Design and Evaluation of Scalable IoT Event Processing Platform

Juan Luis Pérez

Master in Computer Architecture, Network and Systems
Department of Computer Architecture
Universitat Politècnica de Catalunya

Advisor: David Carrera

December 2014
# Contents

1 Introduction 1
   1.1 Domain and Background 1
   1.2 Motivation 1
   1.3 Goals 3
   1.4 Document Structure 4

2 Background 7
   2.1 IoT, WoT 7
   2.2 Service Objects 8
   2.3 Related Work 9

3 Platform 13
   3.1 Architecture 13
   3.2 Jersey 14
   3.3 Jackson 15
   3.4 Jetty 15
   3.5 Couchbase 15
   3.6 Elasticsearch 16

4 Implementation details 17
   4.1 API implementation 17
      4.1.1 HTTP methods supported 19
   4.2 External dependencies 38
   4.3 Couchbase buckets and views 38
   4.4 Data model 39
      4.4.1 Initial Data Model 39
      4.4.2 Modified Data Model 44
   4.5 Search Capabilities 45
      4.5.1 Integration between Elasticsearch and Couchbase 45
      4.5.2 Elasticsearch Index for data queries 46
      4.5.3 Search Options 48
CONTENTS

5 Evaluation 53
5.1 Evaluation Methodology 53
5.2 Evaluation Infrastructure 54
5.3 Experiment 1: Scalability of the API with available resources 54
5.4 Experiment 2: Scalability of the API with load 56
5.5 Experiment 3: Scalability of the Data Store 58
5.5.1 Experiment 4: Performance limitations of the Indexing components 59

6 Conclusion 63
6.1 Summary 63
6.2 Suggestions for future work 64

Bibliography 65
List of Figures

2.1 Service Objects example .......................... 9

3.1 servIoTicy architecture diagram ................. 14

4.1 RESTful web service model ........................ 19
4.2 Authorization sequence diagram .................. 20
4.3 createSO sequence diagram ....................... 20
4.4 getAllISOs sequence diagram ...................... 22
4.5 getSO sequence diagram ............................ 23
4.6 getStreams sequence diagram ...................... 25
4.7 putSoData sequence diagram ....................... 27
4.8 getSOData sequence diagram ....................... 29
4.9 getSOData sequence diagram using the search engine 30
4.10 createSubscriptions sequence diagram .......... 31
4.11 getSubscriptions sequence diagram ............. 33
4.12 getSubscriptions sequence diagram using the search engine 35
4.13 getLastUpdate sequence diagram ................. 36
4.14 getLastUpdate sequence diagram using the search engine 37
4.15 API Couchbase buckets .......................... 38
4.16 API Couchbase buckets to support searches .... 39
4.17 XDCR configuration ............................... 46

5.1 API scalability with variations of the number of instances for the Web Tier and ElasticSearch ................. 56
5.2 API scalability with variations of the number of instances for the Web Tier and the Load Level ......................... 57
5.3 Resource consumption (CPU, Memory) for the REST API Web tier (Jetty) - Load Level 40 ................. 57
5.4 Distribution of response times (bins) - Couchbase tier ................. 59
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5</td>
<td>Variation of memory Heap usage for Load Level 40, one ElasticSearch instance and a variation of instances provisioned for the Web Tier</td>
<td>60</td>
</tr>
<tr>
<td>5.6</td>
<td>API scalability with variations of the memory Heap configuration for a set of three ElasticSearch instances, a variation of the instances provisioned for the Web Tier and a Load Level 40</td>
<td>61</td>
</tr>
</tbody>
</table>
List of listings

4.1 curl command to create SO ........................................... 21
4.2 Service Object JSON definition ..................................... 21
4.3 Create Service Object API response ............................... 21
4.4 curl command to get all SOs ........................................ 22
4.5 Get all SOs API response ............................................ 23
4.6 curl command to get a SO ........................................... 23
4.7 Get a SO API response ............................................... 24
4.8 Get SO JSON payload API response ................................... 24
4.9 curl command to get streams of a SO ............................. 25
4.10 Get streams API response ......................................... 26
4.11 Get streams JSON payload API response .......................... 26
4.12 curl command to put data into a SO ............................... 27
4.13 Service Object JSON data definition ............................. 28
4.14 Put data API response ............................................. 28
4.15 curl command to get data into a SO ............................... 29
4.16 Get data API response ............................................. 30
4.17 curl command to create subscriptions ............................ 31
4.18 Service Object JSON create subscription definition ............. 32
4.19 create subscriptions API response ................................. 32
4.20 curl command to get the subscriptions of a SO ................. 33
4.21 Put data API response ............................................. 34
4.22 Get subscriptions JSON payload API response .................... 34
4.23 curl command to last update SO ................................ 36
4.24 Put data API response ............................................. 37
4.25 Get last update JSON payload API response ..................... 37
4.26 Service Object creation data model .............................. 40
4.27 Service Object API stored data model ............................ 41
4.28 Sensor Update sample Data Model ............................... 42
4.29 Sensor Update Stream Data Model ............................... 42
4.30 Subscription Data Model .......................................... 43
4.31 Subscription Data Model .......................................... 43
| 4.32 Service Object API stored new data model | 44 |
| 4.33 Sensor Update Stream new data model     | 45 |
| 4.34 Elasticsearch index                     | 48 |
Chapter 1

Introduction

1.1 Domain and Background

The Internet of Things may be a hot topic in the industry but it’s not a new concept. In the early 2000’s, Kevin Ashton was laying the groundwork for what would become the Internet of Things (IoT) at MIT’s AutoID lab [21].

Actually, the Internet of Things (IoT) is an integrated part of the Future Internet and could be defined as a dynamic global network infrastructure with self configuring capabilities based on standard and interoperable communication protocols where physical and virtual ”things” have identities, physical attributes, and virtual personalities and use intelligent interfaces, and are seamlessly integrated into the information network.

In the IoT, ”things” are expected to become active participants in business, information and social processes where they are enabled to interact and communicate among themselves and with the environment by exchanging data and information ”sensed” about the environment, while reacting autonomously to the ”real/physical world” events and influencing it by running processes that trigger actions and create services with or without direct human intervention.

Interfaces in the form of services facilitate interactions with these ”smart things” over the Internet, query and change their state and any information associated with them, taking into account security and privacy issues.

1.2 Motivation

The increasing diffusion of electronic devices with identification, computing, and communication capabilities, such as mobile phones, digital cameras, and music players, is laying ground for the emergence of a highly distributed
service and networking environment. This environment, referred to as the Internet of Things (IoT), is strongly entangled with the physical realm and with the local context. Internet-connected smart objects, integrating RFIDs, sensors, and/or actuators, and deployed in the environment or in human artifacts (e.g., cars), will become the connectors between the ”virtual realm” of Information and Communication Technology (ICT) and the “physical realm”.

From a service and application perspective, the ability to seamlessly integrate the virtual and physical realms is expected to foster a major change in ICT systems. This will come from a profound and ubiquitous integration between two elements. On the one hand data will be coming from the environment and from functions that act on the physical reality. On the other hand, the virtual ICT entities (services) will be acting on the environment and on the functions.

We are now facing a tipping point, whereby, for the first time in history, almost anything can become digitally aware and interconnected. Smart grids, smart cities, smart financial systems, smart transportation—with so much technology available at reasonable cost, the list of possibilities is endless.

This project builds on this potential, as well as on two other trends taking place in the ICT field.

First, mobile personal devices are becoming more and more powerful in terms of processing and storage capabilities. Further, they are being equipped with a variety of sensors, such as accelerometers; sensors to measure temperature, humidity, and light; and video cameras. Such mobile personal devices will be seamlessly interconnected by means of a variety of wireless communication technologies, giving rise to complex and dynamic networks and also interacting with servers and applications residing in the cloud.

The second trend relates to the architecture of participation and the Internet of Services (IoS) approaches to service delivery. Architecture of participation, initially embodied by the so-called Web 2.0, encourages users to take an active role in the provision of content and services, as prosumers. Community-based solutions are emerging, whereby users share data coming from their smart devices, mostly sensors. The concept of Everything-As-A-Service enabled the introduction of new approaches for the reliable sharing of ICT infrastructural resources (e.g., computing, storage, and communication capabilities) and application functionalities. The Everything-As-A-Service approaches have so far, found only limited application in the IoT field.

