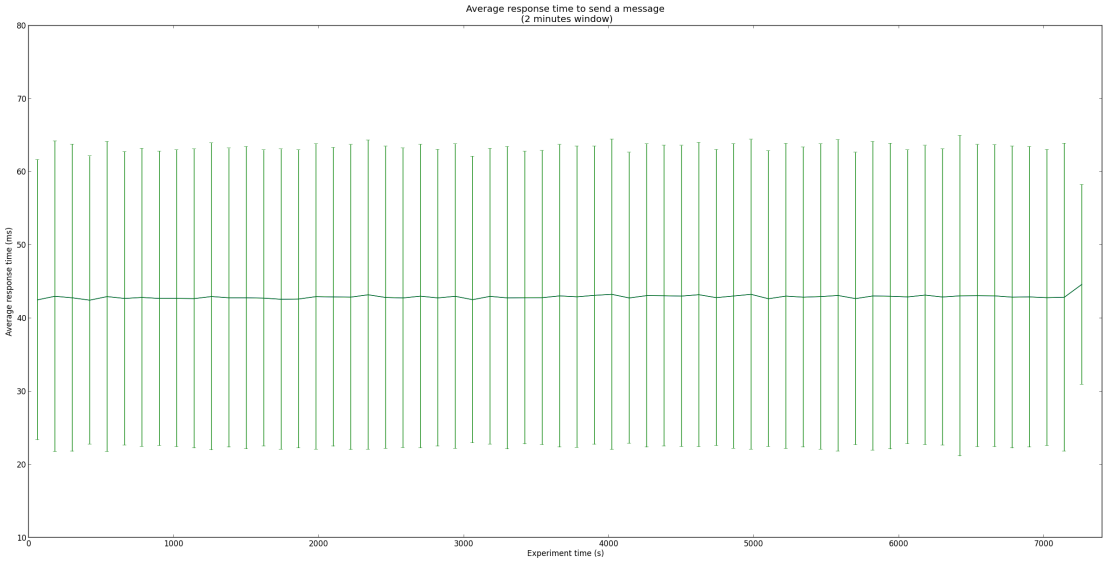

Figure 7.1.: Response time for a send operation

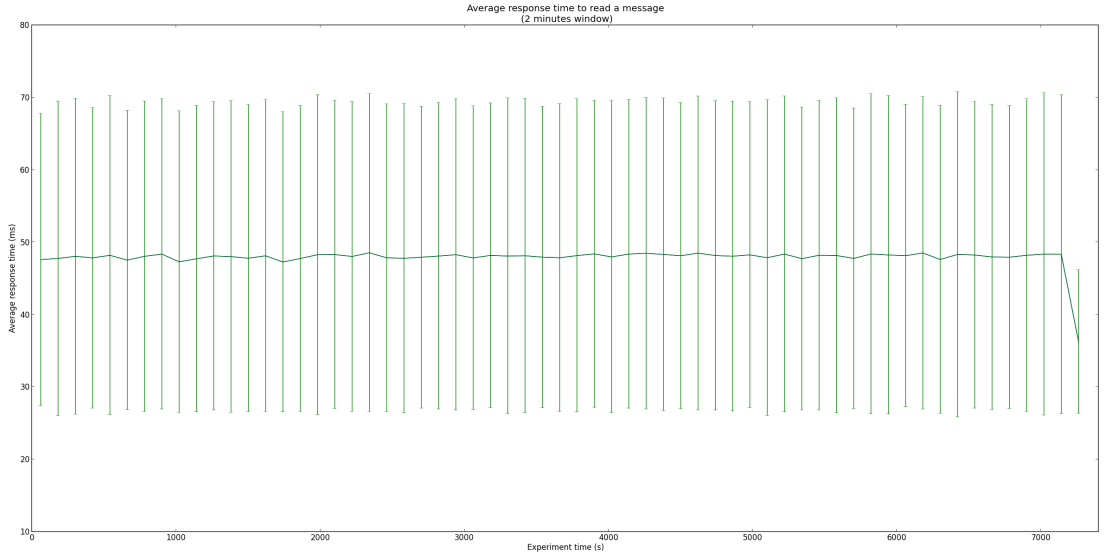

Figure 7.2.: Response time for a read operation

Table 7.2.: Statistical results from experiment No. 1

Metric	**Mean**	**Standard Deviation**	99.9% **CI**
Send RT	$42.9\ ms$	$20.6\ ms$	$42.8 - 42.9\ ms$
Read RT	$48.1\ ms$	$21.4\ ms$	$48.0 - 48.1\ ms$
Queue found RT	$26.9\ ms$	$17.2\ ms$	$26.8 - 26.9\ ms$
Queue not found RT	$22.8\ ms$	$16.3\ ms$	$22.8 - 22.9\ ms$
Send TH	$309\ \frac{msg}{s}$	$2\ \frac{msg}{s}$	$308 - 310\ \frac{msg}{s}$
Read TH	$309\ \frac{msg}{s}$	$2\ \frac{msg}{s}$	$308 - 310 \frac{msg}{s}$

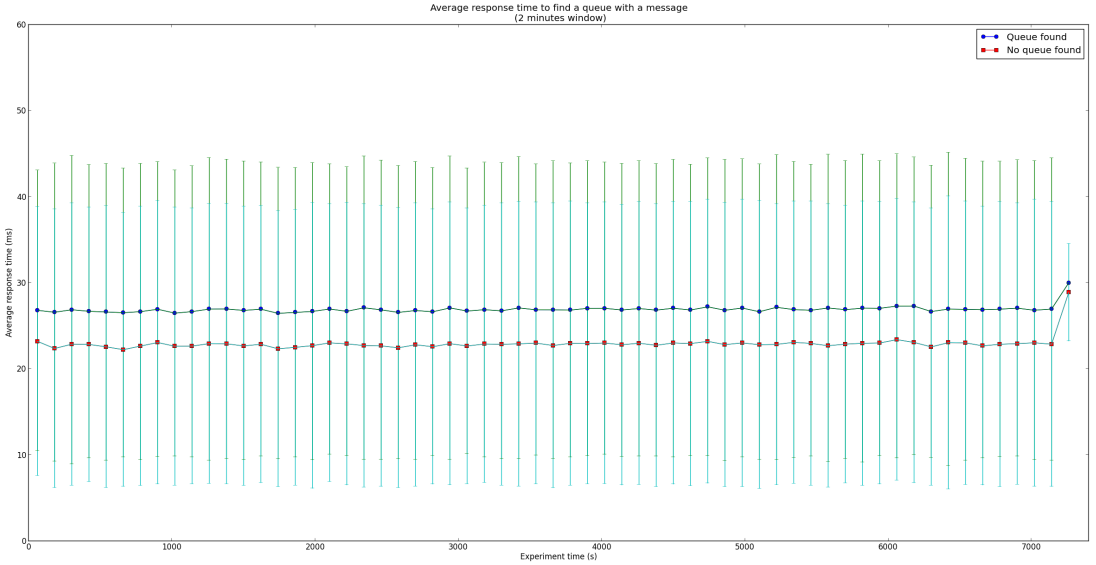

Figure 7.3.: Response time for finding a queue with messages

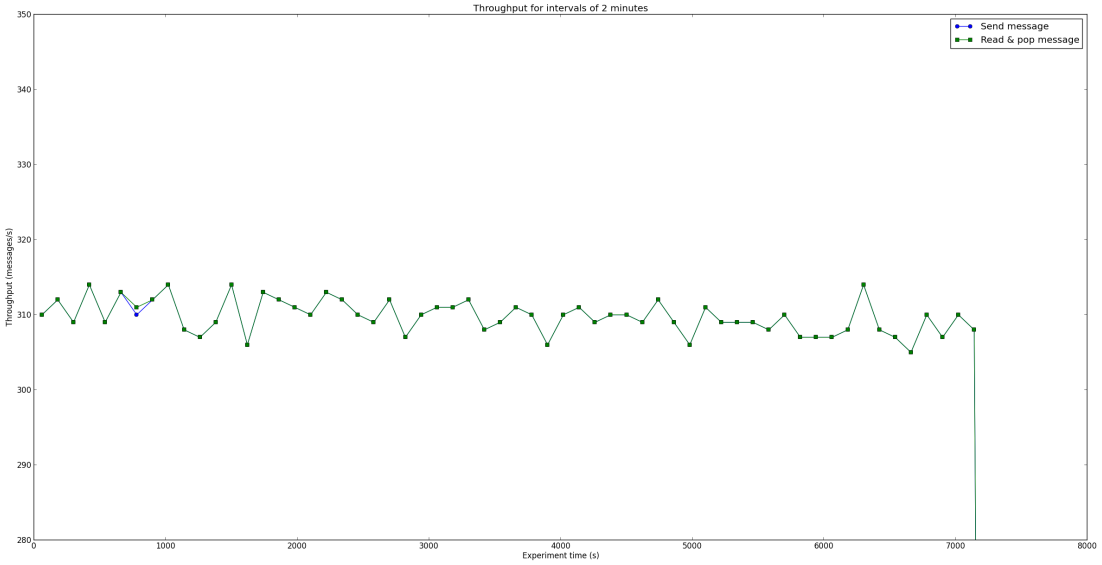

Figure 7.4.: Throughput of send and read operations

size is more controlled throughout the experiment, this leads to the flat behavior observed in the plots. Additionally, it is possible to the see that the queries on the queues are also stable given the implemented indexes in the database.

The controlled environment of the dryads contributes to small absolute variations in the results' metrics, however if the standard deviations are observed as a relative percentage of the mean it is possible to see that the variation is quite significant for the RT, around 50% for the read and send operations and 65% for the queue operations. Nevertheless it is worth noting that these values are in the order of tenths of milliseconds and therefore a higher susceptibility to noise is expected. From the large amount of data points it is possible to see that the obtained mean values are close to the population mean with high probability (i.e. 0.999).

During the experiments it was also possible to observe the resource consumption of the components of the system, although not plotted, observations showed that the system was very intensive in terms of CPU given the large amount of threads in the server and the memory consumption was in the order of 1-2 GB and stable during the duration of the experiment.

As a general remark, it can be inferred from this experiment that under small and controlled loads it is possible to obtain very fast average response times for different operations in the system and this behavior is stable over reasonable periods of time.

7.3. Experiment 2 - Exploring server parameters

7.3.1. Description

From the design, the authors suspect that a significant factor in the system's performance is how the clients are distributed among worker threads and how many worker threads are spawned in each middleware instance. To accommodate N clients with M servers there are two possible extremes: Create $\frac{N}{M}$ threads in each server, each one serving a single client or allocate 1 worker thread per server, each serving $\frac{N}{M}$ clients.

The expectation is that as the number of clients in a thread is increased the response time will grow because each client will suffer a larger waiting time in the queue until its request is serviced or the corresponding response in sent back. However, it is not expected that servicing each client with a single worker thread is the optimal solution since each worker thread in a server implies more context switches in the host machine, as well as an additional concurrent connection to the database which could result in slower transactions. The aim of this second experiment was to explore the effects of varying the two aforementioned parameters and observe if the assumption that the optimal value for response does not lie in an extreme is plausible.

To reduce the number of factors involved, an operational assumption is that the number of database connections in each server is always equal to the number of worker threads. This was decided prior to experimentation as it is clear that having less database connections means that worker threads will have to wait a certain interval before serving a request and therefore this will impact the negatively, and in such case that database connections are a limit then the worker thread number would be adjusted to keep the 1:1 ratio.

Table 7.3.: Setup of the servers for experiment No. 2

Worker threads	Clients per thread
50	2
10	10
5	20
2	50
1	100

For this experiment, two new types of clients were implemented, these are known as Alice and Bob clients. Their behavior is as follows:

Alice

- The client runs continuously until signaled to stop, the running time is controlled by a separate thread that signals all client threads when the experiment is over.
- During its execution the client send a message to a random sample of queues, selected from a list defined at the start of the experiment, with certain message size and without an specific receiver. The modifiable parameters for this client type is the number of queues where the message will be sent to and the message size. For this experiment, the number of queues is one and the message size is the maximum of 2000 characters, selected at random from an English text sample.

Bob

- The client runs continuously until signaled to stop, the running time is controlled by a separate thread that signals all client threads when the experiment is over.
- During its execution the client will check a random queue, selected from a list defined at the start of the experiment, for the top priority message and if any is found then it will pop it from the queue.

For this experiment, the total number of clients to be serviced is kept constant at 200, 100 Alices and 100 Bobs, while the server configuration varies to serve them, in this case there are 2 servers in separate machines. The clients are distributed evenly between worker threads such that each worker has the same number of Alices and Bobs. The values to be explored for the number of worker threads and the number of clients per thread is listed in table 7.3.

For each combination of the parameters, a 30 minute trace was executed with the implemented clients.

7.3.2. Results

Figures 7.5-7.9 illustrate the obtained traces for read and send response time for the different proposed configurations.

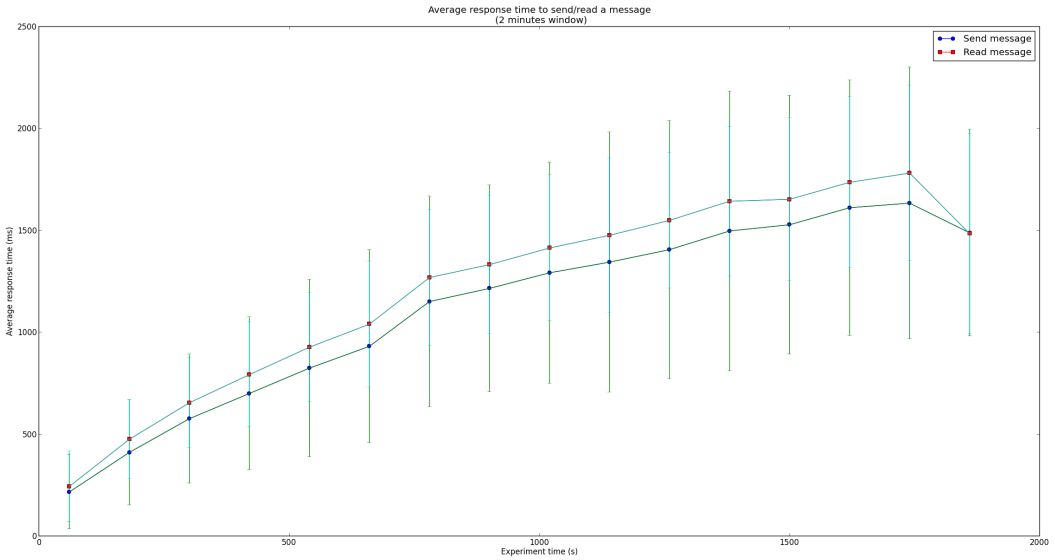

Figure 7.5.: Response time for sending and reading with 50 worker threads and 2 clients per thread

Additionally, figures 7.10 and 7.11 illustrate the variation of the minimum and maximum average response time values in each trace as a function of the server parameters. This minimum and maximum average is taken at the third and second to last intervals respectively for each of the traces, this in order to remove the effects of warm up and cool down times.

7.3.3. Analysis

First, it is important to note the fact that no statistical analysis was presented as in the previous experiment. This is because the system is no longer exhibiting a stationary behavior and therefore it is not correct to determine a response time for this experiment.

However, these results present important details about the behavior of the system. First, the results suggest that the initial proposition that the optimal performance is achieved in a non-extreme configuration is correct. From figures 7.10 and 7.11 it can be seen that the optimal value lies between the configuration with 10 and 5 worker threads. Moreover, it can be seen that having many concurrent threads lead to bigger variations in the response due, possibly due to the number of concurrent transactions in the database which are operating on overlapping sets of rows in the tables for messages and queues. On the other hand, having many clients in a single thread increases the response time in a more stable fashion because it is the result of queuing several clients in a single worker which serves them sequentially.

