Mert Cihat Ocak

Implementation of an Internet of Things Device Management Interface

School of Electrical Engineering

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology.

Espoo, 20.11.2014

Thesis Supervisor:	Prof. Jörg Ott Aalto University
Thesis Instructor:	Jaime Jimenez Oy L M Ericsson Ab

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY SCHOOL OF ELECTRICAL ENGINEERING	ABSTRACT OF THE MASTER'S THESIS

Author: Mert Cihat Ocak

Title: Implementation of an Internet of Things Device Management Interface

Date: 20.11.2014	Language: English	Pages: 84+35

Department of Communications and Networking

Professorship: Radio Communications	Code: S-72

Supervisor: Prof. Jörg Ott, Aalto University

Instructor: Jaime Jimenez, Oy L M Ericsson AB

The Internet is growing from connecting only computers and mobile devices to connecting also objects in the real life to the Internet, to create an *Internet of Things (*IoT*)*. Interconnected Internet of Things unveils the environmental data from these objects to the use of complex applications and systems in the Internet and the cloud, which is supposed to make the boundaries between the real world and the digital world transparent. However, these devices will mainly be resource constrained, simple, sleepy devices such as sensors and the number of devices connected to the network is expected to be very high. Hence, these two factors introduce the problem of device management in IoT networks.

The thesis proposes the design and implementation of a fundamental device management interface for IoT networks and devices, combining IoT-specific features and protocols such as CoAP, LWM2M and Publish/Subscribe with the existing Web frameworks and protocols such HTTP and WebSocket. Real-time management and monitoring of large-scale devices is one of the IoT-specific core features of the interface along with other management features. The interface analyzes the integration of additional features such as anomaly detection in IoT device data and error reporting mechanisms. Moreover, the management interface is designed as a standalone application over the existing Capillary Networks architecture, which targets at providing connectivity for resource constrained devices and optimizing IoT devices with cloud instances. Hence, the management interface extensively uses the features and entities provided by the Capillary Networks via large set of REST APIs.

The design of the interface focuses on the IoT-specific problems of device management, which structures the implementation accordingly. The implementation of the interface is evaluated at the end of the thesis with stress tests and comparison with initial requirements. The evaluation is then followed by possible future work to enhance the interface performance and extendibility to future IoT networks.

Keywords: Internet of Things, device management, web server, CoAP, LWM2M

Acknowledgement

This thesis is the largest personal academic project I have done so far in my life, which ended up taking several months for researching the topic, implementing the system and writing the thesis itself. To achieve one of the most motivating and exciting experiences of my academic life, I received a lot of valuable support from kind people during this process.

Firstly, I would like to thank Professor Jörg Ott for accepting to be my supervisor for the thesis. I really appreciate his comments and academic insight about the topic, which guided me incredibly in the writing process of the thesis.

Secondly, I would like to thank Jaime Jimenez from NomadicLab for his incredible support, guidance, very useful comments, feedback and valuable work experience, as well as being my instructor. I sincerely appreciate working with him in all the related and non-related topics during this time span since he was very helpful to clarify my questions, to advise me and encourage me all the time. I feel very content at the moment for finishing this thesis and it would be much harder to achieve this without the help from Jaime.

Moreover, I would like to thank all my colleagues from NomadicLab and Ericsson for the very enjoyable working experience and environment, especially Nicklas Beijar and Tero Kauppinen for helping me with many technical aspects, Jan Melen for being an understanding project manager and Tony Jokikyyny for being a very kind manager.

Last but not the least, I would like to thank my family and my friends for all their support, some of which were remotely received with great gratitude. Thank you all so much for everything.

Jorvas, November 20, 2014

Mert Cihat Ocak

Table of Contents

Abbreviations and Acronyms

AJAX	Asynchronous JavaScript and XML
BLE	Bluetooth Low Energy
CGW	Capillary Gateway
CN	Capillary Network
CNF	Capillary Network Function
CNM	Capillary Network Manager
CoAP	Constrained Application Protocol
COMMUNE	Cognitive Network Management Under Uncertainty
CSS	Cascading Style Sheets
GBA	Generic Bootstrapping Architecture
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
JS	JavaScript
JSON	JavaScript Object Notation
LWM2M	Lightweight Machine-to-Machine Device Management
MD	Machine Device
MP	Mirror Proxy
MVC	Model-View-Controller
NoSQL	Not-Only Standard Query Language
Pub/Sub	Publish/Subscribe
RAT	Radio Access Technology
RD	Resource Directory
REST	Representational State Transfer
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URL	Uniform Resource Locator
vGW	Virtual Gateway

List of Figures

List of Tables

1 Introduction

The Internet has grown extensively and fast-paced in the last decades, from serving hundreds of hosts to providing billions of interconnected, complex hosting solutions. The evolution of the Internet is still ongoing swiftly with the addition of mobile devices in great numbers connecting to the Internet. Interconnected computers and mobile devices constitute the current Internet architecture in which the people are connected to the Internet regularly.

The next leap in the Internet connectivity is to evolve from connected computers and mobile devices to connected real world objects. Hence, the purpose is to extend the Internet of computers to *Internet of Things.* Connected *things* will combine the real world with the digital world, bringing the environmental information from the objects to the use of the complex systems in the Internet. *Things* in this scope are mainly constrained devices such as sensors or actuators placed around the daily environments and they can collect e.g. temperature readings from a room, heart rate readings from a patient, humidity values and many other uncountable environmental information. The information collected can be processed by different entities in the network and different applications can be retrieved from the data. Moreover, *things* can interchange data between each other to enable different interaction possibilities such as the heating system at home can be turned on automatically when the sensor on the door lock notifies the heaters about incoming home owner or the fridge can send updates to its manufacturer for maintenance purposes.

The application possibilities, which the *Internet of Things* concept offers, are boundless in industry, home automation, health, logistics or any other area by connecting the *things* to the network. However, the constrained nature of the *things* i.e. devices introduces challenges to connect these low-energy, low-processing power and sleepy devices to the network. The current Internet uses mainly HTTP over TCP/IP stack for reliable data transmission in always-connected networks. But the *Internet of Things* requires specialized methods and communication aspects to tackle the connectivity challenges created by the constrained networks.

The research in making the Internet of Things (IoT) real focuses on how to tackle the challenges of constrained networks in different ways and on how to integrate the device and the data to the cloud optimally. These topics are hot topics in many research projects and the Capillary Networks project is one of the research projects which focuses on these research problems and their solutions. Enabling an easily deployable connected devices architecture with using cloud resources extensively is the key objective of the Capillary Networks project.

The number of connected *things* is expected to be in billions, which is supposed to transform the network to an enormous scale. However, this expected large number of constrained devices raises another problem in the IoT network: the device management. Each one of the connected devices needs to be managed and maintained by the network manager either manually or automatically. But the challenges introduced by the connectivity constraints of the devices pushes the research to find new ways of device management for constrained devices, which are different than current Internet entities using always-connected, non-constrained nodes. Hence, the device management in IoT requires lightweight and innovative methods designed in particular for constrained devices.

The management of an IoT network requires manual interaction with the end user for monitoring the network, even though the entire network management is automated. For this purpose, a device management interface in IoT can be implemented as a Web service, which combines the whole IoT network management aspects such as monitoring, manual device controls, error reporting or data presentation. Web service provides easy integration of the device management to the current Internet, high reachability of the interface and innovative graphical ways to manage the large number of devices.