This project aims at enabling and Event Processing Platform that can seamlessly integrate real and virtual worlds through the convergence of the Internet of Services with the Internet of Things. The project will achieve this through the provisioning of an open and scalable infrastructure, in which smart objects can be generated, managed and be searchable to easily build
1.3. GOALS

This project is framed within the servIoTicy [15] platform. servIoTicy is a state-of-the-art infrastructure for hosting Internet of Things (IoT) workloads in the Cloud. It provides multi-tenant data stream processing capabilities, a REST API, data analytics, advanced queries and multi-protocol support in a combination of advanced data-centric technologies.

servIoTicy is part of the development of the COMPOSE [26] project, which aims to develop a more ambitious IoT platform, not only focused on the data management and processing part, but including other aspects such as security, discovery of objects, development tools and composition engines.

This project presents a detailed characterization of the Front-End and Back-End (figure 3.1) of the servIoTicy platform and its design and implementation.

The sources of the servIoTicy are freely available as an open source project [7] in GitHub. The platform is also available for single node testing as a vagrant box, downloadable from a github repository [19]. The sources related to this project are the Front-End, available in the GitHub repositories [14][13][1] and all the Back-End configuration and deployment included in the vagrant box repository mentioned above.

It is worth mentioning that the components performed within the servIoTicy platform has been the mentioned Front-End and Back-End, its design and implementation and its connection with the processing runtime that has been developed by someone else.

1.3 Goals

This project address the problem of managing smart objects and their associated data with the scalability properties imposed by massive geo-distribution. The outcome will be widely accessible and interoperable smart objects, which can be exposed as services and can interact with higher-level services. The project will deal with the following issues:

- Design and implement of an architecture solution.
- Smart object abstract representation.
- Distributed object data store where data associated to objects will be stored, with tuneable availability and consistency.
- Interfaces for the interaction with smart objects:
CHAPTER 1. INTRODUCTION

- Interfaces for searching objects through the object registry as required from high-level services
- Interfaces and basic primitives (implemented using scalable data analytics technologies) for retrieving and processing data associated to Smart Objects, allowing for a mix of real-time and historic data. Services will be allowed to deploy custom search and processing methods through the use of query languages.

- Definition and reference implementation of a set of interfaces and APIs that will define how to access and manage Smart Objects, their associated data and to actuate on physical objects they represent.

- A detailed workload and resource characterization of the serIoTicy major components (REST API tier, Distributed Data Store and Indexing Engine) from a point of view of scalability with the available resources and with the load.

- An evaluation of the efficiency of CouchBase as a distributed data store in terms of response time delivered with the load.

- An insight on the performance impact of a proper configuration of the memory settings in ElasticSearch.

- Lay the groundwork for advanced cloud provisioning strategies and algorithms. The workload characterization described in this master thesis also offers an interesting insight for anybody interested on understanding the resource demands of modern indexing platforms such as ElasticSearch (section 3.6) or distributed data stores such as CouchBase (section 3.5).

Smart Objects will be stored in the Object registry, and their associated data will reside in the Object data store. Both of them will be developed starting from novel scalable distributed storage technologies (NoSQL) for structured and unstructured data. Data management will not be centralized, and novel methods for storing and managing high volume of data will be proposed. Advanced data analytics technologies, such as Elasticsearch, will be extended to go through the mix of real-time and historic data and produce knowledge that can be used by external services.

1.4 Document Structure

This document is organized as follows. Chapter 2 introduces the aspects of the Internet of Things and the Web of Things, explaining its basics and de-
scribing them and also introduces a set of abstractions defined for managing data associated to objects.

The next two chapters describe the actual work completed as part of this Master’s Thesis. On the one hand, chapter 3 introduces the general architecture and components of the platform. On the other hand, chapter 4 describes the main features of the REST API of the platform and its associated data models and its implementation.

The chapter 5 presents the evaluation methodology and the four experiments included in the documentation and its results and conclusions.

Finally, chapter 6 summarizes the results and proposes possible future work related to this project.
Chapter 2

Background

2.1 IoT, WoT

The approach to the IoT taken in this project is the one known as the Web of Things (WoT), where objects are able not only to communicate among themselves, but also they do so utilizing standard web-enabled protocols. Therefore, to communicate with the platform, objects need to be web-enabled (either directly or by using a proxy), and they have to implement the specific protocol to be integrated in the platform.

The Web of Things is a viable solution to build more scalable, open, and flexible IoT applications for various reasons: First, native integration of devices (as opposed to only integrate their data or using a Web page to control them) allows treating devices and their services just like any other Web resource; Second, native integration diminishes the costs to network heterogeneous devices as the Web infrastructure is already in place: HTTP is a highly versatile and omnipresent protocol thanks to its simplicity, powerful and scalable Web servers are freely available as open-source projects, HTTP clients and libraries exist for virtually any programming language and platform; And finally, Web applications are often simpler and faster to develop than classic desktop software applications. Current software for real-world integration and business applications are tailored for specific use cases, thus are often too rigid and closed to be customized by end-users easily.

Any Object which is web-enabled and implements the project communication protocol is known as a Web Object (WO). All WOs will hold a virtual identity in the platform, named a Service Objects (SO). SOs are standard internal platform representations of physical objects. The project specifies a SO API to communicate with WOs through the platform. The SO API is also exposed internally towards the rest of the components within the platform.
Service Objects can be used as mere data endpoints for WOs, but at the same time they can be used to aggregate and combine data coming from different SOs, creating dataflows that are exposed to the users. For instance, a Service Object could be created and deployed to represent aggregated operations across other existing SOs (e.g. calculating the maximum temperature across a large number of existing sensors registered in the platform).

2.2 Service Objects

The project assumes that each physical object (e.g. a smartphone) has different sensors (temperature, GPS, accelerometer...), and each sensor produces data that has one to several dimensions (e.g. wind speed has two dimensions: direction and strenght). Service Objects are the virtual representation of a physical object, and they exist in the platform. Every time one of the sensors of the physical object produces a new reading, it results in a sensor update (SU) being sent to the platform, and it is digested and processed by the Service Object associated to the physical object. In this virtual representation, each sensor is mapped to a sensor stream, and each sensor update contains a set of dimensions, known as channels. So a Service Object can be seen as a virtual entity which has different streams (as many as sensors) and each stream produces updates composed of a tuple of channels.

Each SO deployed in the platform is internally stored and described as a JSON document which contains all the necessary information to provide the data processing logic encapsulated in the SO. The processing logic is expressed using basic logical, string and arithmetic operators. The sources of data for the data processing logic can be Sensor Updates generated by a WO or SO, as well as the result of queries for data stored in the back-end. The communication between the different SOs in the data pipeline is driven by events, and the connections between them to create data paths is built through the use of subscriptions. SOs are deployed into the platform and later on accessed using a RESTful API.

Figure 2.1 illustrates an example in which one Web Object connected to the platform pushes data to be stored and processed. The WO is a Smartphone with GeoLocation capabilities (e.g. GPS enabled). The WO sends a Sensor Update (SU) to the platform everytime that wants to get its position reported. The SU is received in the platform through its virtual counter-part: the Service Object (SO). The platform web-based protocol is used through the SO API to push the SU into the platform. At that point, the platform determines that there is another SO which is subscribed to the data produced by the Smartphone. As such, it is subscribed to the data ingested by the
2.3. RELATED WORK

Figure 2.1: Service Objects example

first SO. The platform takes care of forwarding the SU to the second SO in the pipeline. The functionality of the second SO is to provide GeoFencing (e.g. determining whether a device is inside of a virtual fence defined by four different geographic coordinates). When the second SO gets the SU forwarded, it runs the SU information through its processing logic pipeline and emits another SU, this time containing a boolean value that takes value true when the SmartPhone position is located within the GeoFence. Note that this second SO provides data processing logic within the platform data plane, and that any external users, services or entities can decide to subscribe the second SO to obtain GeoFencing information about the SmartPhone.

The description of the SO includes a processing pipeline composed of different stages that need completed everytime that a Sensor Update is received by the SO. They include input and output data filtering, queries to the back-end and data transformation among others. The COMPOSE platform is responsible to parse the SO description and turn it into a computing entity within the data ingestion pipeline. Internally, the SO logic can access the data contained in the SUs through JSONPath \cite{25} expressions.