An important concern to address is why the behavior is not stationary in such short time span for each experiment, reads should be fast given that there are indexes to support each of the possible queries and surprisingly, sends are faster than reads which suggests that maybe the indexes are not correct in the database. The fact that the slope of the traces gets smaller as the response time gets bigger implies that the database size is likely the responsible for the increase of response time during each of the traces, but

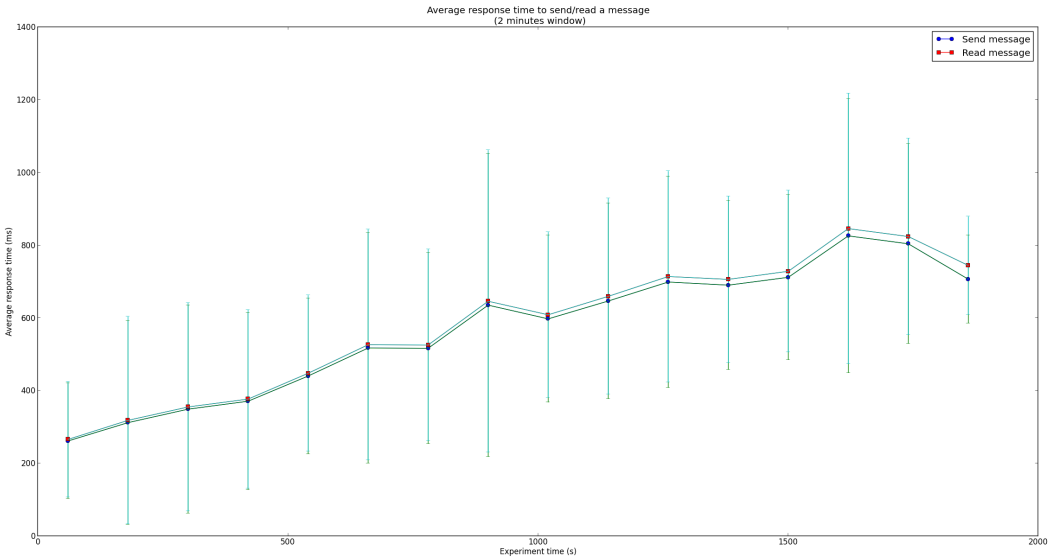

Figure 7.6.: Response time for sending and reading with 10 worker threads and 10 clients per thread

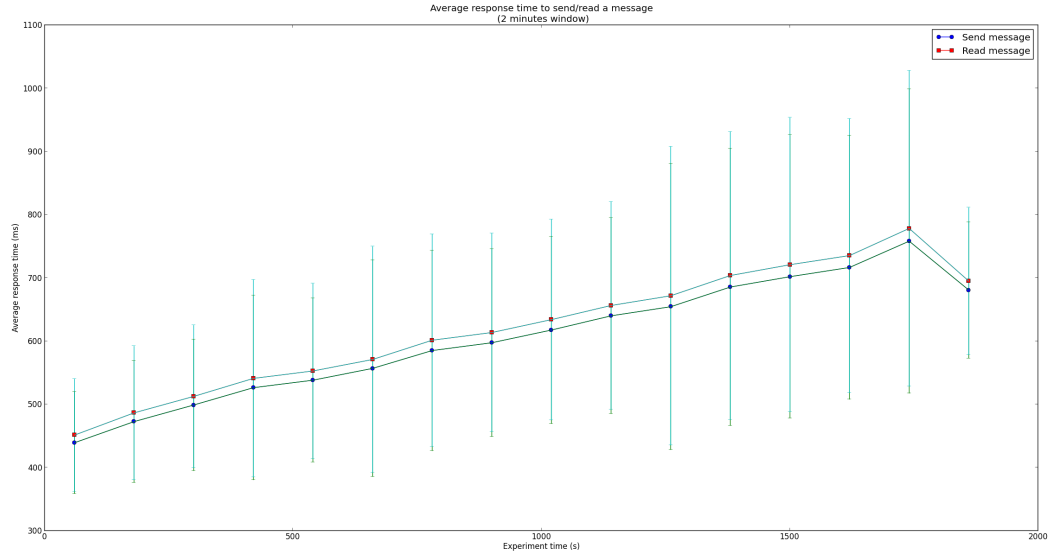

Figure 7.7.: Response time for sending and reading with 5 worker threads and 20 clients per thread

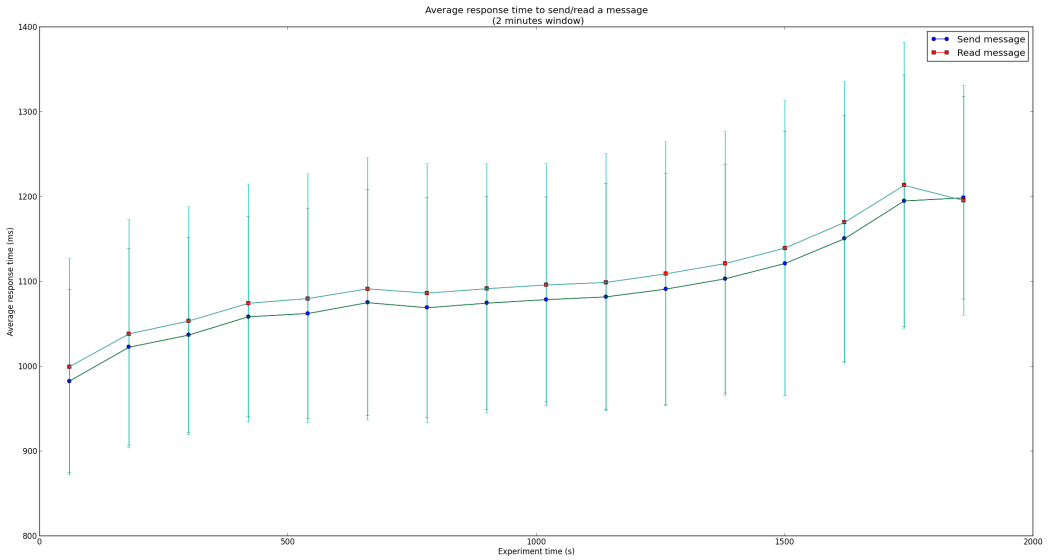

Figure 7.8.: Response time for sending and reading with 2 worker threads and 50 clients per thread

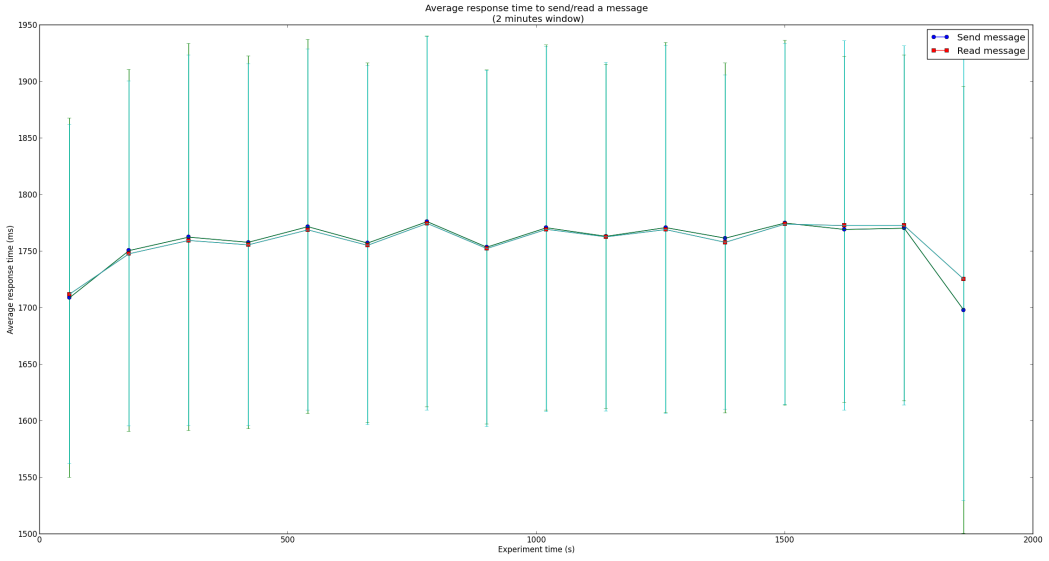

Figure 7.9.: Response time for sending and reading with 1 worker threads and 100 clients per thread

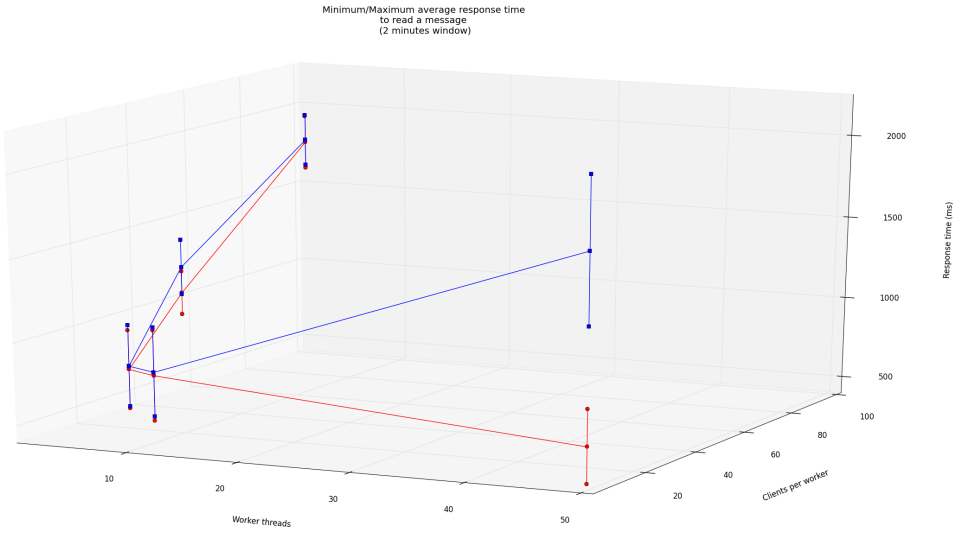

Figure 7.10.: Response time for reading a message as a function of the number of worker threads and clients per thread

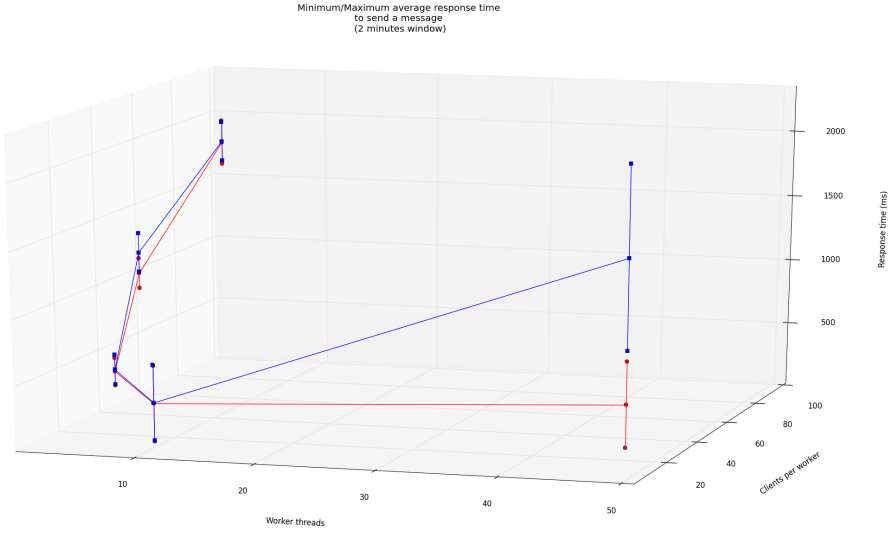

Figure 7.11.: Response time for sending a message as a function of the number of worker threads and clients per thread

Table 7.4.: Setup of the clients for experiment No. 3

Sender clients (Alice)	Reader clients (Bob)
50	50
50	25
50	16
50	12
50	10

this doesn't explain why the sends are almost as slow as the reads. The explanation for this could be the fact that since each worker serves the same number of Alice and Bob clients, then a send operation will likely wait a significant fraction of the time of a read operation before being serviced in a worker, therefore the response time will be close between read and send operations even if the database times for each of these is not similar. Since there is no instrumentation in the server side for this experiment, another experiment will be performed to clarify this phenomena. Additionally, the following experiments will explore if the database size is really the culprit.

7.4. Experiment 3 - Exploring the database size impact

7.4.1. Description

The purpose of this experiment is to explore how the size of the database impacts the response time for read and send operations, and also verify the conjecture presented in the previous analysis which stated that the high send response time was solely due to the fact that every sender client (a.k.a Alice client) shared a worker thread with a reader client (i.e. Bob client).

This experiment is divided in 5 traces of 10 minutes each where the database size is indirectly varied by reducing the number of Bob clients in each step according to the values in table 7.4. In this setup the Alice clients are distributed together in the same set of worker threads split between both servers evenly and the Bob clients share a disjoint set of worker threads. There are 2 servers for this experiment, each one with 25 worker threads with capacity of 2 clients.

7.4.2. Results

Figures 7.12 and 7.13 show the response time over time for the different configurations of clients for read and send operations respectively.

7.4.3. Analysis

From the difference between figure 7.13 and figures 7.5-7.9, it is clearly confirmed that the high response time seen in the later figures is completely due to the waiting time

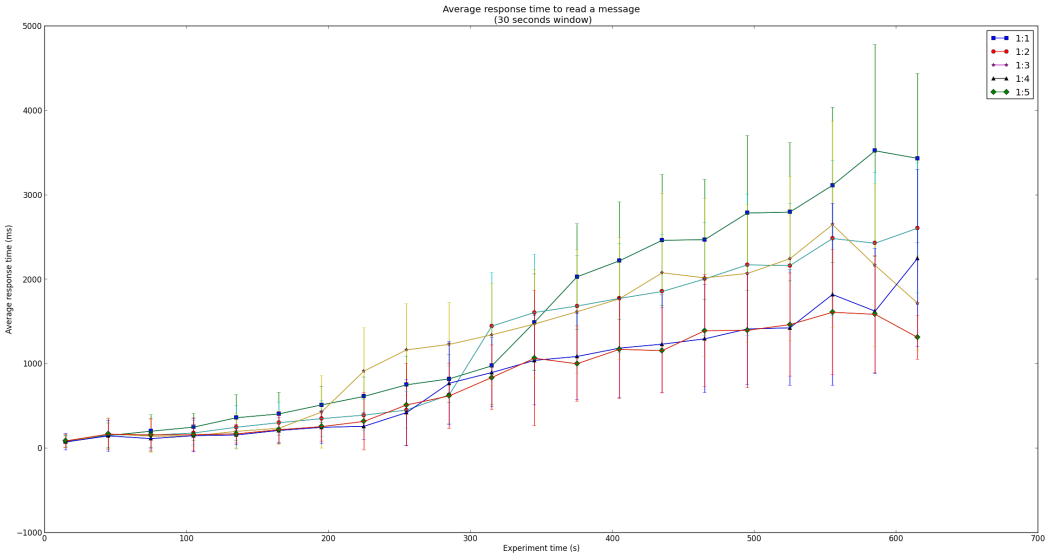

Figure 7.12.: Response time for read operations with different reader:sender ratios

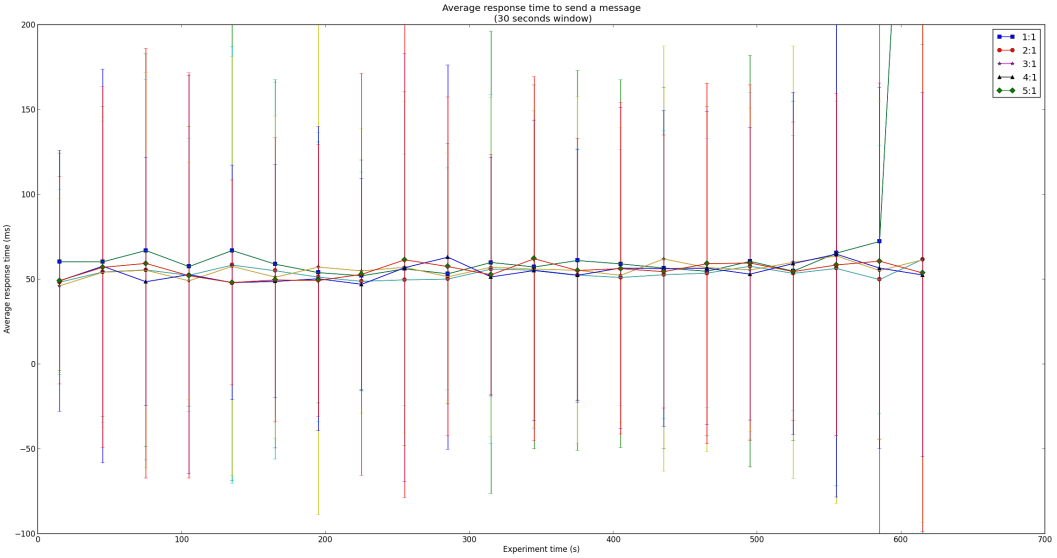

Figure 7.13.: Response time for send operations with different sender:reader ratios

caused by the read operations and not the send operation itself. This exposes a critical feature of the system, and that is that the distribution of clients in the worker threads is significant to the performance of the system. From this result and the results of experiment No. 2 it is possible to infer that the number of clients per thread should be kept reasonably small to ensure good performance, moreover for optimal performance a mechanism could be implemented to ensure that clients with slow or resource intensive behavior do not block other more "casual" clients, for example a load balancer that moves already connected clients between workers depending on their behavior, this is out of the scope of the current implementation but it is an interesting possibility to explore in this particular design for the messaging system.

