Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona
UNIVERSITAT POLITÈCNICA DE CATALUNYA

A work queue server for the Ichnaea project

Author:
Miguel Ibero Carreras

Director:
Lluís A. Belanche Muñoz

Barcelona, November 2014

Acknowledgements

I would like to to thank Lluís Belanche, the director of this project, for giving me the opportunity to work on improving Ichnaea and always being positive and encouraging when I had problems. I would also like to thank Nahuel Velazco, who worked on the website to manage Ichnaea tasks, for always being so helpful.

Finally thanks to my family for supporting me through my studies, and Yvette for always being there for me.

Contents

1 Introduction **8**

1.1 Description 8

1.2 Motivation 8

1.3 Overview 9

1.4 Goals 10

1.5 Structure of the document 11

2 AMQP protocol **12**

2.1 History 12

2.2 AMQP 0-9-1 14

2.2.1 Concepts 15

2.2.2 Transport 23

2.3 AMQP 1.0 28

2.3.1 Type System 29

2.3.2 Transport 32

2.4 RabbitMQ 33

3 Ichnaea AMQP **34**

3.1 R wrapper 34

3.2 Java 36
3.2.1 Shell package 36
3.2.2 Command line interface package 37
3.2.3 Model package 38
3.2.4 Xml package 39
3.2.5 Data package 40
3.2.6 Client package 42
3.2.7 Proto package 44
3.2.8 App package 46
3.3 PHP library 46
3.3.1 Test PHP app 47

4 Analysis of a test run 48

4.1 General setup 48
4.2 Fake request 50
4.2.1 Setup 50
4.2.2 Process 53
4.3 Build models 54
4.4 Predict models 56
4.5 Errors 56
4.5.1 Failing declaration 56
4.5.2 Failing process client 56
4.6 Multiple requests 57
4.7 Test PHP app requests 57
4.8 AMQP 1.0 58

5 Conclusions **61**

5.1 Problems and workarounds 63

5.2 Further development 63

6 Appendix **65**

6.1 User's guide 65

6.1.1 Setup required programs and libraries 66

6.1.2 Compile and Run 66

6.2 AMQP 1.0 support 67

6.3 Developer's guide 68

6.4 PHP setup 69

6.4.1 Install composer 69

6.4.2 Create a RabbitMQ user 70

6.4.3 Usage 70

6.4.4 Test app 71

6.4.5 Possible problems 72

6.5 Ichnaea XML Model Examples 73

6.6 AMQP 0-9-1 Tables 75

6.6.1 Class IDs 75

6.6.2 Method IDs 76

6.6.3 Data types 78

6.7 AMQP 1.0 Tables 79

6.7.1 Data type categories 79

6.7.2 Data types 80

6.8 Bibliography 83

Glossary of terms

- AJAX: Asynchronous JavaScript And XML
- AMQP: Advanced Message Queuing Protocol
- API: Application Programming Interface
- CSV: Comma Separated Values
- HTTP: Hyper Text Transfer Protocol
- IANA: Internet Assigned Numbers Authority
- MIME: Multi-Purpose Internet Mail Extensions
- MST: Microbial Source Tracking
- OASIS: Organization for the Advancement of Structured Information Standards
- PHP: Hypertext Pre-processor
- POSIX: Portable Operating System Interface (UniX)
- SSL: Secure Sockets Layer
- TCP: Transmission Control Protocol
- TLS: Transport Layer Security
- TTL: Time To Live
- UDP: User Datagram Protocol
- UTF: Unicode Transformation Format
- XML: eXtensible Markup Language
- ZIP: lossless data compression format

Chapter 1

Introduction

1.1 Description

This project implements a scalable networking system using the AMQP protocol, that can receive processing requests from a high-traffic website and return the obtained results. In particular these requests will call the Ichnaea project code to predict the microbial source of water samples.

1.2 Motivation

Nowadays, fecal pollution in water is one of the main causes of health problems in the world, and is associated with several thousands of deaths per day, being a main vehicle of pathogen transmission. In this sense, is very important to know whether a waterbody (a river, a lake, etc.) is contaminated or not and, in case it is contaminated, which is the pollution origin.

MST[1] describes different methodological approaches that pursuit the determination of the origin of fecal pollution in water by the use of microbial or chemical indicators.

The original Ichnaea software is a fully computer-based prediction system that is able to make fairly accurate predictions for MST studies. It was developed

[1] Microbial source tracking

David Sànchez of the School of Computer Science, Lluís A. Belanche of the Department of Software of the Technical University of Catalonia and Anicet R. Blanch of the Department of Microbiology of the University of Barcelona (see Sànchez 2012). The system accepts samples showing different concentration levels, uses indicators with different environmental persistence, and can be applied to different geographical or climatic areas.

On the one hand, the Ichnaea core is a set of command line scripts written in the R language that can take a long time and are very cpu-intensive. On the other hand, the target user group for it are microbiologists that want to obtain information about a set of water samples and may not have additional computer knowledge. The motivation for this project is to bridge this gap and implement a more user-friendly frontend that they can use with minimum setup.

1.3 Overview

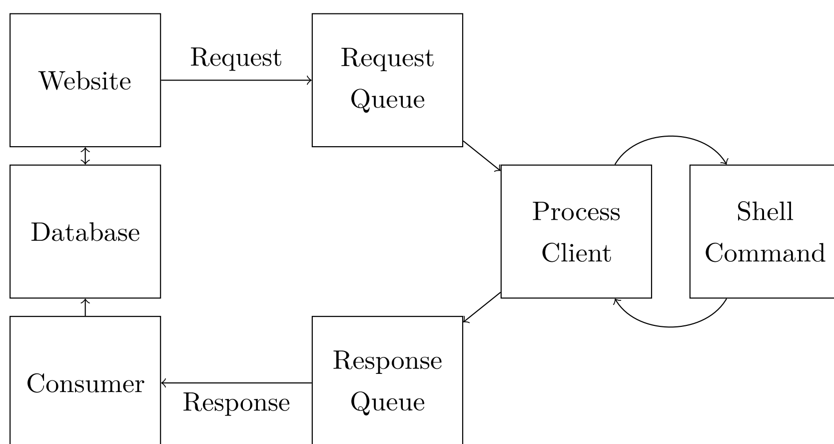

Figure 1.1: Ichnaea System Overview

The frontend was developed in two parts. We worked the communication between the website and the Ichnaea command line scripts. We developed a PHP library to send requests and receive responses encoded in XML format. These messages are quept in queues in a server that uses the AMQP protocol. We also implemented a Java process client that reads these requests and executes the shell commands necessary to run the Ichnaea scripts. Then this client takes

the responses and sends them back to the AMQP server where they wait in a response queue. In the webserver there is a consumer script running that will take these responses and update the website database with the obtained results.

Nahuel Velazco worked on the website (see Velazco Sanchez 2014), which is connected to a database and can be used to manage different users with permisions that can send requests and see response statuses in real time. It also has a system do predefine dataset structures that simplifies the introduction of the request data. These features were implemented with ease of use for the target user group in mind.

1.4 Goals

To produce the described system, this project needed to accomplish the following goals:

- *an easy way to run the original Ichnaea code in multiple servers*

Since the Ichnaea processes are very intensive, we wanted to have an easy way to scale the server processing power by adding more servers and spliting the tasks between them.

- *a queueing server that can run shell commands*

To be able to run the original Ichnaea R scripts we needed the servers to run shell commands and be able to read the responses printed out.

- *an XML definition for the different Ichnaea requests and responses*

To send each processing request and receive each response, we needed to serialize the information. We decided to use the standard XML markup language for this.

- *a server logic to process the different Ichnaea requests*

The original R scripts have two main processes, we needed server code to take the unserialized request, call the corresponding shell command, read the output and generate the correct responses.

- *a PHP library to interact with the Ichnaea server from a website*

The final goal for this project was to send requests and read responses from a website. Since this website uses the PHP programming language, we needed to create a PHP library to interact with it.

1.5 Structure of the document

The first chapter is this introduction, which tries to initiate the reader into the context of the project and its framework.

In the next chapter, we will explain the AMQP protocol and network structure. We will also explain its application layer data format. We will then compare AMQP version 0-9-1 to the new 1.0 version.

The third chapter will explain the developed system in depth. We will describe each of the implemented subsystems and the way they interact, as well as justify different design decisions.

In the fourth chapter we will analyze the packet communication of some usage examples of the system and show examples of the different AMQP protocol data structures.

And finally, the last chapter will expose the conclusions of the project as well as give some insight into possible future work.

After that there is an appendix that includes a brief user's guide, a glossary and more detailed AMQP protocol specification tables.

Chapter 2

AMQP protocol

In this chapter we will describe how the protocol used for Ichnaea message queueing works, the different concepts behind it and the structure of its packets, as well as a short history of it.

AMQP[1] is an open Internet protocol for messaging. It defines a binary TCP application layer that allows for the reliable exchange of messages between two parties. It was originally created for the financial services industry to work on stock transactions, but it can be used with any type of message and has build-in scalable ways of routing and queueing them.

2.1 History

AMQP was designed in 2003 at JPMorgan-Chase by John O'Hara. He was looking for a messaging protocol that could offer high endurance while handling high amounts of data and that could be easily connected to a wide variety of systems. He wanted it to work in an environment where the loss, late arrival or improper processing of data could have a big economic impact. The available commercial products at the time seemed to not be able to match these requirements, therefore most companies in the financial services industry had developed their own custom software to take on the task. These custom solutions

[1] Advanced Message Queuing Protocol

would be replaced after a time and connecting them to each other was always complex and difficult.

O'Hara, on the other hand, thought that standard network protocols, like Ethernet, TCP/IP and HTTP, all had several characteristics in common that would be very beneficial for a messaging protocol. They all were royalty-free, open and had a specification that was defined by an independent entity. He argued that releasing and maintaining libraries that implemented the specified protocol would enable developers to find interesting uses for them quickly. The technical and economic success of these protocols was the consequence of having a strong control over their definition and thinking about real use cases when working on them.

To achieve this goal, he decided that there should be a freely available implementation of the AMQ protocol that was in use at the time in a mission-critical place at JPMorgan. He hired the iMatrix Corporation to work on the first version of the stack called OpenAMQ. The system was deployed in a trading application that had more than 2,000 users.

In 2005 JPMorgan partnered with Red Had to develop Apache Qpid, an open-source implementation of different tools to communicate with AMQP, initially in Java and soon after in C++. Independently Rabbit Technologies created RabbitMQ using the relatively unknown Erlang language and afterwards other companies like Microsoft and StormMQ released similar compatible products. The same year a working group of companies was formed to incentivize adoption of the new AMQP protocol and work on standardizing the definition. AMQP 0-8 was published in June 2006, 0-9 in December 2006, 0-10 was published in February 2008 and 0-9-1 was published in November 2008.

The working group released AMQP 1.0 at a conference in New York on October 2011. At the event there was an interoperability demonstration of different software communicating using the new protocol. The next day an OASIS Technical Committee was formed to advance the 1.0 definition through the international open standards process. After a year and with minimal changes, AMQP 1.0 was approved as an OASIS standard.

Even if the latest AMQP version is 1.0, we will first will proceed to describe the 0-9-1 version of the protocol, since the server we are using primarily communicates that way. Later we will address the changes introduced by the newer version.

2.2 AMQP 0-9-1

The AMQP 0-9-1 protocol is used to publish *messages*, which can contain any type of data, to entry points called *exchanges*. Exchanges then distribute messages to *queues* using rules called bindings. The servers that contain these entities are called *brokers*. When the messages arrive at a queue, they are delivered to subscribed consumers or wait there until a consumer asks for them.

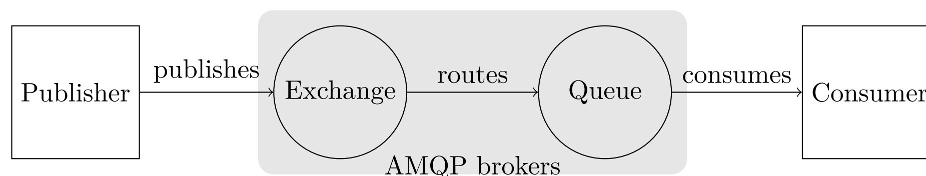

Figure 2.1: AMQP 0-9-1 basic routing

Publishers can specify message attributes when publishing them. Some of these attributes can be used by the broker, but most of them, especially the message content, are not accessible and can only be used by the consumer that receives the message. The most important attribute that the broker uses is the message routing key, a string that all messages need to set.

The AMQP specification has the notion of message acknowledgements to prevent data loss when a message is delivered to an application that fails to process it or when the network is unreliable and the message does not arrive at the destination correctly. When defining a consumer, the application developer can choose between automatic or manual acknowledgements. With manual acknowledgements, the broker will only remove a message from a queue when a notification is received for that message.

AMQP can be defined as a *programmable* protocol since its entities and routing schemes are not defined by a broker administrator but declared by the clients themselves. The protocol has methods to declare queues and exchanges and create the bindings that connect them, as well as the normal methods to send messages and subscribe to queues.

If a client tries to declare an entity with different attributes and the same name as an existing one, an exception with code `406` is raised. Otherwise the server

responds the same way as if the entity was created.

The reasoning behind this decision is to give the maximum amount of power to the application developers, but it also introduces a new set of potential definition conflicts. These problems don't happen very often in practice, since the configuration for a system is usually very fixed and does not change much.

2.2.1 Concepts

2.2.1.1 Exchanges

Exchanges are AMQP entities that act as the entry point for messages. When a message arrives, the exchange routes it into the queues that are decided by an algorithm based in the defined exchange type and bindings rules. Let's discuss each available exchange type.

Default Exchange

The default exchange is a special entity that always exists in the broker and that cannot be removed. It was introduced to make simple applications work out of the box, without having to define any exchange. When a queue is defined, the broker binds it automatically to this exchange with a routing key with the same name as the queue one.

For example, if a client declares a queue with the name of *"test"*, the broker will bind it to the default exchange using *"test"* as the routing key. When a message is published to the default exchange with the routing key *"test"*, it will be routed to the queue *"test"*. In theory all the messages are sent to exchanges, but the default exchange makes it possible to deliver messages directly to queues.

Direct Exchange

A direct exchange routes messages to bound queues based on the message routing key. A message published to a direct exchange will always be routed to only one queue, and for that reason is often used to distribute tasks between multiple processing clients. Using a queue for all the clients is the optimal solution to

reduce waiting time variance, since each time a client is free to work, it will take the last pending task. In this case, the load balancing happens between the processing clients.