2.3 Related Work

Data Centric view of the IoT is not something new for servIoTicy as it was widely covered in the survey presented in \cite{29}. What servIoTicy uniquely provides is an open source solution that challenges the features of commercial solutions such as Xively \cite{20} and Evrythng \cite{6}, while extending their capabilities with the ability to inject user-defined code into its stream processing runtime.

There are other open source platforms for IoT in the market, but they are focused on other aspects of the Internet of Things. The DeviceHive \cite{3} project offers a machine-to-machine (M2M) communication framework for connecting devices to the Internet of Things. It includes easy-to-use Web-
based management software for creating networks, applying security rules and monitoring devices. Devicehub.net [4] is a cloud-based service that stores IoT-related data, provides visualizations of that data and allows users to control IoT devices from a Web page. The IoT Toolkit [8] project provides a variety of tools for integrating multiple IoT-related sensor networks and protocols. The primary project is a Smart Object API, but it also aims to develop an HTTP-to-CoAP Semantic mapping. Mango [4] is a popular open source Machine-to-Machine (M2M) software, which is web-based and supports multiple platforms. Key features include support for multiple protocols and databases, and user-defined events among others. Nimbits [11] can store and process a specific type of data previously time- or geo-stamped. A public platform as a service is available, but it can also be downloaded and deployed on Google App Engine, any J2EE server on Amazon EC2 or on a Raspberry Pi. OpenRemote [12] offers four different integration tools for home-based hobbyists, integrators, distributors, and manufacturers. It supports dozens of different existing protocols, allowing users to create nearly any kind of smart device they can imagine and control it using any device that supports Java. The SiteWhere [16] project provides a complete platform for managing IoT devices, gathering data and integrating that data with external systems. SiteWhere releases can be downloaded or used on Amazon’s cloud. It also integrates with multiple big data tools, including MongoDB and ApacheHBase. Finally, ThingSpeak [18] can process HTTP requests and store and process data. Key features of the open data platform include an open API, real-time data collection, geolocation data, data processing and visualizations, device status messages and plugins.

Deployment of IoT platforms on the Cloud is also covered in the literature. In [23], authors propose strategies for deciding the best approach at the time of making cloud-based deployments of IoT applications using nowadays regular cloud technologies. Another recent work [22] studies the implementation of IoT platforms on top of cloud-based pub/sub communication infrastructures. Finally, authors go one step beyond in [30] by leveraging completely Software Defined Environments for managing the Cloud infrastructures in which IoT applications are deployed.

Few studies present both a characterization of workload and resource consumption for web applications. In [28] Patwardhan et al. perform a CPU Usage breakdown of popular Web benchmarks with emphasis on networking overhead, identifying that network overhead for dynamic applications is negligible, while not for static content. In [31] Ye and Cheng present a similar characterization of resource utilizations as the one presented here, but for Online Multiplayer Games. The work presented in this paper is, to our knowledge, the first performance characterization of a Data Centric IoT
2.3. RELATED WORK

platform for the Cloud.
Chapter 3

Platform

The main focus of the platform data plane is to provide a rich set of features to store and process data through a simple REST API, allowing objects, services and humans to access the information produced by the devices connected to it. The platform described on this project allows for a real time processing of device-generated data, and enables for simple creation of data transformation. Unlike traditional service composition approaches, usually focused on addressing the problems of functional composition of existing services, one of the goals of the platform data plane is to focus on data processing scalability.

As described in more detail below, the Front-End of platform is a Web Tier that implements the REST API that sits at the core the platform. The API contains part the logic of the Service Objects and Data Processing pipes, related to authentication, data storage and data retrieval actions. The data Back-End includes the Data Store that provides scalable, distributed and fault-tolerant properties to the platform, and the Indexing Engine that provides search capabilities across sensors data using different criteria, like timestamps, string patterns or geolocation.

The REST API authenticates the users via an API key. The API key is provided via a pre-authentication mechanism, a simple user-password login web application. The API key is provided for every request and is passed as a query string parameter or via an HTTP header. More detailed information about HTTP authentication can be found in RFC 2617 [24].

3.1 Architecture

A general view of the servIoTicy architecture can be seen in Figure [3.1] In particular it consists of a web Front-End that is a Web Tier that implements
CHAPTER 3. PLATFORM

the REST API that sits at the core of servIoTicy and a data Back-End (CouchBase 3.5) in which both the SO data and metadata will be stored, being the former the SO data repository and the latter the SO registry. As the system will be oriented to IoT stream processing, the central data ingestion component will be a scalable stream ingestion topology (STORM [17]) that processes incoming SU’s in real time while dispatching subscriptions and queries. For advanced text-based search, the data back-end is connected with a search engine platform (Elasticsearch 3.6).

The platform leverages different components to implement the logic of Service Objects and Subscriptions. In this Section we describe in more detail the main properties of each component of the platform architecture related with this master thesis.

Figure 3.1: servIoTicy architecture diagram

3.2 Jersey

The API Web Services are developed using a Java programing language API, JAX-RS (Java Api for RESTful Web Services). JAX-RS provides some an-
notations to aid in mapping a resource class as a web resource. In addition, it provides further annotations to method parameters to pull information out of the request. Jersey is the REST framework providing the JAX-RS Reference Implementation and more. Jersey provides its own APIs that extend the JAX-RS toolkit with additional features and utilities to further simplify RESTful service and client development.

3.3 Jackson

As a JSON processor Java library we use the [Jackson Project][9]. Jackson is a High-performance suite of data-processing tools for Java, including the flagship JSON parsing and generation library, as well as additional modules. The Jackson Project also has handlers to add data format support for JAX-RS implementations like Jersey.

3.4 Jetty

As a HTTP Web Server a Java Servlet container we use Jetty [10]. Jetty is a pure Java-based HTTP server and Java Servlet container. Jetty is often used for machine-to-machine communications, usually within larger software frameworks. Jetty is developed as a free and open source project.

3.5 Couchbase

The API data backed is the CouchBase Server [2]. CouchBase Server is an open source, distributed NoSQL document-oriented database that is optimized for interactive applications. Unlike traditional relational databases, CouchBase stores information in documents rather than table rows.

CouchBase Server has native support for JSON documents. A document is a JSON object consisting of a number of key-value pairs that you define. There is no schema in CouchBase; every JSON document can have its own individual set of keys. Each JSON document can have a different structure, and multiple documents with different structures can be stored in the same CouchBase bucket. Document structure can be changed at any time, without changing other documents in the database.
3.6 Elasticsearch

The API Web Services uses Elasticsearch as search engine. Elasticsearch is a distributed search engine built for the cloud. It provides a distributed, multitenant-capable full-text search engine with a RESTful web interface and schema-free JSON documents. Elasticsearch is developed in Java and is released as open source. Elasticsearch can be used to search all kinds of documents. It provides scalable search, has near real-time search, and supports multi-tenancy.

Elasticsearch is connected with CouchBase to provide search capabilities. The integration between Couchbase and Elasticsearch enables fulltext search, indexing and querying and real-time analytics for variety of use cases such as a content store or aggregation of data from different data sources. This searches are implemented in different parts of the API, for example, the Map Demo uses spatial queries.
Chapter 4

Implementation details

The Web Tier for the REST API is composed of a Servlets Container and a REST Engine. The API contains part the logic of the Service Objects and Data Processing pipes, related to authentication, data storage and data retrieval actions.

The chapter first describes the main features of the API and its described implementation using a general web service model diagram and sequence diagrams for each operation. The chapter follows describing the external dependencies, the Couchbase bucket configuration and a description of the data model used and its modification to support searches.