With respect to the first objective of this experiment, it is not statistically accurate to affirm that the database size affects the response time for read operations because, as pictured in figure 7.12, the response time for the different ratios is not statistically different, however the differences in the mean suggest that as the number of readers diminishes, which in turns means that the database size increases faster over time because the number of senders is constant, the response time also diminishes which is what was proposed for this experiment.

The trend observed from this result reinforced the idea that there was a problem with the read queries in the system, hence the authors started reviewing the implementation and noticed that there was no index for doing the bi-dimensional sorts on message creation time and priority which are required by the read operations, which explains the poor behavior of the read operations and the steep increase in read response time as the database grows in size. To confirm this, a new experiment was performed after creating the missing indexes in the database, this will be presented in section 7.6.

7.5. Experiment 4 - Running under less than full load

7.5.1. Description

Before the missing index was identified and fixed, there was another experiment which explores the effect of the number of clients, which represents the load of the system, on the response time for read and write operations.

In this experiment the Bob and Alice clients, as described in 7.3.1, are used. The experiment is divided in several independent steps with varying number of clients but keeping a one to one ratio, before each step the database is cleaned and the schema redeployed, including the 100 predetermined queues as in previous experiments. The number of clients is increased by 20 (10 Alice, 10 Bob clients) in every step which occurs every 10 minutes, hence the total running time of the experiment was 2 hours.

The idea of this experiment is to isolate the effect of the number of clients on the performance of the system, therefore the server parameters were maintained constant throughout the experiment at the values presented in table 7.5 for both servers, this means that only by the last step of the experiment the system is at full load. It is also worth noting that the Bob and Alice clients were isolated in disjoint sets of worker threads to avoid the interference seen in the results of 7.3.

Table 7.5.: Setup of the server for experiment No. 4

Worker threads	24
Clients per thread	5

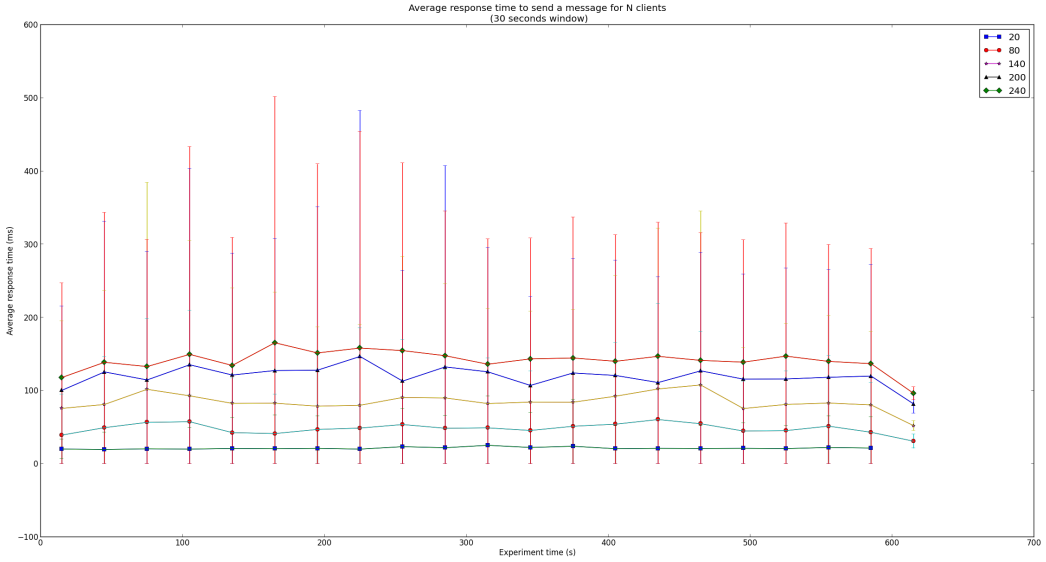

Figure 7.14.: Response time for send operations with different number of clients

The expectation in this experiment is that the response time is proportional to the number of clients and that in the given range this relation can be approximated as a linear function.

7.5.2. Results

Figures 7.14 and 7.15 illustrate the behavior of the response time for send and read operations respectively over the interval of each experiment step. To avoid cluttering the plot, only 5 traces are presented in each figure.

In these figures it is possible to observe that the response time for the send operation is stationary, therefore it is possible to do statistical analysis over the data after removing the time dependency. The results of this statistical analysis are summarized in table 7.6.

Finally, figure 7.16 illustrates the functional dependency between the average response time and the number of clients in the system.

7.5.3. Analysis

From the trace figures, it is deduced that the average response time is proportional to the of clients in the system for a constant server configuration. However, in the case of read operations the difference in the means is not statistically significant as it falls between one standard deviation of the measurements.

As expected, the reads are largely affected by the missing index which causes a steep increase in the response time as the experiment progresses, however in this case it can

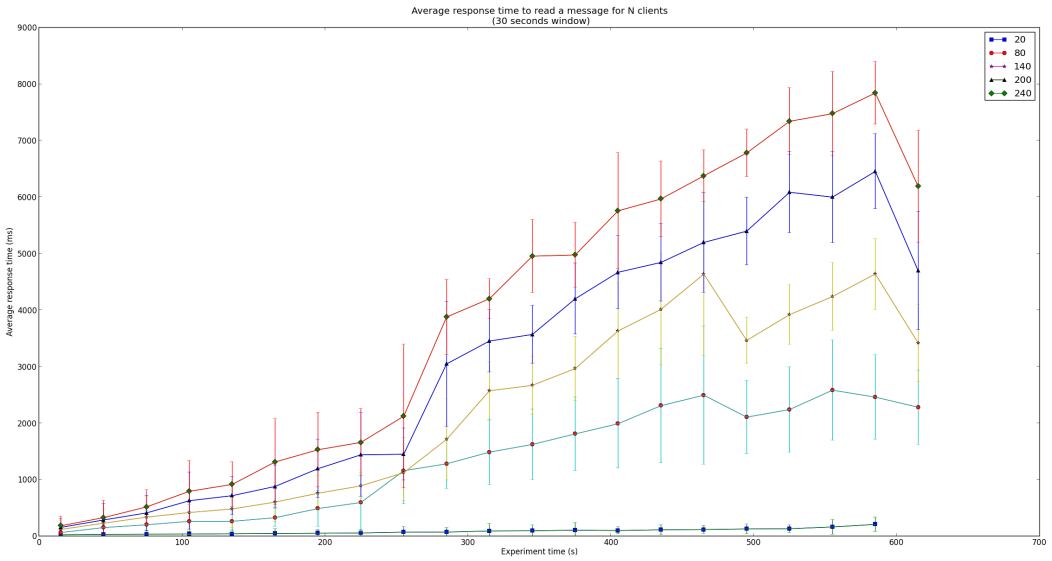

Figure 7.15.: Response time for read operations with different number of clients

Table 7.6.: Statistical results from experiment No. 4 for the send response time

Number of clients	$\overline{X}$ (ms)	$\sqrt{S^2}$ (ms)	95% **CI for** $\overline{X}$ **(t-student)**	95% **CI for** $\sqrt{S^2}$ (χ^2)
20	21.8	44.7	21.6 – 22.0	44.6 – 44.8
40	26.9	58.5	26.6 – 27.1	58.4 – 58.7
60	37.4	98.9	37.1 – 37.8	98.7 – 99.2
80	49.0	101.3	48.6 – 49.3	101.1 – 101.6
100	60.3	111.1	59.9 – 60.7	110.8 – 111.4
120	73.1	133.6	72.7 – 73.6	133.3 – 133.9
140	86.5	153.2	85.9 – 87.0	152.8 – 153.6
160	98.0	159.8	97.33 – 98.4	159.4 – 160.2
180	106.0	159.5	105.5 – 106.6	159.1 – 159.7
200	122.4	188.4	121.8 – 123.1	188.0 – 188.9
220	131.4	182.9	130.7 – 132.0	182.5 – 183.3
240	146.0	215.3	145.3 – 146.8	214.8 – 215.8

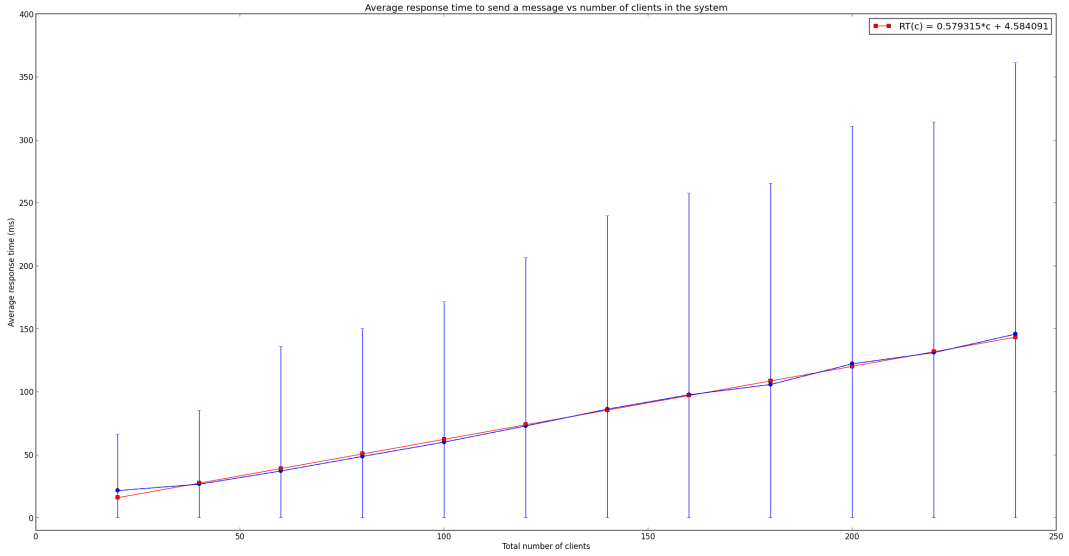

Figure 7.16.: Response time for a send operation as a function of the number of clients

be seen that the difference in the means for different number of clients is statistically significant.

Figure 7.16 confirms the initial hypothesis about a linear relation between number of clients and response time in average, however it still must be noted that this relation is not statistically significant when the standard deviations are considered.

In order to confirm the hypothesis, one could argue that the experiment should be repeated a few times to obtain smaller standard deviations. However, from the results of table 7.6 it is possible to see that the values obtained for the mean and standard deviation for each configuration of the experiment are already accurate with high probability (i.e. 0.95), the mean and standard deviation for each experiments were evaluated under t-student and chi-square tests respectively, assuming that each data point is an iid[1] normal variable which is believed to be a good assumption given the large number of data points for each experiment which is around 20K and the law of large numbers. It is not expected that the standard deviations will shrink as we increase the data points, therefore a new concern is raised because these obtained statistics indicate that the system for this kind of load has a response time with very large variation which makes it difficult to ensure a level of quality for the clients.

Observing the increasing trend for the standard deviations as the number of clients is increased, the authors suspect that the large variance in the response time is due to the concurrency components of the system such as the non-blocking I/O in the worker threads and the parallel database connections. This agree with the fact that it is not a random noise which can be eliminated by increasing the data points and rather a feature of the design and implementation.

[1] Independent identically distributed

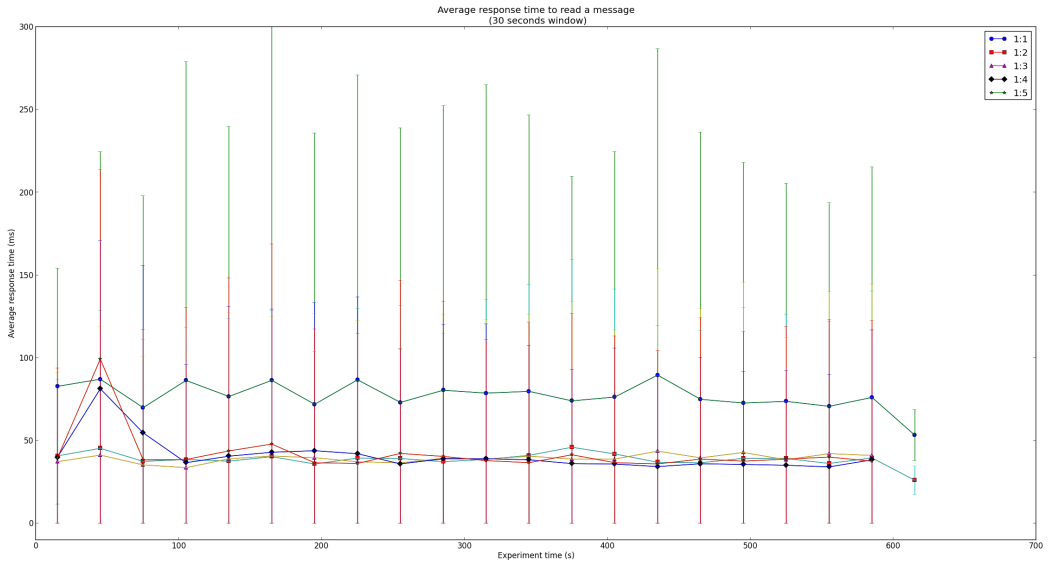

Figure 7.17.: Response time for a read operation with a decreasing number of readers

Table 7.7.: Statistical results from experiment No. 5 for the read response time

Number of readers	$\overline{X}\ (ms)$	$\sqrt{S^2}\ (ms)$	95% **CI for** $\overline{X}$ **(t-student)**	95% **CI for** $\sqrt{S^2}\ (\chi^2)$
50	78.2	169.7	$77.5 - 78.9$	$169.2 - 170.1$
25	39.1	89.9	$38.8 - 39.5$	$89.6 - 90.1$
16	39.6	88.4	$39.2 - 40.0$	$88.1 - 88.7$
12	38.3	74.5	$37.8 - 38.7$	$74.2 - 74.7$
10	39.2	87.3	$38.6 - 39.7$	$86.9 - 87.6$

7.6. Experiment 5 - Revisiting database size impact

7.6.1. Description

After creating new bidimensional index in the database schema, the experiment from section 7.4 was repeated.