This thesis presents the requirements, design and implementation of a Web based IoT device management interface for the Capillary Networks architecture to research the solutions for IoT device management problems. The interface combines several IoT-specific aspects with the existing Web architecture to find solution to the research statement.

1.1 Objective of the Thesis

The management interface design and implementation consist of three main objectives:

1. The management interface is required to be designed so that it can optimally handle large number of devices and large amount of data. Moreover, the features of the device management should be supported for each single device.
2. The interface is needed to present the features of the Capillary Networks architecture to the network manager. To satisfy this, the interface is supposed to gather network management information from different network entities and create the final look. Hence, the interface is required to be designed as a standalone solution.
3. The features specific to IoT device management should be integrated to the interface for constrained network optimization.

To satisfy the objectives, the thesis discusses the implementation of the IoT device management interface done with the use of HTTP, WebSockets, CoAP, LWM2M and specific IoT device management features.

1.2 Scope of the Thesis

The thesis focuses on the design and implementation of a device management interface built on top Capillary Networks architecture and integration of Capillary Networks components. Hence, we omit the integration of other IoT network solutions or protocols which are not used in the Capillary Networks.

The discussion for the implementation of Capillary Network entities is not included as well. The management interface is built upon the existing Capillary Networks architecture and hence, only the relevant features of the architecture is discussed.

The management interface introduces an online map to display the locations of devices (gateways and sensors) on the map. The map is limited to the two-dimensional locations i.e. three-dimensional presentations of device locations are out of the scope. The map is mainly aimed at presenting geographically distributed devices. Moreover, indoor positioning of the devices inside buildings is also not included in the thesis.

Furthermore, the management interface assumes the security of the network i.e. the devices and the data is already established by other means or entities. Therefore, security concerns of the IoT network architecture is skipped from the thesis.

1.3 Structure of the Thesis

The thesis consists of 7 chapters. Chapter 2 presents the background information gathered during the thesis research. Chapter 3 introduces the requirements of the management interface implementation while we present the interface design in Chapter 4. Chapter 5 discusses the implementation in detail both for the frontend and backend of the interface. In Chapter 6, we evaluate the interface performance with obtained measurements from the tests. Finally in Chapter 7, we review the thesis by summarizing the outcomes obtained during the thesis and suggest ideas for future work about the thesis topic.

2 Background

This chapter presents the related technology and literature relevant to the work developed during the thesis. First, the concept of Internet of Things (IoT) and of device management in IoT is discussed thoroughly. Next, two IoT projects, which we got involved during the thesis, are shortly introduced. Several communication protocols and their roles in IoT device management are also presented, followed by the introduction of some data handling solutions in IoT networks. Finally, different features used in IoT device management are presented in detail by referring to related protocols or concepts.

2.1 Internet of Things

The concept of Internet of Things is presented in this section with an aim to cover the idea behind this evolving set of technologies. Requirements and challenges encountered in IoT device management are introduced as well, followed by two project overviews.

2.1.1 Internet of Things Overview

The idea of connecting *things* to the Internet has been discussed and researched extensively by research institutions and industries for the last two decades as the new technology breakthrough. Connecting these *things,* i.e. physical components, to create an Internet of Things (IoT) is predicted to have great impact in daily life and will eventually make many applications possible, such as health care tracking, smart homes, smart industrial plants, logistics and many other examples both in the public and private sectors within [1].

M2M (Machine-to-Machine) with a similar concept of connecting *things* has been in use in different industries since early 1990s [2]. However, M2M and IoT are two distinct concepts: IoT is broader and is an evolution from M2M in terms of connectivity, application and storage. The main purpose of M2M is to maintain the connectivity between a specific *machine* and the remote host in a fixed, proprietary installation, which is generally configured to monitor and control only those specific types of *machines* i.e. vertical solutions [3]. Hence, M2M provides end-to-end connectivity to send the *machine* data to the cloud and to manage only specific, closed and point-to-point systems, such as elevator remote control or fleet management solutions [2]. However, IoT concept broadens the idea of M2M to create a new Internet of connected things horizontally rather than vertical solutions. To establish the horizontal interaction between IoT entities, *things* connected to the network send their data to the cloud, from which humans, computer systems and other *things* read the data, interact with each other and integrate with other standalone applications/solutions [4] (see Figure 2-1). Thus, IoT aims at creating an open, scalable, standards-based, service-oriented network in which very large amount of nodes can communicate and interact with each other, such as receiving updates from your refrigerator to your mobile phone about the food which will go bad soon, heart beat readings of a patient sent to a monitoring node for notifying the doctor in case of anomaly or the sensors in an oven sending readings to the manufacturer for quality assurance.

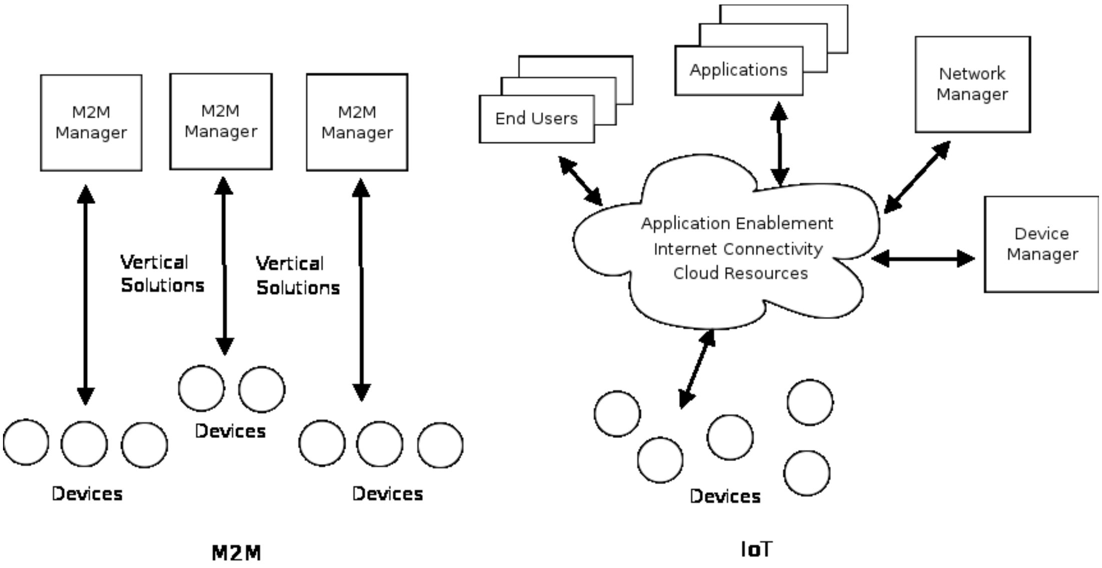


Figure 2-1 M2M and IoT Comparison

To achieve the goals set by the IoT vision, new features and methods are to be introduced to the already existing M2M solutions, some of which are [2]:

- **Connectivity:** The small, constrained devices needs to be connected to the network. Since these devices are generally power-constrained non-cellular devices, low-power connectivity solutions are needed.
- **Communication Protocols:** To decrease the network traffic of the constrained devices, special constrained protocols are needed both for data and control paths since traditional Web protocols (HTTP/TCP) create heavy traffic for constrained devices
- **IP-Based:** Enabling *things* communicating with each other requires each of them to be accessible in the Internet. IPv6 provides large enough namespace for billions of possible connected devices for this purpose.
- **Smart Objects:** To store the data and to enable different network elements to interact with it, the data needs to be in a standardized format. This assures interoperability between different elements of the network.
- **Web APIs:** APIs to retrieve the data and other information from the cloud and the network provides ease of integration by other nodes or free application development on the provided data without the knowledge of the source of the data
- **Post-Processing of the Data:** The immense amount of data stored in the cloud can be related to other entities or can be processed for developing business logics. Moreover, the data can be analyzed for anomaly detection, tracking, data assessment etc.

