# Renewable Energy Production Distribution Map of Catalan Homes

by

Pau Pérez Fabregat

Jun 18, 2014

Thesis Supervisor
Carles Farré Tost
Department of Service and Information System Engineering. ESSI

Informatics Engineering

Universtiat Politècnica de Catalunya (UPC)
BarcelonaTech

Facultat d'Informàtica de Barcelona (FIB)

# Renewable Energy Production Distribution Map of Catalan Homes

by

Pau Pérez Fabregat

Department of Service and Information System Engineering. ESSI
on Jun 18, 2014
Informatics Engineering

## Abstract

Recent development in web technology and infrastructure services together with enhancements in microcontrollers and hardware devices enable the implementation of cheaper IT systems. This enable research centers to build powerful and affordable infrastructures that can ease their work. This is particularly for The Center for Ecological Research and Forestry Applications (CREAF) which is geared towards the creation of new methodological tools in the field of the terrestrial ecology. The upcoming idea of the sensor web – led by the Open Geospatial Consortium (OGC) – offers a new way to obtain data on a more interoperable basis.

The aim of this thesis is to implement a first prototype of a larger project whose goal is to provide a system that enables monitoring the distribution of renewable energy produced in Catalan homes in real-time. A thorough research evaluates the available technologies and lays the foundation of the further development of the project. Through an asynchronous messaging queue the system provides a loosely coupled architecture that enables its scalability. A simple single-page web application offers a real-time data visualization of the data generated by sensor simulators which allow the evaluation of the system while the physical sensor devices are not implemented yet.

# Contents

# Chapter 1

# Introduction

The Center for Ecological Research and Forestry Applications (CREAF) is a public research institution that was created in 1987. The members of the Governing Council of CREAF are the Generalitat of Catalonia, the Autonomous University of Barcelona (UAB), University of Barcelona (UB), the Institute for Research and Technology (IRTA), the Institute of Catalan Studies (IEC) and the Spanish National Research Council (CSIC).

Its objective is to generate knowledge and create new methodological tools in the field of terrestrial ecology in order to improve environmental planning and management in rural and urban areas with special emphasis on forest ecology. This is achieved, among other means, through the development of methodological and conceptual tools designed to facilitate decision-making and improve environmental management.

Since its creation, CREAF has made very important contributions to the field of terrestrial ecology and towards a sustainable management of the environment. This has been achieved through research, development, training and technology transfer. Some of its most relevant contributions include the design and implementation of the Ecological and Forest Inventory of Catalonia (EFIC), innovative at the international level due to the incorporation of new ecological parameters, the production of the Land Cover Map of Catalonia (MCSC), a high-resolution digital map for environmental assessment and territorial planning and management and the development of

the MiraMon©Geographic Information System, widely adopted in Catalan administration and currently used in over thirty countries around the world.

## 1.1 Motivations

The motivation for this master's thesis stems from the idea that the wide development of web technologies, DevOps and infrastructure services make possible the implementation of powerful systems with significant cuts on budget and maintenance costs. Moreover, these technologies often entail considerable improvements in maintainability, performance and reduced complexity.

I am firmly convinced that these technologies and tools can improve research in centres such as CREAF, providing them with better and affordable infrastructures reducing the timespan of common processes. Furthermore, they provide new means for the dissemination of the resulting data.

Nonetheless, I think that computer engineering should be conceived as a tool to push forward the development of other sciences. As recently graduated engineers we can contribute back to society with our knowledge in an attempt to solve problems that benefit us all. Hence, I wish to develop a project in which the outcome could improve the work of public research centres thus making it a useful tool.

Given the interest aroused in topics like distributed computing, sensor networks and resilience systems during in my recent stay in University of Antwerp, I am eager to expand my knowledge further and apply them in a real-world use case.

Within the context of volunteered geographic information (VGI) and renewable energies, CREAF wants to solve the problem of ascertaining the distribution of renewable energy produced in Catalan homes. Nowadays, its performance, time evolution and distribution is unknown thus complicating the decision-making process regarding renewable energy sources in Catalonia.

On the other hand, CREAF wishes to expand its methodological tools by adopting sensor web. The reduced cost of hardware devices like Raspberry Pi and Arduino and their general-purpose features make them an affordable and versatile solution as sensor

devices. Their considerable computational power also facilitates the development of service clients in widely adopted languages. For these reasons, CREAF plans to deploy its own sensor devices in natural surroundings in the near future.

Although CREAF already took some design decisions, the architecture of the system and the software the devices would be shipped with were still to be determined.

Given the mutual interest in the project outlined by CREAF, we set out to design and implement a prototype as a first working solution.

## 1.2 Project Goals

The Renewable Energy Production Distribution Map of Catalan Homes (REDCH) is aimed at developing a system that offers features to registered users who freely share their data as well as other publicly available features. It will visualize the energy production of the clients system and its contribution to the whole Catalan renewable energy production in a real time map, while offering a private analytics dashboard to registered users where they can figure out the actual performance of their system.

Given the extent of the desired product with this thesis we wish to develop a proof of concept – a distributed computing system that will provide essential features, being a simple but functional prototype of the final product. Once built, the system results and metrics will be evaluated and its architecture may eventually become the standard infrastructure basis for future CREAF projects that demand sensor data. As a consequence, all features for registered users are out of the scope.

To sum up, these two general objectives are translated into the following specific goals:

- Provide a command-line interface which allows for the simulation of sensor functionality
- Develop a functional system that stores and processes sensor observations
- Implement a simple public web application to display the observations in a real time map

## 1.3 Methodology

### 1.3.1 Iterative Development

Although advocating agile software methodologies, the concept-proof nature of the thesis – which is developed by just one person – make them an unsuitable choice. We opt instead for a custom adaptation of iterative development.

In conjunction with incremental development, Iterative development is a way of breaking down the development of a software system into smaller chunks and repeated cycles. In each cycle, known as iteration, the slice of functionality is designed, developed, tested, deployed and evaluated. This allows software developers to apply the knowledge acquired in previous iterations, so the first implementation whose goal is to build a bare minimal functional system is iteratively enhanced so as to meet the requirements.

Nevertheless, iterative and incremental development are the basis for Agile Development. Therefore, by adhering to these two practices, we attempt to avoid the agile practices and constraints that may be pointless in this case. Doing so, we aim to progressively enhance the codebase in subsequent iterations, gain insight into the architecture and improve any weak points we may identify until eventually meeting the requirements. Additionally, early results can be achieved and evaluated by CREAF resulting in a smoother collaboration.

### 1.3.2 Test-Driven Development

The chosen methodology also includes Test-Driven Development (TDD), which is a developer practice that involves writing tests before writing the code to be tested. The initially failed test defines the behaviour of the code to be written, then the developer writes the minimum amount of code required to pass the test. Once it passes, it is time to refactor it to remove any duplication. This cycle must be repeated as many times as required to further extend the responsibilities of the code.

Besides validating the correctness of the code, by running the design through test

cases the developer is mainly concerned with the interface of the program rather than its actual implementation.

What we aim for by using TDD in this project is to obtain a more modularized, maintainable, and extensible code. The development of the software in small units leads to smaller, more focused and loosely coupled classes and cleaner interfaces. The main benefit we may get by this means, however, is a greater level of confidence in the code caused by the fact that all code written is covered by at least one test.

Additionally, in this early stage of the project to have a test-covered code is basic practice for the successful evolution of the project. So it can be ensured that the intended behaviour is kept and any defects are caught early in the development process and therefore has a considerably less impact on costs than in later stages.

# Chapter 2

# Analysis

## 2.1 Stakeholders

There are four distinguished groups of stakeholders which will be affected by the outcome of this project. These are summarized as follows.

**Users** These are the clients of the product and producers of the data that feeds the system. They will provide the data gathered from their solar panels or wind mills to the system. To that end, they are responsible for keeping the sensor device working under the required conditions.

**CREAF** This research center is the client of the project and the product owner. The staff in charge of the project is responsible for ensuring that the software and hardware of the sensor devices is updated as needed and for the maintenance of the system. In addition, CREAF must also provide and approve any required funding and infrastructure, as well as monitor the progress of the project.

**Decision makers** Politicians and any person responsible for decisions that may affect the future of renewable energies in Catalonia. They will use the system as a basis for decisions concerning renewable energy production in Catalonia.

**Green activists** They act as a pressure group disseminating the system in their regular campaigns.

## 2.2 Constraints

### 2.2.1 Schedule Constraints

**Description** The project shall be finished by June 2014.

**Rationale** This project will be delivered as a MS Thesis and must be completed before its presentation at the end of June 2014.

### 2.2.2 Budget Constraints

**Development Constraint**

**Description** The project shall be developed by one engineer.

**Rationale** Since there is no budget assigned to the project, it must be developed by the author of this MS Thesis within the time specified in 2.2.1.

**Infrastructure Constraint**

**Description** The system's prototype must not involve any cost.

**Rationale** Due to the lack of budget the project must opt for free services and solutions.

## 2.3 Scope of the Product

As already outlined in 1.2 this MS Thesis is part of a larger project. Given the constraints enumerated in 2.2 the said MS Thesis aims to build the foundation for the further development of the project, focusing on its core features. By doing so, we will be able to draw conclusions and plan the further development of the project

accordingly. Thus, aspects such as the hardware of the sensor device, its distribution among the users or the building of the complete web application fall beyond the scope of this MS Thesis. Therefore, the system will be made up of the following three parts.

Firstly, a sensor simulator that will allow to use the system as if the sensor were already developed. To that end, the simulator will provide a Command Line Interface (CLI) to allow seamless interaction with the system. It will cover the essential use cases for the system to work leaving others out of the scope.

Secondly, a distributed system will process and store the observations generated from the simulators. Eventually, the same system will deal with real observations from the sensor devices. Additionally, it will provide a web interface to manage the sensors and observations as well as the system's settings.

Finally, a simple publicly accessible web application will display these observations in a real-time map. This application will enable to explore the possibilities of a complete Software-as-a-Service (SaaS) by providing insight into the complexities and requirements of building such service.

The scope of the project is formalized through the following features:

- Register simulators as sensors in the system
- Store observations generated from the simulator
- Change the data storage configuration
- Query the stored data
- Clear the stored data
- Show observations in real-time on a map

## 2.4 Requirements

These requirements have been obtained by means of some meetings with the CREAF researcher in charge of the project.

Even though the terms *sensor* and *simulator* may be used interchangeably as explained above, we use the former in the requirements below to keep consistency throughout this Thesis.

### 2.4.1 Functional Requirements

**Requirement** **Insert Sensor**

*Description:* The system shall register the sensors that interact with it.

*Rationale:* The observations within the system must be related to the producer sensor to know the location of the phenomenon, and so the sensors must be registered beforehand.

**Requirement** **Insert Observation**

*Description:* The system shall store the observation measured by the sensors.

*Rationale:* The sensor observations must be stored into the system in order to be queried.

**Requirement** **Update data storage configuration**

*Description:* The system shall enable changes in data storage settings.

*Rationale:* The system must be independent of the parameters of the underlying data storage and allow administrators to update it.

**Requirement Query data store**

*Description:* The system shall allow queries over the stored data.

*Rationale:* The system must allow administrators to query all the raw data including the observations.

**Requirement Clear data store**

*Description:* The system shall allow clearance of all data stored.

*Rationale:* The system must allow administrators to clean up all data stored in the system for maintenance purposes.

**Requirement Data visualization**

*Description:* The system shall enable browse observations in a real-time data visualization.

*Rationale:* Users must be able to browse the location of the observations on a map in real time.

### 2.4.2 Non-functional Requirements

**Usability**

**Requirement** **User-friendly**

*Description:* The web application shall be easy to use by final users.

*Rationale:* The users must be able to use the web application by means of the User Interface (UI) without prior learning.

**Requirement** **Configurable CLI**

*Description:* The simulator's CLI shall enable configuration of all its parameters.

*Rationale:* The simulator's users must be able to choose the service operation parameters in order to simulate the sensors' behaviour in different conditions.

**Performance**

**Requirement** **Observations per hour**

*Description:* The sensor's simulators shall send at least 6 observations per hour.

*Rationale:* The simulators must be able to send at least 6 observation requests per hour to the system as this amount is likely to be changed.

**Requirement** **Real-time**

*Description:* The system shall display a sensor's observation before its next one is received.

*Rationale:* Users must be able to see the observations in real-time. Hence, the time to process and show an observation must be lower than the period between receipt of observations.

**Requirement** **Concurrency**

*Description:* The system shall be reliable processing observations from at least 10 sensors.

*Rationale:* Although budget constraints do not allow to use a full-featured infrastructure the system must be able to deal with reasonable concurrency.

**Requirement Scalability**

*Description:* The system shall be scalable.

*Rationale:* It must be easy to scale the system in order to handle higher loads with more sensors, more users or both.

### Interfaces to External Systems

**Requirement Interoperability**

*Description:* The system shall conform to OGC Sensor Observation Service (SOS).

*Rationale:* The system must offer its data using SOS in order to be interoperable from other independent systems.

### Compliance

**Requirement Licensing**

*Description:* The system and all its components shall adhere to Apache License.

*Rationale:* The system and all components of the final solution must adhere to Apache License to ensure that all software is open-source.

# Chapter 3

# Specification

The following section describes what the system does by detailing its entities.

## 3.1 Use Case Model

This section describes the operations of the systems as events triggered by external actors and their interrelation.

### 3.1.1 Actors

The actors of the system are the following.

**Sensor device** The device responsible for registering itself in the system and sending the measured observations to it.

**User** A person who interacts with the public web application that shows the observations in a data visualization.

### 3.1.2 Use Cases

**Use Case 1** **Insert Sensor**

*Actors:* Sensor device

*Preconditions:* The system is running

*Postconditions:* The sensor is registered and persisted in the system

*Main Success Scenario:*

1. The sensor sends a request to register itself
2. The system stores the information of the sensor in the database
3. The system notifies the sensor when it has been successfully registered

*Extensions:*

1.a The sensor is already registered

    1. The system returns an error response

**Use Case 2** **Insert Observation**

*Actors:* Sensor device

*Preconditions:*

- The system is running
- The sensor is registered in the system

*Postconditions:*

- The observation is persisted in the system
- The observation is sent to the web application tier

*Main Success Scenario:*

1. The sensor sends a request to store the observation
2. The system stores the observation data in the database
3. The system sends the observation data to the web application tier
4. The system notifies the sensor when the observation has been successfully stored

*Extensions:*

1.a The sensor specified in the request is not found

    1. The system returns an error message

**Use Case 3** **Browse data**

*Actors:* User

*Preconditions:* The system is running

*Postconditions:* The web application is shown

*Main Success Scenario:*

1. The user's browser loads the web application

2. Once loaded, the web application establishes a connection against the system

3. The system incorporates observations into the web application as they are available

## 3.2 Conceptual Model

As shown in figure 3-1, the system revolves around the concepts of *Observation* and *Sensor*. Diagram 3-2 describes these concepts and other entities involved in the system as well as their relations, heavily based on the OGC O&M model [12].

To start with, an observation is an aggregation of the following six elements:

**Feature of interest** A representation of a real-world object that carries the observed property, e.g. "Pantà de Sau". Hence, for an in-place instrument this would be the sensor location, whereas for a remote sensor it would be the target location.

**Procedure** Instance of a process which has performed the observation. Despite being usually a physical sensor, it can also be a process that leads to an observation such as a computation or the result of post-processing.

**Observed property** Represent the phenomena under observation. Usually a concept of a formal ontology, e.g. air temperature.

**Phenomenon time** Time when the phenomenon that produces the observation occurs.

**Result time** Time when the observation result has been created. Note that phenomenon and result times may be identical.