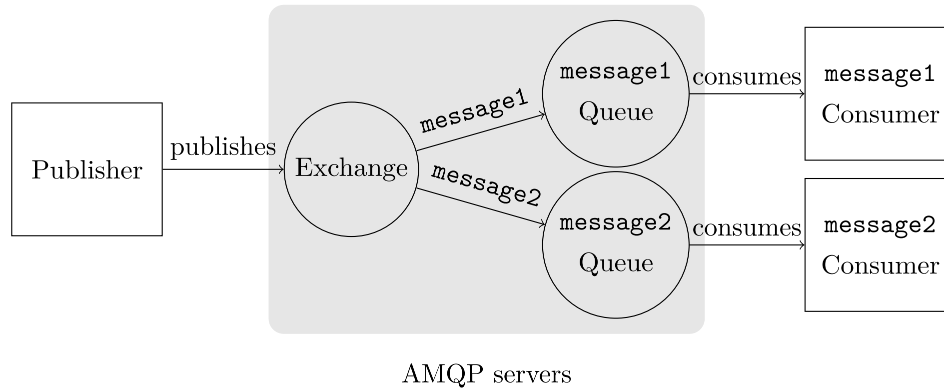

Figure 2.2: AMQP 0-9-1 direct exchange

Fanout Exchange

A fanout exchange ignores the message routing keys, it simply routes to all queues that are bound to it at the same time. If N queues are bound to a fanout exchange, a copy of the received message is delivered to all N queues at the same time. Fanout exchanges are ideal for the broadcast routing of messages.

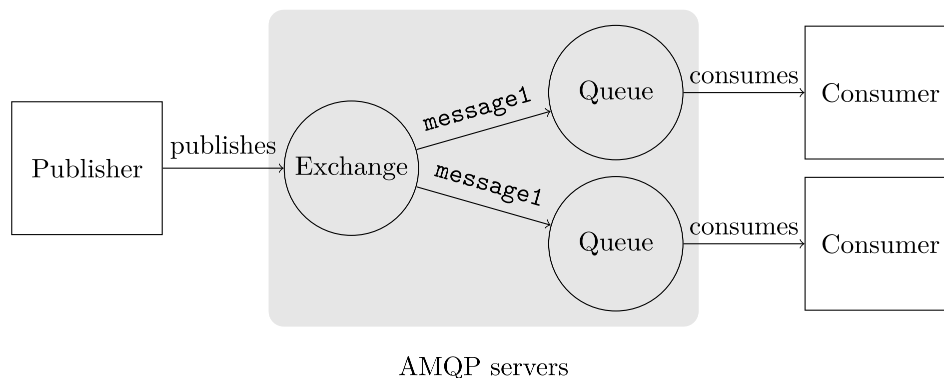

Figure 2.3: AMQP 0-9-1 fanout exchange

Topic Exchange

A topic exchange routes messages to one or many queues based on matching rules between the routing key and the queue binding pattern. This kind of exchange is often used to route multicast messages, in such systems as the ones used to implement publish/subscribe patterns.

Typically topic exchanges should be considered if a problem involves multiple consumers that can decide between receiving different types of messages.

Headers Exchange

A headers exchange ignores the message routing keys and routes the incoming messages basted on matching message headers. This type of exchange is used when the binding rules are more easily expressed in different message headers than in a routing key. It is considered that a message matches a queue when it has a header with the same name and value as the ones defined when binding the queue to the exchange.

Headers exchanges also have the possibility to match more than one header. In these cases the method that binds the queue will need additional arguments. Setting an argument with the name *"x-match"* and the value *"any"* means that the condition is `or`, so if just one of the headers matches, the message will be routed to the bound queue. Using the same argument but with the value *"all"* will make the condition `and`, in that case the message will be routed only if all specified headers match.

This kind of exchange works in a similar way to the direct exchange type, but instead of routing by matching the message routing key, it uses the more advanced message header match. This type is more powerful, since message headers can be any kind of data, not only strings.

2.2.1.2 Queues

Queues are AMQP entities that store messages until they are consumed. They share some properties with exchanges, but have additional options. A queue defined as *durable* will be stored in the broker and will still be there after a

server restart. A queue with the *auto-delete* option set to true will be deleted after the last consumer unsubscribes. Defining a queue as *exclusive* means that it can only be accessed by the current connection and will be deleted afterwards. Finally a queue can define additional arguments like the ones needed for the headers exchange or for features like defining a fixed message TTL.

A queue needs to be declared beforehand to be able to use it. When a client declares a queue, the broker will create it if it does not already exist. If it exists and has the same attributes as the ones defined in the `Queue.Declare` method, the broker will simply respond successfully. If the existing queue has different attributes, the broker will raise a channel-level exception with code 406.

2.2.1.3 Queue Names

Queue names may have a length of up to 255 bytes and are UTF-8 strings. An application may define a queue name when declaring it or ask the broker to generate a unique name by passing an empty string as the queue name. Future `Queue.Declare` methods sent on the same channel will be responded by the broker with the same generated queue name, as a client can only interact with one queue from the same channel.

An AMQP broker will have a number of predefined queues with names starting with *"amq."*. Therefore these names are reserved and trying to declare a queue that violates this rule will get a broker response error with reply code 403 (`ACCESS_REFUSED`).

2.2.1.4 Queue Durability

As we defined before, queues with the *durable* option enabled will be saved to disk and will reappear after a server restart. It is important to understand that a durable queue does not mean that the pending messages themselves will be also saved. If the broker is taken down and then restarted, a durable queue will be automatically re-declared during the startup, but only messages that are marked themselves as *persistent* will be recovered.

2.2.1.5 Bindings

Bindings are rules that exchanges use to route messages to queues. After declaring an exchange E and a queue Q, an AMQP client will normally bind them together. When defining a binding there is an optional routing key attribute that may be used by some of the exchange types. This routing key will act as a filter when the exchange decides if a received message should go to the bound queue.

Having exchanges and bindings enables routing scenarios that would need much more client logic if the client would publish directly to queues. It also improves the performance in these kinds of cases by moving the routing logic to the broker server.

There may be the case that a received AMQP message does not match any bindings and cannot be routed to any queue. If this happens the broker will check the message attributes and either drop it or return it to the publisher.

2.2.1.6 Consumers

Storing messages in queues is useless unless applications can consume them. In AMQP 0-9-1, there are two ways for applications to do this, normally they will have messages delivered to them (*push API*), but they can also fetch messages as needed (*pull API*).

When using the *push API* an application indicates interest in receiving messages from a particular queue by subscribing to it. To subscribe the application sends a `Basic.Consume` method. By default if multiple consumers are listening to the same queue, incoming messages will be delivered in a round-robin manner. On the other hand it is also possible to subscribe as an exclusive consumer and exclude all other consumers from receiving any messages.

The broker will respond to the consume method with an identifier called a consumer tag. This string can later be used by the application to unsubscribe from the queue.

2.2.1.7 Message Acknowledgements

It is possible that an application listening to a queue crashes or has problems during the processing of a received message, or there maybe network issues and the message never arrives. The specification introduces the notion of message acknowledgements, a feature that helps developers build more robust software by preventing these messages from being lost.

When an application subscribes to a queue it can choose between two acknowledgement models. The first one is automatic acknowledgement and tells the broker that it can safely remove a message from the queue just after it is sent to the consumer.

The second possibility is the explicit acknowledgement model. If a broker has a subscribed queue with this model, it will only remove a sent message after it receives a `Basic.Ack` method from the consumer. Consumers that process the data received should consider this option and only send the acknowledgement when the processing finishes. This way if there is an unexpected problem the message will still be in the broker and can be processed by another consumer in the future.

If a consumer fails to send an acknowledgement after an agreed timeout, the AMQP broker will redeliver it to another listening consumer or if there are none available, it will leave the message in the queue until a new consumer subscribes.

2.2.1.8 Rejecting Messages

Processing of a received message in a consumer application may fail. In that case, the application can tell the broker that there was a problem with the message by rejecting it. The application can ask the broker to discard or requeue the message.

To reject a message, a client sends the `Basic.Reject` method indicating the message consumer tag. Since each method only accepts one tag, there is no way to reject multiple messages in the same way as with acknowledgements.

2.2.1.9 Prefetching Messages

It is useful to be able to define the amount of messages each consumer can handle before sending the next acknowledgement. For example in cases where multiple consumers share a queue. This window can be used as a simple load balancing technique. It is also useful to improve the throughput when messages tend to be sent in batches. For example if messages are sent in constant intervals of time due to the type of producer.

2.2.1.10 Message Attributes and Payload

Messages in the AMQP model have attributes. The AMQP 0-9-1 specification defines a number of commonly used attributes and most libraries have custom setters for them, that way application developers do not need to know the exact name for them.

Most attributes are only useful if the applications that receive the messages make use of them, only a limited amount of attributes are actually used by the brokers to decide routing. Headers are a group of attributes and are used in a similar way to HTTP headers. Message attributes are set when a message is published.

In addition to the attributes, AMQP messages carry an arbitrary length byte array, the payload. The payload is opaque to the AMQP brokers, they cannot inspect or modify it. It is possible that a message has an empty payload. Typically payload data is stored in serialized formats like JSON, XML or Protocol Buffers. That way it can contain structured data that can be recovered by the consumer. the *"content-type"* and *"content-encoding"* headers are normally used to define the serialization format for interoperability, but it is not needed if the consumer already knows it.

When preparing a message it can be defined as persistent. This means that the AMQP broker should store it and be able to requeue it if the system is restarted, ensuring that the message is not lost. As we explained before, a durable exchange or queue will not persist the messages by itself, for this to happen the messages need to be persistent. Always keep in mind that publishing persistent messages will affect performance, since storing them takes time.

2.2.1.11 Methods

Operations in AMQP 0-9-1 are done by sending and receiving packages with a number of methods. They are similar to HTTP methods, not object-oriented programming language methods. Each method is defined by a two byte class id and a two byte method id. Classes are logical groupings of these methods[2].

2.2.1.12 Connections

AMQP is an application level protocol that uses TCP for reliable delivery, therefore connections are typically long-lived. Connections can use authentication and security with the TLS-SSL standard. AMQP has a `Connection.Close` method to end a connection gracefully instead of abruptly closing the underlying TCP connection.

2.2.1.13 Channels

Some applications need multiple data streams to an AMQP broker at the same type. Keeping many TCP connections open at the same time can be undesirable as it consumes system resources and makes configuring firewalls more difficult. Thankfully AMQP has the notion of channels, which are like lightweight connections that are multiplexed in the same TCP connection.

When an application uses multiple processing threads, it is a good idea to open a new channel for each one. That way they can receive messages at the same time and be truly concurrent.

Each channel communication is completely separate of the rest. This is achieved marking every AMQP frame with a channel number header. Clients use the packet header to figure out to which communication a frame belongs. For example different channels may listen to different queues and invoke different client handlers.

[2] See Appendix section 6.6 with AMQP 0-9-1 tables for a complete list

2.2.1.14 Virtual Hosts

Virtual hosts in AMQP are very similar to virtual hosts in many popular web servers like Apache. They are like multiple isolated environments inside a real broker. Each virtual host has its own entities. Clients specify the virtual host they want during the AMQP connection negotiation. The default virtualhost is usually the root path "/".

2.2.2 Transport

Now that we understand the concepts behind AMQP, let's look at how the transport layer works. The standard AMQP port number has been assigned by IANA[3] as 5672 for both TCP and UDP. AMQP is primarily TCP based, the UDP port is reserved for used in a future multi-cast implementations.

2.2.2.1 Header

The client starts a connection by sending a protocol header.

0	1	2	3	4	5	6	7
'A'	'M'	'Q'	'P'	0x00	0x00	0x09	0x01

Figure 2.4: AMQP 0-9-1 header format

The AMQP protocol negotiation is compatible with existing protocols such as HTTP that initiate a connection with an constant text string. This strategy is also very useful for firewalls that read the connection start data to decide what rules to apply.

After the client sends this header, the server and client negotiate the version. If the server rejects the protocol, it will write a valid protocol header to the socket and close it. Otherwise the server leaves the socket open and waits for client frames.

[3] Internet Assigned Numbers Authority

2.2.2.2 General Frame Format

All frames start with a 7-octet header composed of a type field (octet), a channel field (short integer) and a size field (long integer).

1 octet	2 octets	4 octets	size octets	1 octet
type	channel	size	payload	0xCE
byte	short	long		byte

Figure 2.5: AMQP 0-9-1 frame format

AMQP has only four groups of frames.

- Type = 1, `METHOD`
- Type = 2, `HEADER`
- Type = 3, `BODY`
- Type = 4, `HEARTBEAT`

Global connection frames, such as `HEARTBEAT` will always be sent through channel 0, frames that only affect a channel will be sent through channels 1-65535. To create a new channel the client will send the global `Channel.Open` method.

The *size* field is the size of the payload, excluding the frame-end octet. The fixed frame end is only used to detect framing errors in incorrect client or server implementations, since AMQP is sent through TCP and already assumes a reliable transport layer protocol.

2.2.2.3 Method Payloads

Method frame bodies consist of an invariant list of argument data fields. All method bodies start with identifier numbers for the class and method.

The *class-id* and *method-id* are constants that are defined in the AMQP class and method specifications[4]. Class id values from `0x0001-0xEFFF` are reserved for AMQP standard classes the rest may be used by implementations for non-standard extension classes.

[4]See Appendix section 6.6 with AMQP 0-9-1 tables

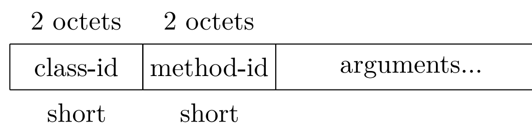

Figure 2.6: AMQP 0-9-1 method payload

Each method has a different list of fixed arguments that are passed after the method-id.

2.2.2.4 Type system

AMQP has two levels of data field specification: the native data fields used for method arguments, and the data fields passed between applications in field tables. Fields tables hold a superset of the native data fields.

The protocol defines native integer types from 8 to 64 bits, which are always unsigned and held in network byte order.

AMQP also defines a native bit field type. Bits are accumulated into whole octets. There is no requirement that all the bit values in a frame be contiguous, but this is generally done to minimize frame sizes.