4.1 API implementation

Service Objects are exposed through a RESTful API that uses HTTP as a transport and that acts as the SO front-end. This basically implies that SOs can be identified unambiguously using uniques URIs. The API provides resource actuations through the four main HTTP operations: GET (retrieve), POST (create), PUT (update) and DELETE. Table 4.1 summarizes the COMPOSE Service Objects API. In the table, soId represents the unique ID associated to each Service Object registered in the platform.
### CHAPTER 4. IMPLEMENTATION DETAILS

<table>
<thead>
<tr>
<th><strong>REST</strong></th>
<th><strong>Target URI</strong></th>
<th><strong>Role</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ServiceObjects</td>
<td>/</td>
<td>createSO, getAllSOs</td>
</tr>
<tr>
<td>ServiceObjectsId</td>
<td>/{soId}</td>
<td>getSO</td>
</tr>
<tr>
<td>Streams</td>
<td>/{soId}/streams</td>
<td>getStreams</td>
</tr>
<tr>
<td>StreamId</td>
<td>/{soId}/streams/{streamId}</td>
<td>putSOData, getSOData</td>
</tr>
<tr>
<td>Subscriptions</td>
<td>/{soId}/streams/{streamId}/subscriptions</td>
<td>createSubscriptions, getSubscriptions</td>
</tr>
<tr>
<td>lastUpdate</td>
<td>/{soId}/streams/{streamId}/lastUpdate</td>
<td>getLastUpdate</td>
</tr>
</tbody>
</table>

| **Table 4.1: API operations** |
|-----------------------------|----------------------------------|
| REST                        | **Target URI**                   | **Role**                  |
| ServiceObjects              | /                                | createSO, getAllSOs       |
| ServiceObjectsId            | /{soId}                          | getSO                     |
| Streams                     | /{soId}/streams                  | getStreams                |
| StreamId                    | /{soId}/streams/{streamId}       | putSOData, getSOData      |
| Subscriptions               | /{soId}/streams/{streamId}/subscriptions| createSubscriptions, getSubscriptions |
| lastUpdate                  | /{soId}/streams/{streamId}/lastUpdate| getLastUpdate             |
4.1. API IMPLEMENTATION

4.1.1 HTTP methods supported

What follows is an explanation of all the operations supported by the API. We show its sequence diagram and show how to use it with the curl command line application. All the API operations are filtered to provide authorization mechanism, the API is authorizing the requests using a user database where the Authorization header is validated. In the following sequence diagram we show the process of validating an API request.

Each request method (GET, POST, PUT) must have a Request header with the provided user authorization key, the API_KEY. With this API_KEY, the engine generates the authorization mechanism querying to the User Database by the userId associated with this API_KEY.
CHAPTER 4. IMPLEMENTATION DETAILS

Figure 4.2: Authorization sequence diagram

createSO
Create a new Service Object definition and register it in the repository. The unique ServiceObject id (soId) is returned on success.

Figure 4.3: createSO sequence diagram

Creating a Service Object is done by posting a SO definition JSON document. The process of creating a Service Object is shown below via curl command line tool.

Request:

```bash
> curl -i -X POST -H "Content-Type: application/json" -H "Authorization: 1YmI4MGEtNTNlMC00MWFhLWEzN2UtYWI2NzEwM2FjY" -d @so.json http://api.servioticy.com/
```
4.1. API IMPLEMENTATION

Listing 4.1: curl command to create SO

Where the definition of a Service Object is as follows:

```json
{
    "name": "Phone",
    "description": "COMPOSE phone",
    "public": "false",
    "URL": "Web Object URL",
    "streams": {
        "position": {
            "channels": {
                "location": {
                    "type": "geo_point",
                    "unit": "degrees"
                }
            },
            "description": "GPS outdoor location",
            "type": "sensor"
        }
    },
    "customFields": {},
    "actions": [],
    "properties": []
}
```

Listing 4.2: Service Object JSON definition

In section 4.4 can be found more information about the Service Object data model.

And the API response is:

```
HTTP/1.1 201 Created
Date: Wed, 05 Mar 2014 15:57:20 GMT
Server: api.servIoticy
Location: http://api.servioticy.com/139403523318497be978754164c27b07e748b53079662
Content-Type: application/json
Transfer-Encoding: chunked
{
    "id": "139403523318497be978754164c27b07e748b53079662",
    "createdAt": 1394035233184
}
```

Listing 4.3: Create Service Object API response

The response includes a JSON payload with the Service Object id and the creation timestamp.
**getAllSOs**

Retrieve all the Service Objects from a given user (identified by the Authorization header).

**Figure 4.4: getAllSOs sequence diagram**

HTTP clients can issue a GET on the root URL and receive all their Service Objects. As we have previously explained, the user authorization is provided by the Authorization header (see section 4.1.1).

In the next listing we show the process of querying the user Service Object via curl command line tool appending the Authorization header via -H flag:

```
> curl -i -H "Authorization: 1YmI4MGEtNTN1MC00MWFhLWEzN2UtYWI2NzEwM2FjY" http://api.servioticy.com
```

**Listing 4.4: curl command to get all SOs**

This GET on the root URL should return the list of the entire user Service Objects. Each Service Object is identified by its id.

The API response is of the form:

```
HTTP/1.1 200 OK
Date: Wed, 05 Mar 2014 16:24:17 GMT
Server: api.servioticy
Content-Type: application/json
Transfer-Encoding: chunked
```
4.1. API IMPLEMENTATION

Listing 4.5: Get all SOs API response

The response JSON payload is an array of the Service Object Ids.

getSO

Retrieve the description of a Service Object.

A GET request to the Service Object id will return its full description. Following the curl command line example:

```bash
> curl -i -H "Authorization: 1YmI4MGEtNTN1MC00MWFhLWEzN2UtYWI2NzEwM2FjY" http://api.servioticy.com/139403523318497be978754164c27b07e748b53079662
```

Listing 4.6: curl command to get a SO

This GET on the root URL of a SO id should return:
CHAPTER 4. IMPLEMENTATION DETAILS

Listing 4.7: Get a SO API response

In the JSON payload we obtain the SO description:

```
{
  "id": "139403523318497be978754164c27b07e748b53079662", "public": "false",
  "createdAt": 1396521151696,
  "updatedAt": 1396521151696,
  "name": "Phone",
  "description": "COMPOSE phone",
  "URL": "Web Object URL",
  "streams": {
    "position": {
      "channels": {
        "location": {
          "type": "geo_point",
          "unit": "degrees"
        }
      },
      "description": "GPS outdoor location",
      "type": "sensor"
    },
    "customFields": {},
    "actions": [],
    "properties": []
  }
}
```

Listing 4.8: Get SO JSON payload API response

With the API generated fields: id, createdAt and updatedAt.
4.1. API IMPLEMENTATION

getStreams

Retrieve the description of the streams in the Service Object definition document.

Figure 4.6: getStreams sequence diagram

The Streams of a Service Object are obtained making a GET request to the URL of the Service Object Id.

In the next listing we show the process of querying the Streams of a Service Object via curl command line tool:

```
> curl -i -H "Authorization: iYmI4MGEtNTNlMC00MWFhLWEzN2UtYWI2NzEwM2FjY http://api.servioticy.com/139403523318497be978754164c27b07e748b5307 9662/streams
```

Listing 4.9: curl command to get streams of a SO

This GET on the root Streams URL will return the list of all the information of each Service Object’s Stream.

```
HTTP/1.1 200 OK
Date: Mon, 07 Apr 2014 06:17:08 GMT
Server: api.servioticy
Content-Type: application/json
```
CHAPTER 4. IMPLEMENTATION DETAILS

Listing 4.10: Get streams API response

In the JSON payload we obtain the Streams description:

```
{
    "streams": [
        {
            "name": "position",
            "channels": {
                "location": {
                    "type": "geo_point",
                    "unit": "degrees"
                },
                "description": "GPS outdoor location",
                "type": "sensor"
            }
        }
    ]
}
```

Listing 4.11: Get streams JSON payload API response

**putSOData**

Store (and process) new data associated to the stream streamId of the ServiceObject soId.
Putting a data definition JSON document to the stream URL does appending data to the Stream of a Service Object. The process of appending data to a Service Object is shown below via curl command line tool.