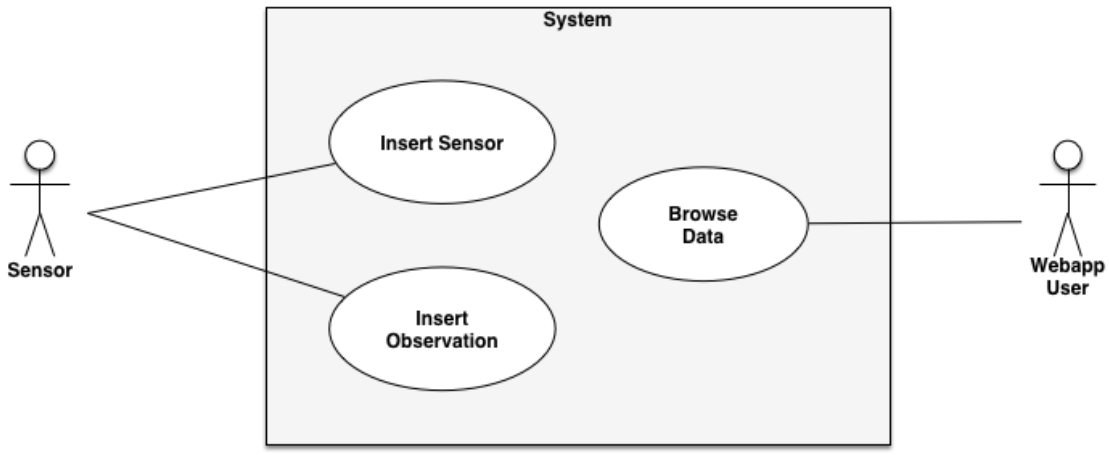



Figure 3-1: System use cases

**Result** Result of the observation, which can be either a scalar value or a complex multi-dimensional array.

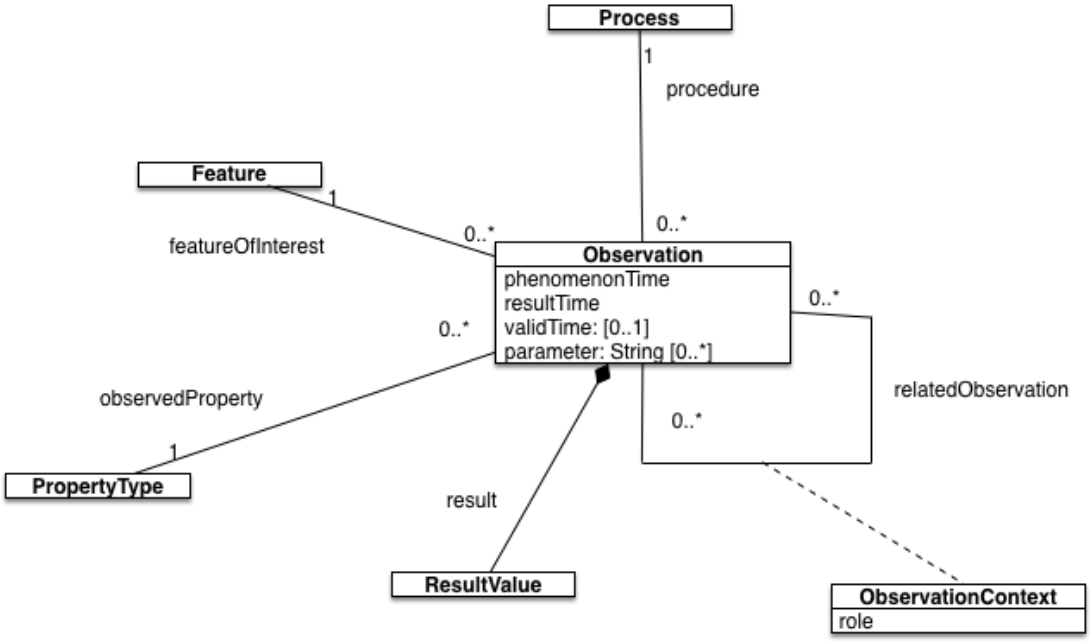



Figure 3-2: Conceptual model

## 3.3 Sequence Diagrams

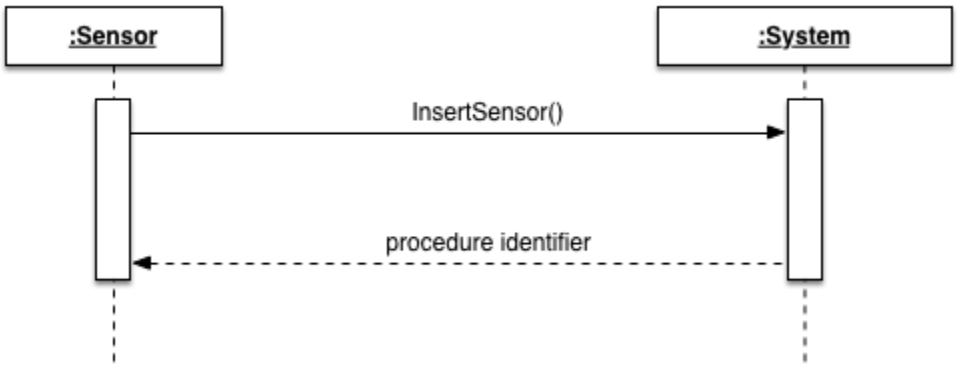



Figure 3-3: Sequence Diagram - Insert Sensor

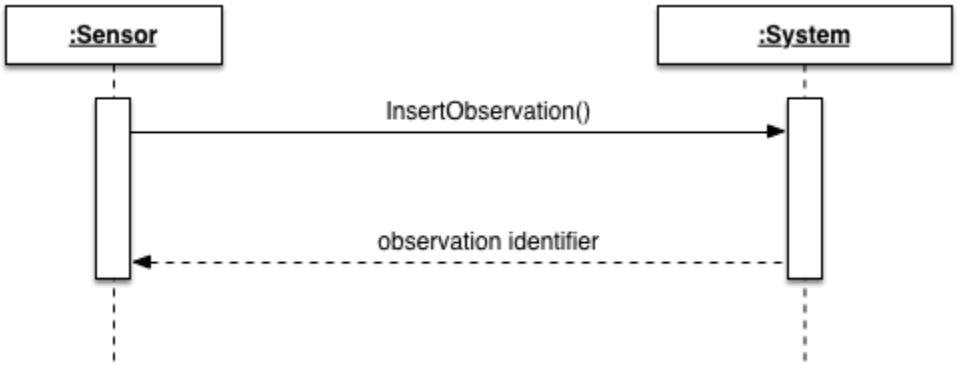



Figure 3-4: Sequence Diagram - Insert Observation

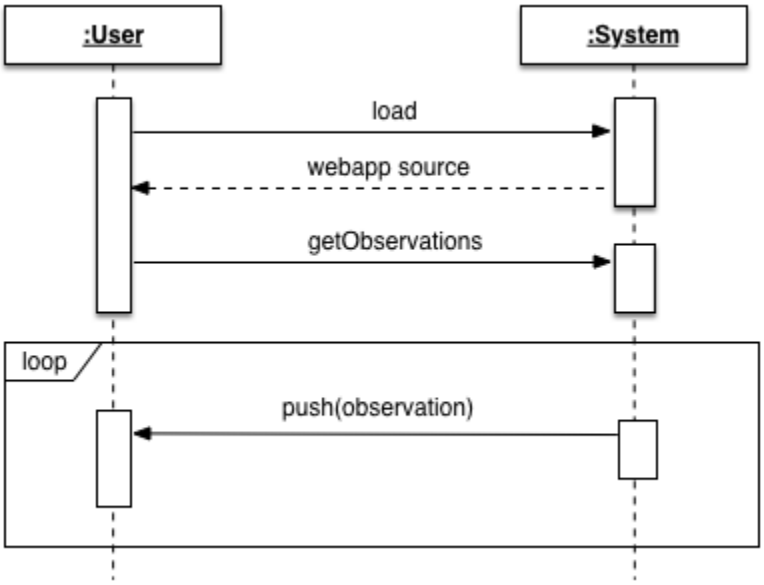



Figure 3-5: Sequence Diagram - Browse Data

# Chapter 4

# Technology research

This chapter aims to synthesize the research carried out over the main knowledge areas involved in the development of this project. It intends to give an overview of the technologies and paradigms considered to support the design decisions the project is based upon. Considering the high-level architecture diagram below, these are the main areas of concern initially identified: public interface, that is, the interaction between the sensors and the web server, database, real-time across the system and web technologies.

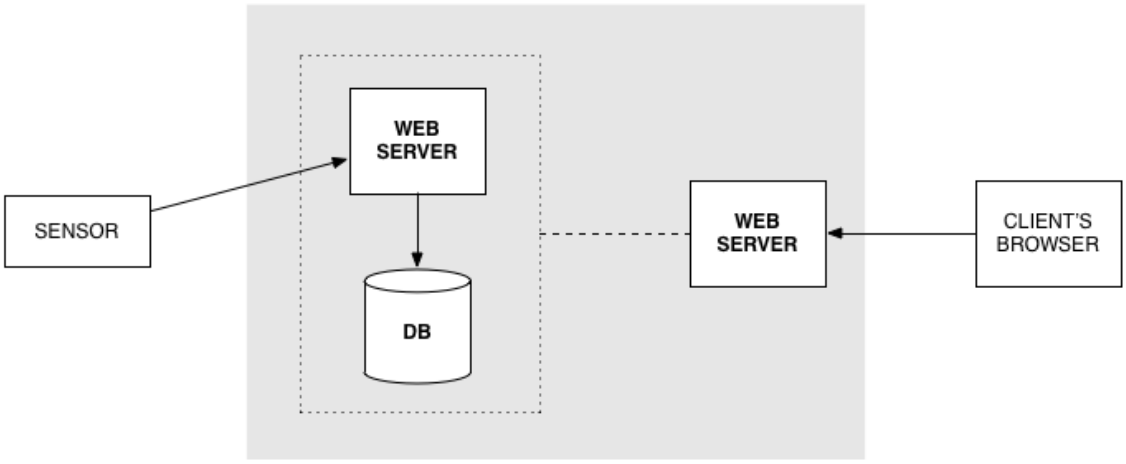



Figure 4-1: High-level required components

## 4.1 Public Interface

### 4.1.1 Client devices

In recent years, there has been a steady reduction in costs of hardware devices. Particularly, some microcontrollers have reached prices under 10$ such as ATmega168, which is the one used by Arduino. Based on a simple microcontroller board and its own development environment, Arduino can be used to develop a vast variety of interactive objects. Its CPU speeds ranging from 8 to 84 Mhz, USB and UART ports, many digital and analog I/O and Flash memory, make it a powerful physical computing device. Even though there are many other affordable microcontroller platforms, Arduino stands out due to its easy-to-use programming environment and cross-platform software. It is especially worth mentioning, however, that this is an open-source physical computing platform. Both its software and the plans of its modules are published under open source licenses.

On the other hand, the decrease in the price of processors for mobile devices with excellent multimedia capabilities led to the foundation of the Raspberry Pi Foundation and the public release of the first Raspberry Pi in 2012. This consists of a single-board computer aimed at teaching computer science basics. Unlike Arduino, it is shipped with 700 MHz ARM processors and as any other computer, it comes with GPU, video and audio outputs and SD storage, but only its Model B has 100 Mbits Ethernet connection. Although it supports some Linux kernel-based operating systems like Debian GNU/Linux and Arch Linux ARM, it is recommendable to run Raspbian, a Debian-based free operating system optimized for the Raspberry Pi hardware. These general purpose features and its credit-card size make it a capable computer which can be used in a wide range of scenarios replacing regular desktop PCs.

Both devices have different aims and capabilities. Arduino is an easy-to-use lower-level physical computing platform, whereas Raspberry Pi beats general purpose PCs in terms of cost and size. Nonetheless, it is not unusual to combine their features attaching them together as a single device, which [11] attempts. While Arduino brings

I/O capabilities that Raspberry Pi lacks, the latter provides computing power.

These features have contributed to their popularity among people involved in technology as well as computing aficionados. They have attracted great interest in the Internet of Things (IoT) community and have had direct impact on its recent growth. Some projects are heavily inspired by Arduino extensibility such as [19], whilst others build their products based on customized Arduino boards.

This is the case of Smart Citizen [16], a whole platform aimed at generating participatory processes of people in cities thus, creating more effective and optimized relationships between services, technology and communities in the urban environment. The core of the platform is the called Smart Citizen Kit, a hardware device shipped with air, temperature, light, sound and humidity sensors plus a Wi-Fi module to serve as an ambient sensor. They started with Arduino shields to develop a prototype until eventually coming up with their own specific-purpose Arduino-compatible data-processing board.

### 4.1.2 Interoperability

Interoperability is the software quality of enabling a system to interact with other systems without the need to write or maintain custom logic. This is often achieved using the same protocols, exchange or file formats, or by means of standardization.

Interoperability has a great impact on several fields such as financial or medical industries where inadequate implementations may lead to important economic costs. It also crucial in science since the outcomes of a research must be operable for others in order to progress towards a common goal. This also applies in the context of this project since the data obtained by the sensors and its underlying infrastructure may be used in other research projects of CREAF. Not less important is the role this project plays within CREAF's efforts towards the Sensor Web [14] as standardization group of the Open Geospatial Consortium (OGC). Therefore, interoperability is a main concern for REDCH.

The OGC's Observations & Measurements (O&M) [12] is the standard data model for storing and publishing sensor data. Based on the Geography Markup Language

(GML) OGC standard, it models the relationship between observation events, the spatial objects under observation, the measured properties and measurement procedure and the captured data resulting from the observations. O&M is one of the open standards developed in the Sensor Web Enablement (SWE) initiative of the OGC.

While O&M provides a system-independent way of sensor data exchange, the Sensor Observation Service (SOS), another SWE standard, defines a Web service interface for sensor data. This standard allows querying observations and sensor metadata, registering and removing sensors, as well as inserting new sensor observations. Furthermore, it defines KVP and SOAP bindings so as to be binding-independent. However, OGC does not provide an implementation but a service interface.

There are currently some open-source implementations of the SOS. The Earth Science Institute of the University of Applied Sciences of the South Switzerland set up istSOS [4] in 2009, an SOS implementation entirely written in Python that includes a RESTful API and a graphical user interface for easing the administration of the service.

52North is a network of partners from research, such as the University of Mnster and the Technische Universitt Dresden, industry, such ESRI Inc. and public administrations such as the IT department of the German Federal Ministry of Transport, Building and Urban Development. It is aimed at bringing innovation into the field of Geoinformatics. 52North SOS [1] is the leading implementation of the Sensor Observation Service. The latest version 4.0, recently released as of this writing, comes with full support of the SOS 2.0 specification. In addition, 52North has developed the SOS RESTful Extension. A SOS 4.0 Add-on that provides a REST binding beyond the standard KVP and SOAP defined by the OGC.

## 4.2 Database

The way the data is handled and stored is a key point of the project. Therefore, it is crucial to choose the Database management system (DBMS) that best fits the features of the underlying data set. It must implement some sort of geographic support as

every observation implicitly belongs to a particular geographic location.

Databases allow to persist a representation of real-world objects and their relations in a structured fashion. Furthermore, they also allow to integrate the data of different applications thus, avoiding data duplication.

Databases may be classified in three general models: hierarchical, network, relational and NoSQL. Relational databases have gained a lot of popularity since their appearance in the late 1970s, becoming the de facto choice regarding data management in IT systems. On the ohter hand, NoSQL systems is a field that has been quickly evolving very fast since its birth in the 2000s.

### 4.2.1 Relational DBMSs

The relational model is based on set theory and predicate logic. Relational databases implement an approximation of these mathematical models using a table-based format. The data is structured in tables that represent relations where the information of a particular entity is represented by a row and the set of fixed attributes of such entity correspond to the columns.

Databases, however, need DBMSs in order to be functional. A DBMS is an especially designed software which enables the creation, querying, update, and management of databases. It is a layer above the OS that abstracts the applications from the database. Hence, applications deal with databases through the DBMS.

Relational DBMSs essentially provide efficient, reliable, convenient, and safe multiuser storage of and access to massive amounts of persistent data. These guarantee consistency by means of robust concurrency models and ACID (Atomicity, Consistency, Isolation, Durability) transactions. Due to decades of development and research relational database systems are relied upon by mission-critical applications that demand strict consistency.

The most popular open-source relational DBMSs are MySQL and PostgreSQL, both with spatial extensions. PostGIS, however, is the most mature solution. It is a PostgreSQL extension that adds support for location awareness enabling queries by geographic location. In addition, PostGIS supports geographic coordinates. As

a consequence of PostGIS' rich feature list, PostgreSQL is the standard choice for Geographic Information Systems (GIS).

However, not every data management or analysis problem is best solved exclusively using a traditional DBMS. There are some problems that are more suitable for other type of systems [9].

### 4.2.2 NoSQL Systems

NoSQL Systems, whose name stands for *Not Only SQL*, differ from traditional relational systems in that they tend to provide a flexible schema rather than a rigid structure. They are also quicker and cheaper to set up geared towards massive scalability and use relaxed consistency in order to provide higher performance and higher availability.

Their downsides are that there is no declarative query language thus, more programming is involved in manipulating the data. Furthermore, because of the relaxed consistency models, their better performance comes at the expense of fewer guarantees about the consistency of the data. Eventually consistent systems are often classified as providing BASE (Basically Available, Soft state, Eventual consistency) properties in contrast to traditional ACID-compliant relational systems.

One of the main goals of NoSQL systems is to enhance horizontal scalability. As the CAP theorem states [8][3], this can only be achieved by relaxing either its consistency or its availability so partition tolerance may be guaranteed. There is no consensus among NoSQL vendors over which pair to choose. Some opt for consistency against availability, while others focus on availability over consistency [10]. Nonetheless, there are few that pick both properties and consequently provide scalability through replication rather than partitioning.

The number of different kinds of NoSQL systems may be generalized in four main categories: MapReduce frameworks, Key-value stores, Graph database systems and Document stores. MapReduce frameworks are typically used in applications that process large amounts of data to do complex analysis, whereas Key-value stores tend to perform a lot of small operations on very small parts of the data. On the other

hand, Graph database systems are designed for storing and operating over very large graphs.

**Document stores** Document stores are very similar to Key-value stores except that the values are documents. Hence, the data model is based on $< key, document >$ pairs where the document is a known type of structure that can contain semistructured data formats such as JSON or XML. Like in key-value stores the basic operations allow a document to be fetched, updated, deleted and inserted based on a given key. Additionally, they also implement a fetch operation based on document contents which is a very implementation-specific feature since there is no standard query language. Few examples of document stores are CouchDB, MongoDB and Amazon's SimpleDB, among many others.

In addition, MongoDB, CouchDB and SimpleGEO support geospatial indexing allowing to query for location-based data. Although not as accurate as PostGIS, these NoSQL systems may fit in some use cases where performance and scalability are critical.

## 4.3 Real-Time in Distributed Systems

As defined in [18] a distributed system is *a collection of independent computers that appears to its users as a single coherent system.* This involves some sort of collaboration between the autonomous components (i.e., computers). Although discussing the advantages and disadvantages is not the topic of this writing, the main benefits of distributed architectures over centralized systems are the greater scalability, improved resilience and higher availability. To do so, distributed systems decouple a single application in a number of components that handle the diverse functionalities of the whole system. This enables the horizontally scaling in an independent way and makes them fail-tolerant. However, this impose other problems as the components need to share state and communicate. Moreover, failure-handling is often a complex task. [5] presents a summary of the main distributed computing concerns.

The challenge that real-time in a distributed environment entails is essentially the management of the system's state, as happens with non-real-time distributed systems. Particularly, real time systems are systems in which the timeliness of the operations is a part of the functional requirements and correctness of the system. However, nearly all systems may be qualified as *soft* real-time, in that there are usually unspoken expectations for the timeliness of operations.

On the other hand, *hard* real-time refers to systems which don't fulfil the requirements when time constraints are not met. This kind of systems are commonly known as Distributed Real-Time systems.

### 4.3.1 Message Passing

Instead of sharing the memory, in distributed systems is generally a better approach to share state by communicating. Message passing is a communication model that involves calling a subroutine by sending a message to an intermediary process. It relies on its infrastructure to invoke the actual code rather than calling subroutine by name as in a traditional procedure call. In such systems, communication is explicit and functions are separated from the specific implementations. (Immediate drawbacks)

An immediate upside of such communication model is the loose coupling between components. With a remote procedure call, the sender process must know the receiver and the complete signature of its procedures beforehand whereas with message passing, sender and receiver are nearly independent. This allows the system's components to be upgraded one at a time, thus giving the system the ability to evolve. Furthermore, it also improves interoperability. If the message is text-based (i.e., JSON, XML) there is no requirement for the components to be built in the same platform, operating system or language.

As all information regarding the state is contained in the messages there is no need for the receiver to store state information thus providing statelessness to the system. Although it comes at the cost of a slightly longer transmission time, this drastically improves the system's scalability.

Message passing systems are categorized in two main groups whether they im-

plement synchronous or asynchronous messaging. In the former, the sender and the receiver must be running at the same time for the message to be passed while in the latter the receiver is not required to be running at the time the sender sends the message. As a consequence, asynchronous messaging requires additional data storing and retransmitting capabilities for components that may not run concurrently.

In regular function calls, the caller waits for the called function to complete. Likewise, in synchronous message passing the sending process remains blocked until the receiving process finishes. The resulting impact on performance is generally unaffordable for large distributed systems, particularly for those with hard real-time constraints.

In contrast, asynchronous messaging is non-blocking. The sending process delivers the function call along with any needed arguments wrapped in a message to the message layer and continues its execution thread. The message layer acts as a intermediary between sender and receiver and so is considered a middleware. It stores the message until the receiver requests messages sent to it. Then, the receiver sends a message back with the result and the message layer stores it until the sender fetches its messages. Asynchronous messaging software is often referred to as Message Queues. Essentially, messaging middlewares store messages in a queue and processes them in a FIFO fashion.

Besides commercial products of well-known vendors such as Oracle or IBM, there are many open-source queueing systems available [17] due to the standardization of AMQP and STOMP protocols. As a consequence, there is no restriction for different programming languages to interact with queueing systems based on these standards. Each one of these has been created for solving specific problems.

The most widely popular and successful running on production systems are Apollo, HornetQ, RabbitMQ and ZeroMQ. The first one is a major rewrite of the Apache's ActiveMQ with better reliability and performance. It is an implementation of the Java Message Service (JMS) specification with support for Enterprise Integration Pattern required for distributed transactions. It supports STOMP, AMQP, MQTT, OpenWire and WebSockets plus SSL support. Additionally, Apollo also provides a

REST Management API.

HornetQ is fully JMS 1.1 compliant, previously integrated in the JBoss Application Server under the name of JBoss Messaging 2.0. Although further developed as a separated project, it provides seamless integration into JBoss. It is focused on reliability and contains lots of features and configurable settings at the expense of a certain degree of complexity. Its REST interface allows it to be used by any programming language, besides the HornetQ client libraries available.