AMQP strings are variable length and represented by an integer length followed by zero or more octets of data. There are two native string types:

- Short strings, stored as an 8-bit unsigned integer length followed by zero or more octets of data. Short strings can carry up to 255 octets of UTF-8 data, but may not contain binary zero octets.
- Long strings, stored as a 32-bit unsigned integer length followed by zero or more octets of data. Long strings can contain any data.

Time stamps are held in the 64-bit POSIX format with an accuracy of one second. By using 64 bits we avoid future wraparound issues associated with 31-bit and 32-bit timestamp values.

Field tables are long fields that contain packed name-value pairs. The name-value pairs are encoded as short string defining the name, an octet defining the value

type and then the value itself. The valid field types for tables are an extension of the native integer, bit, string, and timestamp types. Multi-octet integer fields are always held in network byte order. The available types where extended in version 0-9-1[5].

- Field names start with a letter, `$` or `#` and may continue with letters, `$` or `#`, digits, or underlines, to a maximum length of 128 characters.
- Decimal values are not intended to support floating point values, but rather fixed-point business values such as currency rates and amounts. They are encoded as an octet representing the number of places followed by a long signed integer.
- Duplicate fields are illegal. The behavior of an application that processes a table containing duplicate fields is undefined.

2.2.2.5 Content Framing

Certain specific methods like `Basic.Publish` and `Basic.Deliver`, carry content. Methods that carry content do so unconditionally. Content consists of exactly one content header frame that provides properties for the content followed by zero or more content body frames.

Content frames on a specific channel are always sequential. That is, they can be mixed with frames for other channels, but no two content frames from the same channel can be mixed or overlapped, nor can method frames be sent in the middle of content frames for a single content on the same channel.

The protocol specifies that any non-content frame marks the end of the content. Although the content header specifies the total size of the content and therefore the number of content frames that should follow, this allows for a broker to abort the content sending without having to close the channel.

The Content Header

The *class-id* of the content header frame matches the method frame class id. The *weight* field is unused and will be zero. The *body size* is a 64-bit value that

[5]see Appendix section 6.6 AMQP 0-9-1 tables

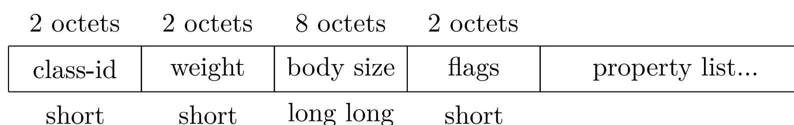

Figure 2.7: AMQP 0-9-1 content header

defines the total size of the content body, that is, the sum of the body sizes for the following content body frames. Zero indicates that there are no content body frames. The *property flags* are an array of bits that indicate the presence or absence of each property value in sequence. The bits are ordered from most high to low, bit 15 indicates the first property. The property flags can specify more than 16 properties. If the last bit (0) is set, this indicates that a further property flags field follows. There can be as many property flags fields as needed. The *property values* are class-specific AMQP data fields. Bit properties are indicated by their respective property flag (1 or 0) and are never present in the property list. The channel number in content frames is never zero, content always needs a valid channel to be transmitted.

2.2.2.6 Content Body

The content body payload is an opaque binary block followed by a frame end octet.

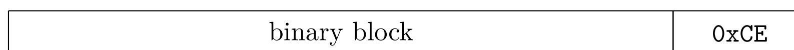

Figure 2.8: AMQP 0-9-1 content body

The content body can be split into as many frames as needed. The maximum size of the frame payload is agreed upon by both ends during connection negotiation.

A client handles a content body that is split into multiple frames by storing these frames as a single set, and either retransmitting them as-is, broken into smaller frames, or concatenated into a single block for delivery to an application.

2.2.2.7 Heartbeat Frames

Heartbeat frames tell the recipient that the sender is still alive. The rate and timing of heartbeat frames is negotiated during connection tuning. Heartbeat frames always have a channel number of zero.

2.2.2.8 Channel Multiplexing

AMQP permits clients to create multiple independent threads of control. Each channel acts as a lightweight virtual connection sharing a single TCP socket.

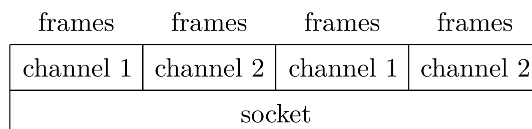

Figure 2.9: AMQP 0-9-1 channel multiplexing

An AMQP connection can support multiple channels. The maximum number of channels is defined at connection negotiation. Each endpoint should balance the traffic on all open channels in a fair fashion. This can be done on a per-frame basis, or on the basis of amount of traffic per channel. A broker normally does not allow one very busy channel to starve the progress of a less busy channel. Client implementations may vary.

2.3 AMQP 1.0

The new AMQP specification introduced a number of changes. The main one being it does not impose a structure of brokers, queues and exchanges, it only specifies a peer-to-peer transport protocol between nodes. Since the applications themselves don't define the routing rules anymore, it is not a programmable protocol in the sense 0-9-1 was. It also implements a new wire-level protocol for the exchange of messages between two endpoints and defines a new more structured type system.

2.3.1 Type System

The AMQP 1.0 type system defines an encoding format for all types that consists of a fixed constructor followed by untyped data. The constructor indicates how to read the untyped data with the typed value. An AMQP encoded data stream always begins with this constructor.

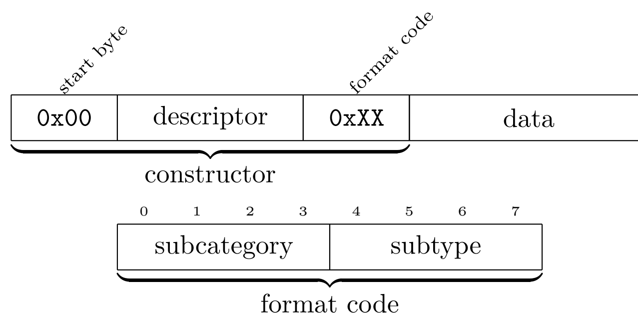

Figure 2.10: AMQP 1.0 encoding format

AMQP values can be annotated with additional semantic information beyond that associated with the primitive type, this is done in the descriptor. The descriptor portion of a described format code[6] is itself any valid AMQP encoded value, including other described values. This allows for the association of an AMQP value with an external type that is not present as an AMQP primitive.

2.3.1.1 Fixed width category

The size of fixed-width data is determined based solely on the subcategory of the format code for the fixed width value.

2.3.1.2 Variable width category

All variable width encodings consist of a size in octets followed by size octets of encoded data. The width of the size for a specific variable width encoding can be computed from the subcategory of the format code.

[6] See the complete list of available format codes in the appendix section 6.7.

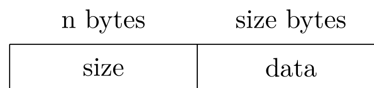

Figure 2.11: AMQP 1.0 variable width encoding format

2.3.1.3 Compound category

All compound encodings consist of a size and a count followed by *count* encoded items each of size *size*. The width of the size and count for a specific compound encoding can be computed from the category of the format code. Each item in the compound data is an encoded type itself with its own constructor.

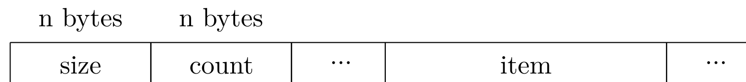

Figure 2.12: AMQP 1.0 compound encoding format

2.3.1.4 Array category

All array encodings consist of a size followed by a count followed by an element constructor followed by *count* elements of encoded data formated as required by the element constructor. This category is different than compound in that all the items share the same constructor.

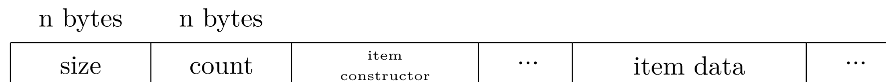

Figure 2.13: AMQP 1.0 array encoding format

2.3.1.5 Example data encoding

Figure 2.14 is an example encoding a list of book information in the AMQP 1.0 format. The data descriptor is of type `sym8` and contains the string `example:book:list` to inform the receiver. The list is of type `list8`, meaning that the size and count fields have 8 bits.

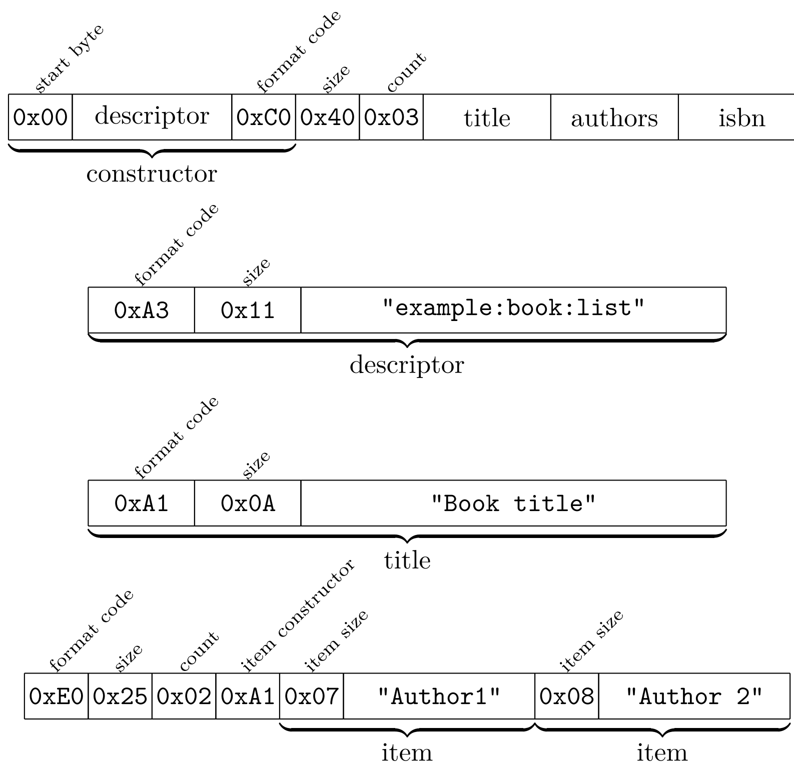

Figure 2.14: AMQP 1.0 encoding example

2.3.2 Transport

When a 1.0 client tries to connect to a server, it will send the same protocol header as the 0-9-1 version, but specifying the new version, `AMQP 0x03 0x01 0x00 0x00`. This is the only section of the protocol that is retro-compatible.

2.3.2.1 Framing

Frames are divided into three distinct areas: a fixed width frame header, a variable width extended header, and a variable width frame body. The fixed width frame header defines the total *size* of the frame in a 4 bytes field at the beginning, then the data offset (*doff*) gives the start position of the body within the frame. The type code indicates the format and purpose of the frame.

Normal AMQP frames will have a `0x00` *type* code, and the beginning of the frame body will have a performative value encoded in the AMQP 1.0 type system. The performative is similar to the 0-9-1 methods and defines the meaning of the frame. After the performative comes an additional payload that depends on the type of frame.

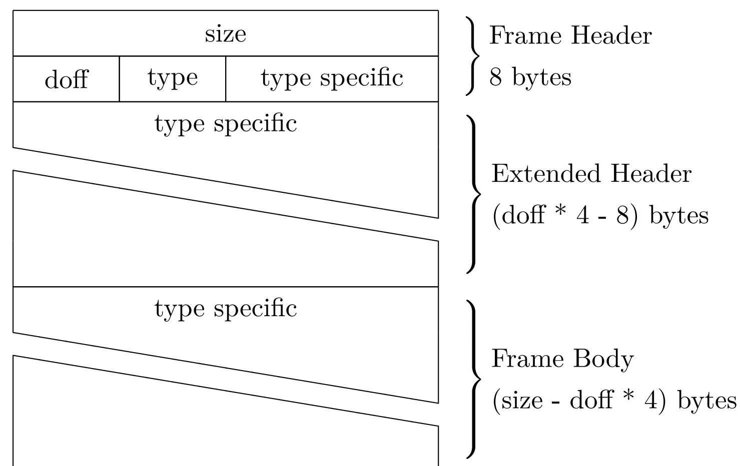

Figure 2.15: AMQP 1.0 framing format

AMQP 1.0 is different in that both ends of the connection are symmetric. Normally when one side sends a frame with a performative, the other side will send the same to confirm. As in the 0-9-1 version, frames can be multiplexed

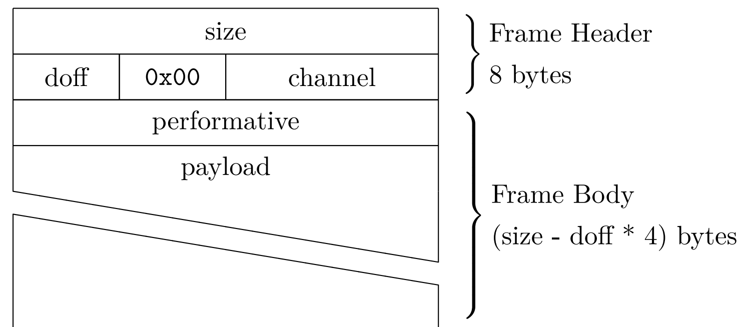

Figure 2.16: AMQP 1.0 AMQP frame

across different channels, allowing multiple links to communicate at the same time.

2.4 RabbitMQ

RabbitMQ is an open-source implementation of a AMQP 0-9-1 compliant message broker written in the Erlang language built on the Open Telecom Platform framework for clustering and failover.

Rabbit Technologies Ltd., develops and provides support for RabbitMQ. Rabbit Technologies started as a joint venture between LShift and CohesiveFT in 2007, and was acquired in April 2010 by SpringSource, a division of VMware. The project became part of GoPivotal in May 2013.

In addition to the AMQP server, RabbitMQ also offers client libraries in multiple languages including Java.

The server additionally has support for AMQP 1.0 through a plugin. Since the new version of the protocol does not have definitions for exchanges and queues, the plugin implements some custom addresses that will publish and consume them.

Chapter 3

Ichnaea AMQP

In this chapter we will describe the developed system in detail, going into each of its layers. The first layer that interacts directly with the scripts, is a bash script wrapper. Over that, we have the actual Java code that receives the processing requests, executes the bash script and sends the responses. Finally we also have written a PHP library to be used by the website that wants to send the processing requests and store the responses.

3.1 R wrapper

The original R Ichnaea code is cumbersome in that it expects a certain directory structure and multiple steps to produce the different outputs needed. The wrapper is a bash shell script that simplifies calling the Ichnaea R code from the command line by adding the familiar command line structure of arguments and parameters. It will also create a new directory structure each time, enabling it to be run multiple concurrent times, which is important if we want task servers to be able to take more than one task at a time. It has four main available processes.