7.6.2. Results

Figures 7.17 and 7.18 are the analogous of figures 7.12 and 7.13 respectively.

Since in this case both read and send response times exhibit an stationary behavior it is possible to perform a statistical analysis of the data, the results of it are summarized in tables 7.7 and 7.8.

Finally, figure **??** illustrates the functional dependency between the read response time and the number of readers.

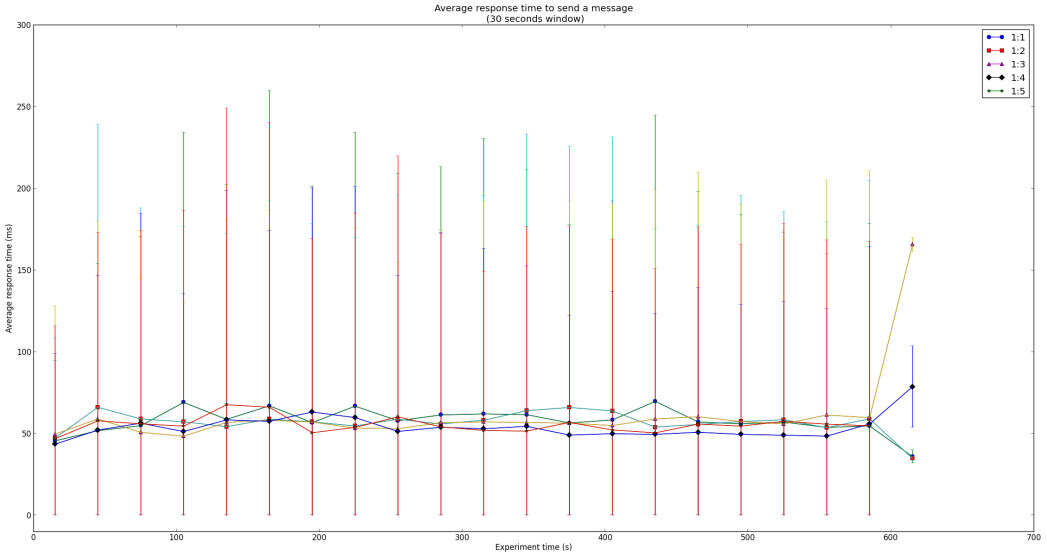

Figure 7.18.: Response time for a send operation with a decreasing number of readers

Table 7.8.: Statistical results from experiment No. 5 for the send response time

Number of readers	$\overline{X}\ (ms)$	$\sqrt{S^2}\ (ms)$	95% **CI for** $\overline{X}$ **(t-student)**	95% **CI for** $\sqrt{S^2}\ (\chi^2)$
50	60.5	151.6	$59.9 - 61.0$	$151.3 - 152.0$
25	58.1	135.6	$57.7 - 58.6$	$135.2 - 135.9$
16	56.6	129.5	$56.2 - 57.1$	$129.2 - 129.8$
12	53.5	105.7	$53.2 - 53.9$	$105.4 - 105.9$
10	55.4	129.3	$54.9 - 55.8$	$129.0 - 129.6$

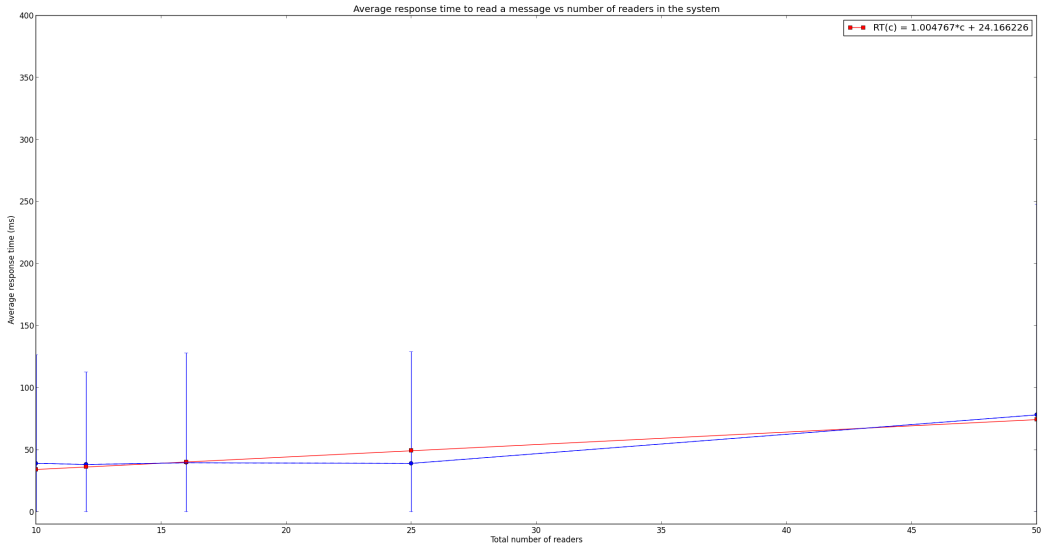

Figure 7.19.: Response time for a read operation as a function of the number of readers

7.6.3. Analysis

By comparing the traces obtained in this experiment with 7.4, it is confirmed that the introduction of a new index corrected the steep increase of the response time as a function of the database size, moreover it significantly reduced the average response time for read operations and brought it to agreement with the expectation of it being lower than the send response time.

An observation of the values in tables 7.7 and 7.8 makes it clear that the system presents a widely varying random behavior, matching the observations made on the last experiment. Unfortunately, it does not provide evidence to the cause. As in the last experiment, the chi-square and t-student statistical tests give provide confidence that the wide variation will not be eliminated by taking more data points.

Figure 7.19 shows that there exists a linear relation between the number of readers, and therefore database size, and the response time of a read operation. However, it seems to contradict the initial assumption that the database size increases the response time because as we increase the number of readers, effectively decreasing the database size, the response time also grows instead of decreasing. In this case, the explanation is that the effect of having more concurrent readers is more significant to the increase of the response time than the database size, this will be explored in the benchmark experiments.

8. Experiments and Benchmarks

8.1. Experimental setup

Following the experiments in the dryad machines, now the focus is shifted towards analyzing the performance of the system in a production-like environment such as the one provided by the Amazon Elastic Cloud Cluster.

In this environment it is expected that the performance will decrease and become more noisy due to the fact that machines used to run the software will be virtual machines which are not isolated as it was the case in the dryad cluster, additionally the hardware specifications of some of them are not as high as the dryad machines. However, this change in environment allows for further exploration of the systems characteristics, and more flexibility and scalability in the experiments.

With respect to the previous experiments, there was a change in the software used, the java virtual machine version was updated to 1.7.0 for the following experiments.

In the Amazon cluster there are several predefined options for the hosting machines[1], given the budget constraints and the observed hardware requirements of the system the following choices were made with respect to the types of machines to be used:

- Database server: The database server requires a significant amount of memory, CPU, network and disk I/O, therefore it was be hosted using a m1.xlarge server instance.

- Servers instances: The server instances are mostly CPU intensive, but also require a non-negligible amount of virtual memory. Therefore they were hosted using m1.large instances.

- Client instances: As the number of clients increases, there is a need for more CPU and network I/O in the hosting machine. However, this value is expected to be less than the requirements of the server and database instances, hence the clients were hosted in m1.medium instances.

For the OS of the instances, Red Hat Enterprise Linux 6 was chosen as it is a known environment for the authors and it is expected to offer adequate support and performance for the experiments.

In this chapter, the organization of the experiment has been slightly modified and the analysis is presented alongside the results for a clearer flow of information.

[1] http://aws.amazon.com/ec2/instance-types/

8.2. Experiment 6 - Varying the load

8.2.1. Description

The next experiment aims to understand how the different factors that characterize the load affect the performance of the system, based on the system's design the following features were chosen as representative of the load:

- Message size: Number of characters in the messages being sent.
- Number of target queues: Number of queues to which a message is sent simultaneously.
- Number of reader clients: Number of clients which only perform read operations, this client is the same Bob client from section 7.3.1.
- Number of sender clients: Number of clients which only perform send operations, this client is the same Alice client from section 7.3.1.

From the previous experiments, it is obvious that the number of concurrent clients has a direct impact in the system performance, more over the balance between them is expected to affect the stability of the system over time.

Since the system was designed as a closed system, i.e. the clients wait for a response before making an new requests, the load for a fixed number of clients can't be controlled by adjusting the rate of operations initiated by the client. Instead, the size of the messages and the number of target queues can be varied to modify the load conditions, the first of these parameters affects the performance since it changes the load on the network and the size of the information stored in the database, the second one also has an effect in the information transmitted over the network but the most significant factor is expected to be its effect on the number of database operations required to send a message.

Given the number of factors and their valid ranges, it was chosen to perform a 2^k factorial experiment to explore the different interactions and the effect of these parameters on the system performance. The values to be evaluated are presented in table 8.1.

For each combination, a 30 minute trace was executed with the following hardware configuration:

- 1 database server with a maximum of 100 connections
- 2 middleware instances, each one with 10 worker threads and 25 clients per worker.
- 2 instances for running the client threads, split evenly and mixed between Alice and Bob clients.

The values were selected under the assumption that the performance behaves approximately linear between both endpoints for all 4 parameters.

Table 8.1.: Setup of the load for experiment No. 6

Parameter	Min	Max
# of Alice clients	120	250
# of Bob clients	120	250
Message size	100	2000
Number of target queues	1	10

Table 8.2.: Statistical results from experiment No. 6 for the read response time

Message size	**Target queues**	**Alice clients**	**Bob clients**	$\overline{X}\ (ms)$	$\sqrt{S^2}\ (ms)$	95% **CI for** $\overline{X}$	95% **CI for** $\sqrt{S^2}$
100	1	120	120	184.4	98.1	$184.2 - 184.6$	$98.0 - 98.3$
100	1	120	250	271.7	64.8	$271.6 - 271.9$	$64.7 - 64.9$
100	1	250	120	256.0	189.5	$255.5 - 256.4$	$189.1 - 189.8$
100	1	250	250	480.0	299.3	$479.2 - 480.7$	$298.8 - 299.8$
100	10	120	120	290.2	103.6	$289.9 - 290.4$	$103.4 - 103.8$
100	10	120	250	538.0	171.4	$537.6 - 538.4$	$171.1 - 171.7$
100	10	250	120	542.2	187.8	$541.5 - 542.9$	$187.4 - 188.3$
100	10	250	250	574.5	188.5	$574.0 - 575.0$	$188.2 - 188.9$
2000	1	120	120	252.0	350.5	$251.1 - 252.9$	$349.9 - 351.2$
2000	1	120	250	270.1	79.9	$270.0 - 270.3$	$79.8 - 80.0$
2000	1	250	120	448.0	578.0	$446.1 - 450.0$	$576.6 - 579.4$
2000	1	250	250	687.2	1271.2	$683.5 - 690.8$	$1268.6 - 1273.8$
2000	10	120	120	310.6	138.2	$310.2 - 311.0$	$137.9 - 138.5$
2000	10	120	250	569.8	210.9	$569.2 - 570.3$	$210.5 - 211.3$
2000	10	250	120	615.4	244.5	$614.4 - 616.4$	$243.8 - 245.2$
2000	10	250	250	652.7	259.5	$652.0 - 653.4$	$259.0 - 260.0$

8.2.2. Results and Analysis

In this section, several traces and metrics which represent the results of this factorial experiment will be presented. As an starting reference point, tables 8.2 to 8.4 contain the values of the response time and throughput for the different configurations, these were obtained assuming a stationary behavior in each of the traces, the validity of this assumption will be illustrated and contested throughout this section and although not completely exact, it is considered an acceptable approximation.

An initial remark derived from these tables is that the response time is proportional to the number of clients in the system, the number of target queues and the message size. However, for the throughput there exists a more complex relationship to be analyzed. Another noteworthy point is that the expectation of having a noisier environment holds in the EC2 infrastructure and the overall performance seems to have decreased by a significant factor in comparison to the dryad environment, this conclusion is made by comparing at the values of experiment 4 for the highest number of clients and the values of the response time for the configuration with 120 senders and readers, 2000 message size and 1 queue in the current experiment, the discrepancy of the mean is of about

Table 8.3.: Statistical results from experiment No. 6 for the send response time

Message size	**Target queues**	**Alice clients**	**Bob clients**	$\overline{X}\ (ms)$	$\sqrt{S^2}\ (ms)$	95% **CI for** $\overline{X}$	95% **CI for** $\sqrt{S^2}$
100	1	120	120	133.3	84.2	$133.1 - 133.4$	$84.1 - 84.3$
100	1	120	250	227.3	52.5	$227.2 - 227.4$	$52.4 - 52.6$
100	1	250	120	246.1	187.3	$245.8 - 246.4$	$187.1 - 187.5$
100	1	250	250	269.8	217.3	$269.4 - 270.2$	$217.1 - 217.6$
100	10	120	120	288.0	101.8	$287.7 - 288.3$	$101.6 - 102.0$
100	10	120	250	488.2	143.6	$487.7 - 488.7$	$143.3 - 144.0$
100	10	250	120	540.8	187.0	$540.3 - 541.3$	$186.7 - 187.4$
100	10	250	250	579.3	187.0	$578.8 - 579.8$	$186.6 - 187.3$
2000	1	120	120	210.4	320.7	$209.7 - 211.1$	$320.2 - 321.2$
2000	1	120	250	263.4	78.4	$263.2 - 263.6$	$78.2 - 78.5$
2000	1	250	120	356.9	510.5	$355.9 - 358.0$	$509.7 - 511.2$
2000	1	250	250	650.5	1200.4	$647.1 - 653.9$	$1198.0 - 1202.8$
2000	10	120	120	310.7	135.4	$310.3 - 311.0$	$135.1 - 135.6$
2000	10	120	250	553.6	203.0	$552.8 - 554.3$	$202.5 - 203.5$
2000	10	250	120	623.7	244.6	$623.0 - 624.3$	$244.1 - 245.1$
2000	10	250	250	666.8	257.5	$666.1 - 667.6$	$257.0 - 258.1$

Table 8.4.: Statistical results from experiment No. 6 for the send throughput

Message size	**Target queues**	**Alice clients**	**Bob clients**	$\overline{X}\ (\frac{msg}{s})$	$\sqrt{S^2}\ (\frac{msg}{s})$	95% **CI for** $\overline{X}$	95% **CI for** $\sqrt{S^2}$
100	1	120	120	899.1	45.4	$878.4 - 919.8$	$34.7 - 65.5$
100	1	120	250	527.6	12.3	$521.9 - 533.2$	$9.4 - 17.8$
100	1	250	120	1015.5	119.6	$961.0 - 1069.9$	$91.5 - 172.8$
100	1	250	250	926.3	99.0	$881.2 - 971.3$	$75.7 - 142.9$
100	10	120	120	416.4	35.6	$400.2 - 432.6$	$27.3 - 51.5$
100	10	120	250	245.6	28.3	$232.8 - 258.5$	$21.7 - 40.9$
100	10	250	120	462.1	53.9	$437.6 - 486.6$	$41.2 - 77.8$
100	10	250	250	430.0	38.6	$412.5 - 447.6$	$29.5 - 55.7$
2000	1	120	120	569.4	41.9	$550.3 - 588.5$	$32.1 - 60.5$
2000	1	120	250	455.3	9.9	$450.8 - 459.7$	$7.5 - 14.2$
2000	1	250	120	699.9	87.0	$660.3 - 739.5$	$66.6 - 125.7$
2000	1	250	250	384.2	48.1	$362.3 - 406.1$	$36.8 - 69.5$
2000	10	120	120	386.1	28.9	$373.0 - 399.3$	$22.1 - 41.7$
2000	10	120	250	216.7	28.9	$203.6 - 229.9$	$22.1 - 41.7$
2000	10	250	120	400.7	44.5	$380.5 - 421.0$	$34.0 - 64.3$
2000	10	250	250	374.8	34.1	$359.2 - 390.3$	$26.1 - 49.3$