RabbitMQ, written in Erlang, is specially suited for high performance distributed applications with great support for concurrency, availability and clustering. With its core fully supporting the AMQP protocol, it can understand STOMP protocol as well through its plug-in architecture. Furthermore, there are client libraries available for multiple languages and integrations with popular frameworks. Finally, its management plugin provides a web console that allows easy administration and detailed resources monitoring.

ZeroMQ, on the other hand, provides a library to create distributed and concurrent applications rather than a message queue. In contrast to the aforementioned message-oriented middlewares, ZeroMQ doesn't require a central server. The sender process handles the routing and the receiver deals with the queueing. This approach enables very low latency and high throughput resulting in substantially better performance. Therefore, it's ideal for large volume of messages like in financial transactions or online games.

It is worth mentioning, however, the MQ Telemetry Transport (MQTT) a pub-sub, simple and lightweight messaging protocol, designed for constrained devices and low-bandwidth, high-latency or unreliable networks. It is fully geared towards the minimisation of network bandwidth and device resource, over reliability and resilience. Therefore, it is especially suitable for the Internet of Things, the sensor web and mobile applications.

Scalability and reliability don't make a difference between all the above products, except for MQTT, as all of them are highly scalable, robust and reliable. With regards to performance, it's really difficult to objectively evaluate even with standardized

benchmarks. However, as already stated, ZeroMQ is by far the most performant message queue when message persistence is not required. Althought there's not much difference between other solutions, RabbitMQ tends to beat others when used with AMQP protocol.

## 4.4 Web Technologies

Since the Sir Tim Berners-Lee's first draft of the World Wide Web back in 1989 and his first proposal for the HyperText Markup Language (HTML) [2] in 1991 the WWW has experienced a tremendous evolution. Since then, HTML has gone through many revisions. The World Wide Web Consortium (W3C) published many iterations of the standard until the specification HTML 4.01 in 1999. It was not until 2004, when the Web Hypertext Application Technology Working Group (WHATWG) was founded to extend HTML so as to allow the creation of web applications. WHATWG published the First Public Working Draft of the HTML5 specification in 2008. Although parts of HTML5 have already been implemented in browsers, it was not until 2012 that W3C designated HTML5 as a Candidate Recommendation. It is scheduled for release as a stable Recommendation by the end of 2014.

### 4.4.1 HTML5

HTML5 is designed to deliver rich content without the need for third-party plugins while being backward compatible. As happened in HTML 4.01 with CSS 1.0, in HTML5 JavaScript is an integral part of the specification along with CSS3. The standard defines multiple JavaScript APIs such as drag and drop, files, webRTC, audio and video, webGL, etc. while others like vibration or accelerometer APIs are still in development.

HTML5 together with XMLHttpRequest API, the technology behind AJAX, already present in major browsers some years ago, has turned web browsers into web application containers. This fact has stimulated the development of JavaScript libraries. Some try to solve cross-browser issues like jQuery. Since its creation in 2006

it has become the standard for manipulating the DOM and for dealing with AJAX requests due to its simplicity and extensibility.

As for JavaScript frameworks, there is a myriad of choices due to the rebirth of the JavaScript community over the last few years. These modern JavaScript frameworks like Backbone.js, Ember.js, AngularJS, CanJS and others provide structure, organization and maintainability to the so called single-page applications (SPA). All of them implement variations of the Model-View-Controller (MVC) pattern.

Much like jQuery improved the DOM manipulation, D3.js happens to be essential as well when it comes to data visualizations. Its concepts provide an abstraction layer that enables the manipulation of documents based on data. It converts data to visualizations using HTML, SVG and CSS with lots of features through a large collection of components and plugins. It is worth mentioning its geographic projections along with layouts and geometry plugins which may come in handy in our project and its further development.

### 4.4.2 Real-Time

The web follows the pull paradigm by design. In HTTP, communication is initiated by the client who sends a request to a server. The server then sends a response back to the client. That is, the client pulls data from the server. While this approach works fine for fetching documents from a server, for which the web was initially designed, it does not work when the server needs to initiate the communication.

The first and most common solution to this issue is what is known as polling, that is, to request information from the server at regular intervals regardless of whether it has new data available. A further improvement of this approach is to make these requests wait until new data available, which is called long polling or comet. Both techniques, however, entail a waste of resources and are clearly non-real-time.

Nonetheless, HTML5 has provided two new techniques to solve this common issue and bring push capabilities to the web. Server Sent Events (SSE) is a W3C specification [15] that defines an API for opening an HTTP connection for receiving push notifications from a server through a half-duplex channel. It is designed to be

extended to work with other notification schemes such as SMS. Furthermore, it has features like automatic reconnection, event identifiers and the ability to send custom events.

With server-sent events, it's possible for a web server to send data to a web page at any time by pushing messages into it. The client initiates the communication by issuing a request to the server when a new EventSource object is instantiated in JavaScript. Then the server pushes the messages as DOM events that can be listened to like any other JavaScript event source.

On the other hand, WebSockets provide a richer protocol with support for bi-directional, full-duplex communication. They have drawn much more attention than server-sent events due to its richer feature set. WebSockets is a TCP-based protocol [6] completely separate from HTTP that provides low-latency two-way communication for browser-based applications.

Its bidirectional capabilities makes it particularly suitable for games, messaging apps and for use cases where near real-time updates in both directions is needed. Additionally, it provides cross origin communication, which enables communication between parties on any domain. As a downside, since it's not HTTP it requires specific infrastructure such as a server enabled to deal with this protocol.

# Chapter 5

# Design

In this chapter we will discuss all design decisions taken regarding the building of REDCH. These are grounded on the detailed research explained in Chapter 4. The high-level architecture already outlined in 3-1 is extended further by first explaining the technology used: the programming languages, frameworks and most relevant libraries. Then, the whole architecture of the system is presented by describing each one of the components the system is comprised of.

The design explained below aims to provide a solid groundwork and a first prototype for the REDCH project. Therefore, not all ideas discussed in previous chapters may be included in the result of this master thesis.

Ruby has been chosen as the main language for the development of the project. This dynamic language focused on simplicity and productivity is often regarded as developer performant. Due to its flexibility and similarity with natural language along with the massive amount of libraries and frameworks available, it allows developers to write applications very quickly. However, these features may hamper its execution performance.

## 5.1 Physical Architecture

The core idea of this architecture is to decouple the data producers from the data consumers by means of an asynchronous message queue enabling push capabilities in

the consumption tier. It is compound of four different servers, as shown in figure 5-1: an application server that receives the observations from the sensors and stores them in the database. That server is also responsible for publishing them into the messaging queue. Then the messaging queue makes them available to the data consumption tier, where the app server sends them to the web clients.

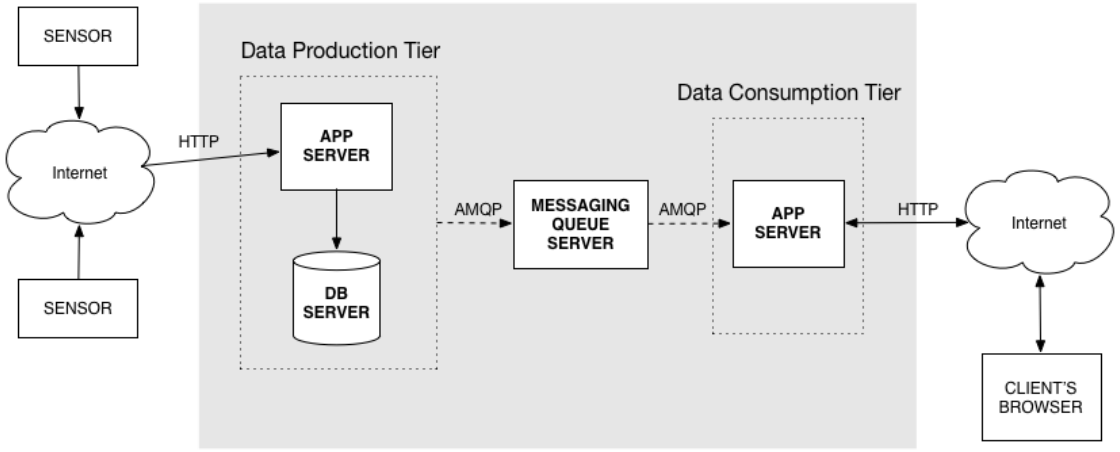



Figure 5-1: Physical Architecture

Such architecture brings about a number of benefits, as outlined in 4. First and foremost, each tier can easily scale out independently. In both tiers, high availability and increased throughput can be provided through redundancy [7], that is, adding more app servers. As for the database and the messaging queue, clusterizing them can provide better performance, increased throughput and storage capacity.

The loosely coupled architecture that the messaging queue provides along with the benefits stated in 4.3.1, not only enables horizontally scaling but also allows components to evolve independently. The only constraint is that both tiers must understand the Advanced Message Queuing Protocol (AMQP). On the other hand, given the wide adoption of such protocol, the particular messaging queue may be replaced without affecting any of the tiers.

Finally, a high-performance asynchronous messaging queue provides real-time capabilities to the system.

## 5.2 Logical Architecture

### 5.2.1 Sensor

As already stated in 3.1.1, although we opt for a solution that may involve RaspberryPi, the development of the sensor device falls beyond the scope of this project. Nonetheless, the system requires some sort of client in order to simulate its functioning in normal conditions.

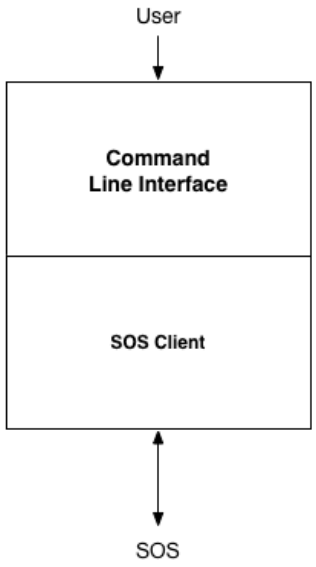



Figure 5-2: Simulator Logic View

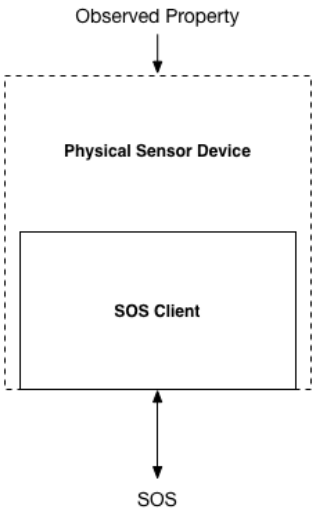



Figure 5-3: Sensor Logic View

A CLI acts as a presentation layer that enables interaction with the underlying SOS API client thus allowing the particular features of the sensor to be simulated. This approach offers the advantage of reusing the SOS client as a component of the final sensor device.

#### Command Line Interface

On one hand, the `Redch` global namespace comprehends the CLI's commands. Similar to the command pattern, each class is named after the command it enables and contains all the logic for that particular command. As 5-4 shows, the `Redch::CLI` class itself is a Thor application that exposes its interface in the command line,

receives the input, executes the commands and prints their output.

Thor is a toolkit for building powerful command-line interfaces, which is used by well-known frameworks and tools such as Bundler, Ruby on Rails or Vagrant. The Thor class exposes a command-suite command-line application like Git that leads to very polished and easy-to-maintain command-line applications. In any Thor subclass, public methods become commands. Furthermore, Thor provides an interface to easily specify options and flags as a command's metadata, along with methods to specify the description of the commands. These are then included in a automatically generated `help` command.

#### Sensor Observation Service Client

On the other hand, each command calls the underlying SOS Client, which in turn makes an HTTP request to the service. Being SOS a RESTful Web Service, the client implements two REST resources: `sensors` and `observations`, using the Ruby's rest-client gem[1].

This widely-used gem abstracts the actual HTTP protocol by exposing methods for each HTTP verb that accept header and query parameters to be passed in. Additionally, it provides a lower-level API that enables specification OF SSL parameters, dealing with cookies, etc. for cases the general API doesn't cover.

Finally, the client uses the Slim template language to render Geography Markup Language (GML), an OGC standard adopted by the International Organization for Standardization (ISO). Each operation has its own template that gets rendered with the data provided by each call. The obtained XML markup makes up the HTTP request body.

### 5.2.2 Messaging Queue

The messaging queue is the central component which drives the data throughout the system and thereby determines the architecture of all other components. A detailed

[1]Libraries in Ruby programming language

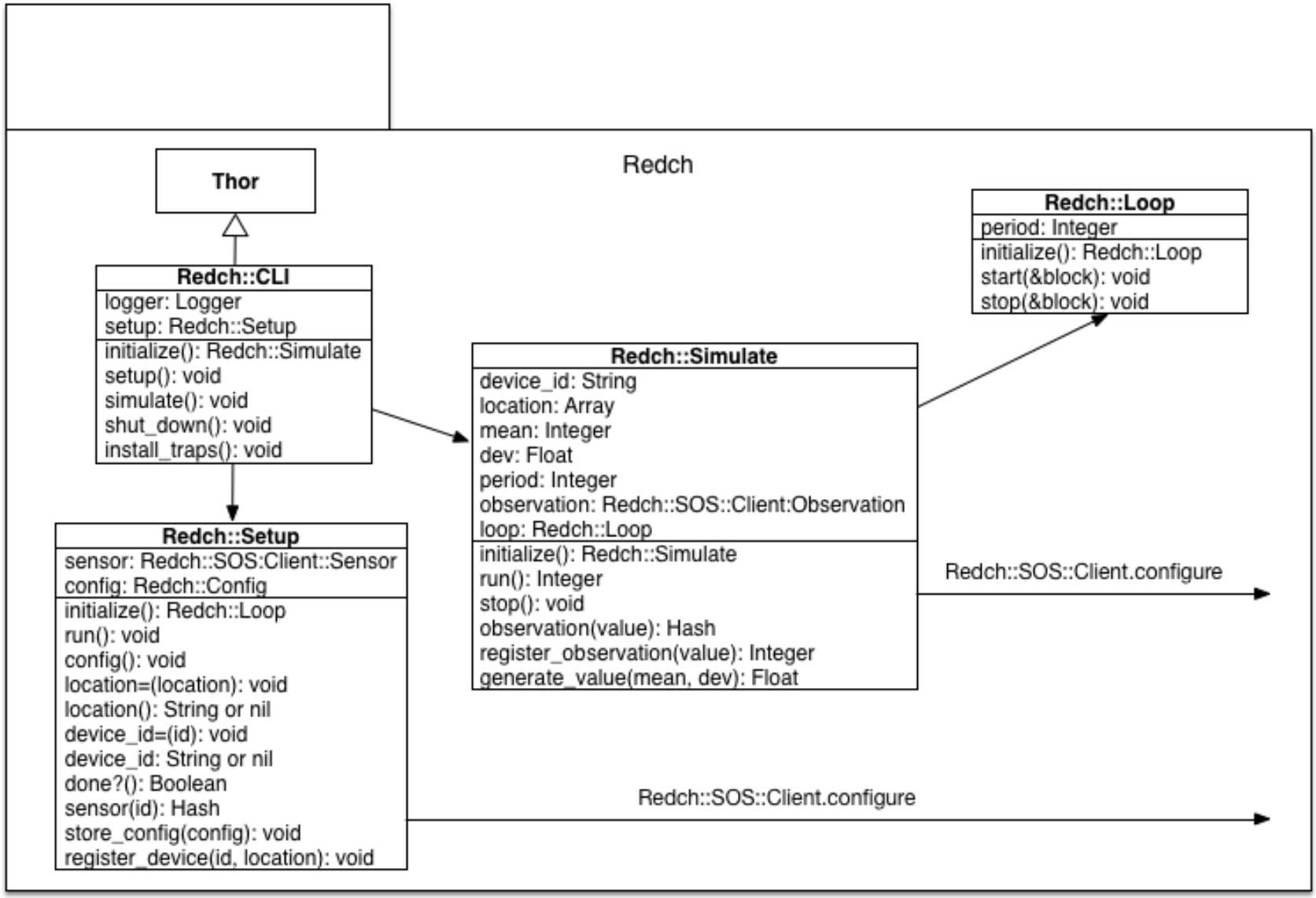



Figure 5-4: CLI class diagram

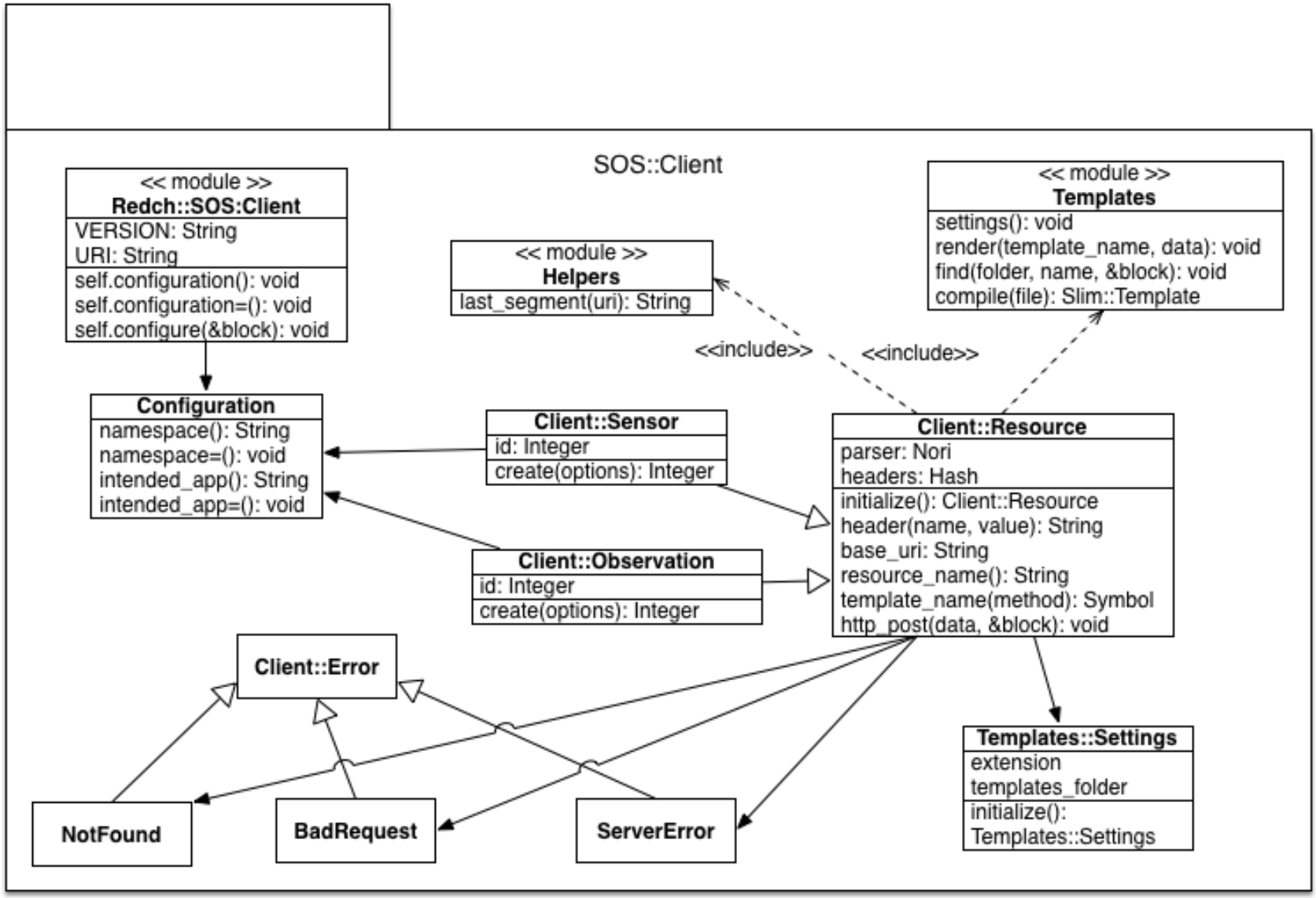



Figure 5-5: SOS Client class diagram

list of most known messaging queue systems has been given in 4.3.1, all of which highly reliable and high-performant. However, usually message queues aren't system bottlenecks but rather message consumers slowed down by database queries or backend systems.