Listing 3.1: Ichnaea R wrapper

```
./ichnaea.sh --debug --verbose --aging=aging/path --models=file.zip
[install|build data.csv|predict data_test.csv|fake duration:
    interval]
```

The `install` process will try to install all the needed R modules to execute the actual R scripts. This was added to simplify the server setup.

The `fake` process accepts two parameters, the total duration and the interval duration. This process is used to test the Java library. In this process, the bash script will not call any R code, it will simply wait the total duration seconds to finish and print a percentage string each interval duration.

The `build` process will call the `section_dataset_building.R` and the `section_models_building.R` R scripts. The input for this process is a dataset file in CSV format. As required parameters it needs `--aging` pointing to a valid aging file directory and `--models` pointing to a path where the results will be stored. The aging directory should contain files with names in the form of `env[column]-[season].txt`. To do this, a temporary directory is created, the scripts and agings are copied there and the result is a ZIP file with the contents of the `data_objects` directory. This way it is possible to execute the wrapper script multiple times without them affecting each other.

Finally, the `predict` process takes a test dataset also in CSV format, the aging directory and the models ZIP file generated by the `build` process as inputs and calls the `section_testing.R` script that will print the prediction results to `stdout`. This is done in a temporary directory the same way as in the `build` section.

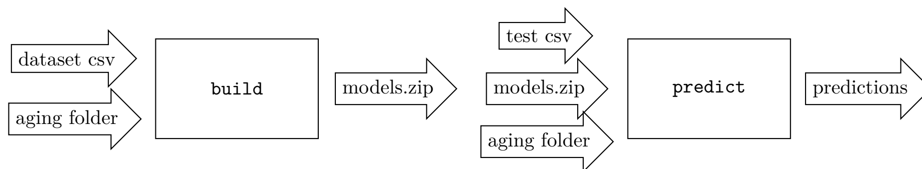

Figure 3.1: Ichnaea bash wrapper inputs and outputs

3.2 Java

The Ichnaea AMQP server is written in Java using the RabbitMQ and Apache Qpid Java AMQP libraries. It consists of multiple Java packages.

3.2.1 Shell package

The `edu.upc.ichnaea.shell` package is a set of classes to abstract interacting with the system shell. It is used to call the Ichnaea bash wrapper in a consistent way. With this classes you can execute a command from a local shell the same way as calling a command in a remote ssh session. To create a secure socket connection we used the Jsch open source library.

The shell interface has a method to run a command and returns a result object from which the output can be read. The shell has also methods to create folders and files. Finally we created the `FakeCommand`, `BuildModelsCommand` and the `PredictModelsCommand` classes to call the bash wrapper script for each process.

Listing 3.2: Ichnaea shell interface

```
public interface ShellInterface {
    public void open() throws IOException;
    public void close() throws IOException;
    public CommandResultInterface run(CommandInterface command)
      throws IOException, InterruptedException;
    public InputStream readFile(String path)
      throws IOException, FileNotFoundException;
    public OutputStream writeFile(String path) throws IOException;
    public void removePath(String path) throws IOException;
    public void createFolder(String path) throws IOException;
    public String getTempPath() throws IOException;
}
```

This package also has the `CommandReader` class to parse the results of a command reading the standard output. This can be done in real time while the command is being executed, which is important as the Ichnaea commands take a long time to complete and the library sends info back while they are still running. Parsing the responses of the actual Ichnaea commands is done in the

`UpdateProgressCommandReader`, `DatasetCommandReader` and `PredictModelsCommandReader` classes. These classes store the results of the command in Ichnaea model classes from the `edu.upc.ichnaea.model` package.

The `UpdateProgressCommandReader` class is used to parse the `fake` and `build` output to find lines that update the progress percentage and estimated finish time. This is already done in the `fake` bash wrapper script and is useful for the finished client website to show a progress bar. In the `build` command that calls the `section_models_building.R` script it is currently not implemented but it should be easy to add by someone that knows the Ichnaea R code.

Listing 3.3: Ichnaea shell progress format

```
percent: 10%
finish: lun may  5 10:10:32 CEST 2014
```

The `DatasetCommandReader` tries to read a data matrix printed by R. It also works when R splits the matrix into multiple parts. The data is then stored in a `edu.upc.ichnaea.model.Dataset` object.

Finally, the `PredictModelsCommandReader` parses the entire output of the `predict` action. It uses the `DatasetCommandReader` to parse the predicted dataset as well as the confusion matrix and it also parses the additional data returned.

3.2.2 Command line interface package

The `edu.upc.ichnaea.cli` package contains classes to simplify the definition of command line parameters and arguments for the Java client. The `Options` class is able to define a list of options that will convert the command line arguments coming from a main method `String[] args` into meaningful values. There are option classes inheriting from `Option` that define the different option types. The `Options` class will also automatically create a text printed out with the `--help` option. This is later used in the `edu.upc.ichnaea.app` classes to add options to change the behavior of the clients without having to recompile the code.

Listing 3.4: Ichnaea cli options example

```
Options options = super.getOptions();
options.add(new StringOption("shell") {
    @Override
    public void setValue(String value) throws
        InvalidOptionException {
        mShell = value;
    }
}.setDescription("The url to the remote shell."));
options.add(new IntegerOption("fork") {
    @Override
    public void setValue(int value) throws InvalidOptionException {
        mFork = value;
    }
}.setDefaultValue(mFork).setDescription(
        "The max amount of processes to spawn."));
options.add(new BooleanOption("verbose") {
    @Override
    public void setValue(boolean value) {
        mVerbose = value;
    }
}.setDefaultValue(mVerbose).setDescription(
        "Print the command output."));
```

3.2.3 Model package

The `edu.upc.ichnaea.model` package contains classes to store all the structured data used in the different processes.

There is a pair of request and response model classes for each possible process that the AMQP Ichnaea server can do. Each request has an unique id string that is maintained in the responses to identify the relation in case there are multiple requests running at the same time. The response classes inherit from a base `ProgressResponse` class that contains the start and predicted end times, as when as the processed percentage. The server will send response objects whenever there is new information about the task process.

These request and response objects contain other smaller model objects. The `Dataset` class represents a two-dimensional matrix with names in the columns. The `DatasetAging` class represents a list of agings, each containing a list of trials,

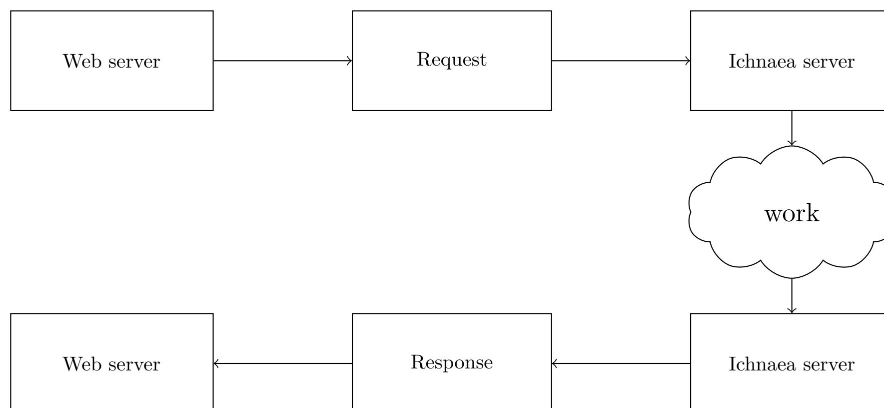

Figure 3.2: Ichnaea request response model

each containing pairs of keys and values.

To do the `fake` action, the client will send a `FakeRequest` and will receive `FakeResponse` objects. `FakeRequest` objects contain duration and interval values that will be used by the server to send responses at the specified times for test purposes.

To do the initial `build` action, the client will send a `BuildModelsRequest` object and will receive `BuildModelsResponse` objects. Each `BuildModelsRequest` object contains a `Dataset` and a `DatasetAging`. The `BuildModelsResponse` contains an array of bytes that contains the data of the output file of the Ichnaea bash script `build` section. This data can be used later to do predictions.

To do the `predict` action, the client will send a `PredictModelsRequest` object and will receive `PredictModelsResponse` objects. Each `PredictModelsRequest` contains an array of bytes that will be filled with the data returned by the `build` action response and a `Dataset` that specifies the column data that wants to be predicted.

3.2.4 Xml package

The `edu.upc.ichnaea.xml` package contains classes that are used to convert from model objects to XML and the other way around. To serialize to XML the `org.w3c.dom` classes are used, to parse the XML `org.xml.sax` classes are used. For each model class there is a class inheriting from `org.xml.sax.ContentHandler` and a

class inheriting from `edu.upc.ichnaea.amqp.xml.XmlReader` that uses the handler to parse the XML and create a model object. Fore each model class there is also a class that inherits from `edu.upc.ichnaea.amqp.xml.XmlWriter` to generate the XML.

Since the `BuildModelResponse` and `PredictModelsRequest` classes contain binary data, the XML classes that deal with them could not send it in strings. It would have been possible to send this data encoded and inside the XML. This would have a number of issues including increasing the XML parsing time. So it was finally decided to send multi-part messages using the standard MIME[1] format. Reading and writing MIME is done using the `javax.mail` library.

Listing 3.5: Ichnaea MIME build-models response

```
--frontier
Content-Type: text/xml

<response end="2012-12-28T22:00:00.000+0100" id="455" progress
    ="1.0"
start="2012-12-27T08:00:00.000+0100" type="build_models"/>
--frontier
Content-Type: application/zip
Content-Transfer-Encoding: base64

cGFjbw==
--frontier--
```

There is a full set of examples of the different models used in appendix section 6.5.

3.2.5 Data package

The `edu.upc.ichnaea.data` package contains classes that are used to convert from model objects to additional formats required by the Ichnaea R scripts. `CsvDatasetReader` and `CsvDatasetWriter` convert `Dataset` objects from and to comma separated value files.

[1] Multi-Purpose Internet Mail Extensions

Listing 3.6: Ichnaea dataset CSV file

```
CLASS;FC;FE;CL;SOMCPH;FTOTAL;FRNAPH;FRNAPH I;FRNAPH II;FRNAPH III;
    FRNAPH IV;RYC2056;COP;ETHYLCOP;EPICOP;CHOL;DiE;FM-FS;Hir;DiC;
    ECP;ECT;GA17;Dentium;Adolescentis;DA;HBSA-Y;HBSA-T
P1-HM01;2,20E+07;1,23E+06;2,40E+05;1,93E+07;1,35E+06;8,40E
    +05;6;33;61;0;3,35E
    +04;5962,2;250,4;113,2;504,9;0,96;40,00;35,00;0,98;75,00;55;2,10
    E+05;1;1;2;;
P1-HM02;2,20E+07;9,30E+05;3,40E+05;6,60E+06;5,60E+05;5,40E
    +05;0;78;22;0;4,00E
    +04;1094,0;435,7;148,8;857,2;0,98;92,00;4,00;0,86;95,00;54;1,40
    E+04;1;1;2;;
```

`AgingFolderReader` and `AgingFolderWriter` convert `DatasetAging` objects from and to folders with aging text files. An aging folder contains a file for each dataset column and each season. Each file contains a list of trials, each trial is a pair of values.

Listing 3.7: Ichnaea aging file

```
# Envelliment BA Estiu (2 assajos)

0    6.91
24   6.05
48   5.99
72   4.60

0    6.91
24   6.05
48   5.99
72   4.52
```

Listing 3.8: Ichnaea aging folder

```
envBA-Estiu.txt     envFC-Estiu.txt          envFRNAPH.IV-Estiu.
    txt
envBA-Hivern.txt    envFC-Hivern.txt         envFRNAPH.IV-Hivern.
    txt
envBE-Estiu.txt     envFE-Estiu.txt          envGA17-Estiu.txt
envBE-Hivern.txt    envFE-Hivern.txt         envGA17-Hivern.txt
envCL-Estiu.txt     envFMFS-Estiu.txt        envHBSA.T-Estiu.txt
envCL-Hivern.txt    envFMFS-Hivern.txt       envHBSA.T-Hivern.txt
envCNFC-Estiu.txt   envFRNAPH-Estiu.txt      envHBSA.Y-Estiu.txt
```

```
envCNFC-Hivern.txt  envFRNAPH-Hivern.txt      envHBSA.Y-Hivern.txt
envDiC-Estiu.txt    envFRNAPH.I-Estiu.txt     envHiR-Estiu.txt
envDiC-Hivern.txt   envFRNAPH.I-Hivern.txt    envHiR-Hivern.txt
envDiE-Estiu.txt    envFRNAPH.II-Estiu.txt    envRYC2056-Estiu.txt
envDiE-Hivern.txt   envFRNAPH.II-Hivern.txt   envRYC2056-Hivern.txt
envECP-Estiu.txt    envFRNAPH.III-Estiu.txt   envSOMCPH-Estiu.txt
envECP-Hivern.txt   envFRNAPH.III-Hivern.txt  envSOMCPH-Hivern.txt
```

3.2.6 Client package

The `edu.upc.ichnaea.client` package contains classes that will send and process the requests and responses. Each processing client has the ability to spawn multiple threads and work on each of them, the amount can be adjusted from the command line. This is useful when there are servers with lots of processing power that can run multiple copies of Ichnaea at the same time.

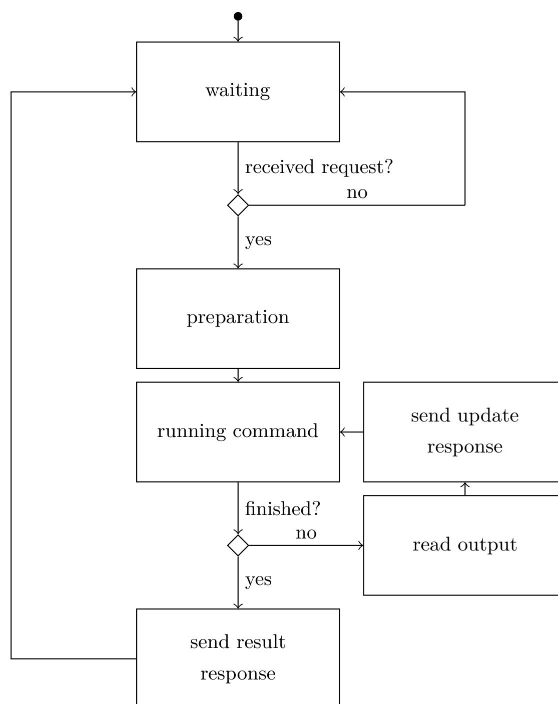

Figure 3.3: Ichnaea process client generic state diagram

`FakeRequestClient` will take a `FakeRequest` object, serialize it and send it to

the `ichnaea.fake.request` exchange on the specified AMQP server. If a `FakeProcessClient` is listening to the `ichnaea.fake.request` queue, it will deserialize the request and call the Ichnaea bash script passing the total time and update interval as arguments. When the bash script prints to the standard output, the process client will create and serialize `FakeResponse` objects. If the `FakeRequestClient` is listening when the `FakeResponse` objects arrive, it will print the received information.