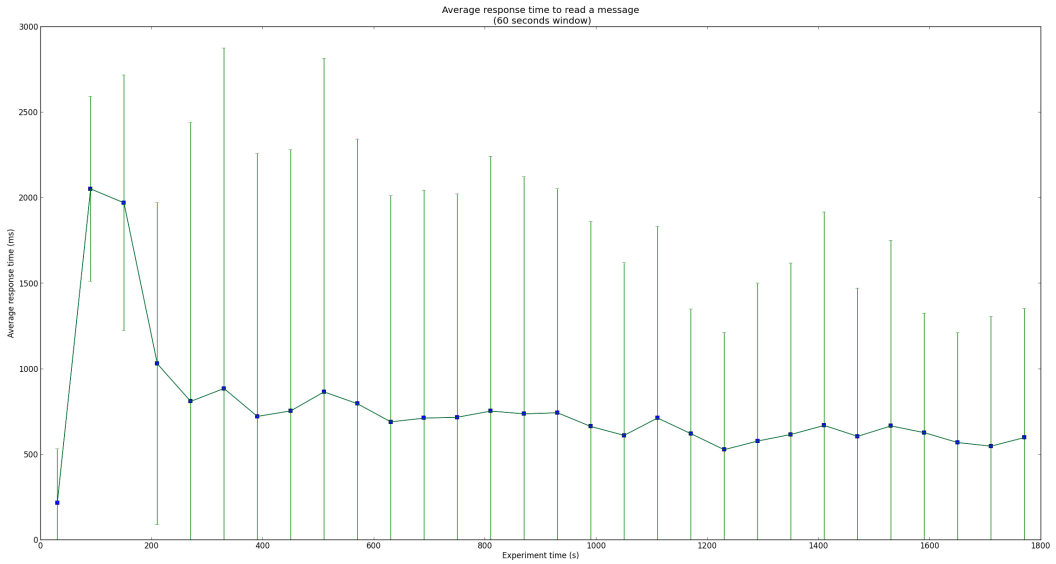

Figure 8.1.: Response time for reading a message of size 2000, sent to one queue when 250 reader and sender clients are present in the system.

$80\ ms$, that is a 50% increase and the difference in the standard deviation is a 100% increase.

Now, it is important to address some values which seem unusually high in the tables. In this case only one of them seems obviously off, this is the standard deviation for the read and send response times for the configuration with 2000 characters message size, 1 queue and 250 sender and reader clients. In order to see if there is an unusual phenomena causing this outlier, the trace was plotted and it is presented in figures 8.1 and 8.2.

From these figures it is observed that no obvious defect was present, from the available information it is unclear how this deviation was attained given that the number of points in the experiment is significant. Note that for the calculation of the statistical metrics, the warm up and cool down intervals were ignored, these were determined as the time before 300 seconds and the time after 1500 seconds respectively.

Nevertheless, figure 8.1 exhibits an interesting property of the response time for read operations in the system. That is an initial peak in the response time at the beginning of the experiments, after observation of the different traces for each configuration the authors found that this is not an isolated feature and that it happens for all configurations. The explanation for this feature is most likely due to the initial disk access in the database before the information is available in main memory, this was found by looking at the disk I/O metrics provided for the database instance.

Now in order to better understand the system behavior, some particular traces will be presented to explain features of the system. Figure 8.4 shows the response time traces for unbalanced number of clients under the highest load configuration for messages and number of queues, namely 2000 characters and 10 queues. On the other hand, figure 8.3 depict the traces when the number of clients are balanced. Additionally the throughput for all four configurations is presented in figure 8.5.

These figures are presented to establish a baseline with the data seen in the last exper-

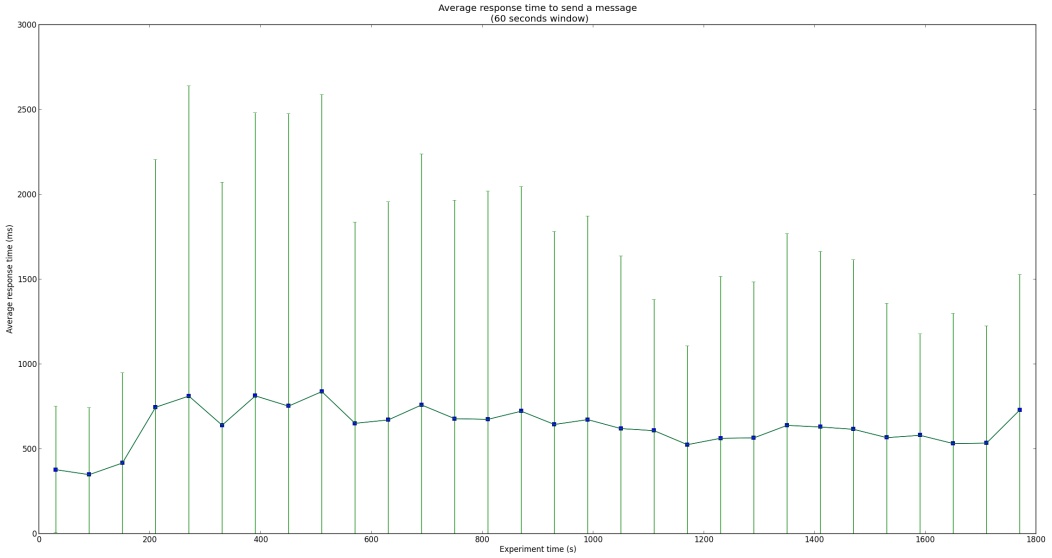

Figure 8.2.: Response time for sending a message of size 2000 to one queue when 250 reader and sender clients are present in the system.

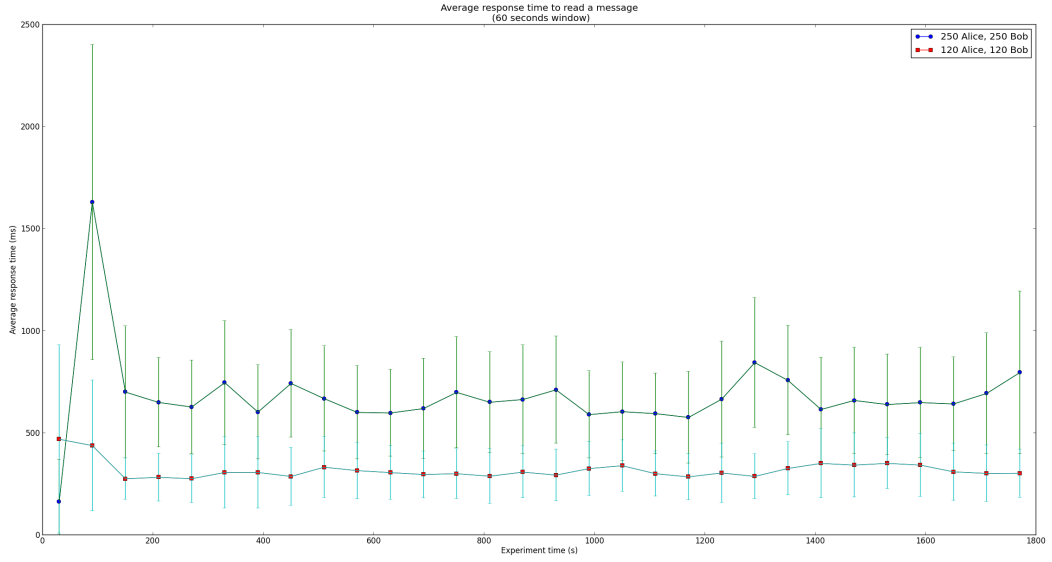

Figure 8.3.: Response time for reading a message with balanced number of clients

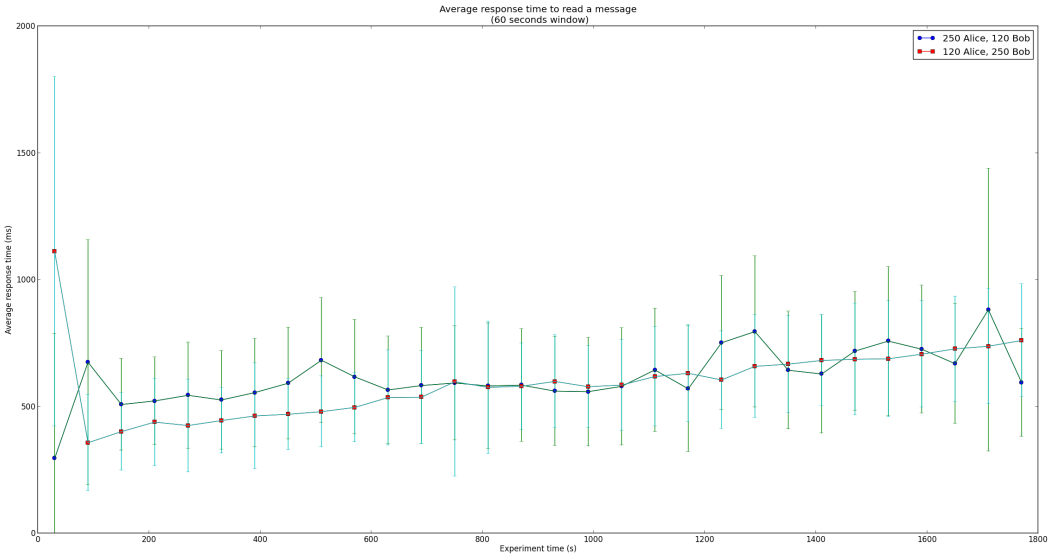

Figure 8.4.: Response time for reading a message with an unbalanced number of clients

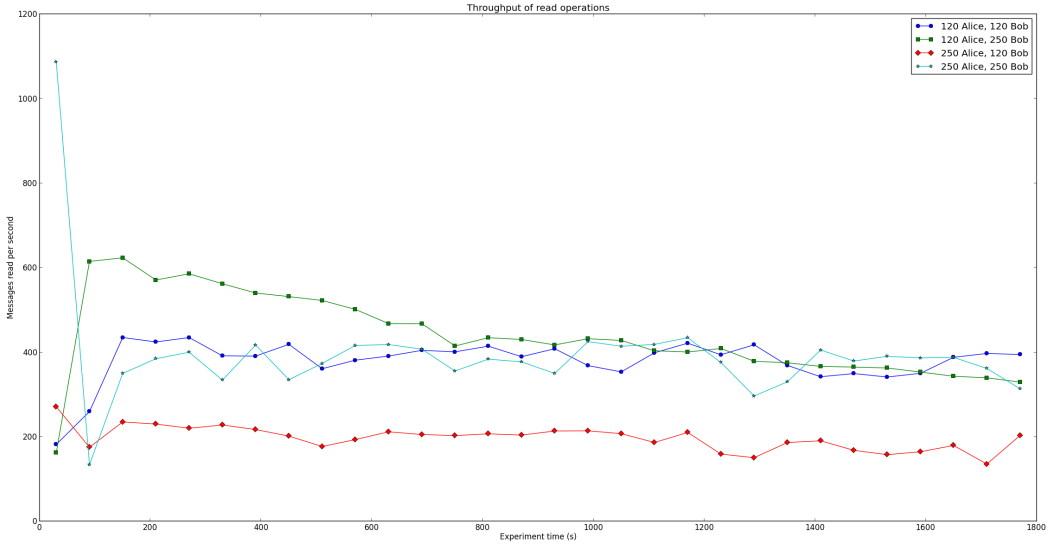

Figure 8.5.: Throughput of reading operations for different number of clients

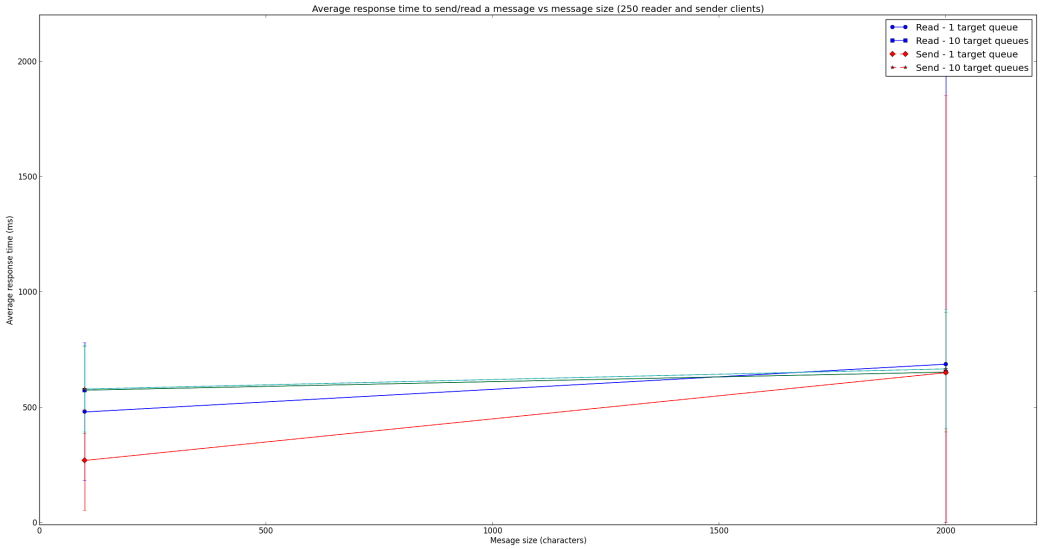

Figure 8.6.: Response time as a function of the message size under maximum client load

iment performed in the dryad, i.e section 7.6. In these results it is possible to observe again that the response time increases in proportion to the number of clients, and in contrast to the last experiment it is also possible worth noting how the database size, which increases in the case of unbalanced client numbers, does affect the response time.

Additionally, figure 8.5 shows more clearly how the reading throughput is largely affected in the case of a small number of readers compared to the case where the number of readers is minimum but equal to the number of senders. This is because of the different values for response time.

An interesting trend to observe is how the message size affects the response time and throughput under the maximum number of clients, this is presented in figure 8.6. The authors expect that the effect of this parameter is not very significant as it can be argued that for modern computing system the difference between 100 and 2000 characters, usually of 2 bytes size, is negligible.