So the choice of an specific message queue depends on the amount of client libraries available, particularly for the languages used in the project, clustering support and the complexity of installation and management. It is also important that the chosen queue has enough high-quality online resources to ease the integration. RabbitMQ, with a rich management web UI and exhaustive documentation including a clustering guide, is the one that best fits our requirements.

This decision has an impact on the design of the data producers and consumers, which must integrate with RabbitMQ using the AMQP protocol. This will be discussed further in following sections.

### 5.2.3 Sensor Observation Service

Given the CREAF's determination towards the SWE initiative and its involvement in the open-source GIS community, it is important to make use of the Sensor Observation Service (SOS). To do so, we opt for 52North SOS 4.0, the leading open-source implementation already integrated by many research institutions throughout the world. In this regard, great efforts are underway to bring latest web standards to the OGC implementations, which may be worth looking into in order to include them in the REDCH project. This is the case of 52North SOS 4.0. While this project uses its beta version, the final version was released less than three months before this writing.

As fully discussed in 4.1.2, SOS provides the level of interoperability the project requires. This specification structures the service with a core and four extensions: Transactional, Enhanced operations, Result handling and bindings. Together, all extensions provide CRUD functionality for sensors, observations and results.

With regard to the bindings, only SOAP and KVP are defined in the specification. In addition, 52North SOS 4.0 implements a RESTful binding as part of the bindings extension which our SOS client will use. By choosing this binding, we aim to build a

lightweight and stateless service client that can run in a resource-constrained sensor device.

Additionally, 52North SOS also provides an administrator GUI that enables changing the settings, de/activation of encodings and bindings as well as queries and clearance of stored data.

In order to integrate with RabbitMQ, a component that handles the data delivery to the messaging queue must be developed and included in the SOS. Once the observation has been stored in the database, this component publishes a message with the observation into the queue. Its logical architecture is shown in figure 5-6.

### 5.2.4 Database

52North's implementation uses Hibernate and Hibernate Spatial persistence framework to allow changing the underlying database management system and database model, which currently supports PostgreSQL/PostGIs, Oracle/Oracle spatial, MySQL and SQL Server DBMSs. Although we have chosen PostgreSQL, the GIS industry standard, REDCH may benefit from the integration of some sort of NoSQL solution.

The system is characterised by an ever-growing data set with small data units. That is, the system is write-intensive and I/O-bound. Given this features, in a real-world scenario REDCH may take advantage of massive NoSQL scalability and higher performance. Furthermore, in this project the impact of relaxed consistency may not be as high as in other systems where high reliability is required.

However, time constraints do not allow further exploration of this possibility since this would require that the whole relational schema migrate to a non-relational one. In addition, this prototype will only deal with a limited number of sensors for testing purposes.

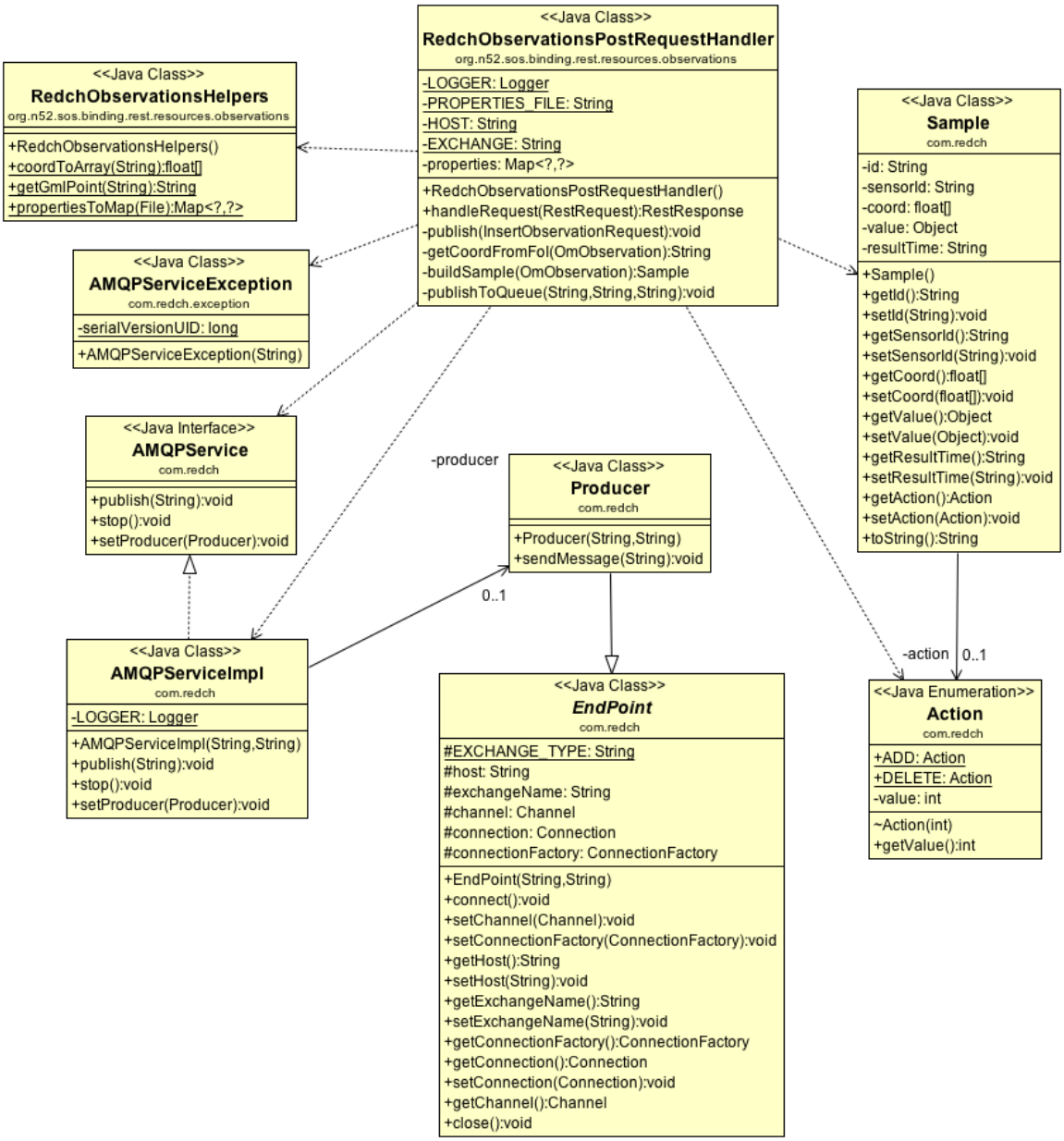



Figure 5-6: SOS AMQP extension

### 5.2.5 Web Application

The web application has two differentiated parts: the application's backend and the Single-Page Application (SPA). While the former pushes AMQP messages received from the queue to the SPA, the latter converts this information into a data visualization. Both components are tied together through a simple Sinatra Application.

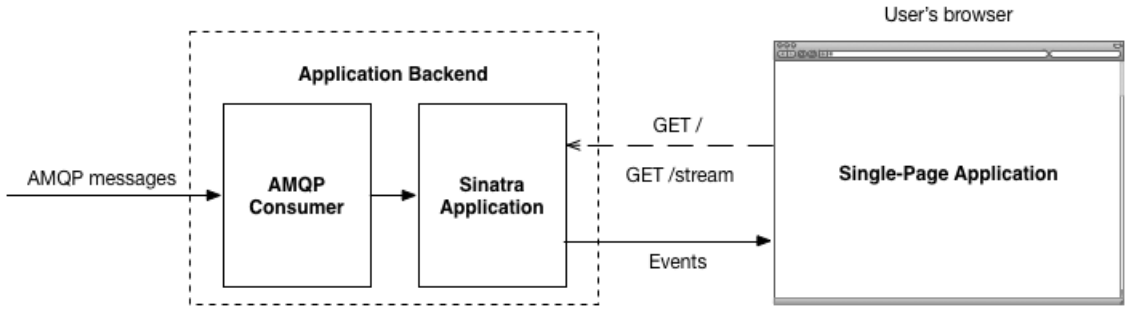



Figure 5-7: Web Application's Logic View

#### Backend

The backend uses the amqp gem. A feature-rich asynchronous RabbitMQ client which is built on top of EventMachine, the most popular event-driven I/O and concurrency library in Ruby. This library implements the Reactor pattern [13], the event handling pattern that constitutes the core of Python's Twisted or Node.js.

Once a message is received, the backend forwards it to each open streaming connection using the HTML5 Server Sent Events API, explained in 4.4.2. Therefore, concurrency is essential for the performance of the backend which, once again, is provided through EventMachine.

SSE has been chosen over WebSockets as the data delivery mechanism because of its much easier implementation and lesser impact on the underlying infrastructure. Moreover, since the data flows only from the backend to the browser, half-duplex one-way communication is enough.

Sinatra is a Web application framework and Domain Specific Language (DSL) that enables quick creation of web applications in Ruby. In contrast with other

frameworks, such as Ruby on Rails, Sinatra does not include a complex ORM nor follows the MVC pattern, focusing instead on being small and flexible.

The Sinatra Application exposes just two endpoints: `GET /` and `GET /stream`. The former is used to download the whole SPA, whereas the latter allows for an SSE streaming connection to be opened.

The class diagram of the whole backend is as follows.

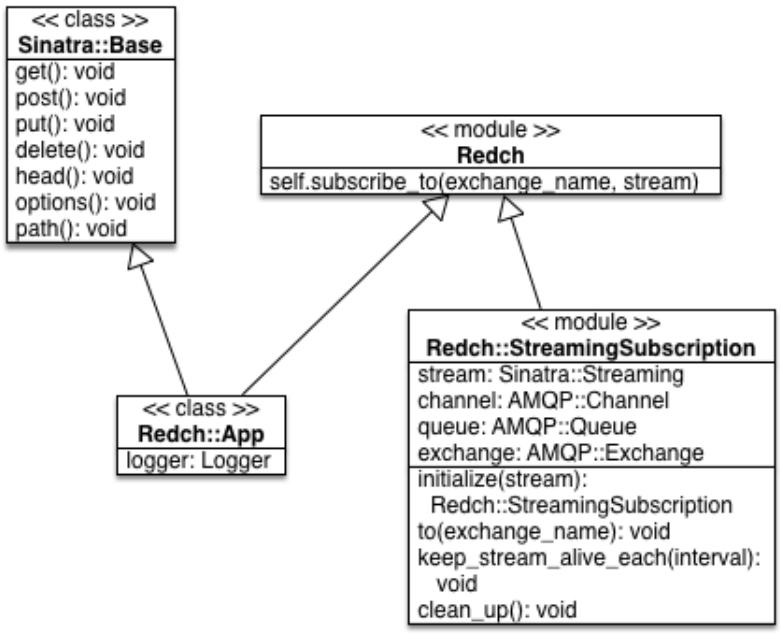



Figure 5-8: Backend class diagram

### Single Page Application

The SPA downloads all necessary source codes —HTML, JS and CSS— at the first request and renders the data visualization, empty at this point. Then, an HTTP streaming connection is opened through which the SSE events are sent. Then, the data visualization is continuously rendered with every event containing an observation by using the D3.js JS library. The state of the application is handled through Backbone.js which structures it as a collection of observation models.

The SPA consists of four different components: the HTML template, the data visualization, the data handling and the SSE client.

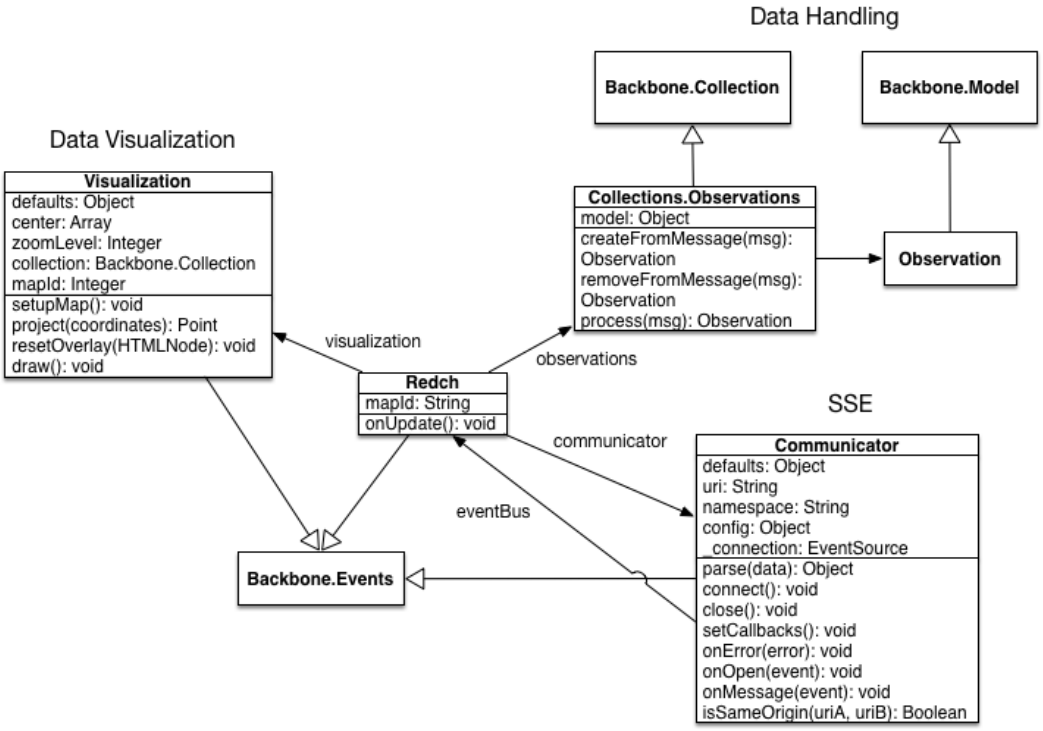



Figure 5-9: SPA class diagram

The single HTML file acts as the template for the application and contains the references to all of its assets: css files, web fonts, and JS libraries. Among these assets, there are the JS and tiles that Mabpox – the interactive maps JS library used – loads.

The data visualization sets up the map and renders circles placed at the exact location where the observation took place, each one corresponding to a different observation. These circles convey the observation's electrical power with the filling color ranging from yellow to red following a linear function.

All SPA pieces work in a fully evented fashion. Whenever a SSE message is received, the `Communicator` publishes the event into the `eventBus` and the observations collection, which is subscribed to said event, process it. When the collection triggers an add, remove or change event the visualization gets updated with new, changed or deleted circles depending on the information contained in the collection at that point.

## 5.3 Sequence Diagrams

This section aims to depict the lifetime of an observation as it goes through all the aforementioned steps from the sensor up until it is visualized in the SPA, thereby giving an overall view of the system's functioning.