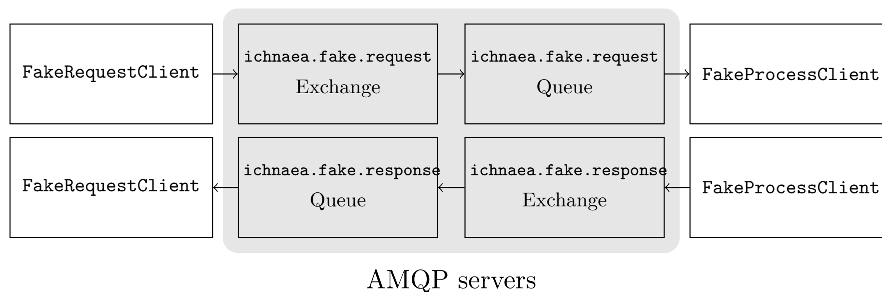

Figure 3.4: Ichnaea fake routing

`BuildModelsRequestClient` will take a `BuildModelsRequest` object, serialize it and send it to the specified AMQP server. If the server is running the `BuildModelsProcessClient` it will deserialize the request, create the files that the Ichnaea bash script needs, create a `ShellInterface` object, run the Ichnaea bash script, listen to its output and send `BuildModelsResponse` objects when possible. If the `BuildModelsRequestClient` is listening when the `BuildModelsResponse` objects arrive, it will print out the received information.

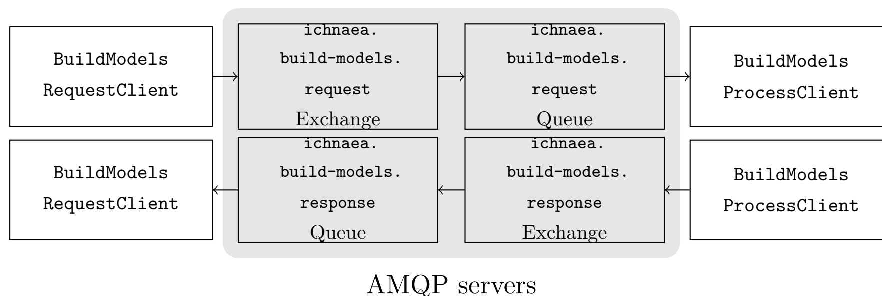

Figure 3.5: Ichnaea build models routing

`PredictModelsRequestClient` will take a `PredictModelsRequest` object, serialize it

and send it to the specified AMQP server. If the server is running the `PredictModelsProcessClient` it will deserialize the request, create the files that the Ichnaea bash script needs, create a `ShellInterface` object, run the Ichnaea bash script, listen to its output and send `PredictModelsResponse` objects when possible. If the `PredictModelsRequestClient` is listening when the `PredictModelsResponse` objects arrive, it will print out the received information.

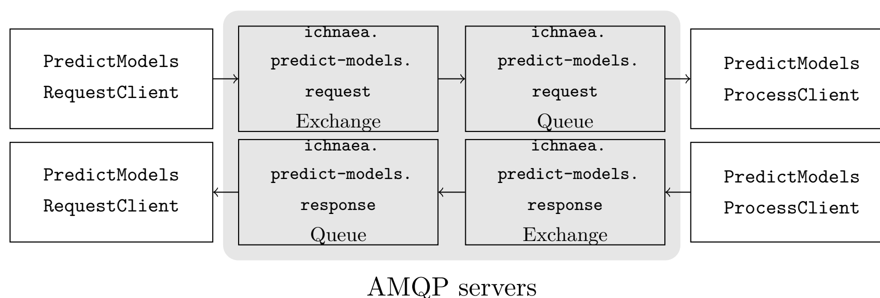

Figure 3.6: Ichnaea predict models routing

3.2.7 Proto package

The `edu.upc.ichnaea.proto` package was introduced the last. Its main purpose is to abstract the methods between the clients and the AMQP client library to be able to use libraries other than RabbitMQ. Initially the Ichnaea clients would call `com.rabbitmq.client` classes directly, this introduced a problem when we wanted to test AMQP version 1.0 as the RabbitMQ client library does not support this version and we needed to use Apache Qpid Proton[2]. To do this we created two interfaces, `ConnectionInterface` and `ChannelInterface`, with the minimum needed methods for the clients to work. We then implemented the interfaces using the RabbitMQ library in the classes `RabbitConnection` and `RabbitChannel`. Next, we created a factory class called `ConnectionFactory` that would return a new `ConnectionInterface` given a set of parameters. Then we replaced all the RabbitMQ dependent code in the clients with the new interfaces and used the factory to create the connection from a set of command line options. Now the entire Ichnaea AMQP code is library independent.

[2]http://qpid.apache.org/proton/

Finally we wrote Qpid implementations for the `proto` interfaces. To implement AMQP 1.0 we took into account how the RabbitMQ 1.0 plugin wires the exchanges and queues, so this mode will only work with a RabbitMQ server. We found a number of problems when communicating between the Qpid client and the server due to fixed expected types in the server and client that would make them crash. For example Qpid sets the `ContentType` field of a message as a `sym8` but RabbitMQ expects a UTF-8 string and gives an error, or when reading a message sent with AMQP 0-9-1, Qpid will fail to parse the message. Ultimately we managed to make publish and consume work, removing these problematic parts that probably will get fixed by the library authors in the future.

Listing 3.9: Ichnaea proto connection interface

```
package edu.upc.ichnaea.amqp.proto;

import java.io.IOException;

public interface ConnectionInterface {
    public void open() throws IOException;
    public void close() throws IOException;
    public void update() throws IOException;
    public ChannelInterface createChannel() throws IOException;
}
```

Listing 3.10: Ichnaea proto channel interface

```
package edu.upc.ichnaea.amqp.proto;

import java.io.IOException;

public interface ChannelInterface {

  public interface Consumer {
        public void handleDelivery(String replyTo, byte[] body)
            throws IOException;
  }

  void exchangeDeclare(String exchange, String type) throws
      IOException;
  void queueDeclare(String queue) throws IOException;
  void queueBind(String queue, String exchange) throws IOException;
  void basicConsume(String queue, Consumer consumer) throws
```

```
        IOException;
    void basicPublish(String exchange, String replyTo, String
        contentType,  byte[] body) throws IOException;
}
```

3.2.8 App package

The `edu.upc.ichnaea.app` package contains classes that combine the other packages into one command line app. These classes will take the command line parameters passed, read them using the `edu.upc.ichnaea.cli` classes, create and execute the correct client from the `edu.upc.ichnaea.client` package.

Listing 3.11: Ichnaea app example execution

```
# the fake:process task
./target/ichnaea-amqp.jar build-models:process -i ../r/files/
    ichnaea.sh --connection-type=Qpid
```

3.3 PHP library

The PHP library implemented can be used by a website to send Ichnaea AMQP requests and listen for its responses. It uses the `php-amqplib` library to interface with the AMQP protocol.

In a similar fashion to the Java library, the PHP library is split up into namespaces. Since there was no way to parse MIME data, we wrote our own MIME parser that can be found in the `Ichnaea\Amqp\Mime` namespace. In the PHP library we coded the CSV and aging folders parsers inside the respective models classes to simplify its usage. The XML readers and writers, found in the `Ichnaea\Amqp\Xml` namespace, were written using the build-in PHP `DOMDocument` classes.

It is important to note that since PHP is single threaded, to listen to the Ichnaea responses, the web server will need to run a PHP script from the command line. This script should store the response data in a database that can be read by the PHP code run from the web server.

3.3.1 Test PHP app

We developed an example application using the Silex PHP server and our PHP library on the backend and the Angular Javascript framework in the frontend. This application can be used to send requests to the server and see the responses and was developed to test the PHP library and the whole Ichnaea workflow. We use AJAX requests to update the website in real-time with the percentages of the tasks that were requested.

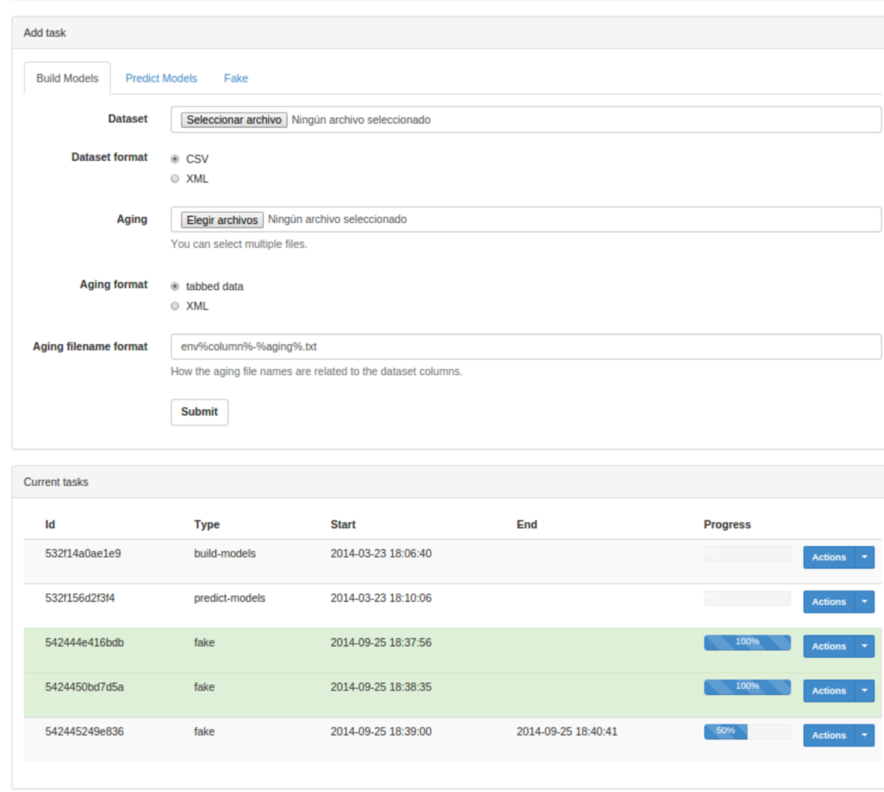

Figure 3.7: Test php app screenshot

Chapter 4

Analysis of a test run

In this chapter, we will analyze the network communication during a test run of the developed system and explain how the different AMQP connections work. We will see both ends, the connection of the Ichnaea request client, which is the publisher of requests and the consumer of responses, and the connection of the Ichnaea process client, which is the consumer of requests and the publisher of responses. To do these tests we used Wireshark on a Linux machine running both the RabbitMQ server and the Ichnaea code. In a real-life setup, these systems could be separated.

4.1 General setup

When a client establishes an AMQP 0-9-1 connection, some steps are always done first. The client first sends the AMQP protocol version signature[1]. If the AMQP server would not support the requested version, it would respond with the version it accepts and close the connection. Since in this example, the server is RabbitMQ and supports 0-9-1, it directly starts with a `Connection.Start`[2] method, which contains a list of the server capabilities. The client responds with `Connection.Start-Ok` listing the clients capabilities. Next, the server sends `Connection.Tune` to setup additional connection information for the client, maximum amount of

[1] see section 2.4 on the AMQP 0-9-1 header format
[2] class-id=10, method-id=10, see figure 2.6 on the method payload

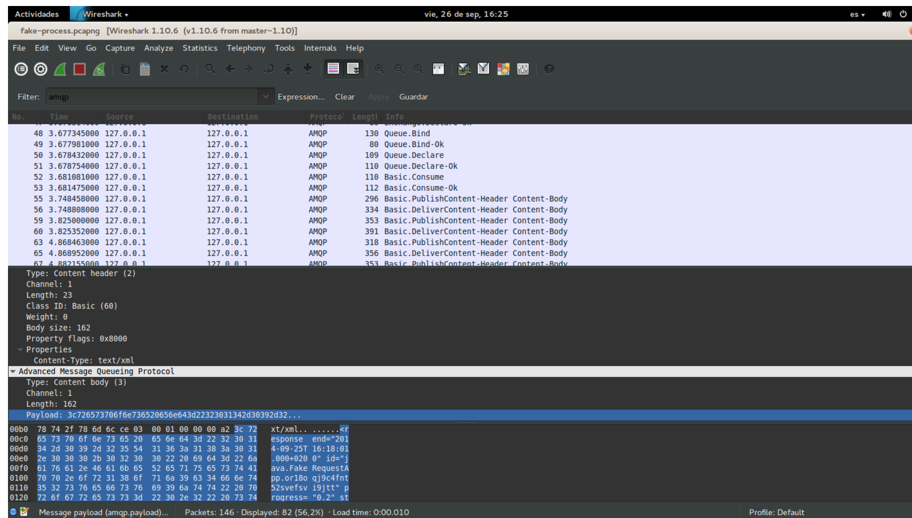

Figure 4.1: Capturing AMQP traffic with Wireshark

channels and frames supported and the heartbeat times expected, the client acknowledges sending `Connection.Tune-Ok`. Finally to finish the setup, the Ichnaea client opens the connection specifying the default virtualhost `/`, virtualhosts are separate environments that can be run on the same AMQP server.