Request:

```
> curl -i -H "Authorization: 1YmI4MGEtNTNlMC00MWFhLWEzN2UtYWI2NzEwM2FjY" -d @data.json http://api.servioticy.com/139403523318497be978754164c27b07e748b53079662/stream(position
```

Listing 4.12: curl command to put data into a SO

Where the data definition is as follows:

```json
{
   "channels": {
      "location": {
         "current-value": "50.818395, 4.40313"
      }
   }
}
```
CHAPTER 4. IMPLEMENTATION DETAILS

Listing 4.13: Service Object JSON data definition

And the response:

```
HTTP/1.1 202 Accepted
Date: Mon, 07 Apr 2014 06:33:46 GMT
Server: api.servIoTicy
Content-Type: application/json
Transfer-Encoding: chunked
{"channels":{"location":{"current-value":"50.818395,4.40313"}},"lastUpdate":1199192935}
```

Listing 4.14: Put data API response

getSOData

Retrieve all the data stored in the platform that is associated to one particular stream (streamId) of the ServiceObject.
4.1. API IMPLEMENTATION

A GET request to the Stream of a Service Object id will return all the data appended to this Stream.

In the next listing we show the process of querying the Stream data via curl command line tool:

```bash
> curl -i -H "Authorization: 1YmI4MGEtNTNlMC00MWFhLWExNy1hZDgzNi9yMjQ4ZDJlNDcyNTg3ZjIyZTA0MGIwY2FmMjE=" http://api.servioticy.com/139403523318497be978754164c27b07e7488b53079662 /streams/position
```

Listing 4.15: curl command to get data into a SO

This GET on the Service Object Stream should return a list with all the data associated it.

The API response is of the form:

```
HTTP/1.1 200 OK
Date: Mon, 07 Apr 2014 06:48:55 GMT
```
Listing 4.16: Get data API response

As can be seen, each data sample is of the form appended previously. After implementing the search functionalities via ElasticSearch API integration, the sequence diagram is as follows.

Figure 4.9: getSOData sequence diagram using the search engine

createSubscriptions

It creates a new external subscription for the stream streamId of the ServiceObject soId.
Subscribing to a stream updates are done by posting a subscription request to the subscriptions resource of a Service Object.

```
> curl -i -H "Authorization: 1YmI4MGEtNTNlMC00MWFhLWEzN2UtYWI2NzEwM2FjY" -d @subscription.json http://api.servioticy.com/139403523318497be978754164c7b07e748b53079662/streams/position/subscriptions
```

Listing 4.17: curl command to create subscriptions

Where the definition of a subscription is as follows:

```json
{
  "callback": "http",
  "destination": "http://somewhere.com:5000/update/@location@",
  "customFields": {
    "aliases": [
      {
        "###": "{$.channels.",
```
"!!": ".current-value}"
},
{
  "@location@": "##location!!"
}
],
"method": "GET"
}

Listing 4.18: Service Object JSON create subscription definition

With the callback type, the destination of the subscription and the customFields, where is placed the information for the destination.

The API response is:

Listing 4.19: create subscriptions API response

The response includes a JSON payload with the Subscription id and the creation timestamp.

getSubscriptions

Retrieve the subscriptions associated with a Service Object stream.
4.1. API IMPLEMENTATION

Figure 4.11: getSubscriptions sequence diagram

A curl command line demonstrating how to get the subscriptions for a Service Object.

```
> curl -i -H "Authorization: 1YmI4MGEtNTN1MC00MWFhLWEzN2UtYWI2NzEwM2FjY" http://api.servIoTicy.com/139403523318497be978754164c27b07e748b53079662 /streams/position/subscriptions
```

Listing 4.20: curl command to get the subscriptions of a SO

Obtaining:

```
HTTP/1.1 200 OK
Date: Mon, 07 Apr 2014 07:40:13 GMT
Server: api.servIoTicy
Content-Type: application/json
Transfer-Encoding: chunked

{"subscriptions": [["id": "1396856076227c89bb779d41d4c899ad615e256a7dc96", "createdAt": 1396856076227, "updatedAt": 1396856076227, "callback": "http", "source": "13965211516962decf0706f4bde92ee668a910
```
Listing 4.21: Put data API response

In the JSON payload we obtain the Subscriptions description:

```json
{
    "subscriptions": [
        {
            "id": "1396856076227c89bb779d41d4c899ad615e256a7dc96",
            "createdAt": 1396856076227,
            "updatedAt": 1396856076227,
            "callback": "http",
            "source": "139403523318497be978754164c27b07e748b53079662",
            "destination": "http://somewhere.com:5000/update/@location@",
            "stream": "position",
            "customFields": {
                "aliases": [
                    {
                        "###": "${.channels.",
                        "!!": ".current-value}"},
                    {
                        "@location@": "##location!!"
                    }
                ],
                "method": "GET"
            }
        }
    ]
}
```

Listing 4.22: Get subscriptions JSON payload API response

We can see, among other things, the createdAt and updatedAt timestamps and the source (the Service Object owner of the subscription).

After implement the search functionalities via ElasticSearch API integration the sequence diagram is as follows.
getLastUpdate

Retrieve the last sample data stored in the platform that is associated to one particular stream (streamId) of the ServiceObject.
GETs request to a last stream update are done by getting a request to the lastUpdate resource of a Service Object.

```bash
> curl -i -H "Authorization: iYmI4MGEtNTNlMC00MWFhLTIzN2UtYWI2NzEwM2FjY" http://api.servioticy.com/139403523318497be978754164c27b07e748b53079662 /streams/position/lastUpdate
```

Listing 4.23: curl command to last update SO

In this case the API response is of the form:

```
HTTP/1.1 200 OK
Date: Mon, 07 Apr 2014 08:39:48 GMT
Server: api.servioticy
Content-Type: application/json
Transfer-Encoding: chunked
"data": [{"channels": {"location": {"current-value": "51.918395, 4.46313"}, "lastUpdate": 1199192945}}]
```
Listing 4.24: Put data API response

And as can be seen in the JSON response payload we found the sample:

```
{
    "channels": {
        "location": {
            "current-value": "50.745395, 4.51313"
        }
    },
    "lastUpdate": 1199192995
}
```

Listing 4.25: Get last update JSON payload API response

After implementing the search functionalities via ElasticSearch, the API sequence diagram is as follows.

Figure 4.14: getLastUpdate sequence diagram using the search engine
CHAPTER 4. IMPLEMENTATION DETAILS

4.2 External dependencies

Most of the client libraries used in the project are related with the external resources and open source software included.

- **com.sun.jersey**: Java client libraries used to include the REST framework Jersey.
- **com.fasterxml.jackson**: Client libraries used to include the High-performance JSON processor Java library Jackson.
- **com.couchbase.client**: The CouchBase java client used to access the NoSQL database.
- **org.elasticsearch**: The ElasticSearch java client used to access the query back-end.

4.3 Couchbase buckets and views

As is explained previously the data backend is provided using the document based NoSQL database, Couchbase. Couchbase Server provides data management services using buckets; these are isolated virtual containers for data. A bucket is a logical grouping of physical resources within a cluster of Couchbase Servers.

In the initial version for API data backend we have two buckets:

- The first one is named “serviceobjects” and supports the storage for all the API components; the Service Objects, its data samples and its subscriptions.
- The other one is named “privatebucket” and supports the storage for the internal API components, for example, the temporary data operational information.

![Figure 4.15: API Couchbase buckets](image-url)
4.4. DATA MODEL

After this first approach, to support searches via ElasticSearch it was necessary to extend the number of buckets in order to permit ElasticSearch queries for Service Object Data documents individually.

Now there are three more buckets:

- "soupdates" to support the storage of all the data samples of all the Service Objects, each of one in an individual document to permit ElasticSearch queries.

- "subscriptions" to support the storage of all the subscriptions.

- "actuations" to support the storage of the actuations.

4.4 Data model

In this section we shown the data model used by the API. Basically, the JSON payload sent to the API to store Service Objects, Sensor Updates and Subscriptions and how they are stored.

4.4.1 Initial Data Model

The first data model shown is a Service Object, it is the core data structure. In the table below we show the basic JSON document of a Service Object created in section 4.1.1.
For this Service Object we have one Stream, the position, with one channel, the location. This Service Object has been defined as public, which means that it can be accessed for every Compose user.

```
{
    "name": "Phone",
    "description": "COMPOSE phone",
    "public": "false",
    "URL": "Web Object URL",
    "streams": {
        "position": {
            "channels": {
                "location": {
                    "type": "geo_point",
                    "unit": "degrees"
                }
            },
            "description": "GPS outdoor location",
            "type": "sensor"
        }
    },
    "customFields": {},
    "actions": [],
    "properties": []
}
```

Listing 4.26: Service Object creation data model

The Service Objects are stored in the engine with additional meta information.