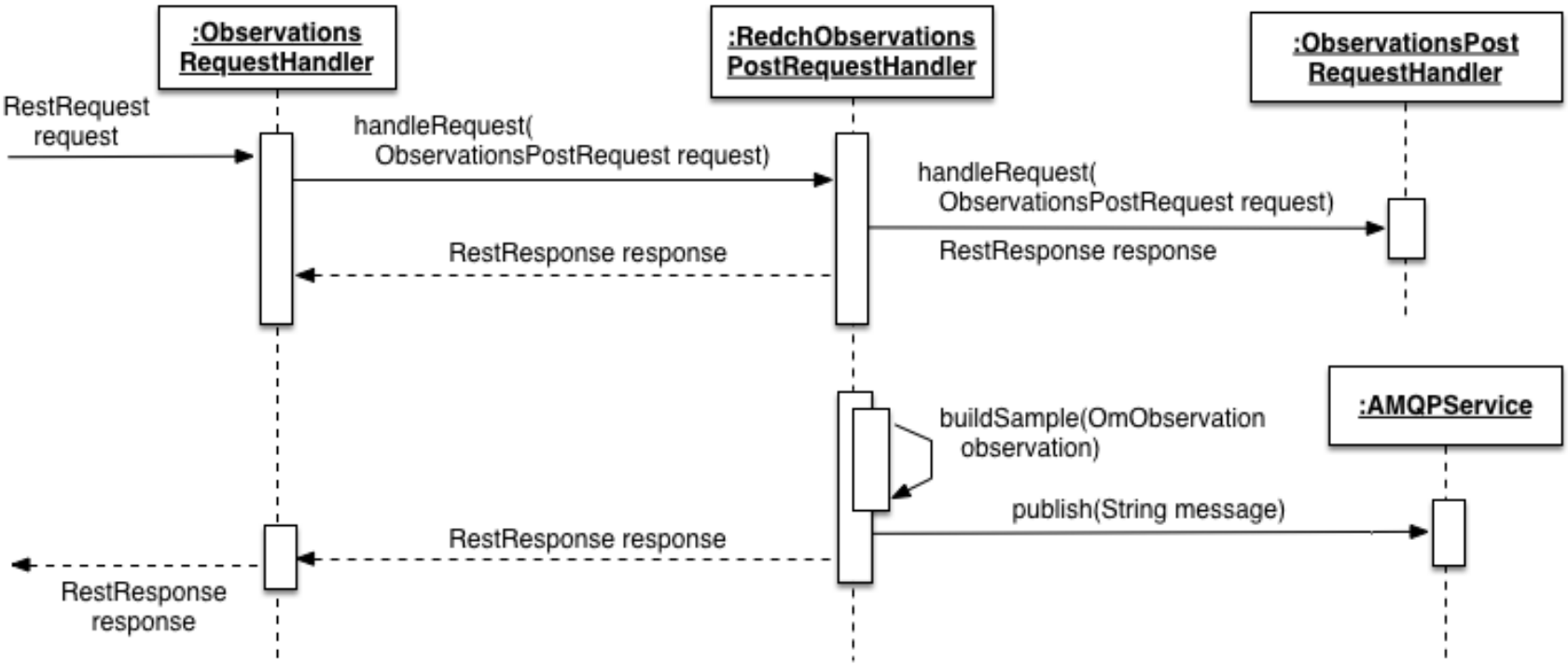



Figure 5-10: Sequence Diagram - Insert Observation

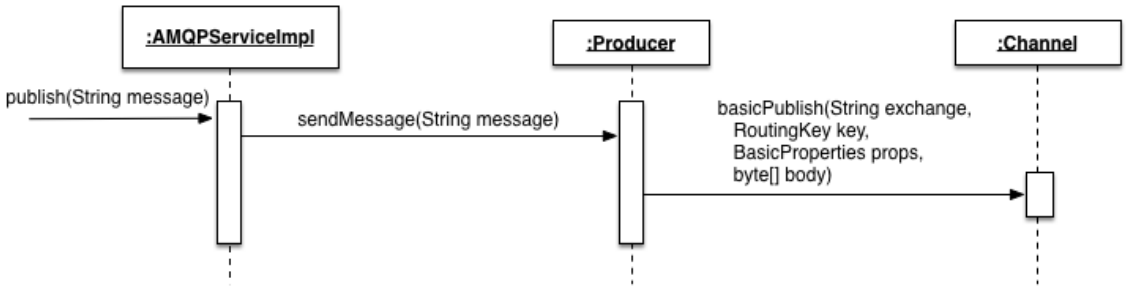



Figure 5-11: Sequence Diagram - Publish Observation

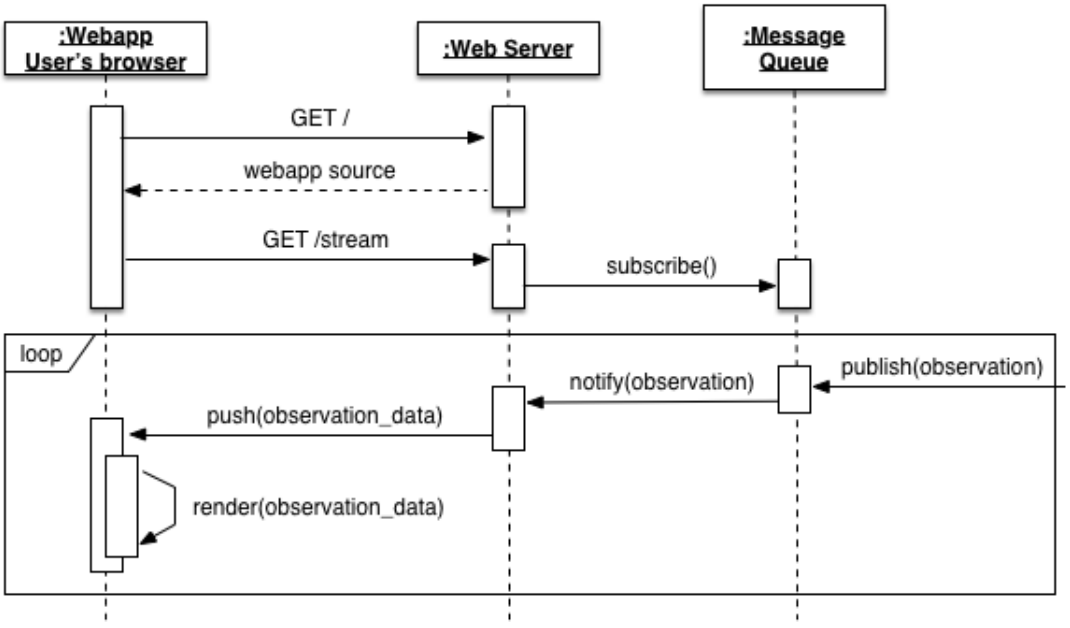



Figure 5-12: Sequence Diagram - Browse Data

# Chapter 6

# Implementation

## 6.1 Development environment setup

The development of this project encompasses a set of diverse tools that aim to ease this process allowing one to focus on the particularities of this project rather than on repetitive and common tasks. What follows is the description and reasons that led to their choice.

**Terminal emulator** iTerm2 has been used as the terminal emulator throughout the project to execute many tools used in this project from the compilation of the customized SOS to the execution of the simulator's CLI. Its rich features such as search, split panes, tabs, 256 colors or OS native notifications support make it a good replacement for the Mac OS X terminal.

**Editors** Given the diversity of languages used in the project different editors have been used in its development. An static language like Java requires the use of a full-featured Integrated Development Environment (IDE) like Eclipse, which provides integration with major frameworks and tools. As for the dynamic languages of the project, Ruby and JavaScript, Sublime Text 2 has been chosen as the main editor, sometimes replaced with Vim. Both are lightweight editors with a rich environment of plugins and focused on the efficiency of the developer.

**Version Control System** Is essential for the sake of the project to store its codebase in a Version Control System (CVS). All source codes as well as this document are kept in multiple Git repositories. In addition, Github has been chosen as the as the code hosting service due to its focus on collaboration and its considerable popularity in the open-source community.

**Virtual Machines** Virtual Machines (VM) have been mainly used in order to employ multiple sensor simulators at once. A tool such as Vagrant has dramatically improved the use of such systems by providing means to easily configure lightweight and portable development environments. It has become as simple as describing the VM in a file and booting it up by typing `vagrant up` in the terminal. The same configuration file can boot the same VM in any other host OS with vagrant installed.

```
VAGRANTFILE_API_VERSION = '2'
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = 'sensor-precise32'
  config.vm.provision :shell, path: 'provisioning.sh'

  config.vm.define :sensor0 do |s0|
    s0.vm.host_name = 'sensor0'
    s0.vm.network :private_network, ip: '192.168.0.2'
  end

  config.vm.define :sensor1 do |s1|
    s1.vm.host_name = 'sensor1'
    s1.vm.network :private_network, ip: '192.168.0.3'
  end

  config.vm.define :sensor2 do |s2|
    s2.vm.host_name = 'sensor2'
    s2.vm.network :private_network, ip: '192.168.0.4'
  end
end
```

Listing 1: Example of a Vagrantfile specifying three VM to host sensor simulators

**Secure Shell** the Secure Shell (SSH) has proven to be essential for the development of the project. Once the aforementioned VMs are running the easiest and fastest way to manage them is by using ssh through the terminal. Likewise, ssh is the only way to remotely manage the production servers.

**Custom tools** Third-party tools not always solve the issues encountered throughout the stages of a project. Rather than final solutions, sometimes is worthwhile considering them as the building blocks of a custom solution. This is the approach followed in the building of the Random Observations Generator[1], a very simple wrapper around the RabbitMQ's Management Command Line Tool plugin. The wrapper is built with Thor and the Open4 gem, which allows to open child processes and handle their pids and I/O streams.

## 6.2 CLI commands

Every implemented SOS operation has its CLI command equivalent. Figure 2 shows the implementation of the command `simulate`. The Thor's `desc` and `option` class methods allow one to define the description of the command and any option such as `period`. The helper shell method `say` outputs the passed message to the terminal. The `simulate` command is then executed from the terminal as:

```
$ redch simulate -p 10
```

Figure 6-1: Example of sensor's simulation from the command line

## 6.3 AMQP Service

The Service Interface pattern is a common and simple pattern for building Java extensible applications. The Service is just a set of programming interfaces and classes that provide access to some specific feature. Considering the implemented

[1]Git repository: `https://github.com/sauloperez/redch-obsgen`

```
desc "simulate", "Simulate a sensor generating electrical power observations in W"
option :period, :aliases => :p
def simulate
  setup
  config = Redch::Config.load
  simulate = Redch::Simulate.new(config.sos.device_id, config.sos.location)
  simulate.period = options[:period].to_i if options[:period]

  say("Sending an observation from #{put_coords(@setup.location)} every #{simulate.period} seconds..
  simulate.run do |value|
    say("Observation with value #{value} sent")
  end
end
```

Listing 2: Implementation of the command `simulate` using Thor

AMQP Service, figure 3 constitutes the Service Provider Interface (SPI), the public interface defined by the service. Then, the particular implementation shown in 4 acts as a AMQP Service provider by conforming to the SPI.

```
// (...)
public interface AMQPService {
  void publish(String message) throws IOException;
  void stop() throws IOException;
  void setProducer(Producer producer);
}
```

Listing 3: AMQPService SPI

```
// (...)
public class AMQPServiceImpl implements AMQPService {
  private static final Logger LOGGER = LoggerFactory.getLogger(AMQPServiceImpl.class);

  private Producer producer;

  public AMQPServiceImpl(String host, String exchangeName) throws IOException, AMQPServiceException
    try {
      this.producer = new Producer(host, exchangeName);
    } catch (AMQPServiceException e) {
      LOGGER.debug("AMQP connection failed");
      throw e;
    }
  }

  public void publish(String message) throws IOException {
    producer.sendMessage(message);
  }

  public void stop() throws IOException {
    producer.close();
  }

  // (...)
}
```

Listing 4: AMQPService implementation

## 6.4 Sinatra's DSL

Sinatra exposes a simple DSL that enables the actions associated to a given endpoint to be specified and the response template to render. Similar to the HTTP verbs methods, it defines a method for each templating engine supported, which accepts the template name as a parameter.

```ruby
class App < Sinatra::Base
  configure do
    # (...)
  end

  get '/' do
    erb :index
  end

  get '/stream', provides: 'text/event-stream' do
    stream :keep_open do |connection|
      p "New connection: #{connection.object_id}"

      Redch.subscribe_to 'samples', stream: connection
    end
  end
end
```

Listing 5: Implementation of the two backend endpoints with Sinatra

## 6.5 Data Joins in D3

Data Joins is what D3.js uses to bind data to elements. Data joined to existing elements produces the *enter* selection, that is, the intersection's set between data and elements. All unbound data produce the *enter* selection, that is, all missing elements. Similarly, all remaining elements produce the *exit* selection which represents elements to be removed. These selections represent the three possible states.

To operate over these three states, one must select the elements and data to be joined. In the third line of the source code 6 all circles of the `this._g` SVG container are selected. This selection is then joined to the array of Backbone observation models `this.collection.models` passed in on instantiation.

As a consequence, data joins lead to a more declarative code allowing targeting operations to specific states without need for branches nor iterations. A good example is found in lines 8 and 24. While the updated circles animate their transition to the new fill color, the circles fade out before being removed.

```
draw: function() {
  var self = this,
      feature = this._g.selectAll("circle")
        .data(this.collection.models),
  //  (...)

  // Update circles that are still present
  feature.transition().duration(200).style("fill", function(model) {
    return color(model.get('value'));
  });

  // Create new circles
  feature.enter()
    .append("circle")
    .style("fill", function(model) {
      return color(model.get('value'));
    })
    .style("fill-opacity", 0.75)
  //  (...)
    });

  // Remove old circles
  feature.exit()
    .transition().duration(250).attr("r",0).remove();
}
```

Listing 6: D3 data joins used in the SPA

## 6.6 Server-Sent Events

### Server-side

From the server-side, HTML5 Server-Sent Events API is a really simple convention over a regular HTTP streaming connection. Together with Sinatra's DSL its implementation is reduced to few lines.

```
1 class App < Sinatra::Base
2   get '/stream', provides: 'text/event-stream' do
3     stream :keep_open do |stream|
4       stream << 'id: Time.now.to_i\n'
5       stream << 'data: a SSE event from Sinatra\n\n'
6     end
7   end
8 end
```

Listing 7: SSE server-side implementation with Sinatra

The Event Stream format is just a plain text response served with Content-Type set to `text/event-stream` and whose data must conform to the SSE format. The format specifies that the response must contain a line beginning with `data:` followed by the message. The message can be broken up in multiple `data:` lines by ending them with a single `"\n"` char. Therefore, `"\n\n"` must be used to end the stream. This is considered a single event, thereby firing only one `message` event on the client-side.

An event can be associated with a unique id by including a line starting with `id:` as in line 4. Likewise, the reconnection-timeout can be changed by including a line beginning with `retry:` followed by the number of milliseconds to wait before the reconnection. In this way, whenever the connection is dropped the browser will attempt to reconnect after the specified time.

What makes SSE even more interesting is the possibility to specify your own event names. If the server sends a line beginning with `event:` followed by a unique name, this event will be associated with that name. Hence, the client can set up a regular event listener to listen to that particular event.

## Client-side

With regard to the client-side, the JavaScript API exposes the `EventSource` object. To subscribe to an event stream, this object must be instantiated passing the URL of the stream. This can be easily encapsulated into a standalone JavaScript object

so the data consumers are not concerned with the details of the API. This is the idea behind the implementation of `communicator.js` partially shown in 8.

```
// (...)
connect: function() {
  if (this._connection) return;
  this._connection = new EventSource(this.uri);
  this.setCallbacks();
}
// (...)
```

Listing 8: SSE connection in `communicator.js`

Next, a handler may be set up for each of the `EventSource`'s basic events: message, open and error. The `onmessage` handler fires and new data becomes available in the `data` property of the event object whenever updates are pushed from the server. Likewise, `onopen` is triggered when the connection has been opened and `onerror` when an error has been encountered.

```
this._connection.onopen = function(e) {
  console.log('New SSE connection opened');
};
this._connection.onmessage = function(e) {
  console.log("Message '" + e.data + "' received");
};
this._connection.onerror = function(e) {
  console.log('An error has occurred');
};
```

Listing 9: Example of SSE event handlers

Furthermore, the application can listen to your specific events setting up a regular `EventListener` as follows.

```
this._connection.addEventListener('sensor-in-sleep-mode', function(e) {
  console.log("This sensor won't send more observations");
});
```

Listing 10: Listen to custom events

# Chapter 7

# Infrastructure

This chapter aims to describe the tools and processes involved in the infrastructure setup, from local development environment to the set of production servers running the REDCH. Firstly, it explains the infrastructure setup and secondly, describes the provisioning and deployment processes.

## 7.1 Amazon AWS setup

Amazon Web Services (AWS) is a cloud computing platform that offers a collection of remote computing services ranging from computing and storage to networking services such as DNS, among others. AWS is a world-wide leader of Infrastructure-as-a-Service (IaaS) providers with numerous companies like Spotify, Heroku, Airbnb, Foursquare, Github, Reddit or Mapbox relying on them.

The whole infrastructure of the system is made up of EC2 instances, virtual servers in Amazon's cloud. They all run a custom Amazon Machine Image (AMI) built from a raw Ubuntu 12.04 LTS with all needed dependencies —Puppet and Ruby 2.0.0— installed. As a result, any new instance booted up with this custom AMI is ready to be provisioned. Starting and stopping machines, as well as configuring their firewall rules is managed through the AWS Management Console, a web UI.

One of the major benefits REDCH can take advantage of is Amazon's Auto Scaling. This service allows to scale the capacity of the EC2 instances up and down

according to a set of predefined conditions. A load balancer, for instance, can automatically spawn app servers during demand spikes and shut them down during low demand periods. REDCH can get the most out of it by exploiting the fact that solar panels don't produce energy at night, thereby minimizing costs. Likewise, less computing power is required under windless conditions.

Although it would be desirable to keep a provider-independent infrastructure, Amazon RDS has been chosen as database server which makes it easier to set up, operate and scale a relational database. Furthermore, it provides automated backups, Multi-AZ replication and monitoring metrics. It has support for MySQL, Oracle, SQL Server and PostrgeSQL, all the DBMSs supported by 52North SOS, being the latter the one it uses by default. However, the PostrgeSQL support is still in beta version due to its recent release in November of 2013.

The final production environment consists of the four servers shown in 7-1. Three EC2 micro instances plus a RDS micro instance. Amazon's free tier includes both services at no cost within the first year. Therefore, EC2 micro instances are limited to 1 low-capacity throttled CPU with 627MB of RAM [1]. All three EC2 instances are attached to an 8GB Elastic Block Storage (EBS), which are storage volumes with built-in redundancy. These volumes host the filesystem of their attached EC2 instances.

Ubuntu 12-04 LTS has been chosen as the OS of all three servers due to its stability and security as well as the inherent benefits of a Linux OS. With regard to the database, Amazon RDS abstract away from the particularities of the underlying hardware by providing the database access as a service.

## 7.2 Provisioning

Once the software is developed, the underlying infrastructure must be configured to host each of the system components. This process, which involves creating directories

[1] AWS Micro Instances Documentation: `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts_micro_instances.html`

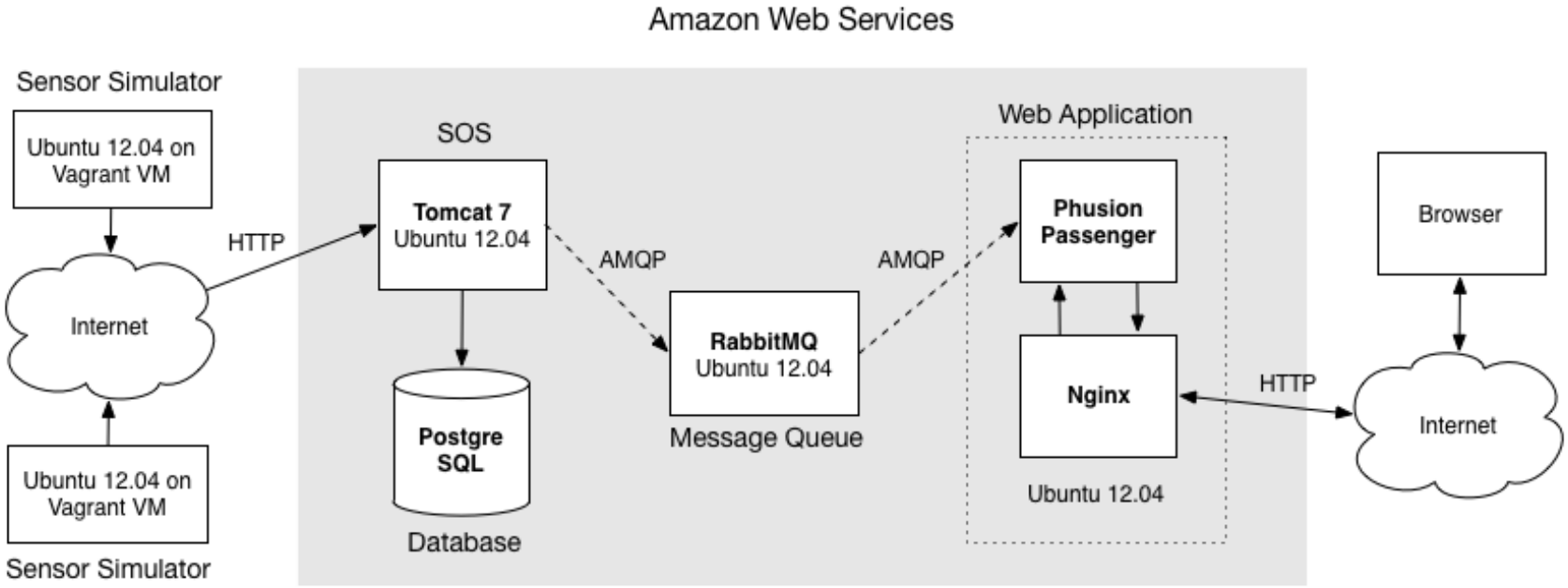



Figure 7-1: Production's infrastructure diagram

and installing packages, may be error-prone when done manually. Besides, the process may need to be repeated several times whenever new servers are set up. Although writing down the build-out process may help, whoever reads that documentation may not be able to figure out the current state of the configuration.

Server automation frameworks formalize systems administration treating infrastructure as code. As a result, infrastructure configuration can be tested and repeated, automating away repetitive tasks while systems administrators focus on architecting and tunning services.

Puppet, among other solutions such as Chef, enables server configuration automation. It automates server provisioning by formalizing its configuration into manifests. Puppet's manifests are text files that contain statements written in a declarative DSL that allows the desired state of the infrastructure to be defined. Once these configurations are deployed, Puppet automatically installs the necessary packages and ensures that the machines files and services match the desired state.

```
1 package { 'apache2':
2   provider => 'apt',
3   ensure   => 'installed'
4 }
5
6 service { 'apache2':
7   ensure => 'running'
8 }
```

Listing 11: Example of Puppet's manifest file

Puppet is mature and widely used and besides having an open source version, there are lots of learning materials available online.

Prior provisioning, any server must have Puppet installed which comes packaged as a ruby gem and therefore, also requires a MRI Ruby interpreter. Puppet can also be installed with any system package manager, but doing so will likely install previous releases missing features and bug fixes.