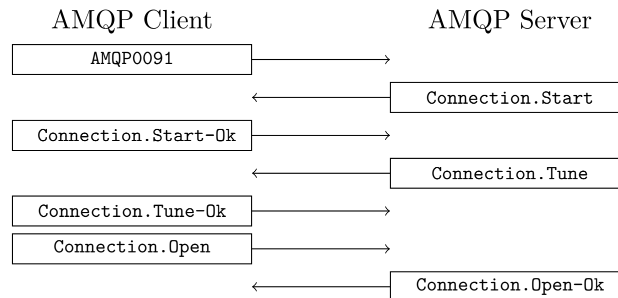

Figure 4.2: Ichnaea AMQP protocol negotiation

type	channel	length	class-id	method-id
0x01	0x0000	404	0x000a	0x000a
arguments				

Figure 4.3: Ichnaea AMQP `Connection.Start` data

4.2 Fake request

4.2.1 Setup

After the `FakeProcessClient` connects to the AMQP server sending the packets described in the general client setup, it will declare the AMQP exchanges and queues it needs. This setup can be done directly on the server, but its easier to code it in the client, that way the server does not need any setup. First of all, the client sets up a new channel 1[3]. This is done to then be able to consume messages on that channel. After the server confirms the new channel, the client declares the `ichnaea.fake.response` exchange with `fanout` mode, the `ichnaea.fake.response` queue and then binds the queue to the exchange. This exchange is where the process client will send the fake responses. The exchange is in `fanout` mode, that way if multiple queues are bound to it, all of them will get the response message. In this case there is only one AMQP server and only one queue, but if there were multiple web servers with multiple databases, it would make sense to send the responses to all of them.

After setting up the response entities, the process client declares the `ichnaea.fake .request` queue and subscribes to it with the `Basic.Consume` method. This means that whenever another client sends a fake request message, the process client will receive it and can start working. After this setup, the process client blocks waiting for incoming requests. The consume method has an additional boolean parameter to define if the AMQP server should wait for an ACK response from the process client to remove the message from the queue. This is useful for situations where it the consumer fails, we want to send the message to the next consumer. In our case, we set `autoack`[4] to `true`, since we always accept the request,

[3]as described in section 2.2.1.13 on AMQP channels
[4]see section on Message Acknowledgements

and in case of failing we will send a response with additional error information. This is useful for the request client, since most errors will come from bad input data.

In a similar way to the process client, the request client will setup the `ichnaea.fake.request` exchange and queue and bind them. The difference is that request exchange is defined as in `direct` mode, since we want each request only to be processed once even if multiple queues were bound in the future. Then the request client declares the `ichnaea.fake.response` queue and listens to it.

Listing 4.1: Ichnaea fake request setup

```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

ConnectionFactory factory = new ConnectionFactory();
factory.setUri("amqp://localhost");
Connection conn = factory.newConnection();
Channel channel = conn.createChannel();
channel.exchangeDeclare("ichnaea.fake.response", "fanout", true);
channel.queueDeclare("ichnaea.fake.response", false, false, false,
    null);
channel.queueBind("ichnaea.fake.response", "ichnaea.fake.response",
     "");
channel.queueDeclare("ichnaea.fake.request", false, false, false,
    null);
```

The additional parameters that can be passed to the RabbitMQ methods for declaring exchanges and queues define additional behaviors such as if they are durable, meaning they survive a server restart, or exclusive, meaning they only work withing the current connection. The additional parameter that is passed to the `queueBind` method is the routing key, in Ichnaea we leave it empty as we will use different exchanges and queues for each request and response. A different architecture could be implemented where we could send every request and response to the same exchange and the exchange could decide to which queue the message should go depending on the routing key.

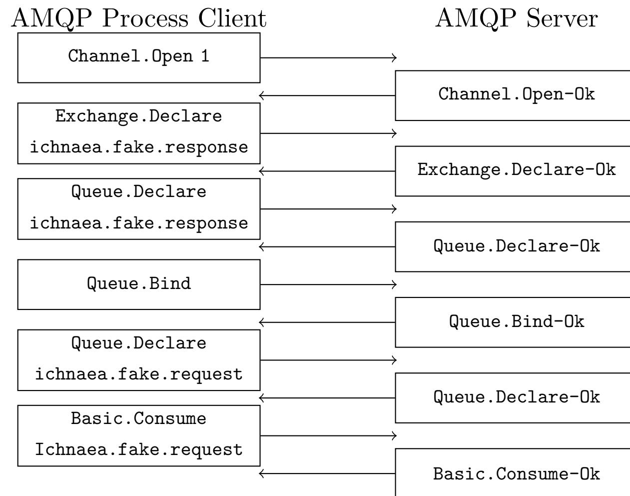

Figure 4.4: Ichnaea AMQP fake process client declaration

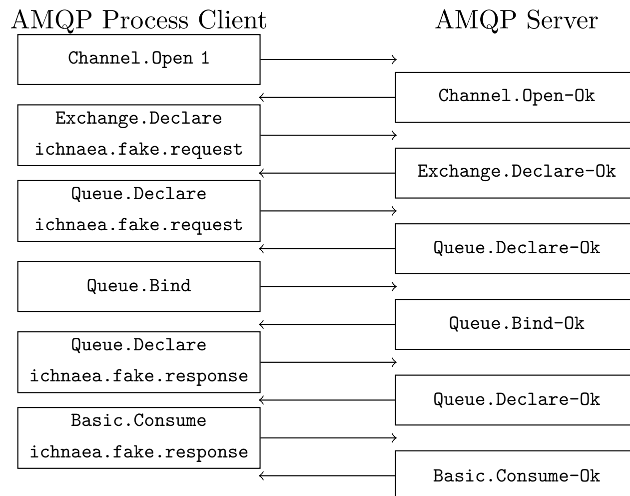

Figure 4.5: Ichnaea AMQP fake request client declaration

4.2.2 Process

The fake request client will start sending one TCP packet containing 3 AMQP frames. The first one is the `Basic.Publish` method to the `ichnaea.fake.request` exchange. The second one is of type `Content-Header`[5] and contains additional info about the content that is going to be sent, in this case it defines the content type as `text/xml` as we are sending XML representations of the requests and responses. Another property is the standard `Reply-To` property, that defines the unique routing key that is uses by the responses to relate them to the request. This routing key is random and generated by the request client. Finally it sends the XML content. When the AMQP server receives the request message, it looks at the routing table and decides that the message should go on the `ichnaea.fake.request` queue to which the process client is subscribed.

If there were no subscribed clients, the message would stay in the queue until some client would subscribe. Since there is a client subscribes, The AMQP server itself sends a TCP packet containing 3 AMQP frames to the process client. The first one is `Basic.Deliver` with the routing information that it came from the `ichnaea.fake.request` exchange, the second and third frames are the same content header and the contend body that where sent to the AMQP server by the request client.

Basic.Publish
Content-Header
Content-Body

Figure 4.6: Ichnaea AMQP fake process publish packet

Listing 4.2: Ichnaea fake request XML

```
<request duration="10.0"
    id="java.FakeRequestApp.or18oqj9c4fnt52svefsvi9jtt"
    interval="1.0" type="fake"/>
```

Once the process client parses the request, it will call the Ichnaea bash script, and since it is a fake request, it will start printing out end time estimations and

[5]type=2, see the AMQP General Frame Format section

percentages of progress. In this test case we requested 10 seconds duration and 1 second updates. So each second the process client will send a response with the percentage of progress and the estimated end time, 10 seconds after the start. These responses are send to the AMQP server and then to the request client in the same way that the request was send, through `Basic.Publish` and `Basic.Deliver` methods with content.

Basic.Deliver
Content-Header
Content-Body

Figure 4.7: Ichnaea AMQP fake process deliver packet

Listing 4.3: Ichnaea fake response XML

```
<response
    id="java.FakeRequestApp.or18oqj9c4fnt52svefsvi9jtt"
    type="fake" progress="0.2"
    start="2014-09-25T16:17:51.000+0200"
    end="2014-09-25T16:18:01.000+0200" />
```

4.3 Build models

The `build-models` workflow is very similar to the `fake` one. In this case the declared exchanges and queues are `ichnaea.build-models.request` and `ichnaea.build-models.response`. Since the `build-models` request contains all the aging and dataset information, it is much bigger than the `fake` one. This makes the TCP layer split the AMQP frames in multiple TCP packets. This also happens when the process client delivers the result data, which is `multipart/mixed` instead of XML, since it contains a base64 encoded ZIP file with the `Rdata` files needed for the `predict-models` step.

In this workflow we can also detect `Heartbeat` packets, that are sent from both clients to the AMQP server during the time the process client is running the Ichnaea R script. This is due to the current implementation of the R code of this step not printing process percentages nor estimated end times. If this were the

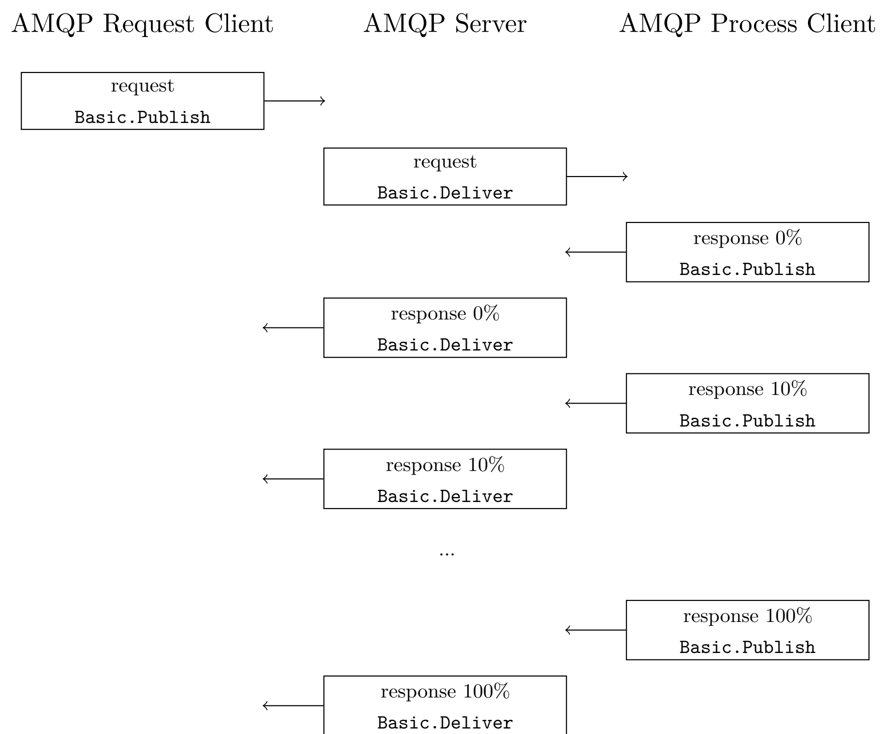

Figure 4.8: Ichnaea AMQP request-response process

case, the process client would take the additional information and send response messages, eliminating the need for heartbeats.

4.4 Predict models

The `predict-models` workflow is also very similar, the declared exchanges and queues are `ichnaea.predict-models.request` and `ichnaea.predict-models.response`. In this case, the R script prints out the total number of steps and the current step it is working on, and with that information the client sends responses each time, therefore there are no heartbeats sent.

4.5 Errors

There are multiple reasons why a process client can fail, from hardware problems to setup problems. We can simulate an error by starting a process client that points to an invalid bash script.

4.5.1 Failing declaration

A possible way of failing would be if another AMQP client would have declared the same exchange or queue our clients are using but with different parameters. To test this situation, we changed an already declared `fanout` exchange to `direct` mode. After the client sending the `Exchange.Declare` method, instead of the server responding with `Exchange.Declare-Ok`, it directly sends `Channel.Close` with a reply code 406 and reply text `PRECONDITION_FAILED` as was defined by the AMQP standard[6].

4.5.2 Failing process client

The process client consume command is sent with automatic acknowledge, so if for some reason the bash script execution fails, the process client sends a

[6]see section 2.2.1.1 on AMQP exchanges

response message with an additional XML property error with the error text printed out by the bash script.

Listing 4.4: Ichnaea process client error response

```
------=_Part_1_753816592.1411823972136
Content-Type: text/xml

<response end="2014-09-27T15:19:32.128+0200" error="failed command
    result: Cannot run program &quot;../Ichnaea.sh&quot;: error=2,
    No existe el archivo o el directorio" id="java.
    BuildModelsRequestApp.t03ai7n91v6b6cr452ttbiv9g2" progress
    ="1.0" start="2014-09-27T15:19:31.931+0200" type="build_models
    "/>
------=_Part_1_753816592.1411823972136--
```

4.6 Multiple requests

Since the implementation of the Ichnaea AMQP clients requires each one to run as a separate process, running multiple clients will create different TCP connections and each AMQP connection will use only one channel, so all channels are number 1. It would also be possible to write one Java program that creates a `build-models` and a `predict-models` client at the same time. Since the AMQP 0-9-1 standard specifies that content frames need to be sequential on the same channel[7], this would mean that the server would have to wait for one message to be finished to send the next one, which is slower. If the application receiving the messages is single-threaded there is no difference, as it will only be able to handle one message at the time. In the case of the Java clients, which are multi-threaded, it would be better to use different channels for each task.

4.7 Test PHP app requests

The final test we did was capturing the traffic sent by the PHP Ichnaea test application. The web application uses AJAX to update the state of the Ichnaea requests without reloading the website, therefore each time an AJAX update

[7]see the AMQP content framing section 2.2.2.5

happens there is a AMQP server connection and disconnection. This behavior could be easily improved by only connecting when there is a AMQP request to be sent.

If we start running the `consumer.php` script, which runs in the background to listen to all the responses and update the database, we see three `Basic.Consume` methods, one for each type of response, being called on the same AMQP channel. In this case it makes sense as the PHP consumer is single-threaded and will handle one response after the other.

When the website sends a request it does the same as the Java request client, a `Basic.Publish` method followed by `Content-Header` and one or more `Content-Body` frames.

The PHP consumer works also very similarly to the Java version, the only difference is that it sends a `Basic.Ack` method frame after receiving a response message. This `ack` package is needed, since in the PHP version we sent the `Basic.Consume` method without the `auto-ack` option. This is useful in case there is some unhandled exception in the php code. In that case the PHP would fail before sending the `ack` frame and the response message would still be in the AMQP queue the next time the consumer is started.

4.8 AMQP 1.0

To test the AMQP 1.0 support we had to do some additional setup[8]. We enabled the 1.0 support plugin on the RabbitMQ server and changed to to using Apache Qpid Proton library which uses 1.0 by default.

When starting up, after the version negotiation there is a SASL negotiation. SASL[9] is a standard way of doing authentication and authorization used in many systems today. It is the build in way in which AMQP 1.0 clients are authenticated. In our test case we accept anonymous logins so the authentication is successful.