This figure shows that the initial assumption was wrong, and the message size does have a significance in the response time. For the evaluated configuration the increase in message size caused an increase in the average response time for both send and reads of about $200 - 400\ ms$. This phenomena can be explained by the fact the network performance in the EC2 environment can be poor and therefore the size of the messages transmitted across the network can play a significant role in the performance

In order to provide a full picture of the interactions of the chosen load parameters, the system of equations for the send and read response times given the factors and its covariate variables was solved and the following linear relations were obtained:

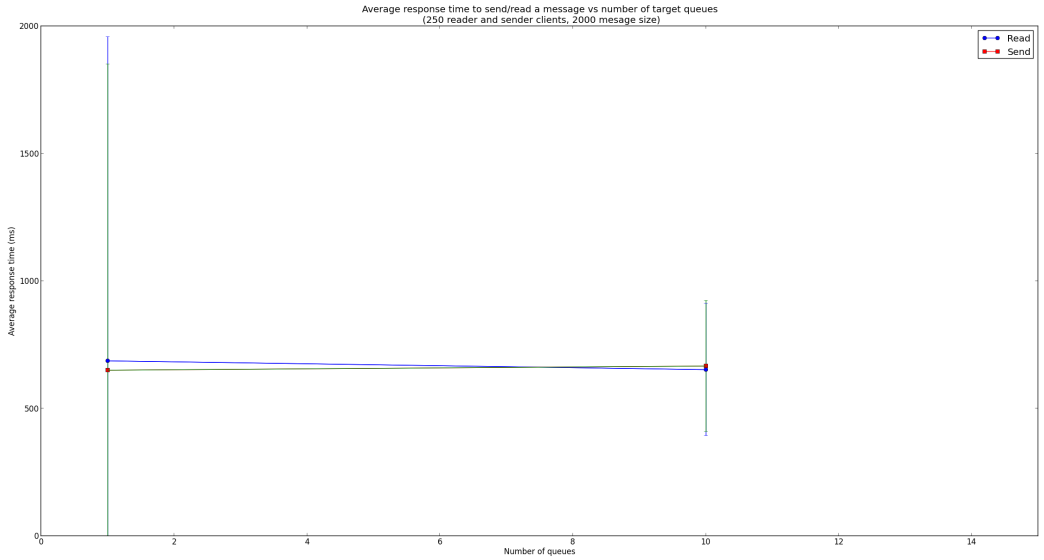

Figure 8.7.: Response time as a function of the number of target queues under maximum client load

$$\begin{aligned} RTread(m,q,s,r) = 0.0544m - 55.8q - 1.74s - 1.29r \\ -0.0078mq + 0.00036ms - 0.0013mr + 0.86qs + 0.82qr \\ +0.04sr + 0.000011mqs + 0.00015mqr + 0.000012msr \\ -0.0091qsr - 0.0000013mqsr + 205.3\ [ms] \end{aligned}$$

$$\begin{aligned} RTsend(m,q,s,r) = 0.21m - 14.1q + 2.6s + 2.42r - \\ 0.026mq - 0.0024ms - 0.003mr + 0.35qs + 0.29qr \\ -0.018sr + 0.00031mqs + 0.00036mqr + 0.000044msr \\ -0.0019qsr - 0.0000048mqsr - 121.7\ [ms] \end{aligned}$$

Looking closely at these equations, it is possible to see that the initial assumption of linearity is wrong and the response time does not behave as a hyperplane given the 4 load parameters studied in this experiments. This is deduced from the fact that the equations offer contradictory coefficients to what has been observed in the plots and the initial hypothesis. For example, for the read time it indicates a significant inversely proportional relation between the number of queues and the response time, however figure 8.7 shows that this relation should be proportional and it is actually not very significant.

8.3. Experiment 7 - Scaling the server

8.3.1. Description

This final experiment aims to explore the scaling properties of the middleware, benefiting from the flexibility of the EC2 infrastructure this experiment runs up to 5000 parallel

Table 8.5.: Setup of the server for experiment No. 7

Parameter	Min	Max
Clients per thread	1	50
Threads per server	10	20
Servers	1	5

Table 8.6.: Statistical results from experiment No. 7 for the read response time

Clients per thread	**Threads per server**	**Servers**	$\overline{X}\ (ms)$	$\sqrt{S^2}\ (ms)$	95% **CI for** $\overline{X}$	95% **CI for** $\sqrt{S^2}$
1	10	1	18.5	51.2	$18.3 - 18.6$	$51.1 - 51.3$
1	10	5	71.3	324.9	$70.3 - 72.2$	$324.2 - 325.6$
1	20	1	28.2	109.4	$27.9 - 28.5$	$109.2 - 109.6$
1	20	5	141.5	397.3	$140.3 - 142.6$	$396.5 - 398.1$
50	10	1	856.0	747.9	$853.5 - 858.4$	$746.2 - 749.6$
50	10	5	2766.1	2291.8	$2760.2 - 2772.1$	$2287.6 - 2296.0$
50	20	1	1165.4	1448.3	$1161.1 - 1169.7$	$1445.3 - 1451.4$
50	20	5	5603.9	2852.4	$5596.5 - 5611.4$	$2847.1 - 2857.7$

client threads and 5 middleware instances.

The parameters whose impact on the response time will be studied are the following:

- Number of clients per thread
- Number of threads per server
- Number of servers

As in the previous experiment, a 2^k factorial experiment will be performed to analyze the influence of each one of them in the performance of the system. The range of values for the parameters is summarized in table 8.5. In the following experiments the load was always kept at that maximum capacity of the server instance, i.e. the total number of clients in the system is $Clients = \frac{Clients}{Thread} * \frac{Threads}{Server} * Servers$.

The clients for this experiment are the usual Bob and Alice clients used in previous experiments with a fixed message size of 2000 characters and one target queue. The clients were evenly distributed among 5 machines to reduce the load in each machine and allow higher request rates.

8.3.2. Results and Analysis

The first step when analyzing the data was to plot the traces for response time and throughput for send and read operations, these were observed to be stationary and therefore the next step was the extraction of statistical metric from the data points. This information is contained in tables 8.6 to 8.9.

From the data in these tables, the authors deduce the following remarks about the system's performance:

Table 8.7.: Statistical results from experiment No. 7 for the send response time

Clients per thread	**Threads per server**	**Servers**	$\overline{X}\ (ms)$	$\sqrt{S^2}\ (ms)$	95% **CI for** $\overline{X}$	95% **CI for** $\sqrt{S^2}$
1	10	1	13.7	40.0	$13.6 - 13.9$	$39.9 - 40.1$
1	10	5	33.3	96.0	$33.1 - 33.5$	$95.8 - 96.1$
1	20	1	19.4	53.1	$19.3 - 19.5$	$53.0 - 53.2$
1	20	5	52.9	114.1	$52.7 - 53.1$	$114.0 - 114.3$
50	10	1	824.7	730.3	$822.4 - 827.0$	$728.7 - 731.9$
50	10	5	2741.5	2304.5	$2735.5 - 2747.4$	$2300.3 - 2308.7$
50	20	1	1156.8	1445.4	$1152.9 - 1160.6$	$1442.7 - 1448.2$
50	20	5	5217.9	2818.9	$5210.9 - 5225.0$	$2813.9 - 2823.9$

Table 8.8.: Statistical results from experiment No. 7 for the read throughput

Clients per thread	**Threads per server**	**Servers**	$\overline{X}\ (\frac{msg}{s})$	$\sqrt{S^2}\ (\frac{msg}{s})$	95% **CI for** $\overline{X}$	95% **CI for** $\sqrt{S^2}$
1	10	1	267.4	20.9	$257.9 - 276.9$	$16.0 - 30.2$
1	10	5	349.6	45.8	$328.8 - 370.5$	$35.0 - 66.1$
1	20	1	352.3	21.6	$342.4 - 362.1$	$16.5 - 31.1$
1	20	5	352.9	67.7	$322.1 - 383.7$	$51.8 - 97.7$
50	10	1	292.0	78.7	$256.2 - 327.8$	$60.2 - 113.6$
50	10	5	452.9	52.1	$429.2 - 476.6$	$39.9 - 75.2$
50	20	1	343.2	30.5	$329.3 - 357.0$	$23.3 - 44.1$
50	20	5	448.0	48.0	$426.2 - 469.9$	$36.7 - 69.3$

Table 8.9.: Statistical results from experiment No. 7 for the send throughput

Clients per thread	**Threads per server**	**Servers**	$\overline{X}\ (\frac{msg}{s})$	$\sqrt{S^2}\ (\frac{msg}{s})$	95% **CI for** $\overline{X}$	95% **CI for** $\sqrt{S^2}$
1	10	1	356.0	27.7	$343.4 - 368.7$	$21.2 - 40.0$
1	10	5	746.5	68.5	$715.4 - 777.7$	$52.4 - 98.9$
1	20	1	508.8	29.5	$495.4 - 522.2$	$22.5 - 42.5$
1	20	5	941.2	121.2	$886.1 - 996.4$	$92.7 - 175.0$
50	10	1	303.0	82.1	$265.6 - 340.4$	$62.8 - 118.6$
50	10	5	456.0	47.2	$434.5 - 477.5$	$36.1 - 68.2$
50	20	1	432.5	39.7	$414.4 - 450.5$	$30.4 - 57.3$
50	20	5	482.0	42.1	$462.8 - 501.1$	$32.2 - 60.8$

- The throughput for read operations in the system is capped at a value around 500 messages per second, as seen in table 8.8 even though the number of available servers or threads is increased, the average read throughput varies only slightly around this value.

- On the other hand, the send operation benefits more from the scaling of the system and it finds its best value when each client is serviced in its own thread and the number of servers and threads is high. In this configuration, even though the total number of clients is small, i.e. 50 readers, 50 senders, the total system throughput is double the value of the throughput of the configuration when the number of senders is 2500. This is a very important value as it exhibits where the bottlenecks of the system are and how to avoid them, as it was proposed in initial experiments this provides evidence to the hypothesis that having a considerable number of clients in a worker thread is one of the factors that most impacts the response time for read and send operations.

- The response time is heavily dependent on the number of clients and this effect occludes the effect of the changes in the server configuration. This can be seen by looking at the evolution of the average response times in tables 8.6 and 8.7 which increases as downward for each row in the table, this is the same direction as the increase on the total number of clients that the system can service.

- Reinforcing propositions made in previous experiments, it is possible to see that the standard deviation increases proportionally with the concurrency of the system, i.e. increasing the number of threads or servers increases the standard deviation more significantly than increasing the number of clients per thread because the latter involves a sequential behavior, i.e. waiting in queue for service.

The traces for the best and worst performing configurations are presented in figures 8.8 nd 8.9. This was determined by the maximum and minimum values of the throughput of the send operations.

Observation of these traces reveal that the best system is not only fast but also more stationary than the worst configuration, in the worst configuration it is possible to observe random peaks outside of the warm up and cool down times which suggest that this configuration is unstable, most likely due to the fact that many clients share a small number of workers and therefore interference created by any intensive operation can propagate and significantly build up the response time as seen in 8.9.

In order to understand better what the results found in this experiment suggest about the systems behavior, the systems of linear equations for the response time and throughput were solved and the following linear functions found, where c is the number of clients per thread, t is the number of threads and s the number of servers.

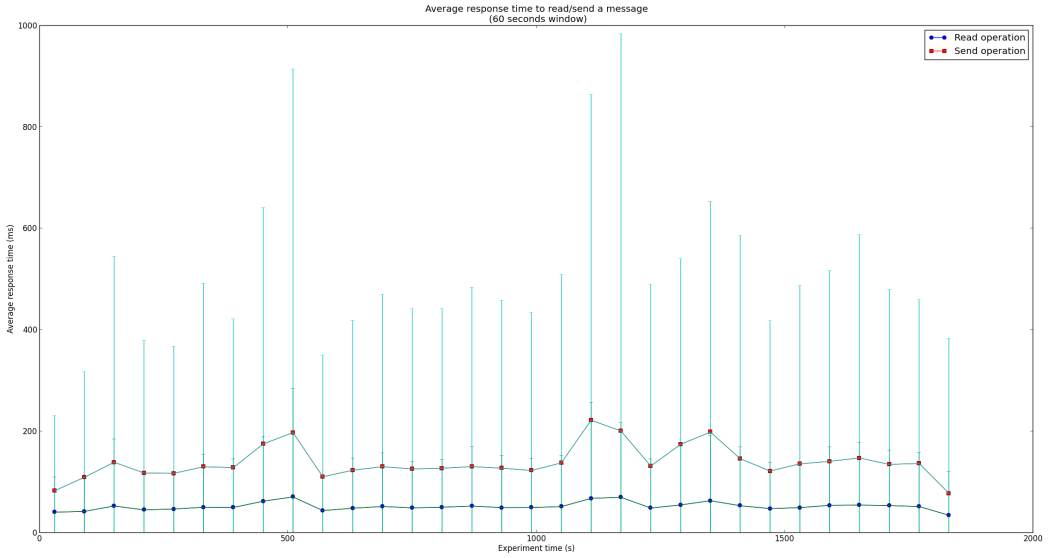

Figure 8.8.: Response time trace for send and read operations in a configuration with 5 servers, 10 threads and 50 clients per thread

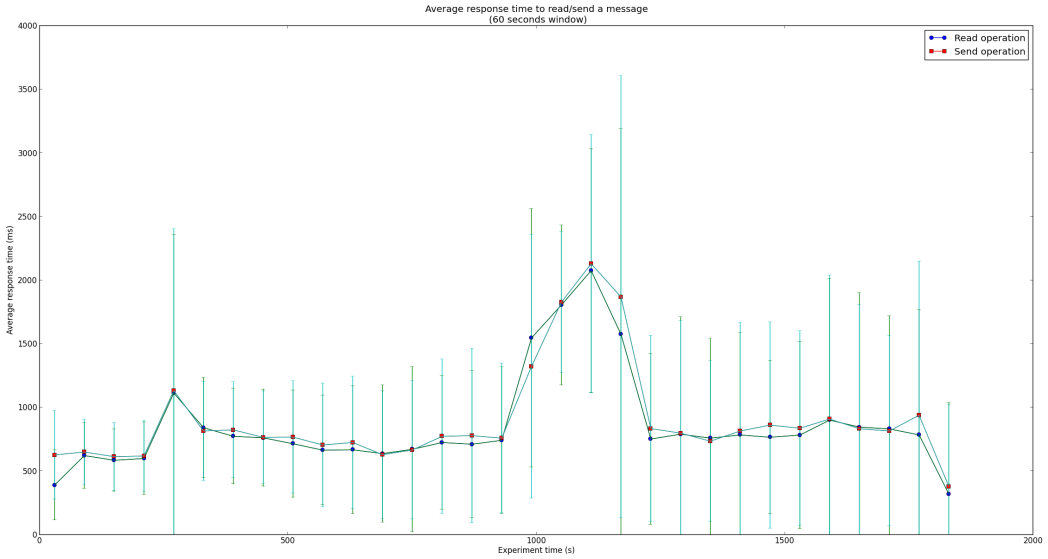

Figure 8.9.: Response time trace for send and read operations in a single server configuration with 10 threads and 50 clients per thread

$$RTread(c,t,s) = 14.1c + 0.1t + 1.2s - 0.6ct - 3.11cs + 0.25ts + 1.26cts - 3.37\ [ms] \tag{8.1}$$

$$RTsend(c,t,s) = 11.1c + 0.6t + 2.6s - 0.4ct - 1.2cs - 0.74ts + 1.1cts - 4.50\ [ms] \tag{8.2}$$

$$THread(c,t,s) = 0.9c + 10.6t + 40.7s - 0.08ct + 0.27cs - 2.05ts + 0.013cts + 140.63\ [\frac{msg}{s}] \tag{8.3}$$

$$THsend(c,t,s) = -0.14c + 14.2t + 87.6s + 0.03ct - 0.47cs + 1.22ts - 0.07cts + 116.2\ [\frac{msg}{s}] \tag{8.4}$$

In this case, the results seems to be more linear than in the previous experiment and this equations provide a clearer picture of the systems behavior. As proposed, the equations prove that the as the number of threads, clients per thread and server is increased the response time grows accordingly and that the biggest contribution comes from how many clients per thread are allocated. Additionally, for the throughput the equations indicate that increasing the number of clients per thread doesn't help the throughput and this achieved by increasing the number of threads per server or servers instead.

8.4. Experiment 8 - System trace

8.4.1. Description

The aim of this experiment is to analyze the system behavior and performance during a long running period in order to expose any time dependent behaviors that were likely not observed in previous experiments. Additionally, this experiment allows the exploration of other operations besides the basic reads and sends from previous ones.

The setup for this experiment is in similar fashion as the very first from section 7.2, there are however new client behaviors to be introduced, in order to facilitate the presentation of this experiment they were given particular names.