Puppet enables provision machines by either applying the configuration directly or compiling into a catalog and distributing it to the target system via a client-server

paradigm.

### Production Software

Most of the software used in the development environment is also used in production. This is the case of SOS, for instance, where both Tomcat 7 and PostgreSQL have been chosen to run also in production. However, the production web application stack differs from the one used in development. Nginx plus Passenger has been chosen over Thin, the server included in Sinatra for development purposes.

Passenger is a mature and feature-rich application server widely used in many production scenarios. Therefore, is easy to find learning materials and support on the internet. Nginx on the other hand, is a high-performance web server and load balancer that can handle high concurrency.

Placed in front of the application server, Nginx acts as a reverse-proxy. It deals with all incoming requests serving static files efficiently and passing them to the application layer. Passenger then processes the requests and returns a response.

## 7.3 Deployment

Once the configuration is applied, the target server is up and ready. Next, the code release must be transferred to the production environment to make the application available for use.

Capistrano is a remote multi-server automation tool that enables the execution of arbitrary tasks on remote servers over SSH. It aims to allow reliable deployment of web applications to any number of machines simultaneously. All of its features enforce sane development workflows.

The general configuration of the application is set in the `deploy.rb` file. Then, each particular stage overrides it in its own file. By following this convention, Capistrano infers the environment names and enables tasks to be run on each of them as in 7-2.

The configuration can be tailored to fit the needs of the project by writing extra

```
1  set :application, 'redch'
2  set :repo_url, 'git://github.com/sauloperez/redch-webapp.git'
3  set :ssh_options, {
4    forward_agent: true
5  }
6
7  ask :branch, proc { `git rev-parse --abbrev-ref HEAD`.chomp }
8
9  set :deploy_to, '/var/redch'
10 set :use_sudo, false
11 set :deploy_via, :copy
12 set :copy_strategy, :export
13
14 SSHKit.config.command_map[:rake]  = "bundle exec rake"
15
16 after 'deploy:publishing', 'nginx:restart'
17 after 'deploy:publishing', 'passenger:restart'
```

Listing 12: Capistrano's deployment definition

```
$ cap staging deploy
...
$ cap production deploy:rollback
```

Figure 7-2: Execution of commands in different environments

tasks. Capistrano provides the *deploy* and *rollback* flows that invoke several hooks for the developer to hook up custom tasks into the flow. In line 16 of listing 12 Nginx is restarted right after the release has been published and before the deploy's leftovers are cleaned up.

Therefore, some custom Capistrano tasks have been developed to ease the execution of common operations such as starting and stopping tomcat. The listing 13 illustrates Nginx-related tasks used in the web application's deployment. As Capistrano essentially executes commands in a remote server, these custom tasks can written to target any specific purposes such as listing servers' uptimes, checking their load, etc.

```
namespace :nginx do
  desc 'Start nginx'
  task :start do
    on roles(:app), in: :sequence, wait: 5 do
      sudo 'start nginx'
    end
  end

  desc 'Stop nginx'
  task :stop do
    on roles(:app), in: :sequence, wait: 5 do
      sudo 'stop nginx'
    end
  end

  desc 'Restart nginx'
  task :restart do
    on roles(:app), in: :sequence, wait: 5 do
      sudo 'restart nginx'
    end
  end
end
```

Listing 13: Content of `lib/capistrano/tasks/nginx.cap`

# Chapter 8

# Performance Testing

Performance testing deals with any process of determining the quality attributes of a system such as responsiveness, stability and reliability under certain workloads. It is usually used to verify that the system meets its specifications.

Load testing is the simplest form of performance testing. Tests are conducted to assess the behaviour of the system under a particular load, which is defined by a particular number of transactions and a certain level of concurrency within a specified duration. In the context of a web system, the transactions are a round-trip of HTTP requests to a particular endpoint while the concurrency level is the number of requests performed at a time. As a result, the test outputs the response times of these requests, thus uncovering possible bottlenecks of the system.

There are numerous variables involved in the execution of a web system such as network reliability, performance of the underlying hardware, availability of third-party services, etc. Therefore, the following load tests do not aim to give a thorough assessment of the performance of the system but rather its behaviour under conditions similar to a real scenarios while making some reasonable assumptions.

The tests have been conducted using two simple yet powerful and mature open-source tools: Apache Bench and Gnuplot. Apache Bench is a server benchmarking tool that focuses on showing how many requests per second a system is able to serve. It provides basic statistic such as mean, median, minimum and maximum of the measured magnitudes. Gnuplot, on the other hand, makes it easy to draw charts from

diverse input text formats by means of its own scripting language or an interactive console. Additionally, Apache Bench can output data in Gnuplot-compatible format, which allows both tools to be easily integrated.

Since the system has two entry points, the one used by the sensors and the web application, two different load test have been performed. By doing so, we can assess the performance of the Sensor Observation Service and the Web application.

## 8.1 Web Application

First and foremost, the variables involved in determining the response times of the requests must be defined. These are the concurrency level and the number of requests per test.

**Concurrency** We distinguish two different levels of concurrency. Firstly, when a browser loads a web page it starts multiple connections to the server to load the resources. The number of simultaneous connections is a built-in browser parameter that for most of the web browsers defaults to 6. Besides, concurrency in load testing often refers to the number of users issuing requests to the system at a time. Each progressive increase in this variable defines the workloads the system will be tested with.

**Number of requests** This is the total number of requests issued to the system for each execution of the test. However, each time a user loads a web page the browser makes as many requests as assets the page contains. That is, the browser loads each of the CSS files, images and JS scripts the HTML lists, one per request.

In this particular case, the single HTML page of the application contains 46 assets, loading 717 KB of data. These resources contain the JS application source files plus the map tiles and other resources fetched by the Mapbox library. Moreover, as the user interacts with the map, more tiles are loaded by the browser. Nevertheless, these requests don't hit our system but instead the Mapbox's servers. Therefore, they are not taken into account although they have an impact on the perceived performance

of the system. The size of each resource is assumed to be the average, 717 KB / 46 resources = 15,6 KB.

### Baseline

To serve as comparison the baseline is defined as a test with a single request to load the HTML page followed by 46 requests with 6 concurrent connections to load the assets of 15,6 KB each. This tries to mimic the behaviour of the browser while simplifying the intricacies of its concurrency and assuming the content is instantly rendered.

### 10-User Scenario

In the following scenario the concurrency is increased progressively in order to simulate more demanding situations, first 10 users, then 50. For each of these, half of the required requests point to the HTML and the other half to the assets thereby simulating requests that impact the server differently.

Therefore, the first test issues 10 concurrent requests to load the HTML, while 460 more requests with 60 concurrent connections load the assets.

### 30-User Scenario

To test the load equivalent to 30 users, this scenario involves 30 concurrent requests plus 1380 requests with 180 concurrent connections. Again, this tries to be slightly more realistic than just requesting the plain HTML document.

## 8.2 Sensor Observation Service

This test aims to assess the performance of the system while receiving requests from the sensors. This test impacts the SOS and the database and so the bottleneck is likely to be one of these components. In contrast with the web application load

test and since SOS is essentially a set of HTTP endpoints, regular requests with an increasing level of concurrency are enough to assess the performance of the system.

Nevertheless, the sensors are required to perform only a request every 10 minutes, resulting in 6 requests per hour. If higher resolution was required requests would also be evenly distributed. Thus, the test take into account this time spans.

### Baseline

For this test, the baseline is defined as a single POST request to the `/observations` endpoint. The two available endpoints, `/sensors` and `/observations`, have fairly similar behaviour. Since they process requests with similar payload size and store results in the database, there is no need for testing both endpoints as it provides no valuable insight.

### First scenario

In this scenario the load is composed of 10 sensors sending observations every minute concurrently. Thus, the resolution of the observations is then increased up until 60 observations per hour. This covers the worse-case scenario where all available sensors have been turned on at the same time, thereby sending their observations simultaneously. For the sake of brevity, only the execution with the worst mean response time is shown in 8.3, while the response times of all executions is summarized in the chart 8-3.

It is worth noting that all requests use the same POST data file, as it is the only way to simulate a POST request with Apache Bench. Splitting the test into different steps, one for each particular sensor wouldn't allow Apache Bench to compute aggregated statistics.

### Second scenario

Finally, in this scenario the SOS performance is tested under a more demanding load. To do so, the test sends 500 requests with 30 concurrent connections in a single

execution. Thus, by assuming that every sensor sends an observation per minute and given that the system receives multiple batches of concurrent connections within a minute, this scenario simulates much more than 30 sensors.

## 8.3 Results

### Web Application

As shown in 8.3, the application assets are loaded in 240ms. This, together with the response time of a request for the HTML document, 431ms, means that a single user waits an average 671ms until the whole application is fully loaded. The requests per second the system is able to serve while dealing with the assets is 24.94 req/sec.

When the concurrency level is increased up to 10, the application loading time reaches $2966ms + 4419ms = 7385ms$, on the average. However, as shown in 8.3 and contrary to expectations, the throughput is reduced to 13.58 req/sec. The chart 8-1 more closely examines all the response times, which happen to be slightly disparate in some cases.

Nevertheless, the system reaches its tipping point when the concurrency is increased to 180 connections and 1380 requests for assets are issued. In this case, Apache Bench's output 8.3 lists 54 failed requests and the mean response time is increased by more than 30%. The related chart 8-2 perfectly illustrates how the system is unable to reliably process all the requests. Besides the initial burst on the response, the standard deviation rises gradually as time goes by until reaching its maximum capacity serving requests in around 30 seconds. Then, the system suddenly brakes and serves only HTTP error responses which have a negligible response time.

In spite of some unusually long response times, the system can reliably handle around 60 concurrent connections. In overall terms, the results show a somewhat poor performance. It takes more than 7 seconds to load the application with 10 concurrent users, a highly likely scenario.

## Sensor Observation Service

As expected, both SOS endpoints have a similar response when a single request is issued to them. While `/sensors` response time is 510ms, it is 654ms for `observations`. This is a reasonable difference since to insert an observation, the SOS must first look up the related sensor.

When some concurrency is added, the results of the first scenario 8.3 show an increased response time – 1103ms – but a better throughput, with 9 requests per second. When all response times of the 7 executions are considered together, as in 8-3, the deviation turns out to be remarkably high. While most of the response times fall within the one-second threshold, a significant number of measures are around 2 seconds and few of them fall beyond 3 seconds.

When tested with heavy load of 500 requests and 30 concurrent connections, the system provides to be far more unreliable than the first scenario. The chart 8-4 conveys a dramatic increase in the deviation of the measures. Furthermore, the two failed requests shown in 8.3 make clear that the system has already reached its limit in terms of concurrency.

It should also be mentioned the high value of the mean connection waiting time. Unlike the web application test, it is almost equal to the mean connection processing time across all scenarios, including the baseline. This is directly related to the particular kind of workload this server does, mostly bound to the CPU and IO. Both are scarce resources in a AWS micro instance.

Finally, the first scenario provides that the system meets the requirements in terms of observation processing. It is able to handle 10 concurrent connections reliably.

```
(...)

Document Path:          /test_asset
Document Length:        15600 bytes

Concurrency Level:      6
Time taken for tests:   1.845 seconds
Complete requests:      46
Failed requests:        0
Keep-Alive requests:    46
Total transferred:      729330 bytes
HTML transferred:       717600 bytes
Requests per second:    24.94 [#/sec] (mean)
Time per request:       240.603 [ms] (mean)
Time per request:       40.100 [ms] (mean, across all concurrent requests)
Transfer rate:          386.12 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    8  19.9      0      64
Processing:    87  223 241.9    142    1448
Waiting:       54  110  26.9     99     189
Total:         87  231 249.4    142    1448

Percentage of the requests served within a certain time (ms)
  50%    142
  66%    159
  75%    259
  80%    265
  90%    378
  95%    748
  98%   1448
  99%   1448
 100%   1448 (longest request)
```

Listing 14: Web application baseline results

```
(...)

Document Path:          /
Document Length:        2406 bytes

Concurrency Level:      10
Time taken for tests:   2.967 seconds
Complete requests:      10
Failed requests:        0
Keep-Alive requests:    10
Total transferred:      27460 bytes
HTML transferred:       24060 bytes
Requests per second:    3.37 [#/sec] (mean)
Time per request:       2966.874 [ms] (mean)
Time per request:       296.687 [ms] (mean, across all concurrent requests)
Transfer rate:          9.04 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:      100  106   2.9    107     110
Processing:   241  529 821.4    276    2867
Waiting:      236  380 370.9    270    1435
Total:        345  636 819.3    383    2967

Percentage of the requests served within a certain time (ms)
  50%    383
  66%    390
  75%    397
  80%    404
  90%   2967
  95%   2967
  98%   2967
  99%   2967
 100%   2967 (longest request)
```

Listing 15: Output of 10 concurrent connections to root of the Web application

```
(...)

Document Path:          /test_asset
Document Length:        15600 bytes

Concurrency Level:      60
Time taken for tests:   33.881 seconds
Complete requests:      460
Failed requests:        0
Keep-Alive requests:    460
Total transferred:      7293300 bytes
HTML transferred:       7176000 bytes
Requests per second:    13.58 [#/sec] (mean)
Time per request:       4419.294 [ms] (mean)
Time per request:       73.655 [ms] (mean, across all concurrent requests)
Transfer rate:          210.22 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0   17  48.0      0     212
Processing:   401 2677 2836.6   1979   31183
Waiting:       71  431 699.3    217    7835
Total:        401 2695 2849.7   1993   31183

Percentage of the requests served within a certain time (ms)
  50%   1993
  66%   2608
  75%   3115
  80%   3535
  90%   4596
  95%   7347
  98%  11424
  99%  18008
 100%  31183 (longest request)
```

Listing 16: Output of 460 requests with 60 concurrent connections to load the assets

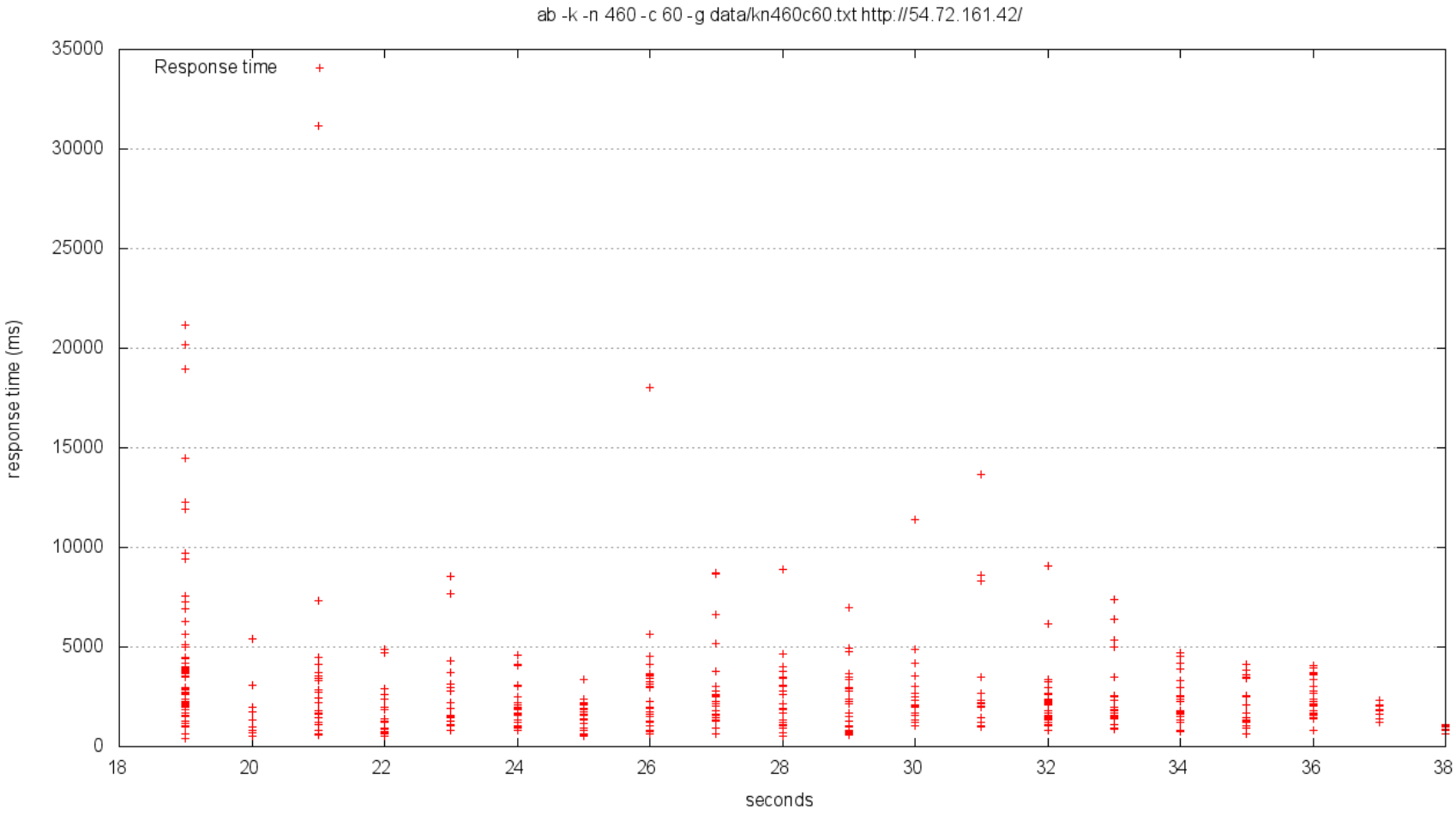



Figure 8-1: Response time plot of 460 requests with 60 concurrent connections

```
(...)

Document Path:          /
Document Length:        2406 bytes

Concurrency Level:      30
Time taken for tests:   1.607 seconds
Complete requests:      30
Failed requests:        0
Keep-Alive requests:    30
Total transferred:      82380 bytes
HTML transferred:       72180 bytes
Requests per second:    18.67 [#/sec] (mean)
Time per request:       1606.587 [ms] (mean)
Time per request:       53.553 [ms] (mean, across all concurrent requests)
Transfer rate:          50.07 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:      240  260  10.9    266     279
Processing:   350  752 443.5    457    1356
Waiting:      347  747 444.8    453    1354
Total:        600 1013 437.1    726    1606

Percentage of the requests served within a certain time (ms)
  50%    726
  66%   1555
  75%   1561
  80%   1572
  90%   1602
  95%   1604
  98%   1606
  99%   1606
 100%   1606 (longest request)
```

Listing 17: Output of 30 concurrent connections to the root of the web application

```
(...)

Document Path:          /test_asset
Document Length:        15600 bytes

Concurrency Level:      180
Time taken for tests:   102.233 seconds
Complete requests:      1380
Failed requests:        54
   (Connect: 0, Receive: 0, Length: 54, Exceptions: 0)
Keep-Alive requests:    1326
Total transferred:      21187354 bytes
HTML transferred:       20841574 bytes
Requests per second:    13.50 [#/sec] (mean)
Time per request:       13334.721 [ms] (mean)
Time per request:       74.082 [ms] (mean, across all concurrent requests)
Transfer rate:          202.39 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0  128 652.6      0    4900
Processing:     0 6618 17501.4    831   95570
Waiting:       54 1260 6032.2    193   49972
Total:          0 6745 17697.4    843   97127

Percentage of the requests served within a certain time (ms)
  50%    843
  66%   1348
  75%   1843
  80%   2547
  90%  21964
  95%  45340
  98%  86613
  99%  91950
 100%  97127 (longest request)
```