One of the main purposes of IoT is the construction of a horizontal, versatile, scalable and accessible data architecture where large number of devices and humans interact with each other. However, the vast amount and distribution of the devices over the network creates technical challenges in device management of devices in IoT. Device monitoring and control, network monitoring and management, device or resource identification and discovery are some of the challenges the infrastructure of IoT is required to find solutions for.

To achieve the management of IoT devices in distributed and constrained networks, specific constrained protocols with IoT specific features are needed. Moreover, the network architecture requires to provide and store the device management information retrieved from both the devices and the management nodes. The integration of the management protocols and management data to the Internet can be achieved by using Web services, which creates human-readable, interactive management interfaces.

The device management interfaces are often placed at the end of the IoT management network and provides all the relevant information to the end user i.e. the device manager. Handling the vast amount of devices and the data, along with several different communication protocols and cloud resources, device management interfaces are required to be the wrapper applications in the network combining different protocols, features and data handling methods.

2.1.2 Project Overview

During the duration of the thesis, we were involved in two IoT projects, which are Capillary Networks and COMMUNE.

Capillary Networks

Predictions for future IoT architecture propose that billions of devices will be connected to the Internet, as stated in the previous section. But the majority of these devices are expected to be non-cellular, basic devices (e.g. sensors, actuators) using only short-range radio technologies (Wi-Fi Low Energy, BLE, IEEE 802.15.4 etc). A possible scenario for non-cellular devices to connect to the cellular network is the use of capillary gateways. In this case, a capillary gateway is a device which is capable of communicating over short-range radio and providing network connectivity (either cellular or fixed network connection) to non-cellular devices. Moreover, a capillary network refers to the local short-range network created between non-cellular devices and the capillary gateway. By making easily deployable capillary networks possible and creating end-to-end connectivity between non-cellular devices and the end user, capillary networks are supposed to be an important concept for making IoT real [5].

CGWs can be connected to the Internet using either wired or wireless backhaul. For this project, however, LTE was chosen to be the backhaul technology with the motivations such as [6]:

- 3GPP CGW is easier and cheaper to deploy since there is no need for LAN or

fiber cable.
- 3GPP is currently the best solution in case of mobile capillary networks.
- Connecting capillary networks to core cellular network enables cellular network features to be used for capillary networks.

The Capillary Networks project aims at providing connectivity to devices connected via CGW, which current 3GPP devices do not support. Following are some other features offered by the project outcomes, which do not currently exist in 3GPP solutions:

- Enabling end-to-end management and connectivity over cellular network for very simple and cheap devices e.g. sensors or actuators.
- Smart CGW selection between capillary networks.
- Increased QoS for capillary networks.
- Possibility to have middleware with different functionalities in the core network for capillary networks e.g. proxies.
- Providing cloud resource optimization for capillary networks management and data.

Features and components of the Capillary Networks project are discussed in detail in Section 2.5.

COMMUNE

COMMUNE is a joint Celtic-Plus project with partners from Finland, Spain, Poland, Slovenia and Ireland. The project focuses on the problem of how to manage future networks under uncertainty and proposes solutions based on cognitive networking techniques to reduce uncertainty with critical management situations [7].

While knowledge based reasoning approaches in order to detect system faults are widely used in current networks, the solutions still include human interaction to deal with uncertainties in complex networks. To decrease the need for human interaction, machine-learning techniques may be relevant to deal with network management under uncertainty.

In the concept of COMMUNE, cross domain management should require less human interaction in the future as the number of nodes connected to the network and the amount of data produced will be immense such as the number of IoT devices. From IoT device management perspective, COMMUNE offers automatic anomaly detection in the device data as a cognitive method (see Section 2.4.4).

2.2 Communication Methods in IoT Device Management

IoT device management requires extensive communication of the network manager with the constrained devices and with the other entities in the network. Management commands from the manager are transmitted to the device via different nodes using different communication methods/protocols. Being generally in the cloud as a Web server, the management node is expected to support HTTP [8] to integrate seamlessly with other cloud entities. Since HTTP does not support real-time communications directly, the WebSocket protocol can be used between several entities for real-time device monitoring. Moreover, the manager may be limited to use specific protocols to communicate directly with the constrained devices due to low power, connectivity or other constraints. For this purpose, the manager can use CoAP (Constrained Application Protocol) [9] as a protocol for constrained environments either for data interchange from the devices or for management commands. For the management part in particular, a lightweight device management protocol working over CoAP would complete the functionality of the IoT device management. LWM2M (Lightweight Machine-to-Machine Device Management) [10] satisfies the requirement of working over CoAP and retains many features for IoT device management. As it has been shown, there are several different protocols that are needed for IoT device management to integrate with the rest of the network. This section discusses these communication methods and protocols, which are to be used for a complete and operable IoT device management solution.

2.2.1 HTTP

HTTP (Hypertext Transfer Protocol) is one of the widely used application layer protocols for transferring data over the Internet [8]. HTTP architecture defines a server (generally referred as *web server*), which serves the data and a client (generally a *web browser*), which requests the data from the server. Hence, HTTP follows a *request-response* scheme i.e. *request* from the client is sent to the server and the server replies with a *respond* to the client.

Since HTTP is an application layer protocol, it can be used on top of any reliable transport level protocol such as TCP or RTP [8]. The current web servers in the Internet mostly utilize HTTP on top of TCP/IP stacks.

Resources and Path Conversion

The web server stores the web content, which is sent to the client in HTTP response. The web content can include static HTML files, images, messages, videos, on-demand dynamic content and many other types. Each single piece of the web content is defined as a *resource* on the web server i.e. any content that has an identity on the web server is a *resource* [11].

Resources on the web server are accessible via uniform resource locator (URL). URLs indicate a reference to the unique resource on the web server and follow the format below [12]:

PROTOCOL://HOST_NAME:PORT /RESOURCE_PATH?RESOURCE_INPUT

PROTOCOL refers to the application layer protocol used (e.g. *http*) while HOST_NAME is the webserver address (e.g. *www.ericsson.com*). PORT is the port number accepting the requests in the server (80 for default http requests). RESOURCE_PATH defines the resource location on the server whereas RESOURCE_INPUT is the input given by the user agent for the requested resource.

In practice, webserver backend implementation is responsible of directing URLs into logical functions and of responding the user with the requested resource. Hence, the requested URL may not refer to a simple static file but rather can also refer to a more complex data acquisition in the server.