The API assigns a unique identifier at creation time, the id, which is necessary to access it.

All the timestamps, the `creationAt` and the `updatedAt` fields, are stored UNIX time format (i.e. the number of seconds that have elapsed since 00:00:00 Coordinated Universal Time, UTC, Thursday, 1 January 1070).

As outlined below, all the Sensor Updates are stored in one single document. The data field is the reference where the engine stores this link between the stream and its Sensor Updates.

We also can see the subscription array field that is the link between a Service Object Stream and its entire Subscriptions definition document.

```
{
    "id": "139403523318497be978754164c27b07e748b53079662",
    "userId": "2d8d385681024a1a80f0f14d2179b1b9",
}
```
4.4. DATA MODEL

Listing 4.27: Service Object API stored data model

The Sensor Update data model has not modification from the data sent to the API engine to the data stored. No meta information is added by the API engine. Basically the API engine grabs each Sensor Update sample and appends it to a list with all others. Each individual sample of this list is referenced by the lastUpdate Sensor Update field.

In Listing 4.28 we can see a Service Object Stream sensor update sample and in Listing 4.29 the list of Sensors Updates stored by the API engine.

```json
{}
```

```json
{
    "channels": {
        "location": {
            "current-value": "50.745395, 4.51313"
        }
    },
    "lastUpdate": 1199192995
}
```
CHAPTER 4. IMPLEMENTATION DETAILS

Listing 4.28: Sensor Update sample Data Model

```json
{
  "1199192935": {
    "channels": {
      "location": {
        "current-value": "50.818395, 4.40313"
      }
    },
    "lastUpdate": 1199192935
  },
  "1199192945": {
    "channels": {
      "location": {
        "current-value": "51.918395, 4.46313"
      }
    },
    "lastUpdate": 1199192945
  },
  "1199192995": {
    "channels": {
      "location": {
        "current-value": "50.745395, 745395"
      }
    },
    "lastUpdate": 1199192995
  }
}
```

Listing 4.29: Sensor Update Stream Data Model

The subscription data model consists of three main fields. The callback, which specifies the type of executable code that will be passed, the destination and the customFields, the code that will be interpreted by the destination.

In Listing 4.30 we can see an example of the subscription data model. In this case, a Service Object Stream is subscribed to an http destination located in the URL http://somewhere.com:5000/update/@location@, passing the customField code.

```json
{"callback": "http",
 "destination": "http://somewhere.com:5000/update/@location@",
 "customFields": {
   "aliases": [
```
4.4. DATA MODEL

Listing 4.30: Subscription Data Model

For this Subscription the API engine stores the following definition document:

```json
{

  "##": "\$channels.",
  "!!": ".current-value"

},

  {  
    "@location@": "#location!!"
  
  ],
  "method": "GET"

}
```

Listing 4.31: Subscription Data Model

Were we can see extra meta fields: the timestamps, createdAt and updatedAt, the Subscription’s id, listed in the Service Object document and referred in this Subscription document via the source field, and the stream to which belongs.
4.4.2 Modified Data Model

To support searches via ElasticSearch the data model was modified. ElasticSearch searches documents. It doesn’t have the characteristic of search range of field values inside a document. For this reason the new data model is based on that value (an update, a subscription) is considered as a document.

From the point of view of the Service Objects meta information stored by the engine (see Listing 4.27) the reference to the data document and the reference to the subscription array disappears.

```
{
    "id": "139403523318497be978754164c27b07e748b53079662",
    "userId": "2d8d385681024a1a80f0f14d2179b1b",
    "public": "false",
    "createdAt": 1396521151696,
    "updatedAt": 1396856076240,
    "name": "Phone",
    "description": "COMPOSE phone",
    "URL": "Web Object URL",
    "streams": {
        "position": {
            "channels": {
                "location": {
                    "type": "geo_point",
                    "unit": "degrees"
                }
            },
            "description": "GPS outdoor location",
            "type": "sensor"
        }
    },
    "customFields": {},
    "actions": [],
    "properties": []
}
```

Listing 4.32: Service Object API stored new data model

Now all the data updates of a stream of a Service Object are stored in an individual document. The Sensor Update data model has not modification from the data sent to the API engine (as the initial data model), but now the API engine stores for each Sensor Update a document. And to reference each Sensor Update with its Service Object Stream the engine applies a document key based on; the Service Object id, the Stream Id and the lastUpdate field.
4.5. **SEARCH CAPABILITIES**

4.5.1 **Integration between Elasticsearch and Couchbase**

The integration of these two components is done through the CouchBase Plug-in for Elasticsearch, which exposes the Elasticsearch cluster as an external CouchBase cluster, and leverages CouchBase’s native XDCR replication
protocol to keep data synchronized between the data back-end and the search back-end. The following capture shows the configuration of the replication tasks in CouchBase.

Figure 4.17: XDCR configuration

### 4.5.2 Elasticsearch Index for data queries

The following Listing provide the description of the current implementation of Elasticsearch Indices deployed on top of the data stored in the ”soupdates” bucket in Couchbase. The index is designed to provide search capabilities without duplicating the data between the data and search back-ends. It also provides spatial search capabilities for the channels named ”location” and stored in ”geojson” format.

```json
{
    "settings": {
        "index": {
            "analysis": {
                "analyzer": {
                    "analyzer_startswith": {
                        "tokenizer": "keyword",
                        "filter": "lowercase"
                    }
                }
            }
        }
    }
}
```
4.5. SEARCH CAPABILITIES

```json
}

"mappings": {
    "couchbaseCheckpoint": {
        "_source": {
            "includes": [
                "doc.*"
            ]
        },
        "dynamic_templates": [
            {
                "store_no_index": {
                    "match": "*",
                    "mapping": {
                        "store": "no",
                        "index": "no",
                        "include_in_all": false
                    }
                }
            }
        ]
    },
    "_default_": {
        "_source": {
            "includes": [
                "meta.*"
            ]
        },
        "properties": {
            "doc": {
                "properties": {
                    "lastUpdate": {
                        "type": "date"
                    },
                    "channels": {
                        "properties": {
                            "location": {
                                "properties": {
                                    "current-value": {
                                        "type": "geo_point"
                                    }
                                }
                            }
                        }
                    }
                }
            },
            "meta": {
                "properties": {
```
Here we explain the most relevant parts of our Elasticsearch index to understand what do.

In the settings section we configure our custom analyzer used for our index named “analyzer_startswith”. Analyzers are composed of a single Tokenizer and zero or more TokenFilters. We setup an analyzer of type keyword that “tokenizes” an entire stream as a single token. Token filters accept a stream of tokens from a tokenizer and can operate with the tokens. In our case we modify the tokens to lowercasing them. A token filter of type lowercase normalizes the token text to lower case.

In the following section we define the Elasticsearch Mapping. Mapping is the process of defining how a document should be mapped to the Search Engine, including its searchable characteristics such as which fields are searchable and if/how they are tokenized.

Our mapping marks all documents coming from Couchbase to be indexed and not stored within Elasticsearch. This default is there because Couchbase itself is really fast in getting the document and it does not need Elasticsearch to store the document. We also ensure that the sensor updates fields “last-Update” and “channels.location.current-value” are of type date and geojson respectively. If we don’t set this fields, we could have problems if there were sensor updates incorrectly formatted. Finally set the mapping analyzer to the previously declared.

4.5.3 Search Options

The following is a list of different types of data queries supported by the platform:

- **geodistance**: Search for Updates within a given distance for a given coordinate. Incompatible with GeoBoundingBox. Requires a **location** channel
4.5. SEARCH CAPABILITIES

(in GeoJson format) to exist, which will be used to keep the location of the sensor updates.