Listing 18: Output of 1380 requests with 180 concurrent connections to load the assets

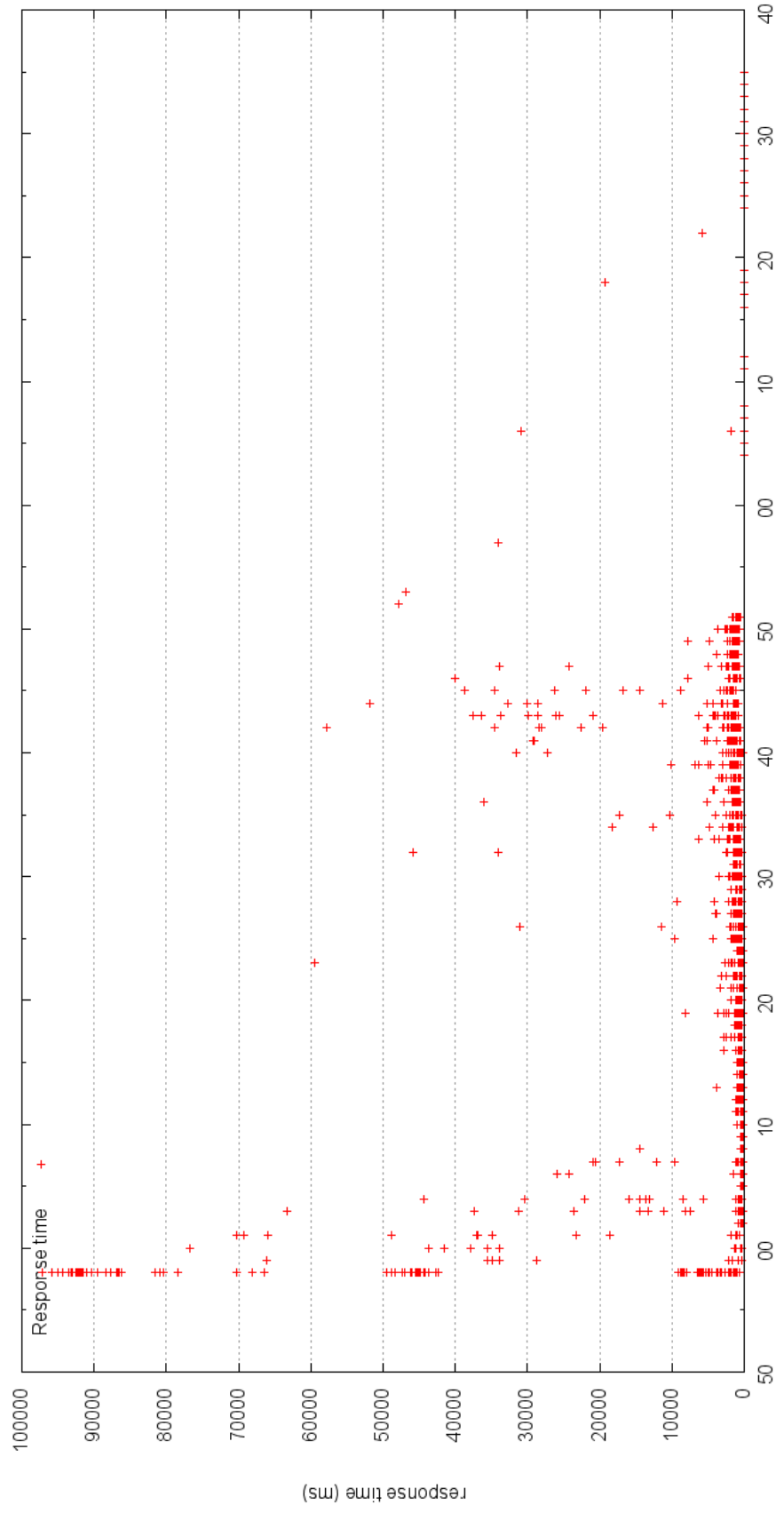



Figure 8-2: Response time plot of 1380 requests with 180 concurrent connections

```
(...)

Document Path:          /webapp/sos/rest/sensors
Document Length:        890 bytes

Concurrency Level:      1
Time taken for tests:   0.511 seconds
Complete requests:      1
Failed requests:        0
Total transferred:      1210 bytes
Total body sent:        3003
HTML transferred:       890 bytes
Requests per second:    1.96 [#/sec] (mean)
Time per request:       510.607 [ms] (mean)
Time per request:       510.607 [ms] (mean, across all concurrent requests)
Transfer rate:          2.31 [Kbytes/sec] received
                        5.74 kb/s sent
                        8.06 kb/s total

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       48   48   0.0     48      48
Processing:   463  463   0.0    463     463
Waiting:      462  462   0.0    462     462
Total:        511  511   0.0    511     511

(...)
```

Listing 19: Output of a POST request to /sensors

```
(...)

Document Path:          /webapp/sos/rest/observations
Document Length:        854 bytes

Concurrency Level:      1
Time taken for tests:   0.655 seconds
Complete requests:      1
Failed requests:        0
Total transferred:      1210 bytes
Total body sent:        2204
HTML transferred:       854 bytes
Requests per second:    1.53 [#/sec] (mean)
Time per request:       654.535 [ms] (mean)
Time per request:       654.535 [ms] (mean, across all concurrent requests)
Transfer rate:          1.81 [Kbytes/sec] received
                        3.29 kb/s sent
                        5.09 kb/s total

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       59   59   0.0     59      59
Processing:   595  595   0.0    595     595
Waiting:      592  592   0.0    592     592
Total:        654  654   0.0    654     654

(...)
```

Listing 20: Output of a POST request to /observations

```
(...)

Document Path:          /webapp/sos/rest/observations
Document Length:        1000 bytes

Concurrency Level:      10
Time taken for tests:   6.621 seconds
Complete requests:      60
Failed requests:        0
Non-2xx responses:      60
Total transferred:      70920 bytes
Total body sent:        132240
HTML transferred:       60000 bytes
Requests per second:    9.06 [#/sec] (mean)
Time per request:       1103.555 [ms] (mean)
Time per request:       110.355 [ms] (mean, across all concurrent requests)
Transfer rate:          10.46 [Kbytes/sec] received
                        19.50 kb/s sent
                        29.96 kb/s total

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       84  405 532.5    144    2382
Processing:   214  595 514.2    365    1916
Waiting:      213  595 514.2    365    1916
Total:        316 1000 672.8    611    2808

(...)
```

Listing 21: Output of 60 POST request to /observations with 10 concurrent connections

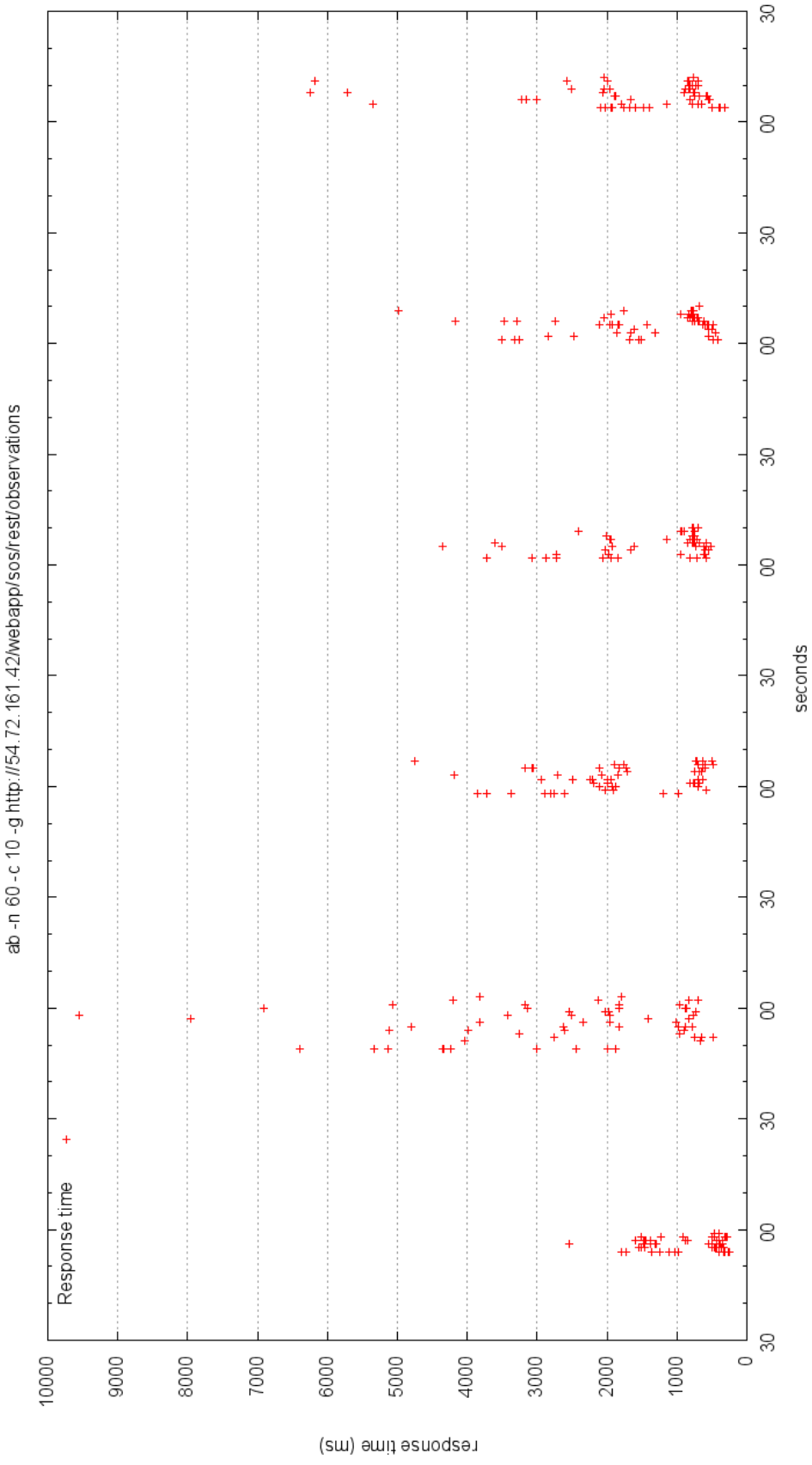



Figure 8-3: 6 executions of 10 concurrent connections with 1 minute timespan

```
(...)

Document Path:          /webapp/sos/rest/observations
Document Length:        1000 bytes

Concurrency Level:      30
Time taken for tests:   63.616 seconds
Complete requests:      500
Failed requests:        2
   (Connect: 0, Receive: 0, Length: 2, Exceptions: 0)
Non-2xx responses:      498
Total transferred:      588636 bytes
Total body sent:        1102000
HTML transferred:       498000 bytes
Requests per second:    7.86 [#/sec] (mean)
Time per request:       3816.968 [ms] (mean)
Time per request:       127.232 [ms] (mean, across all concurrent requests)
Transfer rate:          9.04 [Kbytes/sec] received
                        16.92 kb/s sent
                        25.95 kb/s total

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       53 1076 915.3    551    4895
Processing:   170 2477 2986.4   1296   21714
Waiting:        0 2391 2732.4   1205   19575
Total:        242 3554 3152.2   2658   22155

(...)
```

Listing 22: Output of 500 POST request to /observations with 30 concurrent connections

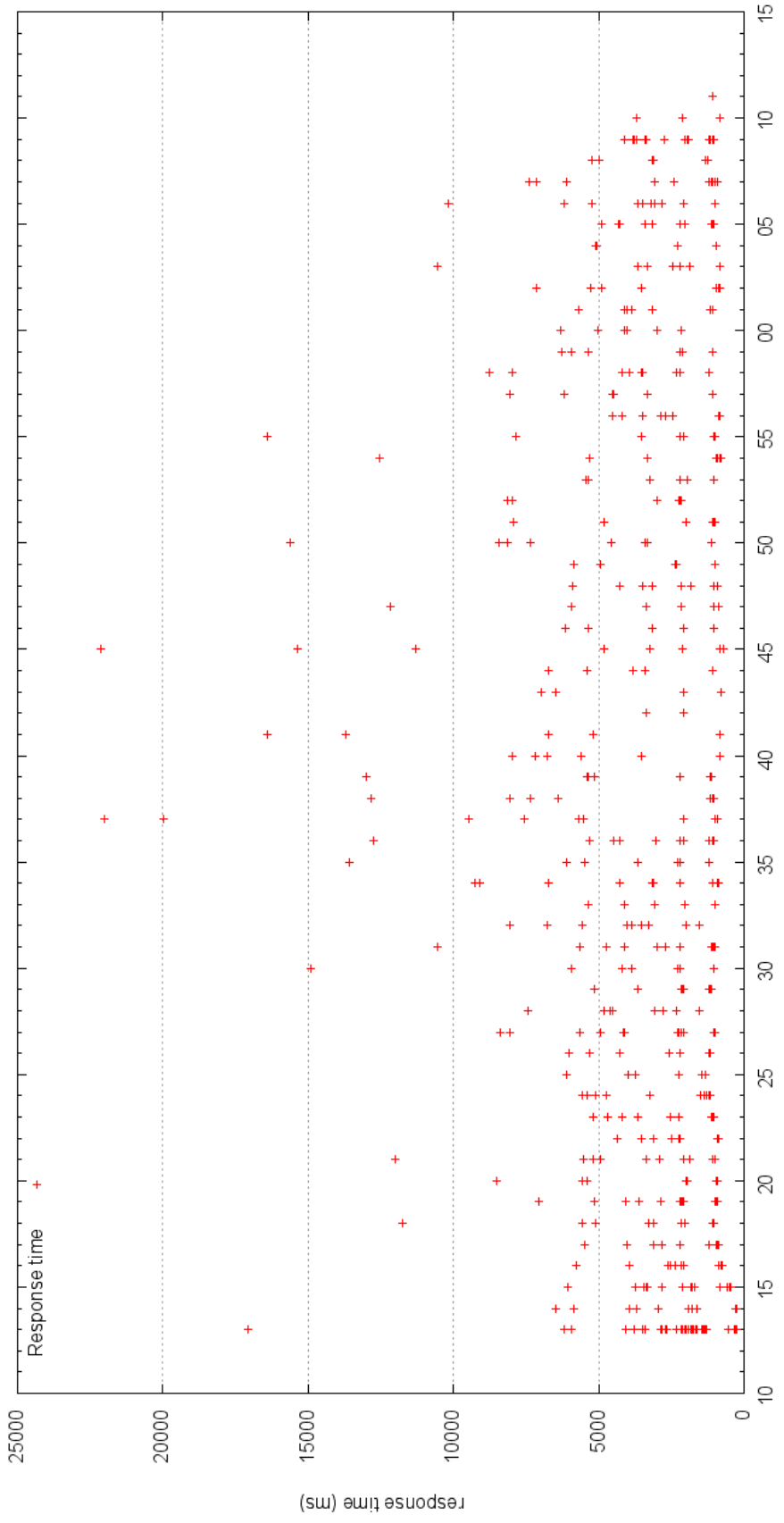


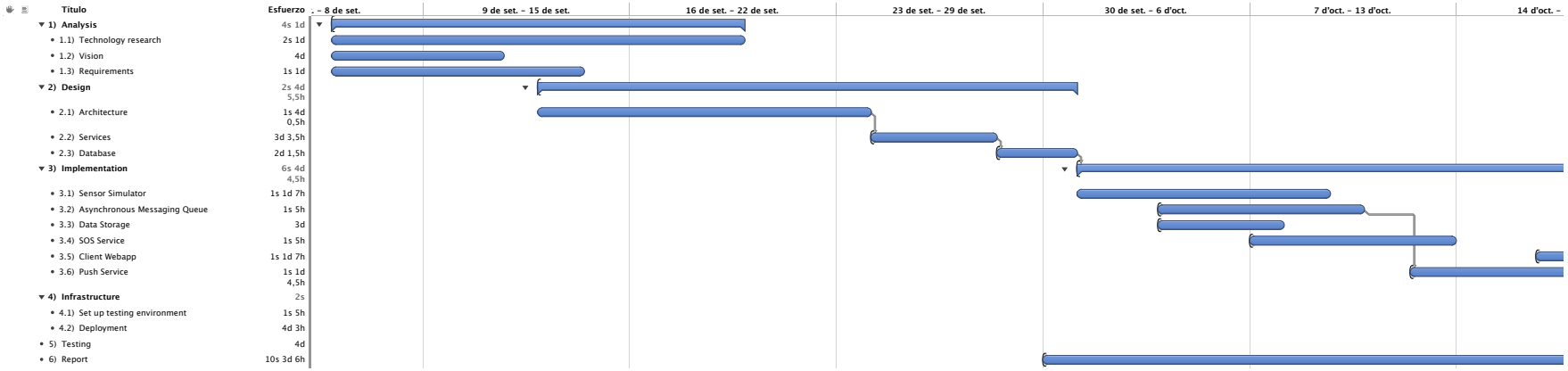

Figure 8-4: Response time plot of 500 requests with 30 concurrent connections

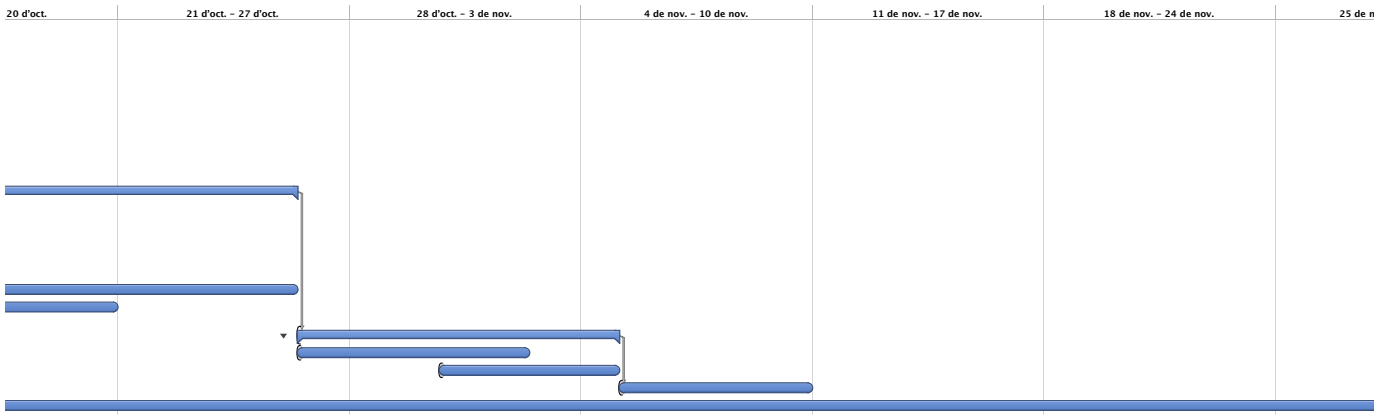
# Chapter 9

# Project Management

## 9.1 Planning

The project started with full-time dedication in September 2013 and was planned to be finished by December 2nd, 2013. The first steps were to study its feasibility in terms of technology, to outline the vision and specify the requirements with CREAF. Next steps included the design and implementation of the identified major components of the solution. It would end with the infrastructure setup, testing and the writing of the current document.