The client then sends `open` performative to create a connection and `begin` to establish a channel to communicate. The server responds with the same frames

[8]see the appendix section 6.2 on how to set it up
[9]Simple Authentication and Security Layer

and we have an established link.

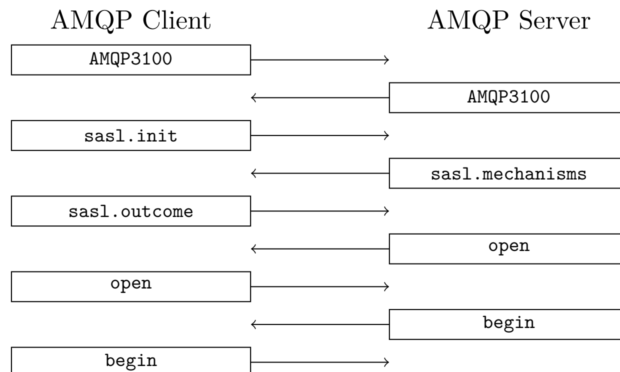

Figure 4.9: Ichnaea AMQP 1.0 protocol negotiation

In AMQP 1.0 there are no ways to declare queues and exchanges, so this step is skipped altogether. RabbitMQ AMQP 1.0 implements custom addresses for sending to exchanges (`/exchange/[name]`) and consuming from queues (`/queue/[name]`).

To consume a queue, we send an `attach` performative with the queue address and the role type receiver to the server, who then responds with the same but with sender role. And after that the client sends a `flow` performative. This frame is used for flow control and establishes the way the receiver would like to get the incoming messages. In our test case, the client is defining link-credit as 1024, meaning the server can only send that amount of messages before getting acknowledgements from the client.

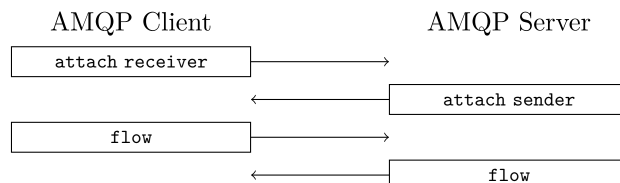

Figure 4.10: Ichnaea AMQP 1.0 consume

When sending messages with AMQP 1.0 the client will also start with an `attach` performative, but this time declaring itself as the sender for the link. In the cases where a client is consuming and also sending, it will call `begin` two times

to create two different channels, and then send each `attach` on each different channel. Once the link is established, the client sends a `transfer` frame with the message data. In our case we're sending the entire Ichnaea request or response XML withing a binary type, but as we described in section 2.3.1 about the new 1.0 type system, this data could be much more structured now than with version 0-9-1. After the transfer the server will respond with a `disposition` performative that declares if the message was acknowledged or there was an error.

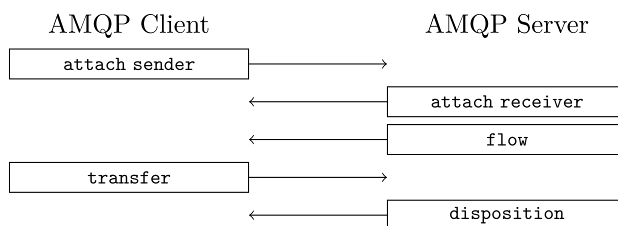

Figure 4.11: Ichnaea AMQP 1.0 publish

When consuming messages from the server, AMQP 1.0 works the same way but in reverse. The server sends `transfer` frames and the client responds with `disposition` frames.

Chapter 5

Conclusions

In this chapter we will discuss the results obtained by the developed system and compare them to the goals described in the introduction. We will discuss some of the encountered problems and describe ideas for future development.

The developed system managed to fulfill the goals defined in the introduction.

- *an easy way to run the original Ichnaea code in multiple servers*

The implemented infrastructure can scale easily by adding more machines and running more process clients. As shown in section 4.2.2, each process client will listen to the same RabbitMQ server queue, and process the oldest pending requests when it is free.

A good solution to scale the process clients could be done using cloud computing services like Amazon EC2 or the UPC computing cluster. We would need to set up a virtual machine image using Linux that would run the process clients on start up. This solution would be optimal as we could adjust the number of process clients and processing power instantaneously depending on the current demand.

On the requesting side it is easy to send requests to by using the Java and PHP libraries. Requests could scale in the same way HTTP services are scaled using load balancers to split the requests between different machines.

- *a queueing server that can run shell commands*

In our developed system the task queueing is done in the RabbitMQ server, as described in section 2.2.1.2.

To run shell commands we implemented the `edu.upc.ichnaea.shell` Java package, explained in section 3.2.1. It is used in the AMQP process clients to run the Ichnaea bash script wrapper, and it could be used in the future to setup new tasks.

Listing 5.1: Ichnaea shell example

```
ShellInterface shell = new SecureShell("ssh://user:pass@host");
CommandInterface cmd = new Command("ls");
CommandResultInterface result = shell.run(cmd);
new CommandReader(result){
  protected void onLineRead(String line) {
    // called on every line read
  }
};
```

- *an XML definition for the different Ichnaea requests and responses*

Using the developed XML parsers and writers, described in section 3.2.4, we can serialize the Ichnaea request and response data. This system could also be used to store serialized requests and responses in a generic way. There is a complete list of XML models used in section 6.5.

- *a server logic to process the different Ichnaea requests*

The `edu.upc.ichnaea.client` package, described in section 3.2.6, implements the logic to request and process the `build-models` and `predict-models` tasks, as well as the test `fake` task.

- *a PHP library to interact with the Ichnaea server from a website*

The PHP library we implemented, described in section 3.3, is used in the PHP application developed by Nahuel Velazco. We also created a simple PHP app to test the library.

In addition to the initial goals, we implemented the system without forcing a particular AMQP library and enablet it to switch between RabbitMQ and and Apache Qpid client libraries. Having done this, we were able to analyze the way the developed software communicates and compare both protocol versions.

5.1 Problems and workarounds

During the development of the queue system we found some issues that had to be addressed.

When working with the shell in Java we originally would read the output of an executed command after the command finished. This proved to be a problem when reading Ichnaea output since the tasks took a long time and would print out valuable information while running. We changed this to a streamed approach that enables reading the output while the command is running. That way we can send progress updates and the consumer client can update the database and the website before.

Originally the whole Java code was bundled with the RabbitMQ client library. Since we wanted to test the system using the AMQP protocol version 1.0 and the library does not support this, we had to abstract the AMQP client funcionality in an interface. This interface was then implemented using RabbitMQ for protocol version 0-9-1 and Apache Qpid for protocol version 1.0.

5.2 Further development

A good improvement to the system would be that the Ichnaea R scripts would print estimated end times and percentages of work done. The system is already prepared to detect lines with this information as it is already working in the `fake` mode. Adding this would make the website able to show a progress bar and estimated end time, which would be much more user friendly.

It would be much better if the `build-models` step could save its output data in a exportable format. Right now the data is saved in the R native binary `Rdata` format. That's the reason why we pack the `build-models` response in a zip file

instead of adding the data in the XML response. If the data was readable, the website could show additional data about the `build-models` response.

A part that was not implemented due to time constraints was the AMQP 1.0 support for the PHP library. In general the 1.0 support in RabbitMQ should improve in the future for it to be a viable option, because right now the plugin will not work with certain message configurations that are actually valid.

An interesting further development would be to work on mobile clients for the system. These clients could work as a frontend for the website or directly send and receive AMQP data. They could allow microbiology researchers to send data to be processed by Ichnaea directly from the place where the measurements are being taken.

Chapter 6

Appendix

In this chapter we have the system user and developer guides that can be used to setup and continue developing the Ichnaea AMQP server. We also added a glossary of terms used in the document and in detail AMQP protocol tables referenced in chapter 2.

6.1 User's guide

These are the instructions to compile, run and test the Ichnaea AMQP server. This guide was tested on Ubuntu Linux 14.04.

To run the original Ichnaea R scripts, we have to install the R interpreter and modules, in Debian-based Linux systems those are available through `apt-get`. Additionally Ichnaea requires some R modules that are not present in the default R installation. We added a simple way to install all the required modules through the R bash wrapper script using the `--install` argument.

```
# install the R interpreter
sudo apt-get install r-base-core

# install required R modules
sudo ../r/files/ichnaea.sh --install --debug
```

Now we can run the wrapper script from the command line with the test data to see if everything is working.

```
# fake request
../r/files/ichnaea.sh --debug fake 10:1

# build models request
../r/files/ichnaea.sh --debug --aging ../r/fixtures/aging --models
    /tmp/cyprus_models.zip build ../r/fixtures/cyprus.csv

# predict models request
../r/files/ichnaea.sh --debug --models ../r/fixtures/cyprus_models.
    zip predict ../r/fixtures/cyprus_test.csv
```

6.1.1 Setup required programs and libraries

TO compile the java part of the Ichnaea AMQP server, we will have to install a Java Development Toolkit and additional tools as well as the RabbitMQ server.

```
# install oracle jdk 7
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java7-installer

# install rabbitmq-server
sudo apt-get install rabbitmq-server

# install maven 3.0
sudo apt-get install maven
```

6.1.2 Compile and Run

Now we can build the Ichnaea Java code using Maven. This will build an executable file in `target/ichnaea-amqp.jar`. Use the `-h` command line argument to get all the available options.

```
mvn clean compile assembly:single
```

Test the Java clients running the different request and process tasks.

```
# the fake:process task
./target/ichnaea-amqp.jar fake:process \
    -i ../r/files/ichnaea.sh --verbose

# the fake:request task
./target/ichnaea-amqp.jar fake:request \
    --duration=10 --interval=1

# the build-models:process task
./target/ichnaea-amqp.jar build-models:process \
    -i ../r/files/ichnaea.sh --verbose

# the build-models:request task
./target/ichnaea-amqp.jar build-models:request \
    --aging=../r/fixtures/aging/env%column%-%aging%.txt \
    --dataset=../r/fixtures/cyprus.csv \
    --output=/tmp/cyprus_models.zip

# the predict-models:process task
./target/ichnaea-amqp.jar predict-models:process \
    -i ../r/files/ichnaea.sh --verbose

# the predict-models:request task
./target/ichnaea-amqp.jar predict-models:request \
    --data=../r/fixtures/cyprus_models.zip \
    --dataset=../r/fixtures/cyprus_test.csv \
    --output=/tmp/cyprus_result.txt
```

6.2 AMQP 1.0 support

To test AMQP 1.0 we will need the latest Wireshark version.

```
sudo apt-get remove wireshark
sudo add-apt-repository ppa:dreibh/ppa
sudo apt-get update
sudo apt-get install wireshark
```

We also will need to enable the AMQP 1.0 plugin on the RabbitMQ server.

```
sudo rabbitmq-plugins enable rabbitmq_amqp1_0
sudo service rabbitmq-server restart
```

For the clients to use the AMQP 1.0 protocol, we will have to specify connection type `Qpid`, that way they will use Apache Qpid Messenger instead of the RabbitMQ client library.

```
# the fake:process task
./target/ichnaea-amqp.jar fake:process \
    -i ../r/files/ichnaea.sh --connection-type=Qpid

# the fake:request task
./target/ichnaea-amqp.jar fake:request \
    --duration=10 --interval=1 --connection-type=Qpid
```

6.3 Developer's guide

To work on the Java source code you will need to install eclipse IDE.

```
sudo apt-get install eclipse
```

Run eclipse and select the `Help -> Check for updates` menu to update it to the latest version. Install the `m2eclipse` plugin for maven integration adding the URL `http://download.eclipse.org/technology/m2e/releases` to the window that appears after clicking the `Help -> Install new software` menu. Create a new empty workspace and enter the following commands to create the `ichnaea-amqp` project.

```
# add maven repo to eclipse
mvn -Declipse.workspace=path/to/eclipse/workspace eclipse:configure
    -workspace

# create eclipse project file
mvn eclipse:eclipse
```

Import the `ichnaea-amqp` project from the `Import... -> Existing projects into Workspace` menu. When developing, you can run the following maven commands to compile and run the code in one step.

```
# run the unit tests
mvn test
```

```
# listen to new build-models requests
mvn exec:java -Dexec.args="build-models:process -i ../r/files/
    ichnaea.sh"

# issue a fake test build-models request
mvn exec:java -Dexec.args="build-models:request --fake 10:1"

# debug a build-models request
mvn exec:java -Dexec.args="build-models:request --debug \
    --dataset ../r/fixtures/cyprus.csv \
    --aging ../r/fixtures/aging/env%column%-%aging%.txt"
```

6.4 PHP setup

The PHP library can be found in the `amqp/php` folder. The following commands should be run from there.

6.4.1 Install composer

Composer is a tool to manage PHP dependencies. To use the library you need to install it and use it to update the dependent libraries.

```
# install composer
curl -sS https://getcomposer.org/installer | php

# install dependencies
php composer.phar install --no-dev
```

Once the dependencies are installed, you can run the unit tests to confirm the library is working fine.

```
./vendor/phpunit/phpunit/phpunit.php ./src/
```

6.4.2 Create a RabbitMQ user

The Java library does not need a RabbitMQ user, but the PHP one does. To create a test user with all the permissions do the following

```
rabbitmqctl add_user test test
rabbitmqctl add_vhost /
rabbitmqctl set_permissions -p / test ".*" ".*" ".*"
```

6.4.3 Usage

This is an example of a website sending a `BuildModelsRequest`. The dataset can be any one of the following: a string with the CSV content, a `\SplFileObject` file handler or a `\SplFileInfo` object.

```
use Ichnaea\Amqp\Connection;
use Ichnaea\Amqp\Model\BuildModelsRequest;

$conn = new Connection("test:test@localhost");
$conn->open();
$req = BuildModelsRequest::fromArray(array(
    'id'        => 'test'
    'dataset'   => "csv_data"
));
$conn->send($req);
$conn->close();
```

Listening to responses has always to be done in a separate process as it will block execution.

```
$conn = new AmqpConnection("test:test@localhost");
$conn->open();

$amqp->listenForBuildModelResponse(function(BuildModelsResponse
    $resp) use ($db) {
    print "Received build-models response ".$resp->getId().
        " ".intval($resp->getProgress()*100)."%\n";
    $data = $resp->toArray();
    // save data
});
```

```
$amqp->listenForProcessModelResponse(function(ProcessModelsResponse
     $resp) use ($db) {
    print "Received process-models response ".$resp->getId().
        " ".intval($resp->getProgress()*100)."%\n";
    $data = $resp->toArray();
    // save data
});

$amqp->wait();
```

6.4.4 Test app

In the `php/app` folder there is a test app, to install it do the following. The test application uses Sqlite with the database in a file in the `php/app/db` folder, so please make sure that the apache user has write permissions to that folder.

```
# install app dependencies
php composer.phar install
# build database
php ./bin/build_database.php
```

Change `config.php` to point to a running AMQP server, the default is the same host the website is running on. Configure the web server to point to the `php/app/www` folder. Run the three Ichnaea Java process clients.

```
./target/ichnaea-amqp.jar fake:process \
    -i ../r/files/ichnaea.sh --verbose
./target/ichnaea-amqp.jar build-models:process \
    -i ../r/files/ichnaea.sh --verbose
./target/ichnaea-amqp.jar predict-models:process \
    -i ../r/files/ichnaea.sh --verbose
```

Run the PHP response process consumer. This will listen to responses and update the database. That way the changes will show up in the application.

```
php ./bin/consumer.php
```

Load the website and send a fake request. The fake process client should start printing out the bash script output and the PHP consumer should start receiving responses. The listing on the bottom of the website should start showing a progress bar.