- Larry client: This is the client described in 7.2.1, which sends a message to a random queue and receiver and then waits until a message addressed to him appears in any of the system queues, proceeds to read it (and pop it) and then bounce it back to another random client by placing it in a random queue.
- Tola (client) and Maruja (server) clients: These clients represent a pairwise client-server interaction where the Tola clients make requests to its predefined Maruja counterpart which is constantly checking the queue for messages, once the server (Maruja) finds a message, it pops it and puts back a response message in the queue, this allows the client (Tola) to make a new request after it finds the response in the queue.
- Rick and Carl clients: These clients represent another kind of client-server interaction that is not pairwise as the case before, the clients Carl make requests by

putting messages in a predefined queue and the Rick clients service them by popping them out of the very same queue. However, in this case there is no response back from the Rick clients and the Carl clients are continuously making requests without waiting to check if the previous one have already been serviced.

For this experiment the number of clients is as follows:

- 50 Larry clients
- 15 Maruja clients
- 15 Tola clients
- 15 Carl clients
- 5 Rick clients

These are executed in a single machine given that their number is small, two middleware instances are setup for this each experiment each with capacity of 50 clients distributed in 25 threads, i.e. 2 clients per thread.

The number of queues used in this experiment is 20 distributed as follows:

- 18 queues are used for the random message sending of the Larry clients.
- 1 queue is used for the interactions between Tola and Maruja clients.
- 1 queue is used for the client-server interaction of the Carl-Rick clients.

The experiment starts with an empty database, apart from the initialized queues, and was executed uninterrupted for 2 hours.

8.4.2. Results

The first set of figures 8.10, 8.11 and 8.12 illustrate the response time for the 3 basic operations in the client, i.e. read(pop) a message, send a message and query for queues with available messages for the client respectively.

Following, is a set of traces that show the database component of the response time for the different basic operations, in this case there is a differentiation between the response time when the result is positive or not, e.g. when a queue with message is found. This information is contained in figures 8.13 through 8.17.

Finally, figure 8.18 shows a trace of the number of messages per queue in the database.

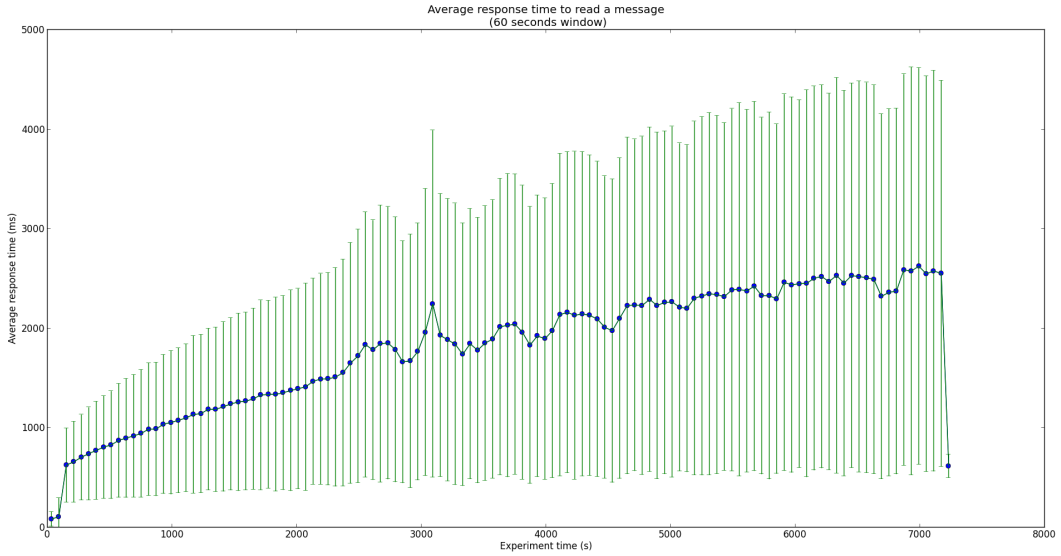

Figure 8.10.: Response time trace for a read operation

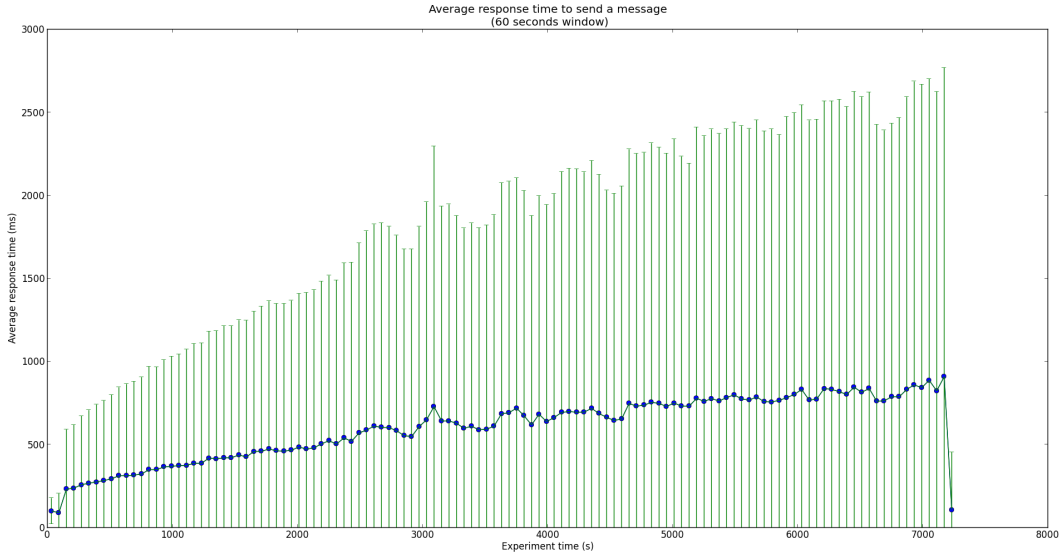

Figure 8.11.: Response time trace for a send operation

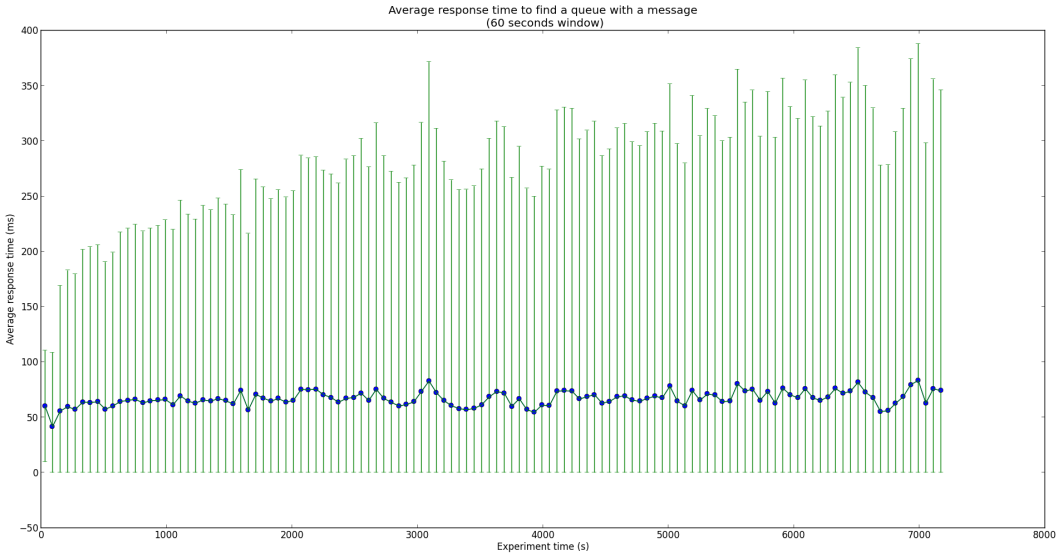

Figure 8.12.: Response time trace for finding a queue with messages

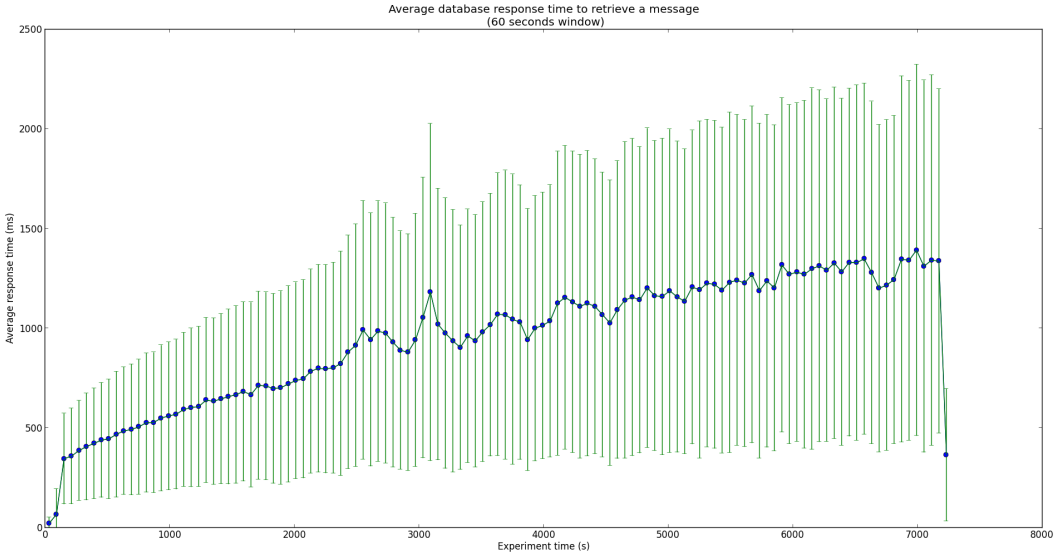

Figure 8.13.: Database response time trace for a read operation when a message is ready

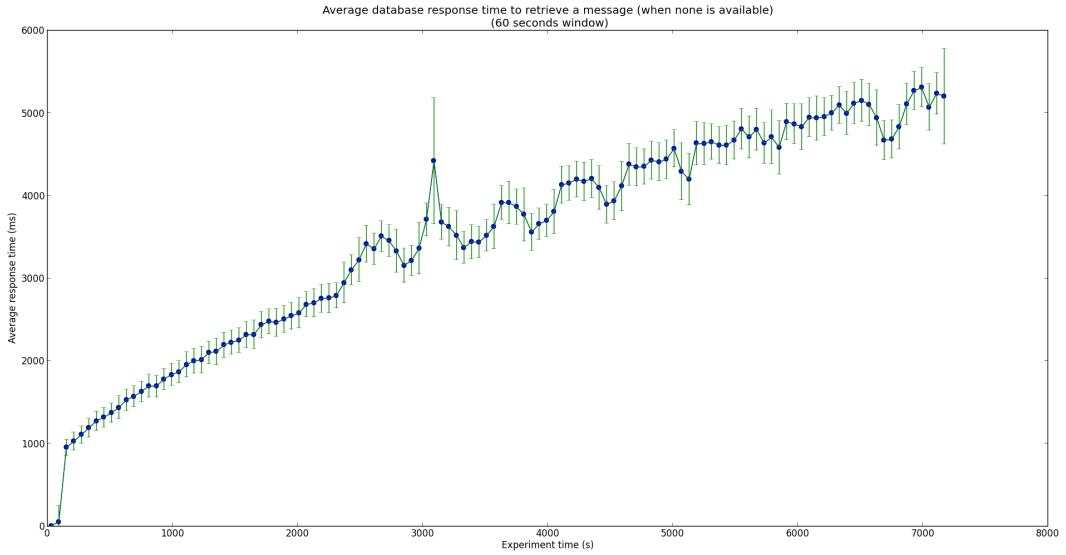

Figure 8.14.: Database response time trace for a read operation when no message is ready

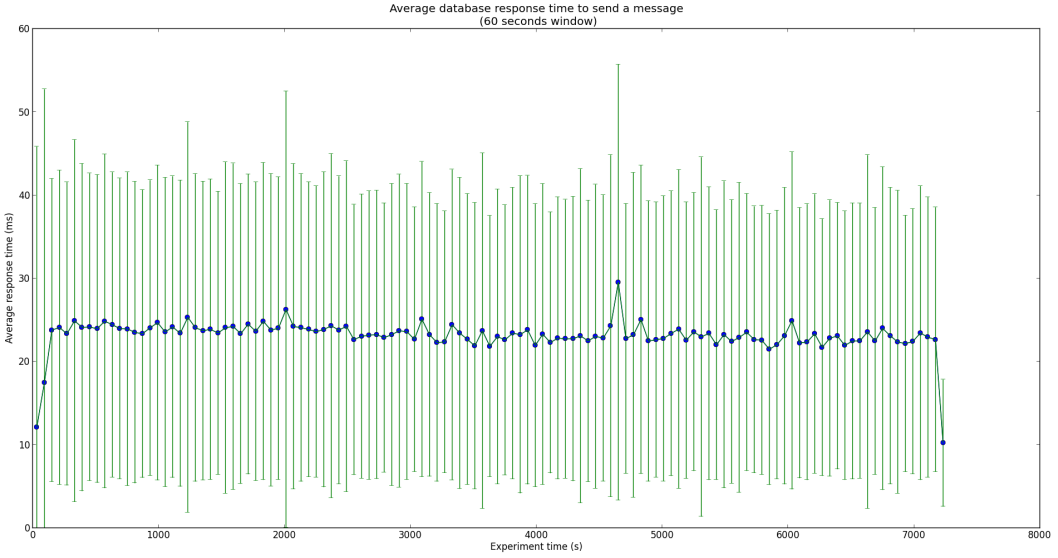

Figure 8.15.: Database response time trace for a send operation

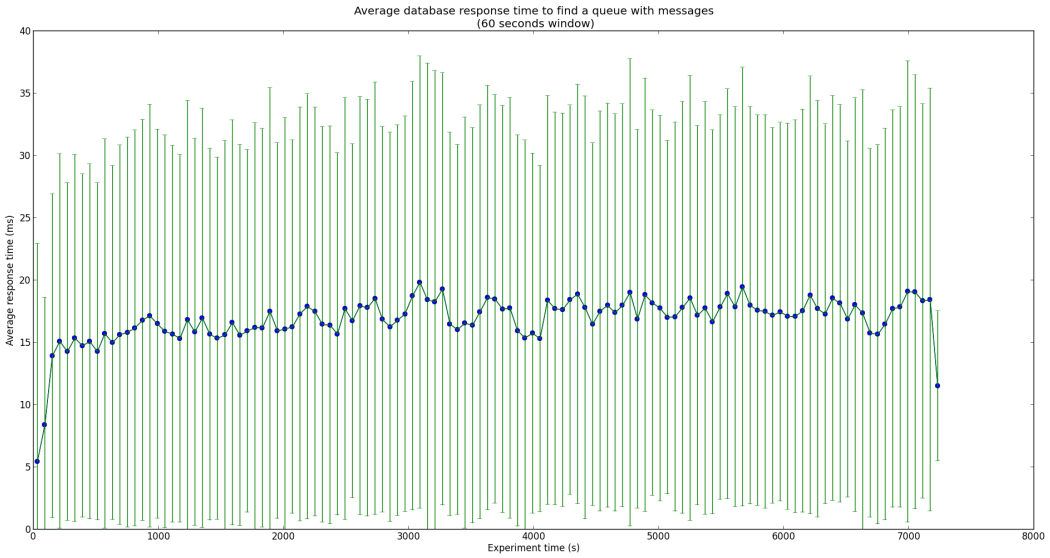

Figure 8.16.: Database response time trace for finding a queue with messages when one exists

Figure 8.17.: Database response time trace for finding a queue with messages when none has messages for the client

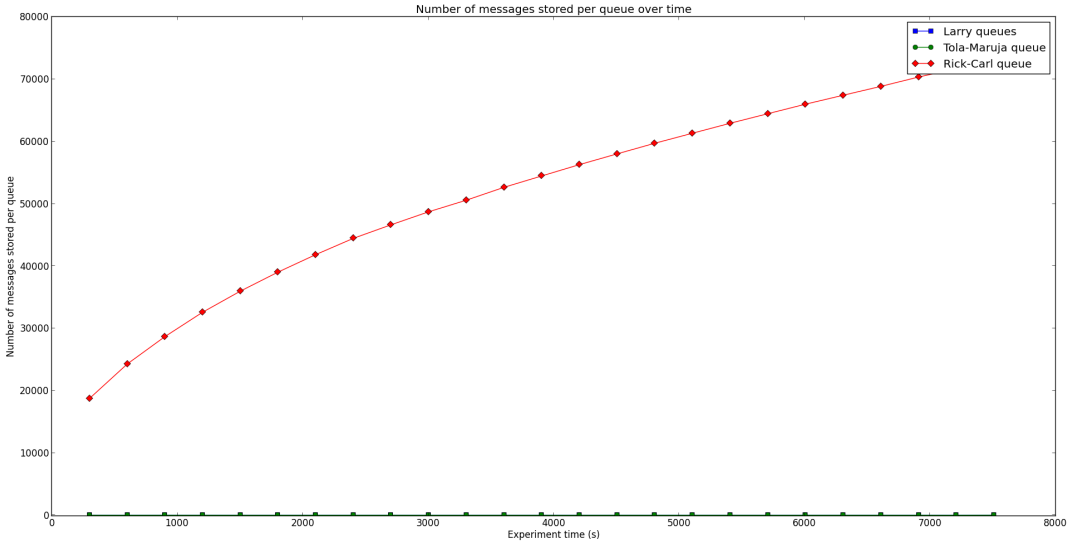

Figure 8.18.: Database size trace, measured in number of messages per queue

8.4.3. Analysis

The first characteristic observed in this experiment and that clearly differs from previous ones is that the behavior of the system is no longer stationary because the response times for both the send and read operations grow over time. Even though the response time for the queue query is not.