Methods

As defined in [8], there are six main HTTP request methods which indicate to the server the action to be performed on the requested resource. Each request is accompanied with one type of method and the method type is written in the request header. The list of HTTP methods is presented in Table 2.1.

The methods available, GET method is the most common one which basically asks the requested resource from the server. GET request data is encoded in the URL itself. However, the request data is encoded in message bodies in case of POST and PUT methods. The server processes the message data in POST and PUT requests (e.g. written to the database) and returns the appropriate status code. Less common methods HEAD, DELETE and TRACE are mentioned in Table 2.1.

HTTP methods can have two different features, safe or idempotent. A safe method should not modify any data in the server other than just retrieving the information, such as GET and HEAD. To perform changes in the server, “unsafe” methods should be chosen [8]. Moreover, idempotent methods should have the same side-effects in single or multiple requests to the server i.e. the user can send multiple requests to the server while expecting the same server response.

Table 2.1 Most Common HTTP Methods

Method	Description	Safe	Idempotent
GET	Get the requested resource.	Yes	Yes
POST	Post data to the server for processing	No	No
PUT	Store the resource in the server	No	Yes
HEAD	Get only the HTTP header, no body	Yes	Yes
DELETE	Delete the resource in the server	No	Yes
TRACE	Trace the request to the server	Yes	Yes

Status Codes

Each HTTP request receives a response code from the server indicating the result of the operation performed in the backend. The response code is embedded in the HTTP response. If the server responds with data, the data is attached to the response as well.

HTTP response code is a 3-digit result code in which the first digit defines the class of the response [8]. The last two digits can be defined depending on the implementation; however, there are widely accepted and used HTTP codes available. The list of status codes are presented in Table 2.2:

Table 2.2 HTTP Status Codes

Class	Category	Description	Example
1xx	Informational	Request received, continuing the process	101 Switching Protocols
2xx	Success	The action successfully performed	200 OK
3xx	Redirection	Further action needed	302 Found
4xx	Client Error	The request cannot be handled	404 Not Found
5xx	Server Error	The server failed to fulfill a valid request	500 Internal server error

2.2.2 REST

Representational state transfer (REST) is an client-server type architectural style for network applications, which consists of several design principles and has defined the current basis of the World Wide Web [13]. Although REST is not a protocol not has been standardized within HTTP standards, the Web architecture has evolved on REST architecture using the distributed system design defined with REST [14].

REST relies on three main design principles: addressability, uniform interface and statelessness [15]. Addressability refers to data elements in the server being accessible via uniform interfaces to the clients. Data elements in REST are abstracted as *resources*, which makes REST a Resource Oriented Architecture [13]. As stated in 2.2.1, resources can be any type of information stored on the server and accessible via a uniform and standard interface (i.e. URI) [16]. Advantages of using a uniform interface are familiarity of the web server operations to the clients and interoperability of the request and responses [16]. Statelessness of REST architecture assures that requests are performed with the information provided only in that request i.e. the server never relies on the data from previous requests to perform the operation for a new request. Statelessness provides several advantages for implementation such as scalability and load balancing [16].

As mentioned above, REST is widely used in current the Internet applications along with HTTP, though REST can be applied on other protocols as well. Support of extensive representation formats (e.g. HTML, JSON, XML etc.) and HTTP methods as uniform interfaces (e.g. GET, POST, PUT etc.) can be referred as only two of the main reasons of the wide use of REST architecture over HTTP in web technologies.

Web services combine REST architecture to the implementation with the following convention:

- Resources stored on the server are given unique IDs, which are URIs.
- Request from the client identifies the resource by providing the resource ID in the request.
- The representation of the resource is prepared and sent in the response to the client.
- The requests are not directly referred to previous requests (statelessness).

Table 2.3 presents examples of message flows in a web service, which deploys REST architecture.

Table 2.3 REST architecture implementation example in a Web service

Resource	Action	HTTP Method	Client -> Server	Server -> Client
Sensors	Get: List of sensors	GET: http://iotman.com/deviceList	None	[{"devices":"111, 222"}]
Gateways	Update: GW information	POST: http://iotman.com/gateways	cmd=set_log&node =1	[{"status":"Success"}]

2.2.3 The WebSocket Protocol

The WebSocket protocol is a client-server based communication protocol, which enables two-way (bidirectional) communication over a single TCP connection between the client (generally a web page or a web browser) and the server [17]. The protocol is developed as part of HTML5 initiative to decrease the complexity of bidirectional communication in the Web and to provide a simpler form of full-duplex connection to also enable server initiated communication.

Bidirectional communication in HTTP can be achieved by the client frequently polling the server for any updates (HTTP polling or HTTP long polling). However, this approach is not efficient with the following reasons [17]:

- For each request, the server needs to use different TCP connections and handshakes.
- Each request repeats HTTP headers and the same applies to the response.
- The client needs to map the outgoing HTTP connections to the incoming connections.

The WebSocket protocol eliminates the need of HTTP polling by the client since the open TCP connection can be used to send information either by the client or by the server. Bidirectional nature of the WebSocket enables the build of scalable and real-time Web applications, in which the server can initiate the communication as well [18].

The protocol is designed to be compliant with the current HTTP architecture of the Web i.e. it runs over HTTP ports 80 and 443 as well as HTTP proxies or other intermediate entities [17]. Protocol switch between HTTP and WebSocket is performed during the “handshaking” process. However, WebSocket is not limited to HTTP or any other application layer protocol since it is an independent TCP-based protocol.

The WebSocket protocol consists of two parts, from which “handshaking” is the first part. To establish the WebSocket connection with the server and to switch protocol from HTTP to WebSocket, the client sends a handshake request to the server via HTTP, which can look like as follows:

```
GET /deviceList HTTP/1.1
Host: server.iotmanagement.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: TS4irnQyQTfEfeRKFyqN8g==
Origin: http://iotmanagement.com
```

The response to the handshake request from the server would be as follows:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: J7/IIGO7dfhs8KWl+098tqUDSeE=
```

A successful handshake creates the TCP connection between the client and the server and utilizes the WebSocket protocol over this connection by switching from HTTP (note the HTTP status code 101 in the response). The second part, “data transfer” follows the successful handshake. The client and the server uses the TCP connection for transmitting data in a bidirectional manner, in which data messages are sent in frames over this connection. One data frame can look like as follows:

```
{"name":"networkElements","args":["[{"id":"G002127","latitude":59.40442269
,"longitude":17.95359135,"service":"gateway","vgwip":"1.1.1.1"}]"]}
```

The standardized API [19] of the WebSocket protocol enabled many WebSocket implementations being integrated into existing Web technologies (e.g. PHP, Perl, Python) or into evolving Web solutions (e.g. Node.js). Moreover, server-initiated data transmission feature of WebSocket introduced the extensive use of Publish/Subscribe schemes in the evolving technologies. It enabled servers subscribing to clients or vice versa to inform the other party instantly by sending a notification. The feature of publish/subscribe is a crucial requirement for real-time device management in IoT, as to be discussed in Section 2.4.1.

2.2.4 CoAP

Constrained Application Protocol (CoAP) is a RESTful application layer protocol specifically designed for constrained devices and constrained networks [9]. It decreases the effects of the difficulties created by the constrained nature of these networks (e.g. low-power, lossy). CoAP utilizes a request/response based architecture between a CoAP client and a CoAP server while including key concepts of the Web (similar to HTTP) such as media types, URIs and method types [9]. Moreover, it introduces new solutions for M2M specific problems such as resource discovery of the constrained nodes, smaller message overhead compared to HTTP and asynchronous transfer model.

Message Transaction Model

CoAP uses UDP or SMS (Short Message Service) bindings as transport protocol to avoid the overhead created by connection oriented protocols such as TCP. As the protocol stack of CoAP shown in Figure 2-2 presents, CoAP *mainly* works over IPv6 using 6LoWPAN (IPv6 over Low power Wireless Personal Area Networks) [20] and RPL for multi-hop cases (IPv6 Routing Protocol for Low-Power and Lossy Networks) [21] as opposed to HTTP over TCP/IP stack. Hence, including HTTP features for constrained devices does not indicate CoAP being a compressed version of HTTP but rather being a re-designed protocol specialized for constrained environments [22].

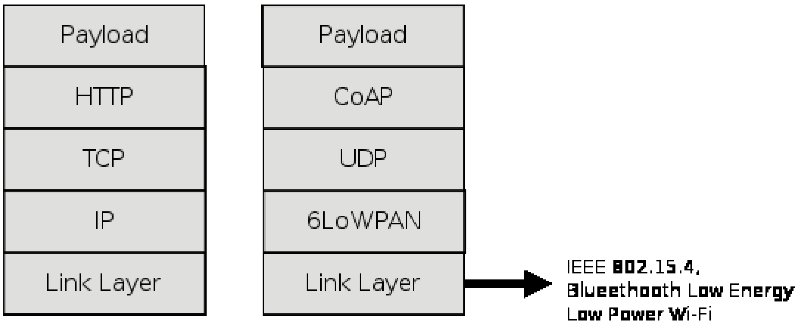


Figure 2-2 Protocol Stacks of HTTP vs CoAP

Since UDP is a best-effort protocol, the optional reliability in CoAP is ensured with different types of CoAP messages in the message layer: *confirmable* (CON), *non-confirmable* (NON), *acknowledgement* (ACK) and *reset* (RST). Confirmable messages require the receiver to send an ACK to the sender to ensure the reliability. However, non-confirmable CoAP messages do not require any ACK. If the response for a CoAP confirmable message is immediately available, the response can be carried in the payload of ACK message and this is called piggybacked response [9]. CoAP message types are presented in Table 2.4.

Table 2.4 CoAP Message Types

Type	Definition	Reliable	Indicator
CON	Confirmable	YES	0
NON	Non-confirmable	NO	1
ACK	Acknowledgement	NO	2
RST	Reset	NO	3

Each CoAP message consists of a four-byte binary header followed by a sequence of options and payload (see Figure 2-3). The header contains 2-bit CoAP version, 2-bit message type indicator, 4-bit token length indicator for the variable length token, 8-bit message code (explained in Section “Message Codes and Methods” below) and 16-bit message ID. The header is followed by the token value, which is used to correlate requests and responses. Token value is followed by options if there is any. Following the options comes the Payload Marker (0xFF) to indicate the end of options and the start of optional payload. This compact header design decreases the overhead significantly compared to HTTP, which leads to decreased energy consumption and response time for constrained devices [22].

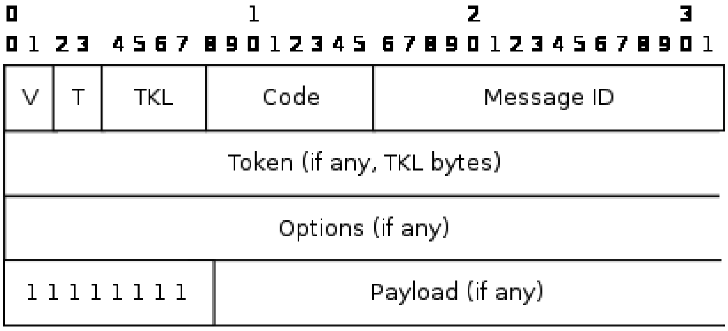


Figure 2-3 CoAP Message format

Figure 2-4 presents a communication example between a CoAP client and a CoAP server using CON messages. Response for the CON message is piggybacked in the ACK in both cases while the timeout in CoAP client triggers the request resent to the server in the second case.

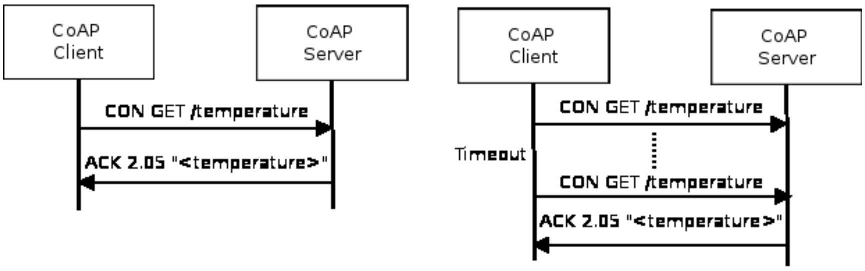


Figure 2-4 CoAP request-response example

URIs and Discovery of Resources

Since CoAP is a RESTful protocol, information in constrained devices are stored as resources and accessible via URIs. Hence, CoAP defines a URI scheme for CoAP resources (e.g. temperature value in a specific room) which is similar to HTTP URLs (see Chapter 2.2.1):

coap://HOST_ADDRESS:PORT_NUMBER/PATH?QUERY

As an example, following URI refers to the *temperature* resource on the requested host and the path.

```
coap://ericsson.com:5683/rd/jorvas/room/541/temperature/
```

However, machines communicating with each other are required to discover each other's resources in M2M environments i.e. resources and corresponding URIs are needed to be available for other machines. To solve this problem, CoAP introduces the well-known resource path */.well-known/* in CoAP servers. Each COAP server is suggested to provide the *well-known* path for resource discovery in the network [9]. The *well-known* URI is an important aspect for CoAP since it provides uniform and interoperable resource discovery in CoAP networks [23]. For instance, following URI refers to the list of resources in the given host address i.e. to *well-known*:

```
coap://[2001:db8::2:1]/.well-known/core
```

Message Codes and Methods

Message code attribute in CoAP message header is used to define different types of request types or response codes. In case of a CoAP request, the attribute is used to determine the message method of the request from one of the following methods: *GET, POST, PUT* and *DELETE.* These methods perform similar way to the same methods in HTTP (see Chapter 2.2.1).

When the CoAP message is a response to a request, 8-bit message code attribute is used to indicate the response codes. Response codes are also similar to HTTP with the difference of a dot between the class of the response code and the sub-class, as some of them are presented in Table 2.5.

Table 2.5 CoAP Response Code Examples

2.xx Success		4.xx Client Error		5.xx Server Error	
2.01	Created	4.00	Bad Request	5.00	Internal Server Error
2.02	Deleted	4.01	Unauthorized	5.01	Not Implemented
2.03	Valid	4.02	Bad Option	5.02	Bad Gateway
2.04	Changed	4.03	Forbidden	5.03	Service Unavailable
2.05	Content	4.04	Not Found	5.04	Gateway Timeout

Observe/Notify in CoAP

CoAP introduces an asynchronous publish/subscribe mechanism to enable server-initiated communication, which is called "Observe/Notify" [24]. In HTTP, requests are always client-initiated which means that the client needs to perform the same request frequently (polling) to determine the changes in the server. However, this approach is not optimal for power-constrained environments. To overcome this problem, CoAP enables a communication scheme in which the server can initiate the communication to inform the registered client about the updates in the server data or management command.

To start the observer/notify mechanism, the client indicates in a CoAP request its interest to “observe” the changes in the CoAP server by specifying the “Observe” option in the message options. This way, the client starts observing the resource on the server and if the resource is updated, the server “notifies” the client with the new information. Notification message is a regular CoAP response (e.g. message code 2.05 Content) with the token ID of the first observe message and an additional observe message ID (see Figure 2-5).

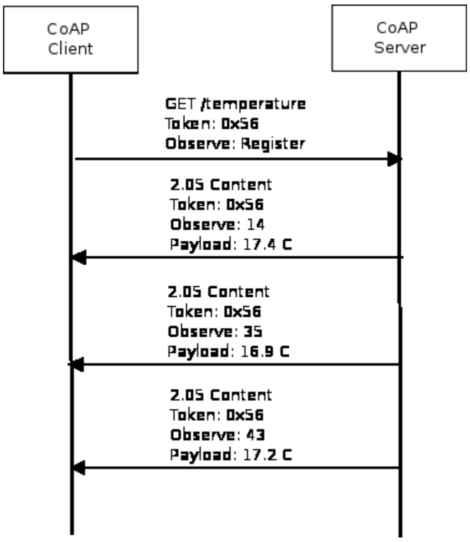


Figure 2-5 CoAP Resource Observe-Notify Scheme

2.2.5 LWM2M

OMA DM Lightweight (LWM2M) is a light and compact device management protocol that is used for managing IoT devices and their resources [10]. LWM2M runs on top of CoAP as an application layer communication protocol, hence, LWM2M is compatible with any constrained device, which runs CoAP as the transport protocol (see Figure 2-6). The main approach for LWM2M is to provide a set of interfaces for managing the constrained devices. Monitoring, managing and provisioning the massive amount of IoT devices require a standardized lightweight management protocol to maintain the control of the IoT network.

LWM2M offers a simple object based resource model, which allows several operations on the resources, such as resource creation, retrieval (read), update, deletion, configuration, execution, observation and notification [10]. As a device management protocol, LWM2M supports all basic device management functionalities, which include but not limited to access control, firmware update, connectivity, location, device meta data etc. Next sub-sections discuss the architecture of LWM2M and the resource model in IoT device management.

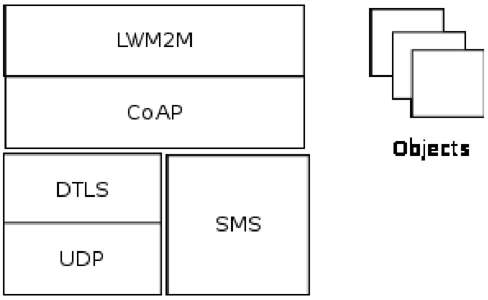


Figure 2-6 LWM2M Protocol Stack

Architecture and Interfaces

LWM2M architecture defines three logical components:

- **LWM2M Client**: It contains several LWM2M objects with several resources. LWM2M Server can execute commands on these resources to manage the client, commands such as to read, to delete or to update the resources. LWM2M Clients are generally the constrained devices (sensors, actuators etc.).
- **LWM2M Server**: It manages LWM2M Clients by sending management commands to them. The LWM2M Bootstrap Server configures the access control for a specific LWM2M Server on the constrained device.
- **LWM2M Bootstrap Server**: It is used to manage the initial configuration parameters of LWM2M Clients during the bootstrapping process. It is only entitled to configure the device to give access to specific LWM2M Servers, hence, management of the device does not involve the bootstrap server after the bootstrap process.

To maintain the communication between mentioned components above, following LWM2M interfaces are defined in the standard:

- **Bootstrap**: LWM2M Bootstrap Server sets the initial configuration on LWM2M Client when the client device bootstraps. For this interface, the client sends a *"Request Bootstrap"* message to the bootstrap server and the server performs "*Write*" and "*Delete*" on the client's access control objects to register one or more LWM2M Servers.
- **Client Registration**: LWM2M Client registers to one or more LWM2M Servers when the bootstrapping is completed.
- **Device Management and Service Enablement**: LWM2M Server can send management commands to LWM2M Clients to perform several management actions on LWM2M resources of the client. Access control object of the client determines the set of actions the server can perform, which is already set during the bootstrapping process.
- **Information Reporting**: As a feature of CoAP Observe-Notify mechanism [24], LWM2M Clients can initiate the communication to LWM2M Server and report information in the form of notifications.

Figure 2-7 presents the LWM2M architecture and the interfaces between the components, while Table 2.6 shows all available operations on LWM2M objects in each interface.

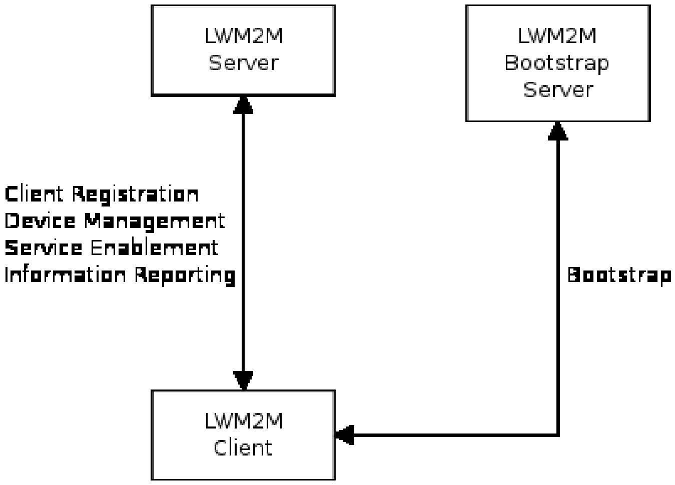


Figure 2-7 LWM2M Components and Interfaces

Table 2.6 LWM2M Interfaces and Operations

Interface	Direction	Operation
Bootstrap	Uplink	Request Bootstrap
Bootstrap	Downlink	Write, Delete
Client Registration	Uplink	Register, Update, De-register
Device Management and Service Enablement	Downlink	Create, Read, Write, Delete, Execute, Write Attributes, Discover
Information Reporting	Uplink	Notify
Information Reporting	Downlink	Observe, Cancel Observation

Object and Resource Model

The information in LWM2M Clients is stored as Resources and Resources are grouped into different Objects on the client. Hence, a LWM2M Client may have any number of Resources, each of which are grouped under an Object (see Figure 2-8).