Example payload:

```json
{
    "geodistance": true,
    "geodistancevalue": 300,
    "pointlat": 43.15,
    "pointlon": 15.43,
    "geodistanceunit": "km"
}
```

**timerange**: Search for Updates with a `lastUpdate` field value within a range of timestamps.

Example payloads:

```json
{
    "timerange": true,
    "rangeto": 1396859760
}
```

```json
{
    "timerange": true,
    "rangefrom": 1396859660,
}
```

**numericrange**: Search for Updates with a given field value within a range of values.

Example payloads:

```json
{
    "numericrange": true,
    "rangefrom": 13,
    "rangeto": 17,
    "numericrangefield": "channels.age.current-value"
}
```

```json
{
    "numericrange": true,
    "rangefrom": 13,
    "rangeto": 17,
    "numericrangefield": "channels.age.current-value"
}
```
CHAPTER 4. IMPLEMENTATION DETAILS

"numericrangefield": "channels.age.current-value"

{
  "numericrange": true,
  "rangeto": 17,
  "numericrangefield": "channels.age.current-value"
}

match: Search for Updates that match in any of its fields with the terms listed (any of them).

Example payloads:

{
  "match": true,
  "matchfield": "channels.name.current-value",
  "matchstring": "Peter John"
}

Example of complex query that restricts:

- Field channels.age.current-value between 13 and 17.
- lastUpdate of the items above 1396859660
- Containing the terms "John" or "Peter" in the field channels.name.current-value.
- Located within a GeoFence.

Corresponding payload:

{
  "numericrange": true,
  "rangefrom": 1396859660,
  "rangeto": 17,
  "numericrangefield": "channels.age.current-value",
  "timerange": true,
  "match": true,
  "matchfield": "channels.name.current-value",
  "matchstring": "Peter John",
  "geoboundingbox": true,
  "geoboxupperleftlat": 43.15,}
4.5. SEARCH CAPABILITIES

```
{"geoboxbottomrightlat": 47.15,
"geoboxbottomrightlon": 15.47
}
```
Chapter 5

Evaluation

The goal of our evaluation was to explore the following questions:

- Experiment 1: How does throughput and response time scale for the servIoTicy API with the amount of available resources?
- Experiment 2: How does throughput and response time scale for the servIoTicy API when the load increases?
- Experiment 3: How does the response time delivered by CouchBase scale with the API pressure?
- Experiment 4: How does ElasticSearch scale with the load and how does its configured heap size affects its performance?

To provide answers to these questions, we developed a set of four different experiments that are presented in the following subsections.

5.1 Evaluation Methodology

For the client emulation we used Httperf [27] specifying a session workload generator, designed to simulate a real users progress through a site. This type of testing is useful for estimating the actual performance that a web server or an API will achieve in practice. A file was created containing the sequence of requests to be performed, the number and sequence of URI’s and request method. We created a file containing the list of all existing Service Objects, randomly sorted and not repeated more than 8 times.

For such purpose, 6 parallel Httperf processes were used, each one emulating 300 clients (at which 100 new sessions were created per second) for a total of 1800 clients, each one issuing requests at a variable rate. It was
verified that each Httperf process had no internal bottlenecks. The target request rate was changed from 1 request per second to 40 requests per second, resulting in an overall target load of 1,800 to 72,000 requests per second.

Requests are distributed across different Service Objects a specified before, and the in all cases the API call to perform was 'lastUpdate' (see Table 4.1 for more details). This call is interesting because it involves a query to the search engine to retrieve the latest timestamp for all the stored updates for a given sensor, and then the actual retrieve of the sensor update that is returned to the client. This operation actually is more complex than updates from the point of view of the API: updates are a difficult task for the Stream Processing Topology, that is out of the scope of this paper, but involve less work for the API.

ServIoTicy was populated for the experiments with 52,388 Service Objects and 261,940 Sensor Updates, an average of 5 Sensor Updates per Service Object.

5.2 Evaluation Infrastructure

The tests have been run on two sets of nodes: one set for running the client emulators and one set for running the servers of the system under test. The 'server' set was composed of 16 two-way 4-core Xeon L5630 @2.13GHz Linux boxes, for a total of 8 cores per node and 16 hardware threads because hyperthreading was enabled. Each 'server' machine was enabled with 24GB of RAM. The 'client' set was composed of 2 two-way 6-core Xeon E5-2620 @2.00GHz Linux boxes, for a total of 12 cores per node and 24 hardware threads because hyperthreading was enabled. Each 'server' machine was enabled with 64GB of RAM. All nodes were connected using GbE links to a non blocking 48port Cisco 3750-X switch.

For the software stack, all nodes were running Ubuntu 12.04LTS, and we used Httperf v0.9.1, nginx 1.2.8, Jetty v9.2.3, Jackson v2.3.1, Jersey v1.18, Couchbase 2.2.0 Enterprise Edition, Elasticsearch 1.3.4, and Couchbase Transport Plugin for Elasticsearch v2.0.0.

5.3 Experiment 1: Scalability of the API with available resources

The goal of this experiment is to analyze how the throughput and the response time scale for the servIoTicy API when the number of instances for different servIoTicy components varies. To this end we vary the number of
Jetty nodes that serve the API and the number of Elasticsearch servers that perform the request search. We run the same workload changing the request rate target from 1 request per second to 40 requests per second and compare the results. Notice that as this is a closed loop system, request rate target is not always achieved as the response time delivered by the servers will influence the rate at which the client emulators will issue requests.

For this experiment we explore the scalability of the web tier of the IoT REST API for a combination of 1 to 12 Web Servers (Jetty) and a combination of 1 to 3 Elasticsearch servers. A static configuration of 2 CouchBase servers, continuously synchronized with the Elasticsearch tier are used also.

Figure 5.1(a) shows the results for highest target load explored, a request rate of 40 requests per second each client, resulting in an overall request rate target of 72,000 requests per second. As it can be observed, the Web Tier offers almost linear scalability as the resources committed to this tier increase. This can be observed for the case in which 3 Elasticsearch servers are allocated. This can be considered an expected result because the Web Tier is stateless and does not benefit from any kind of session affinity, what could result in a performance impact for some conventional Web Applications, but this is not the usual case for REST APIs.

As the number of Elasticsearch servers is reduced, from 3 to one, it can be observed how the indexing tier gradually becomes the bottleneck, dropping the overall performance of the servIoTicy API from around 25,000 requests processed per second for 12 Jetty servers and 3 Elasticsearch nodes, to roughly 2,500 requests processed per second in the case that the same 12 Jetty servers are allocated for the Web Tier and only one instance of Elasticsearch is used.

Looking at the results for response time scalability, shown in Figure 5.1(b), it can be clearly observed the same behavior, what is expected for a closed loop system as our testing platform is. In this case, it can be seen how the platform can easily achieve a baseline response time of around 100ms per request for any configuration of the Web tier that is provisioned with 8 or more Jetty servers. But when the number of Elasticsearch servers is varied, from 3 instances to 1 instance, we observe again how the delay introduced per request in the indexing layer starts increasing, increasing substantially the response time and therefore reducing the maximum throughput delivered by servIoTicy as it had been seen in Figure 5.1(a).

The reason why the experiment does not explore a different number of CouchBase servers is that we have observed that the data tier is not a bottleneck in most situations. CouchBase seems to deliver very low response time for a very high number of requests per second, and therefore the usual case is that either the Elasticsearch tier or the Web tier are responsible for
performance bottlenecks. For a better understanding of this statement we refer the reader to Experiment 4 below in this Section.

Figure 5.1: API scalability with variations of the number of instances for the Web Tier and ElasticSearch

### 5.4 Experiment 2: Scalability of the API with load

Once the scalability of the platform with the allocated instances for the Web and Indexing tier has been explored, we look now in more detail how the throughput and response time scale for servIoTicy with load variations. To do this we set the number of Elasticsearch instances to a fixed number, and we modify the target request rate in the range that goes from from 1 request per second to 40 request per second and client. As in Experiment 1, we then vary the number of Jetty servers serving the API. In particular, and based on the results obtained for experiment 5.3 we fix the number of Elasticsearch instances to three, as it provides a good compromise between scalability and resource usage, and ensures that ElasticSearch will not be the bottleneck for most of the load levels to be tested.

Figure 5.2(a) and Figure 5.2(b) show respectively the throughput and response time associated with each number of Jetty servers for a range of load levels. As it can be observed, the more instances are provisioned for the REST API tier, the higher the maximum achieved throughput is, as more instances are operating in parallel and therefore more requests per second can be processed. A similar behavior can be observed for the response time: there is a baseline response time of about 100ms that is only achievable for those load levels and configurations that are not saturated. Any configuration that provides a too low number of instances for the Web Tier as to keep the
5.4. EXPERIMENT 2: SCALABILITY OF THE API WITH LOAD

(a) Throughput

(b) Response Time

Figure 5.2: API scalability with variations of the number of instances for the Web Tier and the Load Level

(a) 2 Jetty Instances - Jetty

(b) 12 Jetty Instances - Jetty

(c) 2 Jetty Instances - ES + CB

(d) 12 Jetty Instances - ES + CB

Figure 5.3: Resource consumption (CPU, Memory) for the REST API Web tier (Jetty) - Load Level 40
equilibrium of the system, quickly results in a system with growing waiting queues for the requests and ends up with increased response times.