6.4.5 Possible problems

To see in real-time what the RabbitMQ server is doing, you can enable the management plugin by entering the following commands.

```
sudo rabbitmq-plugins enable rabbitmq_management
sudo service rabbitmq-server restart
```

After that, open the web browser at `http://localhost:55672` and enter with user `guest` password `guest`. There you will be able to see edit active exchanges and queues.

By default the server will block connections[1] if it does not have enough memory or disk space. You can change those settings by editing the `/etc/rabbitmq/rabbitmq.config` file.

```
[
    {rabbit, [
      {disk_free_limit, {mem_relative, 0.1}},
      {vm_memory_high_watermark, 0.9}
    ]}
].
```

[1] see http://stackoverflow.com/questions/10427028/rabbitmq-connection-in-blocking-state

6.5 Ichnaea XML Model Examples

These are examples of valid XML representations used to send Ichnaea requests and responses between the AMQP clients.

Listing 6.1: Dataset

```
<dataset>
  <column name="test"><value>1</value><value>2</value><value>3</
      value></column>
  <column name="test2"><value>3</value><value>4</value></column>
  <column name="test3"><value>5</value><value>6</value><value>7</
      value></column>
</dataset>
```

Listing 6.2: Aging

```
<aging>
  <trial>
    <value key="0">4.4</value>
    <value key="10">40</value>
    <value key="50">120.6</value>
  </trial>
  <trial>
    <value key="0">5.4</value>
    <value key="10">43</value>
    <value key="50">123.6</value>
  </trial>
</aging>
```

Listing 6.3: Dataset Aging

```
<agings>
  <column name="test">
    <aging><!-- aging content --></aging>
  </column>
  <column name="test2">
    <aging><!-- aging content --></aging>
  </column>
  <column name="test3">
    <aging><!-- aging content --></aging>
  </column>
</agings>
```

Listing 6.4: Fake Request

```
<request id="431" type="fake" duration="10" interval="1" >
</request>
```

Listing 6.5: Fake Response

```
<response id="431" type="fake" progress="1.0"
  start="2012-12-27T08:00:00.000+0100"  end="2014-12-28T22
      :00:00.000+0100" />
```

Listing 6.6: Build Models Request

```
<request id="432" type="build_models">
  <dataset><!-- dataset content --></dataset>
</request>
```

Listing 6.7: Build Models Response

```
--frontier\n";
Content-Type: text/xml

<response id="432" type="build_models" progress="1.0"
  start="2012-12-27T08:00:00.000+0100"
  end="2014-12-28T22:00:00.000+0100" />
--frontier
Content-Type: application/zip
Content-Transfer-Encoding: base64

cGFjbw==
--frontier--
```

Listing 6.8: Predict Models Request

```
--frontier
Content-Type: text/xml

<request id="433" type="predict_models">
  <dataset><!-- dataset content --></dataset>
</request>
```

```
--frontier
Content-Type: application/zip
Content-Transfer-Encoding: base64

cGFjbw==
--frontier--
```

Listing 6.9: Predict Models Response

```
<response id="433" progress="1.0" type="predict_models"
  start="2012-12-27T08:00:00.000+0100"
  end="2014-12-28T22:00:00.000+0100" >
  <result name="ERROR2" predictedSamples="1" testError="0.0"
      totalSamples="1">
    <dataset><!-- dataset content --></dataset>
    <confusionMatrix>
      <column name="-1">
        <value>0</value>";
        <value>0</value>
      </column>
      <column name="1">
        <value>0</value>
        <value>1</value>
      </column>
    </confusionMatrix>
  </result>
</response>
```

6.6 AMQP 0-9-1 Tables

6.6.1 Class IDs

Class	ID	Description
`Connection`	10	work with socket connections
`Channel`	20	work with channels
`Exchange`	40	work with exchanges

Class	ID	Description
`Queue`	50	work with queues
`Basic`	60	work with basic content
`Tx`	90	work with transactions

6.6.2 Method IDs

Method	ID	Description
`Connection.Start`	10	start connection negotiation
`Connection.Start-Ok`	11	select security mechanism and locale
`Connection.Secure`	20	security mechanism challenge
`Connection.Secure-Ok`	21	security mechanism response
`Connection.Tune`	30	propose connection tuning parameters
`Connection.Tune-Ok`	31	negotiate connection tuning parameters
`Connection.Open`	40	open connection to virtual host
`Connection.Open-Ok`	41	signal that connection is ready
`Connection.Close`	50	request a connection close
`Connection.Close-Ok`	51	confirm a connection close
`Channel.Open`	10	open a channel for use
`Channel.Open-Ok`	11	signal that the channel is ready
`Channel.Flow`	20	enable/disable flow from peer
`Channel.Flow-Ok`	21	confirm a flow method
`Channel.Close`	40	request a channel close
`Channel.Close-Ok`	41	confirm a channel close

Method	ID	Description
`Exchange.Declare`	10	verify exchange exists, create if needed
`Exchange.Declare-Ok`	11	confirm exchange declaration
`Exchange.Delete`	20	delete an exchange
`Exchange.Delete-Ok`	21	confirm deletion of an exchange
`Queue.Declare`	10	declare queue, create if needed
`Queue.Declare-Ok`	11	confirms a queue definition
`Queue.Bind`	20	bind queue to an exchange
`Queue.Bind-Ok`	21	confirm bind successful
`Queue.Unbind`	50	unbind a queue from an exchange
`Queue.Unbind-Ok`	51	confirm unbind successful
`Queue.Purge`	30	purge a queue
`Queue.Purge-Ok`	31	confirms a queue purge
`Queue.Delete`	40	delete a queue
`Queue.Delete-Ok`	41	confirm deletion of a queue
`Basic.Qos`	10	specify quality of service
`Basic.Qos-Ok`	11	confirm the requested qos
`Basic.Consume`	20	start a queue consumer
`Basic.Consume-Ok`	21	confirm a new consumer
`Basic.Cancel`	30	end a queue consumer
`Basic.Cancel-Ok`	31	confirm a canceled consumer
`Basic.Publish`	40	publish a message
`Basic.Return`	50	return a failed message
`Basic.Deliver`	60	notify the client of a consumer message
`Basic.Get`	70	direct access to a queue

Method	ID	Description
`Basic.Get-Ok`	71	provide client with a message
`Basic.Get-Empty`	72	indicate no messages available
`Basic.Ack`	80	acknowledge one or more messages
`Basic.Reject`	90	reject an incoming message
`Basic.Recover-Async`	100	redeliver unacknowledged messages
`Basic.Recover`	110	redeliver unacknowledged messages
`Basic.Recover-Ok`	111	confirm recovery
`Tx.Select`	10	select standard transaction mode
`Tx.Select-Ok`	11	confirm transaction mode
`Tx.Commit`	20	commit the current transaction
`Tx.Commit-Ok`	21	confirm a successful commit
`Tx.Rollback`	30	abandon the current transaction
`Tx.Rollback-Ok`	31	confirm successful rollback

6.6.3 Data types

0-9	0-9-1	Type
	`t`	Boolean
	`b`	Signed 8-bit
	`B`	Unsigned 8-bit
	`U`	Signed 16-bit
	`u`	Unsigned 16-bit
`I`	`I`	Signed 32-bit
	`i`	Unsigned 32-bit
	`L`	Signed 64-bit

0-9	0-9-1	Type
	`l`	Unsigned 64-bit
	`f`	32-bit float
	`d`	64-bit float
`D`	`D`	Decimal
	`s`	Short string
`S`	`S`	Long string
	`A`	Nested Array
`T`	`T`	Timestamp (u64)
`F`	`F`	Nested Table
`V`	`V`	Void

6.7 AMQP 1.0 Tables

6.7.1 Data type categories

Subcategory	Category	Format
0x4	Fixed Width	Zero octets of data.
0x5	Fixed Width	One octet of data.
0x6	Fixed Width	Two octets of data.
0x7	Fixed Width	Four octets of data.
0x8	Fixed Width	Eight octets of data.
0x9	Fixed Width	Sixteen octets of data.
0xA	Variable Width	One octet of size, 0-255 octets of data.
0xB	Variable Width	Four octets of size, 0-4294967295 octets of data.

Subcategory	Category	Format
0xC	Compound	One octet each of size and count, 0-255 distinctly typed values.
0xD	Compound	Four octets each of size and count, 0-4294967295 distinctly typed values.
0xE	Array	One octet each of size and count, 0-255 uniformly typed values.
0xF	Array	Four octets each of size and count, 0-4294967295 uniformly typed values.

6.7.2 Data types

Type	Encoding	Code	Category	Description
null		0x40	fixed/0	the `null` value
boolean		0x56	fixed/1	boolean with the octet 0x00 being false and octet 0x01 being true
boolean	true	0x41	fixed/0	the boolean value `true`
boolean	false	0x42	fixed/0	the boolean value `false`
ubyte		0x50	fixed/1	8-bit unsigned integer
ushort		0x60	fixed/2	16-bit unsigned integer in network byte order
uint		0x70	fixed/4	32-bit unsigned integer in network byte order
uint	smalluint	0x52	fixed/1	unsigned integer value in the range 0 to 255 inclusive
uint	uint0	0x43	fixed/0	the uint value 0
ulong		0x80	fixed/8	64-bit unsigned integer in network byte order
ulong	smallulong	0x53	fixed/1	unsigned long value in the range 0 to 255 inclusive

Type	Encoding	Code	Category	Description
ulong	ulong0	0x44	fixed/0	the ulong value 0
byte		0x51	fixed/1	8-bit two's-complement integer
short		0x61	fixed/2	16-bit two's-complement integer in network byte order
int		0x71	fixed/4	32-bit two's-complement integer in network byte order
int	smallint	0x54	fixed/1	signed integer value in the range -128 to 127 inclusive
long		0x81	fixed/8	64-bit two's-complement integer in network byte order
long	smalllong	0x55	fixed/1	signed long value in the range -128 to 127 inclusive
float	ieee-754	0x72	fixed/4	IEEE 754-2008 binary32
double	ieee-754	0x82	fixed/8	IEEE 754-2008 binary64
decimal32	ieee-754	0x74	fixed/4	IEEE 754-2008 decimal32 using the Binary Integer Decimal encoding
decimal64	ieee-754	0x84	fixed/8	IEEE 754-2008 decimal64 using the Binary Integer Decimal encoding
decimal128	ieee-754	0x94	fixed/16	IEEE 754-2008 decimal128 using the Binary Integer Decimal encoding
char	utf32	0x73	fixed/1	a UTF-32BE encoded unicode character
timestamp	ms64	0x83	fixed/8	64-bit signed integer representing milliseconds since the unix epoch
uuid		0x98	fixed/16	UUID as defined in section 4.1.2 of RFC-4122
binary	vbin8	0xa0	variable/1	up to $2^8 - 1$ octets of binary data
binary	vbin32	0xb0	variable/4	up to $2^3 2 - 1$ octets of binary data
string	str8-utf8	0xa1	variable/1	up to $2^8 - 1$ octets worth of UTF-8 unicode (with no byte order mark)

Type	Encoding	Code	Category	Description
string	str32-utf8	0xb1	variable/4	up to $2^32 - 1$ octets worth of UTF-8 unicode (with no byte order mark)
symbol	sym8	0xa3	variable/1	up to $2^8 - 1$ seven bit ASCII characters representing a symbolic value
symbol	sym32	0xb3	variable/4	up to $2^32 - 1$ seven bit ASCII characters representing a symbolic value
list	list0	0x45	fixed/0	the empty list (i.e. the list with no elements)
list	list8	0xc0	compound/1	up to $2^8 - 1$ list elements with total size less than 2 8 octets
list	list32	0xd0	compound/4	up to $2^32 - 1$ list elements with total size less than 2 32 octets
map	map8	0xc1	compound/1	up to $2^8 - 1$ octets of encoded map data
map	map32	0xd1	compound/4	up to $2^32 - 1$ octets of encoded map data
array	array8	0xe0	array/1	up to $2^8 - 1$ array elements with total size less than 2 8 octets
array	array32	0xf0	array/4	up to $2^32 - 1$ array elements with total size less than 2 32 octets

6.8 Bibliography

"Advanced Message Queuing Protocol." 2011. *1.0 Specification*. October 7. http://www.amqp.org/specification/1.0/amqp-org-download.

Aiyagari, Sanjay, and others. 2008. "Advanced Message Queuing Protocol." *0-9-1 Specification*. November 13. http://www.amqp.org/specification/0-9-1/amqp-org-download.

"AMQP 0-9-1 Model Explained." http://www.amqp.org/specification/1.0/amqp-org-download.

O'Hara, John, JP Morgan. 2007. "Toward a Commodity Enterprise Middleware." *Queue - API Design*, May.

Pèrez Pèrez, Aitor. 2014. "ICHNAEA 2.0: a Software for Microbiology Modeling." http://hdl.handle.net/2099.1/20697.

"Php-Amqplib Documentation." https://github.com/videlalvaro/php-amqplib.

"RabbitMQ Documentation." https://www.rabbitmq.com/documentation.html.

Sànchez, David. 2012. "A Software System for the Microbial Source Tracking Problem." http://hdl.handle.net/2099.1/16066.

Velazco Sanchez, Nahuel. 2014. "Aplicació I Serveis Web Per Ichnaea Software." http://hdl.handle.net/2099.1/21944.