Título
Esfuerzo
. – 8 de set.
9 de set. – 15 de set.
16 de set. – 22 de set.
23 de set. – 29 de set.
30 de set. – 6 d'oct.
7 d'oct. – 13 d'oct.
14 d'oct. –
1) Analysis 4s 1d
1.1) Technology research 2s 1d
1.2) Vision 4d
1.3) Requirements 1s 1d
2) Design 2s 4d 5,5h
2.1) Architecture 1s 4d 0,5h
2.2) Services 3d 3,5h
2.3) Database 2d 1,5h
3) Implementation 6s 4d 4,5h
3.1) Sensor Simulator 1s 1d 7h
3.2) Asynchronous Messaging Queue 1s 5h
3.3) Data Storage 3d
3.4) SOS Service 1s 5h
3.5) Client Webapp 1s 1d 7h
3.6) Push Service 1s 1d 4,5h
4) Infrastructure 2s
4.1) Set up testing environment 1s 5h
4.2) Deployment 4d 3h
5) Testing 4d
6) Report 10s 3d 6h


20 d'oct.
21 d'oct. – 27 d'oct.
28 d'oct. – 3 de nov.
4 de nov. – 10 de nov.
11 de nov. – 17 de nov.
18 de nov. – 24 de nov.
25 de n

## 9.2 Cost

### Human Resources

The human resources involved in the project must be considered in order to forecast the costs of the project. These are an analyst who will be in charge of the Analysis and Design, a Developer who will implement the design and a System administrator who will set up the infrastructure. Therefore, considering these human resources and the initial planning, the overall cost is 15250€, as detailed below.

| **Resource** | **Cost/Hour** | **Hours** | **Cost** |
|---|---|---|---|
| Analyst | 40€/h | 200 | 8000€ |
| Developer | 25€/h | 250 | 6250€ |
| SysAdmin | 20€/h | 50 | 1000€ |
| **Overall** | | | **15250€** |

Table 9.1: Cost of human resources

Besides the time spent on the different stages of the development, additional time must be considered in order to write the current document. To that end, 90 additional hours plus the $200h + 250h + 45h = 495h$ invested by these human resources must be allocated, totalling $495h + 100h = 600h$.

### Material Resources

With regard to the software used in the development of the project, as all the frameworks, tools, code editors and languages have open-source licenses they don't involve any cost. Regarding the infrastructure, as it relies only on AWS free tier no cost is expected.

As for the development computer, the costs associated with its energy consumption plus its amortization must be taken into account. Being 1500 the cost of computer and an amortization in 4 years, its cost per hour would the number of work hours in these years divided by its overall cost, $1500€/8064h = 0.18€/h$. Considering that

the power consumption of the computer is $64W/h$ and that the current price of the energy is about $0.20€/kWh$, the cost of the energy consumed by the computer is $0.064KW/h \times 0.20€/kWh = 0.0128€/h$. The overall cost of the material resources is detailed in the table below.

| **Resource** | **Cost/Hour** | **Hours** | **Cost** |
|---|---|---|---|
| Computer | 0.18€/h | 600 | 108€ |
| Energy | 0.0128€/h | 600 | 7,68€ |
| **Overall** | | | **115,68€** |

Table 9.2: Cost of material resources

Lastly, taking into account human and material resources the total cost of the project amounts $15250€ + 115,68€ = 15365,68€$.

| **Resource** | **Cost** |
|---|---|
| Human | 15250€ |
| Material | 115,68€ |
| **Overall** | **15365,68€** |

Table 9.3: Total cost of the project

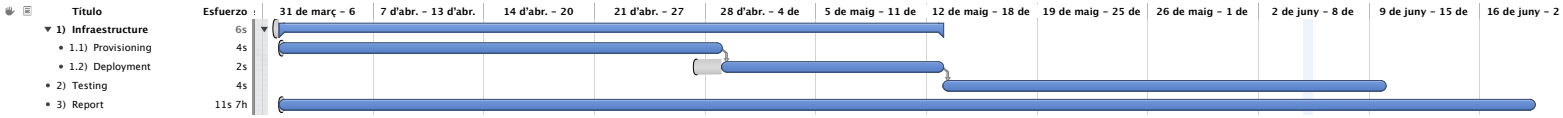
## 9.3 Execution

Unfortunately, initial planning has suffered a few setbacks during its execution. It was first delayed at the beginning of December due to my need of finding a job and the time I spent working on a technical test required for a job offer. The major delay was caused by the impact the full-time job had in the dedication time, causing the project to be nearly stopped for several weeks. Although working on it occasionally, it was not until allocating 2 hours every day and full-time dedication on weekends that the project took effectively off. As a consequence, the planning for the remaining tasks was defined as follows.

Regarding the cost, AWS free tier provided not to be enough to fulfil the needs of

the required infrastructure. Using three servers exceeded the maximum of 750 hours of EC2 micro instance usage. Thus, the overall cost has been 20.27$ so far, missing the cost of June 2014. Assuming that the cost for both months was the same the total cost of the infrastructure would be $2 \times 20.27\$ = 40,54\$$, that is 29.92€.

Moreover, another delay was encountered while executing this final planning. A disproportionately large Amazon AWS bill was received for what seemed to be either an attack on the system or a billing error. This required infrastructure to be shut down until the issue was resolved. Although having some impact, fortunately it didn't excessively affect planning.


Título
Esfuerzo
31 de març – 6
7 d'abr. – 13 d'abr.
14 d'abr. – 20
21 d'abr. – 27
28 d'abr. – 4 de
5 de maig – 11 de
12 de maig – 18 de
19 de maig – 25 de
26 de maig – 1 de
2 de juny – 8 de
9 de juny – 15 de
16 de juny – 2
1) Infraestructure
6s
1.1) Provisioning
4s
1.2) Deployment
2s
2) Testing
4s
3) Report
11s 7h

# Chapter 10

# Conclusions

## 10.1 Conclusions

The number of open-source software and modern web technologies used in this MS Thesis have proven to be a viable solution for building IT infrastructure for public research centers. These technologies have been combined together to build a distributed system as a proof-of-concept for REDCH, a larger initiative that aims to provide a valuable insight into the actual production of renewable energies at a small scale in Catalonia.

The introduction presents the motivations behind this initiative of the CREAF and outlines the main goals of this project.

Next, a thorough analysis defines the boundaries of this thesis by providing its scope and requirements. This is detailed further in Chapter 3 with a formal specification of the use cases and the whole conceptual model around the measurement and processing of observations.

Then, Chapter 4 details the findings of the research process that had been carried out to later support the design decisions taken in Chapter 5. These chapters are particularly relevant due to the fact that the chosen technologies are the basis for its further development.

Chapters 6 and 7 provide insight into the implementation and the infrastructure the system runs on. Particular attention is given to the automation of common

processes such as provisioning, deployment and maintenance tasks, which provide reliability and confidence to the system managers. Then, the evaluation of the resulting system in terms of performance is described in Chapter 8.

In spite of the difficulties that determining the scope of the product entailed, the final delimitation we came up with together with CREAF has proven to be adequate. It has been enough to explore each individual part of the project and demonstrate their potential. Specifically, although being rather simple the web application shows how a data visualization can be enriched with a full-featured application. As for the future sensors, the development of the simulator has allowed to better understand the challenges and requirements their design may involve.

On the other hand, the key point of using a messaging queue has been a very successful decision, in that has enabled a loosely coupled and scalable architecture that allows both ends of the queue, the SOS and the web application, to evolve independently. But as downside, this has brought some complexity that affects the resilience of the system. Implementing a more robust redundancy-based resilience mechanism would have improved the overall quality of the system.

As for the infrastructure, the complexity of setting the servers up surpassed the initial estimation causing a great impact on the time invested for that matter. While running the services in a development environment is often very easy, there are numerous variables involved when it comes to a production environment. Furthermore, it was the least-known of the fields involved in the project and the one that required the deepest understanding of the architecture. This led us to the conclusions that being the infrastructure critical for the proper functioning of the system, much attention has to be paid to the administration of the system. Otherwise, its impact on cost will increase as the time goes by.

Regarding the methodology, the outcome of the iterative development is a clean and maintainable codebase. A first iteration laid out each component and allowed to get the insight upon which the second iteration set the system up and ready.

Finally, all the goals of the project have been successfully reached and all the requirements in Chapter 2 were met. We are able to simulate sensors with a command-

line interface and the observations are stored, processed and displayed in real time.

## 10.2 Further Work

Considering the current state of the product, we identify some unresolved issues and steps that would be worth exploring in further research.

From the point of view of the implementation, there are a couple of aspects of the current architecture that would be interesting to investigate. First, given the event-driven nature of the web application's back-end, it may be worth replacing its implementation with Node.js. Its non-blocking I/O design that claims to maximize throughput and efficiency, makes it suitable for scalable networking applications. This seems to be a natural fit for the features of this project and may even surpass the EventMachine's high performance. However, this has not been possible due to the time constraints and our total lack of awareness of this platform.

Regarding the messaging queue, given that RabbitMQ's messages acknowledgement is not used may be beneficial to implement messaging with Redis instead. It is essentially a very high-performance key/value store for structured data that brings many other features such as pub/sub capabilities. These, however, don't include message acknowledgement for the sake of performance. Furthermore, Replacing RabbitMQ with Redis would enable to implement bulk observation retrieval thereby allowing to populate the map when a new browser is connected. Nevertheless, Redis pub/sub simplicity compared to RabbitMQ queue features may impact on future decisions as the system's usage grows.

From the infrastructure perspective, it may be beneficial at the early stages of the project to lean towards a Platfor-as-a-service (PaaS) hosting rather than the current IaaS. As a result, it would require far less systems administration knowledge and it would simplify deployments even more, but this comes at the expense of higher cost and less control over the product. In any case, this a possibility that is worth studying.

Finally, as next step, the system's poor performance must be addressed by switch-

ing to more reliable and powerful servers. Regarding the sensors, the physical sensor devices must be implemented considering the ideas brought in the research as the starting point. Besides, the SOS must be upgraded to the 52North SOS 4.0 final release.

After that, it would be recommended to start using the system with a small subset of real users while the ideas exposed above are considered prior to a public release. Meanwhile, it would be valuable to look for the involvement of public institutions, other research centers and specially the Open Geospatial Consortium so as to ensure the success of the project. Once a steady number of active users use the system, it would be the time to explore using AWS Auto Scaling. Finally, at a much later stage, the big data set would benefit from migrating to a NoSQL database, thereby increasing the scalability of the system.

# Appendix A

# Instruction Manual

## A.1 Web Application

### Installation

First, clone the repo:

```
git clone git@github.com:sauloperez/redch-webapp.git
```

Next, install its dependencies:

```
bundle install
```

REDCH Webapp gets observations from a RabbitMQ, so make sure the RabbitMQ server is running and accessible from within your network. For Mac OS X users this is done be typing:

```
rabbitmq-server
```

While for Ubuntu users this is done with:

```
sudo /etc/init.d/rabbitmq-server start
```

Besides, you must load the appropriate Procfile containing the values for the required env variables. It must contain the following:

```
AMQP_HOST=<RabbitMQ_server_host>
```

Name this file after the environment, e.g. `development` and save it wherever you like from your directory tree. You can edit the development file in `https://github.com/sauloperez/redch-webapp/blob/master/development`.

You can find further documentation in Process Types and the Procfile from Heroku Dev Center and from its Github repo.

## Usage

### Development

Now we are ready to start the webapp. From the root folder type the following in your terminal, using the path of your environment file:

```
foreman start -e <path_to_env_file>
```

Nevertheless, it is recommended to have a development environment file per machine ignored by git, so any customizations can be made for that machine.

That's all. The webapp is up and running. Point your browser to `http://localhost:3000` and you will see the real time map.

### Production

In production the deployment process is automatized using Capistrano. To deploy just type the following command from your machine:

```
cap production deploy
```

This essentially runs commands on the remote server through SSH. Once the process is finished, point your browser to the production server.

Additionaly, some tasks to manage production services are provided. Each service has its own start, stop and restart actions:

```
cap production nginx:restart
cap production passenger:stop
cap production rabbitmq:start

# Or open an SSH connection
cap production utils:ssh
```

To list all available tasks type:

```
cap production -T
```

### Testing

Testing covers both frontend and backend of the app. Jasmine has been chosen for the former while RSpec for the latter.

To test the frontend start up the server as stated above and point your browser to `http://localhost:3000/SpecRunner.html`. You will get immediate results of how many test are passing (hopefully all of them).

As for the backend, type the following in your terminal:

```
rspec spec
```

This will execute all tests contained in the `/spec` folder.

## A.2 CLI Client

### Installation

You must clone the repo

```
$ git clone git@github.com:sauloperez/redch.git
```

Although not mandatory, it is highly recommended to add the executable in your PATH environment variable. To do so, create a softlink in a suitable system

folder pointing to `/bin/redch` of the previously cloned repository. In Mac OS X `/usr/local/bin` might be a good choice. You can do that with the following command:

```
$ ln -s ~/redch/bin/redch /usr/local/bin/.
```

Then, make sure your PATH variable looks up into the folder that contains the softlink. If not, add it. Doing so `redch` will be globally accessible.

In case of working with Bash shell this should be set in the `~/.bash_profile` file. Find further details in .bash_profile vs .bashrc

```
# Prepend the variable with the right path
PATH="/usr/local/bin:/usr/local/sbin:$PATH"
```

Lastly, load the changes

```
$ source ~/.bash_profile
```

## Usage

The command-line interface comes with the methods `setup` and `simulate` that you can use as follows.

### Setup

If no coordinates are provided, the `setup` command will pick up a random location near by Trrega within a range of 90 Km as the sensor location. It uses the first MAC address of the system as device unique identifier.

```
$ redch setup -c '41.65, 2.13'
```

### Simulate

The `simulate` command loads the configuration set by the `setup` and issues randomly generated observations for each time period indefinitely. If not specified a period of 2 seconds will be used.

```
# Issue a post request each second
$ redch simulate -p 1
```

If the setup defaults suit you, you can skip the `setup` and just type the `simulate` command. It always executes the setup before the simulation if no configuration is found.

```
# Set up the sensor with its defaults and simulate observations
$ redch simulate
```

### Help

You can always list the available commands with the `help` command or the `-h` flag

```
$ redch help
Commands:
  redch help [COMMAND]  # Describe available commands or a single one
  redch setup     # Sets up the environment to enable the use of ...
  redch simulate  # Simulate a sensor generating electrical power ...
```

Or find out the details of a particular command

```
$ redch help setup
Usage:
  redch setup

Options:
  c, [--coordinates=COORDINATES]

Sets up the environment to enable the use of the device
```

## A.3 Sensor Observation Service

### Installation

#### Database

You must create a Postgres PostGIS database named `sos`. To do so, connect to postgres and create the database. Then, connect to it and add the PostGIS extension.

```
$ psql -h localhost


=# CREATE DATABASE sos;
=# \connect sos;
=# CREATE EXTENSION postgis;


# Quit from the DB
=# \q
```

#### AMQP Service extension

This SOS implementation has been extended to suit the requirements of the REDCH project. A RabbitMQ client wrapper has been added as extension.

Contained in the amqp-service submodule of the `extensions` module, it comes with an example properties file which can be found in the config folder. These properties are the following:

```
# URL of the RabbitMQ server
redch.amqp.host=192.168.0.20


# Name of the exchange to where the observations should be published
redch.amqp.exchange=observations
```

An updated copy of this file must be stored in the same folder named as `redch.properties`.

**Note:** Make sure you compile the `amqp-service` submodule whenever you change it before compiling the whole SOS. You can do so with the `mvn install` command.

**Integration** This service has been integrated with the SOS by using a subclass of the corresponding request handler. As every observation received must be published into the queue, the ObservationsPostRequestHandler with RedchObservationsPostRequestHandler of the rest binding, which connects to the AMQP Service.

## Usage

### Development

First of all, deploy the Java Webapp with the command `mvn clean tomcat:deploy` or `mvn clean tomcat:undeploy tomcat:deploy` if it already exists. Then configure it using the webapp. Just browse to <`tomcat_url`>`/webapp` and follow the wizard's steps.

REDCH SOS talks to RabbitMQ and Postgresql. So besides setting them up and running, make sure the `redch.amqp.host` property of the `redch.properties` file points to the right the RabbitMQ server. As for the DB, make sure the field `Host` under **Datasource settings** section of the SOS administrative backend points to the right host.

### Production

In production the deployment process is automatized using Capistrano. To deploy just type the following command from your machine:

```
cap production deploy
```

This essentially runs commands on the remote server through SSH. Once the process is finished, point your browser to <`production_server`>`/webapp` and fill up the wizard fields like in development.

In addition, it also provides tasks to manage Tomcat. You can list all available tasks typing:

```
$ cap production -T
...
cap tomcat:compile                # Compile tomcat
cap tomcat:deploy                 # Deploy tomcat
cap tomcat:restart                # Restart tomcat
cap tomcat:start                  # Start tomcat
cap tomcat:stop                   # Stop tomcat
cap tomcat:undeploy               # Undeploy tomcat
```

So, you can run any of these by tying, for instance:

```
cap production tomcat: restart
```

## Testing

The AMQPService extensions as well as the SOS itself come with unit tests made with JUnit. It is recommended to run them from your IDE of choice. Most of them have JUnit plugins available.

The tests can be found in the `src/test` folder of every module.

# Bibliography


[1] 52North SOS 4.0. `https://wiki.52north.org/bin/view/SensorWeb/SensorObservationServiceIVDocumentation`, 2014.

[2] Tim Berners-Lee. Html tags. *CERN*, 1991.

[3] Julian Browne. Brewer's CAP Theorem. `http://www.julianbrowne.com/article/viewer/brewers-cap-theorem`, January 2009.

[4] M. Cannata, M. Antonovic, M. Molinari, and M. Pozzoni. istSOS. *University of Applied Sciences of the South Switzerland*, 2009.

[5] Peter Deutsch. The eight fallacies of distributed computing. 1997.

[6] I. Fette and A. Melnikov. The WebSocket Protocol - RFC 455. `http://tools.ietf.org/html/rfc6455`, December 2011.

[7] Vincenzo De Florio. On the Constituent Attributes of Software and Organisational Resilience. *Interdisciplinary Science Reviews*, June 2013.

[8] Seth Gilbert and Nancy Lynch. Brewers Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *ACM SIGACT News*, 2002.

[9] Todd Hoff. What The Heck Are You Actually Using NoSQL For? *High Scalability*, December 2010.

[10] Nathan Hurst. Visual Guide to NoSQL Systems. `http://blog.nahurst.com/visual-guide-to-nosql-systems`, March 2010.

[11] Dexter Industries. Arduberry. `http://www.dexterindustries.com/Arduberry/`, 2014.

[12] Geographic information - observations and measurements. *OGC Abstract Specification*, September 2013.

[13] Douglas C. Schmidt. Reactor. An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events. 1995.

[14] The OGC's Sensor Web Enablement (SWE) Initiative. *Open Geospatial Consortium*.

[15] Server-sent events, w3c candidate recommendation. `http://www.w3.org/TR/2012/CR-eventsource-20121211/`, December 2012.

[16] Smart Citizen. `http://www.smartcitizen.me/`, 2012.

[17] Lukasz Strzalkowski. Queues. `http://queues.io`, 2014.

[18] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, second edition, 2007.

[19] Telefónica. Thinking things. `http://www.thinkingthings.telefonica.com/`, 2013.