Looking at Figure 5.2(a) and taking a particular load level, some interesting scalability properties can be observed. Take for instance load level 40 as an example. It can be seen how for this load level, the system is initially constrained by the capacity of the Web Tier. This fact can be observed because as more instances are added to the Web Tier, the system perfectly scales-out, going for instance from around 2,000 replies per second for a load level 40 and 1 Jetty instance to around 4,000 replies per second when the capacity of the Web Tier is doubled to two instances. This behavior is generally observed until other limits are reached. An example of this situation can be seen for the same load level and 10 or 12 instances allocated for the Web Tier. In these scenarios, the scalability exhibited by the system is not linear, and therefore, other limiting factors must be reach.

To understand the situation, we monitor the resource consumption of the platform components for different load levels. Figure 5.3 shows the average CPU and memory consumption for the Web Tier instances (the cases in which 2 instances and 12 instances are provisioned) and for the combination of ElasticSearch and CouchBase components, when the load level is 40. As it can be observed, for the case in which 2 Jetty instances are provisioned (Figure 5.3(a)) shows how the servers hosting those instances are clearly overloaded, with a total CPU consumption above 90% all the time. When we compare these numbers with the resource consumption observed for ElasticSearch and CouchBase (Figure 5.3(c)), it can be clearly seen how those components are not suffering from the same overload. For the case in which 12 Jetty instances are used, the situation is the opposite, showing a lower CPU utilization for the Web Tier (Figure 5.3(b)) than the data tier (Figure 5.3(d)).

Notice that in all cases, memory is not a limiting factor for the tests included in this experiment. Nevertheless, this is not always the situation, and we refer the reader to Experiment 4 for an example of effects of memory consumption on the performance of the platform.

5.5 Experiment 3: Scalability of the Data Store

The goal of this third experiment is to evaluate the horizontal scalability of the Couchbase tier and its impact in servIoTicy performance. The motivation for this experiment is the fact that as we went through all the tests included in
this paper, we realized that CouchBase was delivering very low response times independently of the number of instances that we were provisioning. Even 12 Jetty instances for the Web Tier were apparently unable to saturate a single instance of Couchbase. To put some light on the response times delivered by Couchbase on our experiments and to understand by how much this tier was responsible for the baseline response time observed for servIoTicy, we built the response time histograms that can be observed in Figure 5.4.

In this experiment take the more demanding load level (40), we provision 12 Jetty servers for the Web Tier and we compare the behavior of the Couchbase tier when two and three instances of Couchbase are provisioned. Results are organized in response time bins on the x-axis, and in the y-axis we show the percentage of the total responses generated in the experiment that fell in that bin. There are two facts to remark in this experiment: First, the confirmation that Couchbase is delivering very low response times, with all the requests being served in less than one millisecond; Second, that when we compare the response time delivered by two instances against the observed times for three instances, it can be seen that no difference is noticeable, indicating that the instances deliver perfect horizontal scalability and that very rarely they will become the bottleneck for servIoTicy deployments.

![Histograms showing response times for 2 and 3 Couchbase instances](image)

Figure 5.4: Distribution of response times (bins) - Couchbase tier

### 5.5.1 Experiment 4: Performance limitations of the Indexing components

In this experiment we evaluate how ElasticSearch performance is affected by the Heap Size in that is configured in Java when running the indexing instances. For this purpose we look in detail the execution of the servIoTicy workload this time using a single Elasticsearch instance and a load 40.
The first thing to explore is how the memory Heap consumption is affected by the load level that reaches the ElasticSearch tier in servIoTicy. Figure 5.5(a) and 5.5(b) show the heap memory overhead produced for a load of 40 when the Web Tier is provisioned with 1 and with 12 Jetty servers respectively. As it can be observed, when 1 single Jetty instance is provisioned, as it acts as a bottleneck for the platform, the load that reaches the ElasticSearch tier is low and therefore the memory Heap utilization is low. In contrast, when 12 Jetty instances are provisioned and a much higher demand reaches the indexing tier, the Heap utilization becomes extremely high and, as it will be show below, this situation results in a significant performance degradation.

To quantify the performance impact of memory Heap configuration on ElasticSearch, we set the number of Elasticsearch instances to three and we analyze for a load level 40, the impact of different Heap memory configurations (ranging from 2GB to 8GB per instance) as we vary the number of Jetty servers. Figure 5.6(a) and Figure 5.6(b) show the throughput and response time delivered by servIoTicy under these configurations. As it can be observed, the capacity of the system to deliver sustained performance is seriously affected by the memory allocated to the ElasticSearch instances. As an example, the same number of Jetty and ElasticSearch instances, using 8GB of Hep per indexing instance allows servIoTicy to deliver up to 25,000 replies per second, while changing the Heap size to 2GB produces a drop on performance that results in less than 5,000 replies per second.
5.5. EXPERIMENT 3: SCALABILITY OF THE DATA STORE

Figure 5.6: API scalability with variations of the memory Heap configuration for a set of three ElasticSearch instances, a variation of the instances provisioned for the Web Tier and a Load Level 40.
Chapter 6

Conclusion

6.1 Summary

In this document we have presented the servIoTicy API for Service Objects, from the specification of the existing methods and operations to the details of the actual implementation. The Service Objects abstraction represent the virtual counter-parts of any existing physical device. Service Objects API provides the interfaces to store, retrieve and process data associated to one physical device. They are also used to dynamically construct the servIoTicy data plane that can ingest, transform and output sensor updates as they arrive into the platform.

We also have presented a detailed characterization of the resource demand observed for the different components of the servIoTicy platform. ServIoTicy is a state-of-the-art platform for IoT services, that integrates multi-protocol channels to communicate with the platform on the edge and data management and processing capabilities at its core. The characterization has revealed interesting details about the three main components involved in the process of storing and retrieving data: the REST API (implemented using Jackson on Jetty), the data store (CouchBase) and the search and indexing engine (ElasticSearch). We’ve observed how the REST API is generally the bottleneck, clearly CPU-bound, being ElasticSearch the second component more demanding in terms of CPU resources. CouchBase has delivered impressive performance and very low response times across different configurations, allowing one single instance of this data store to fulfill the demand in terms of requests per second of up to 12 API instances. ElasticSearch has shown to be very sensitive to memory configurations: degradation on its performance could be observed both for lack of free CPU resources as because of a limited amount of available memory. This work is a first stage
toward the construction of resource allocation policies for highly distributed
data-centric platforms such as servIoTicy.

6.2 Suggestions for future work

Even though the main goals of the project have been successfully completed, there is still room for improvement and future work. Previous chapters have already pointed out some of the ideas to improve the platform, which are again summarized and extended in the following paragraphs.

Performance and scalability is a critical requirement for the data processing infrastructure. For this reason, current and future research will put the focus on evaluating different strategies to improve the performance and scalability of the platform. Several tweaks to the data model and data organization have taken place due to the needs of the platform and the requirements of an efficient integration with the ElasticSearch engine.

The proposed technologies will have to scale with the size of data sets and provide reliability and customizable QoS objectives in the space of performance, consistency and availability.

Following steps are to characterize the data processing engine in the platform, which is Apache STORM using a novel code injection technique to allow for multitenant processing of data streams; to characterize the brokering layer to support multi-protocol messaging (implemented using Apache Apollo).

Another option to improve the platform would be the development and evaluation of dynamic resource allocation policies for deploying servIoTicy in the cloud. This isn’t trivial if the deployment is performed in a heterogeneous environment, where, for instance, we have to allocate a new ElasticSearch node depending on the free nodes and demanding.
Bibliography

[6] evrythng. URL evrythng.com
[8] IoT Toolkit. URL iot-toolkit.com
[17] Storm. URL http://storm-project.net