The explanation for this can be easily seen if figure 8.18 is taken into account as it clearly displays the same growing trend for the database size. On the other hand the queue query does not grow is because it operates on a separate table, i.e. the message-queue association table, than the read and send queries.

However, it must be pointed out that the database time itself is not growing in all cases over time, for example in 8.15 in can be seen that the database time to send a message remains constant. This is expected as the database size is still not considerably large (i.e. below 100k rows) and therefore the insertion operations should be independent of table size in this range. On the other hand the read operations are clearly affected by the size of the tables, especially the operations to find a message when there is none available. This operation is mostly performed by Tola-Maruja clients which are aggressively checking the queues for a request or response from its counterpart. The fact that the response time, as seen by the client, grows in the same way as the read response time even though the database time is constant lies in the issue that the clients are sharing worker threads and therefore the operation times are not independent between operations as it was seen in initial experiments.

Finally, it is important to note that from these traces there is still a high variance in the results, it is still not completely clear how much of it can be accounted to random noise from the infrastructure and how much it is really a feature of the system design, although the previous experiment suggest that the latter is a significant factor in it.

9. Conclusions

9.1. Summary of the system performance

In this section, a list of statements about the system behavior and performance is presented, this was compiled from the observations and analysis made in all experiments.

- The biggest factor affecting the system performance from the load point of view is the number of clients that are connected to the system at a given point, however this can not crash the system as there are hard limits configured for the number of connected clients.

- The distribution of clients in a middleware instance plays an important role on the quality of service seen by each client in terms of response time and throughput, this is because of a bottleneck in the system created by the fact that each worker thread servers its clients one at a time. This bottleneck is more obvious in scenarios where clients with intensive operations, such as reads in a reasonably sized database, are allocated in the same worker threads as clients doing lighter operation, such as message sending, because the response time for the sender client degrades significantly not because of an increase in the database response time but the waiting time in the worker thread.

- The performance of the system in most of the experimental scenarios was found to be significantly noise and therefore not very precise, however statistical tests suggest that the calculated performance values are accurate with high confidence. However, the authors doubt that this statement is completely true because the validity of these tests depend on the i.i.d. property of the data points, however this does not hold in most cases as it is not plausible to think that the actions of one client do not affect the response time of others. Nevertheless, the authors consider that it was the best assumption that could be done with the available information about the system and given the high amount of data points it is expected that the obtained values are valid.

- The observed variance of the response time in the different experiments could be explained by the effect of concurrency in the system, concurrency factors such as the non-blocking I/O and multiple database connections seem to increase the standard deviation of the response time of the experiments which reinforces the proposed explanation.

- In the dryad environment, the usual response time was $50\ ms$ for both read and send operations in average, with send operations having slightly lower response times. However, this response time only holds for small deployments with client

number below 100, as the number of clients increase the response time increases linearly, at least in the observed vicinity of this value.

- Even though the performance in the EC2 cluster was expected to be considerably worse than in the dryad infrastructure, the results from the first EC2 experiment suggest that the values are closer than expected, at least in average. On the other hand, it is clear that the variation of the results does increase significantly in the EC2 cluster due to the non-isolation and non-homogeneity of the environment where the software is executing. These remarks can be made assuming linear relations between the different evaluated parameters and the performance metrics.

- The response time for operations in the amazon cluster with a load of around 500 clients is $500\ ms$ in average for send and read operations, this for a particular server configuration which is known to have good performance because it keeps the number of clients per thread low.

- The system as it was designed appears to be reasonably scalable as it is possible to increase the number of threads per server and servers to obtain increased throughput in send operations, although the read operations scale slowly most likely due to a bottleneck in the database queries associated with it. From the experiments related to server configuration, it was found that the biggest factor impacting performance is how many clients per thread are allocated and that in order to increase throughput the best bet is in increasing the number of servers and threads, this also allows higher server capacity.

9.2. Proposed improvements

- In order to reduce the impact of server configuration in the performance of the system and improve scalability, a fundamental architectural change must be made to remove the sequential servicing of the clients in a worker thread. One possible new architecture could implement a third layer of threads that take care of servicing the requests in the database and then returning the request to the second layer which just takes care of servicing the network I/O component of the system. However, this could increase the resource usage of the system and possibly introduce more non-deterministic behavior as it is made more concurrent.

- Another bottleneck to be improved is the performance of the database read operations as the database size increases, it is clear that multiple concurrent read operations on tables of relatively small size (i.e. less than 100k records) can lead to very high response times in the order of seconds. It is not clear to the authors if a database schema change would be necessary or if there exists some adjusting in the queries or database configuration which would improve this situation.

- Finally, it would be worthwhile to explore the possibility of modifying the design in a way that it preserves average performance but improves the variability of it towards a more stable deterministic behavior.

Appendices

A. Management Console Screenshots

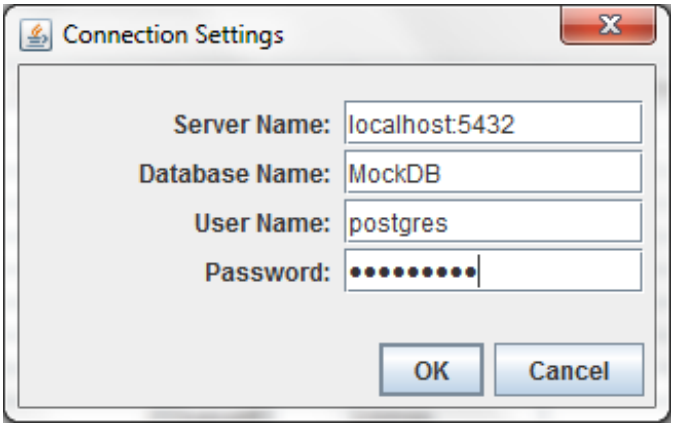

Figure A.1.: Screenshot of the connection settings dialog in the management console.

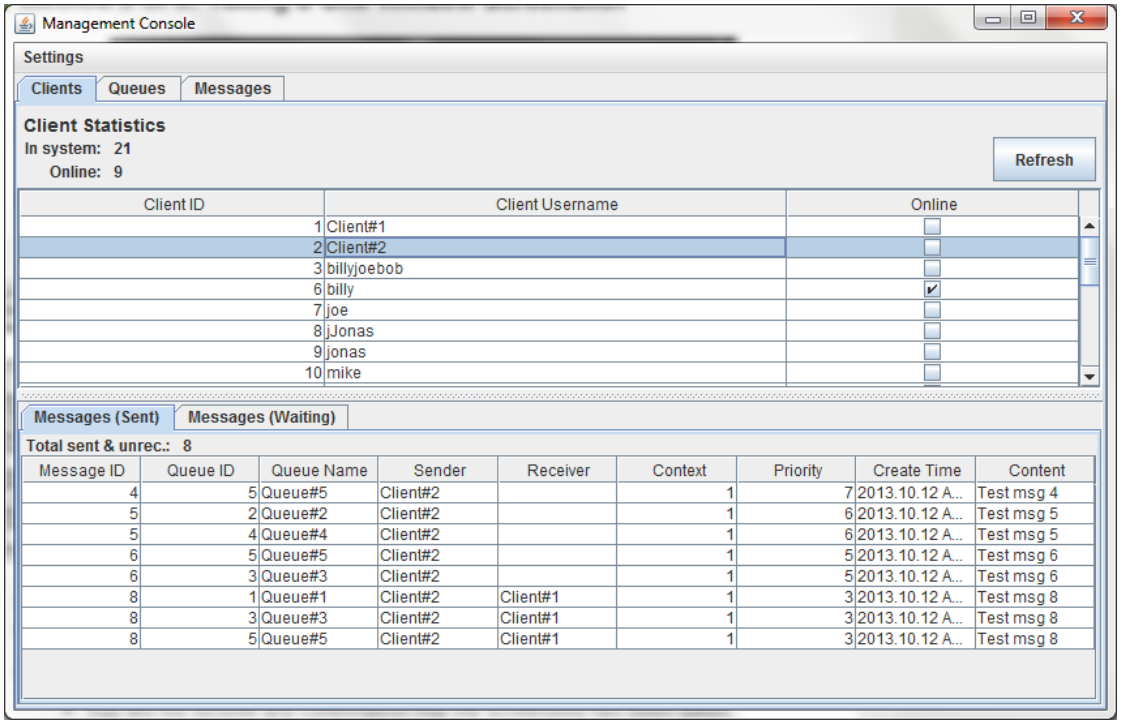

Figure A.2.: Screenshot of the client overview in the management console.

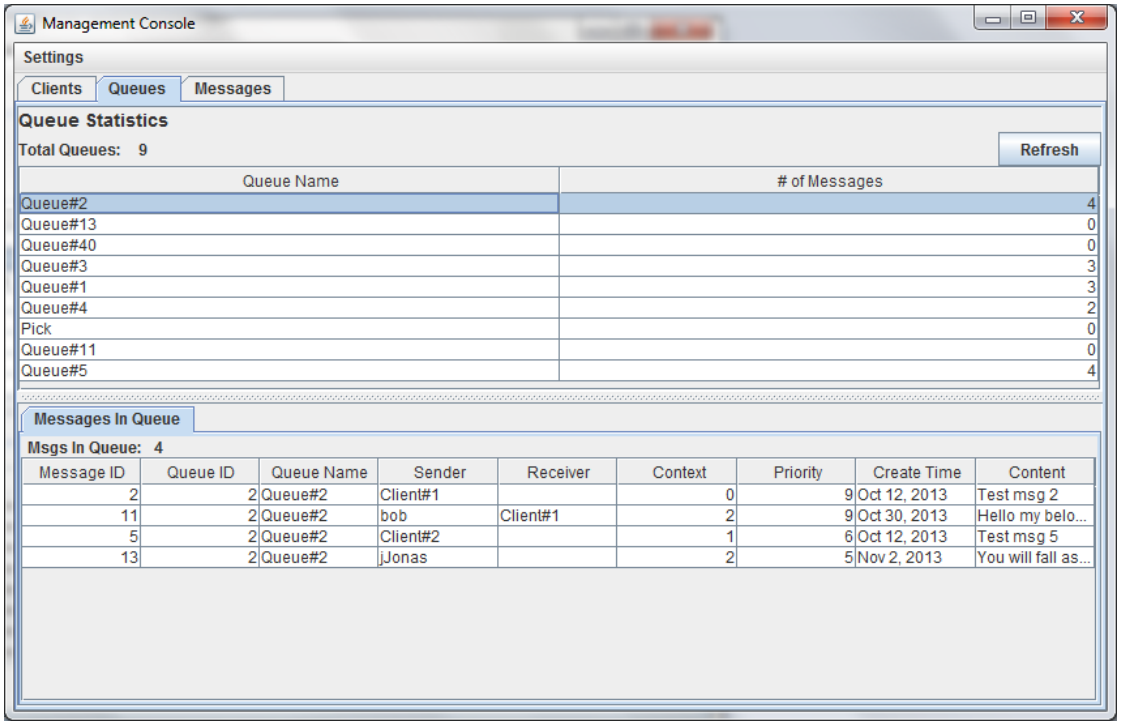

Figure A.3.: Screenshot of the queue overview in the management console.

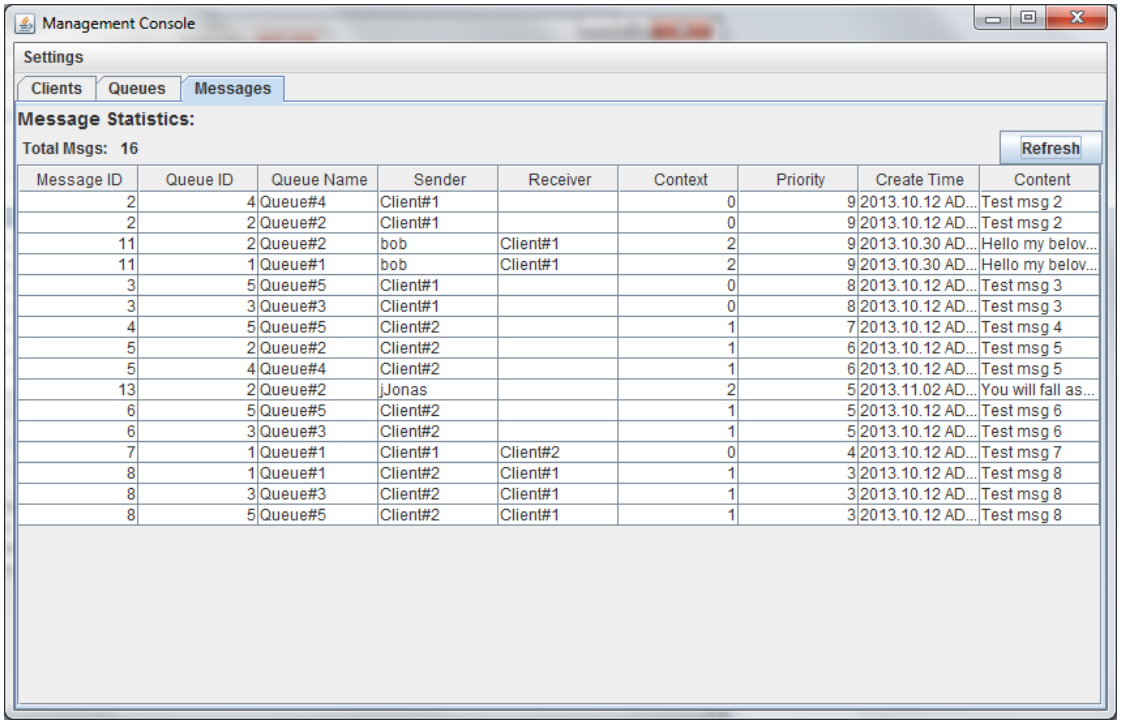

Figure A.4.: Screenshot of the message overview in the management console.