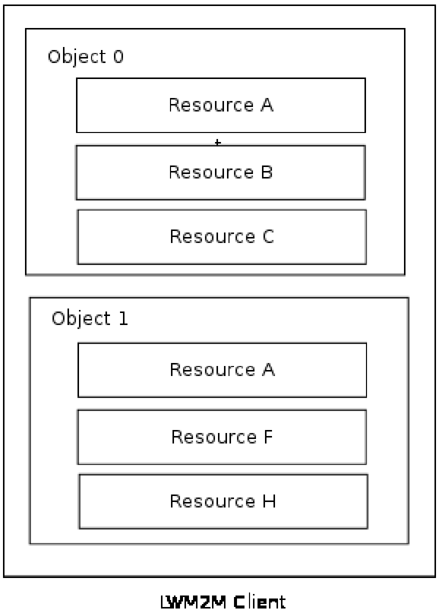


Figure 2-8 LWM2M Client, Object and Resource Overview

LWM2M Server performs the operations mentioned in Table 2.6 on the objects and resources to manage the device and its resources. Hence, the server can perform any allowed operation on the client. For example, LWM2M Server can read "*Serial Number*" resource (ID 2) or "*Battery Level*" resource (ID 9) from *Device* Object (ID 3) of the client with the ID 1895 (see Figure 2-9). Moreover, it can also delete the "*Access Control Owner"* resource (ID 3) from *Access Control* Object (ID 2) (see Figure 2-9).

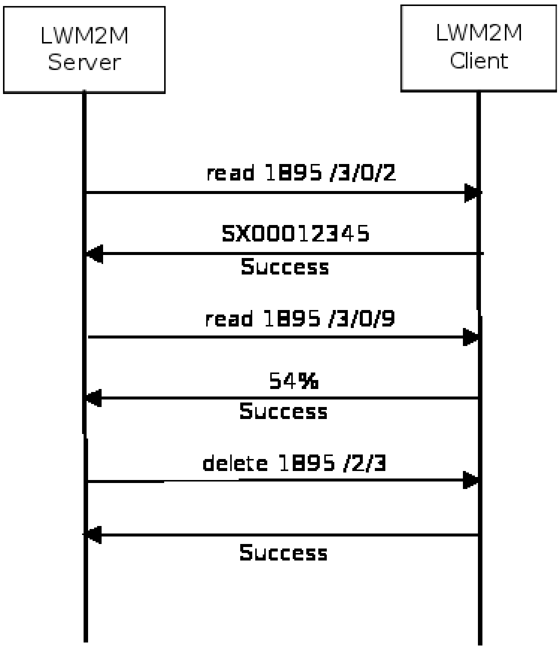


Figure 2-9 LWM2M Request-Response Example

The format of LWM2M objects is standardized by IPSO (IP Smart Objects) Alliance [25] to ensure the interoperability of the objects between different applications and environments.

2.3 Data Handling for IoT Device Management

Section 2.2 discusses the communication methods to be used for management of many IoT devices and the possible ways to move the data or management information from the constrained device to the management nodes and vice versa. However, the format of transmitted data also retains critical importance since the resources (power, connectivity, processing capability) are limited at the constrained node. Hence, the data formats used to communicate with the constrained device for the data exchange are required to be lightweight and versatile for optimum resource consumption. Moreover, the large amount of data from the many constrained nodes readings needs to be stored in the cloud for post-processing purposes (data analysis, error/anomaly detection, pattern recognition etc.). The storage method of the data determines the practicality and methodology of how the post-processing functions are performed. For these purposes, a data-store model (NoSQL) and a data-transmission/store model (JSON) are discussed in this section.

2.3.1 NoSQL

The large amount of data created by the emerging technologies (IoT, Mobile or Cloud Computing) has created new expectations from database management systems, which traditional relational database designs (e.g. MySQL, PostgreSQL) do not possess: high concurrent reading and writing rates, high scalability and availability, dynamic schemas, efficient very large data storage, easiness to expand and distributed architecture [26]. To satisfy these expectations for Big Data management, new type of database systems, which are very different than relational databases in design, have appeared and are referred with the term *NoSQL (Not Only SQL)* in general.

The design of NoSQL databases can be different from each other but each one of them shares similar advantages over relational databases:

- *Schema-free:* Relational databases require a fixed table (relation) defined before storing data (e.g. rows, columns, tables). If a new type is added to the table, the entire database must be re-altered. However in NoSQL databases, either new data values or new data types can be added dynamically without altering the entire database leading to great flexibility [27].
- *Horizontal Scalability:* Relational databases are vertically scalable, which is a single-server model for the host machine while NoSQL databases are horizontally scalable, meaning the load can be distributed automatically between different hosts without any complicated configuration procedure [28]. Horizontal scalability provides diverse distributed NoSQL databases.
- *Eventual Consistency:* NoSQL databases can support very high rate of concurrent create, read, update and delete (CRUD) operations effectively [29].

NoSQL databases are grouped into different groups according to their design architecture, some of which are:

- **Key-Value Store**: A key value corresponds to a value like a dictionary, without any structure or relation. Any type of information can be stored in either key or value attribute. Each key or value can store different data structure than other key-value pair, which provides high flexibility, scalability and faster query than SQL databases. Some example implementations are Redis, MemcacheDB, Oracle NoSQL etc.
- **Document Based**: The structure is similar to key-value store NoSQL but the value of the key is stored as a document in either JSON or XML format. This approach enables complex structured information trees to be stored in documents. Some examples are MongoDB, Elasticsearch, Couchbase Server etc.
- **Column Oriented**: The structure is similar to key-value store too. However in column oriented NoSQL databases, each key can be associated with one or more key-value pairs constructing a two-dimensional array i.e. value of a key can be another key-value pair. This nested, unstructured data architecture allows very large, aggregated data storage possibility. Examples include Cassandra, Hadoop (HBase), Apache Flink etc.
- **Graph Based**: The architecture is based on tree-like structures (i.e. graphs) where the objects (the data) are connected to other objects through static or dynamic relations. The data pieces are connected to each other and the relations between the objects are used to retrieve the data. Social networks (e.g. Facebook, LinkedIn) can be modeled as graph based databases. Some examples are OrientDB, Neo4J, Infinite Graph etc.

Different type of NoSQL introduces different type of advantages for IoT data storage over relational databases. For this thesis, several type of NoSQL databases are used for different purposes (see Section 2.5.2 IoT Framework and Section 5.2.3).

2.3.2 JSON

JSON (JavaScript Object Notation) is a widely used, lightweight, text-based, language-independent data exchange format [30]. It provides a nested data structure format for interchanging complex data structures but still is simple to parse even by the constrained devices considering the low overhead in JSON messages and JSON parsers requiring very little processor power.

JSON defines four primitive types: *strings, numbers, booleans, null* and two structured types: *objects* and *arrays* [30]. Objects can be other JSON messages inserted into the outer JSON message so that nested JSON objects can be created. Moreover, JSON supports arrays of any supported format (i.e. strings, numbers, booleans and objects) to be created. Both arrays and objects assure a detailed, structured model for aggregated data interchange. An example JSON message is presented below:

```
{
   "node": “2”,
   “devices”: “280:e103:1:2a9c”,
   "params": {
      "2":  {
        "lat": 41.35342516,
        "lng":  2.13135839,
        "280:e103:1:2a9c ": {
              "lat": 41.35361845,
              "lng": 2.13043571,
         }
      }
   }
}
```

Being lightweight makes JSON a good candidate for IoT device management message interchange format because many constrained devices can parse JSON messages without much power consumption. JSON messages can be attached to CoAP payload and LWM2M resources can be retrieved as JSON as well. Web services utilize JSON heavily since JSON integrates very well JavaScript. Hence, using the uniform JSON as the data interchange format in all protocols (CoAP, LWM2M, HTTP) decreases the complexity of integration between constrained environments and the Web significantly.

2.4 Features of IoT Device Management

The constrained environment in IoT devices and networks imposes IoT device management schemes to introduce the IoT-friendly (i.e. low power consumption, low processing power, lossy networks) management features. For instance, frequent polling of data from low-power devices is not optimal for power consumption and the IoT device management is expected to avoid such situations. In this section, we discuss several features of IoT device management which are addressed to optimize the use of constrained environments and to maximize the IoT data readings.

2.4.1 Publish/Subscribe

Publish/subscribe is a paradigm discussed in the concept of Information Centric Networking (ICN). ICN redefines the network purpose to provide information dissemination throughout the highly scalable distributed nodes rather than to provide pair-wise communication between the end points [31]. In the data-oriented network definition of ICN, data distribution between nodes can be achieved by nodes subscribing to information updates on other nodes and information being published to the subscribers by the information source [32].

The large number of IoT devices, the amount of data and management commands transmitted and the constrained environment of the devices make Publish/Subscribe mechanism a valuable solution for data distribution in IoT networks. Several reasons can be enlisted as why the Publish/Subscribe approach in ICN should be applied in IoT device management too:

- Data retrieval from power-constrained devices is a challenge since common polling mechanisms towards the device create unnecessary connections if the data on the device has not changed. Hence, the optimum way is the device ‘informing’ the network only if the data on the constrained device changes. This can be achieved by the network nodes subscribing to the device (the publisher) and it decreases the power consumption dramatically.

- The data retrieved from the device can be applied to several processes in the network, such as aggregation and abstraction, and the result of these procedures is transmitted to other entities. The information can be delivered with Publish/Subscribe scheme for event-driven, flexible data management to ensure scalable and dynamic network topology [33]. This approach proposes IoT networks to be an important part of ICN i.e. the Future Internet.
- Publish/Subscribe schemes introduce loosely coupled, asynchronous relations between the publisher and the subscriber [34]. The loosely coupled relation between the publisher and the subscriber permits the IoT device management to scale the management resources optimally in the cloud. However, the device management is required to track the list of subscribers and publishers in such case.

Considering the REST approach of client requesting the information from the server is not the efficient way to distribute the data in the constrained networks, CoAP introduces Publish/Subscribe paradigm as Observe/Notify for constrained devices [24]. Any client can observe any resource stored on the device and notifications are sent for information updates from the CoAP server (see Section 2.2.4 Observe/Notify in CoAP). Moreover, LWM2M supports Observe/Notify scheme as well since it runs on CoAP (see Section 2.2.5).

Observe/Notify scheme in CoAP and LWM2M introduces the advantages of Publish/Subscribe discussed above to IoT device management. Moreover, event-driven approach of Observe/Notify provides real-time network monitoring and controlling in IoT networks. Network manager by observing the resources on the device is notified instantly for information updates such as errors, logs, network topology changes and data updates, which is an important asset for real-time IoT device management.

2.4.2 Aggregation

Constrained characteristics of IoT devices require the management nodes of the IoT networks to be smarter to decrease the power consumption and processing required on the device [35]. Aggregation of data and management commands is one of the ways to achieve that target by means of decreasing the communication with the device and implementing the aggregation logic in the network.

In data aggregation model, the data from the device can be aggregated in different parts of the network before reaching to the cloud storage. A gateway or a proxy in the network aggregates the data from different resources according to the characteristics of the data such as the devices in the same neighborhood, same type of devices or similar data reading values. This type of aggregated data simplifies the post-processing data analysis by giving the insight about the entire set of data rather than individual data points [36]. It as well decreases the total amount of messages for the data points by grouping them into one message.

Aggregation of management commands implies a more important aspect for IoT device management. The network manager can send aggregated, different management commands to e.g. gateways and gateways capable of interpreting them split the messages to be sent to different machine devices. This decreases the traffic on the network since the several management commands are transmitted in one aggregated message. Moreover, grouping the devices and gateways to manager-defined clusters also allows the manager to send one group message to the entire group. Different from the aggregated message, group message is defined to be the same for the group elements. The distribution of the group message to each group element is performed in the network node where the group information is stored such as in a gateway or a proxy. During the research for this thesis, several patent applications have been prepared regarding the aggregation and group management for CoAP and LWM2M.

The main target of both aggregated and group messages is to decrease the communication required in the network by combining several messages, which leads to less power consumption and a more compact communication model.

2.4.3 Prioritization

As stated in Section 2.4.2, IoT device management nodes are required to be smart to increase the efficiency of the constrained networks. Prioritization of the messages in the network is yet another way to achieve that target.

Prioritization in communication networks allows several possibilities for different types of messages: High priority messages must reach the destination, medium priority messages are important but less sensitive to delay while low priority messages are less important with additional data [37]. To take the message priorities into account, new features can be added to routing protocols such as priority queues in each node or identity based priority assignments [37] [38].

The concept of prioritization in IoT device management indicates that either the management server or other entity in the network assigns different priority levels for messages transmitted to/from the device (e.g. CoAP or LWM2M messages). Each message with its own priority level allows the network entity (e.g. gateway, management node etc.) to classify the messages and act according to those levels. For instance, critical error messages from the device can be assigned high priority and can bypass the possible queues in the proxies, any aggregation node or re-routed in case of congestion to the management node. Similarly, firmware updates from the LWM2M server can be prioritized for critical security updates as well.

The critical asset the prioritization introduces to IoT device management is the support of classified messages. This implies additional benefits for real-time device management, increased monitoring and controlling capability over the entire IoT network.

2.4.4 Anomaly Detection

Constrained devices, typically sensors or actuators, transmit their data readings to the network frequently and the amount of the entire data becomes massive considering the large amount of devices. However, data readings of the devices can encounter anomalies due to the device hardware problems or other external factors. The device may not send any error messages regarding this; hence, the management node is required to detect the anomalies only from the acquired data readings in that case.

There can be several reasons why the device data readings can encounter anomalies, such as sudden, short time temperature increase in a temperature sensor due to external heat or light sensor displaying light appearance in a dark room. Thus, anomalies do not always refer to hardware or software errors on the device but can refer to unusual, abnormal environmental factors.

Detecting the possible anomalies is an important asset for the IoT device management to ensure the reliability of the data. The acquired data readings from devices are used for post-processing, analysis and decision-making. Hence, the users of the data should be assured about the data reliability, which can be achieved by anomaly detection algorithms.

The challenge is, though, the amount of the data and the variety of the data sets. Anomaly detection algorithms are generally sequence or pattern based, which use statistics, classifiers or machine learning techniques to detect the anomalies [39] [40]. With the immense variety of IoT device data types, it is still a big challenge to perform anomaly detection in any type of data i.e. anomaly detection algorithms are tailored for specific types of data sets, which creates integration complexity for other sets of data. At the time of the research, anomaly detection in IoT is still an immature topic. However, IoT device management can include anomaly detection support for a specific type of data set for the proof of concept work.

2.4.5 Error Reporting

Machine devices are generally geographically wide-spread, which makes them unreachable in case of an error on the device. This introduces the need of remote error monitoring for IoT network. However, machine devices are not connected to the network all the time, which might lead to latencies in error reporting. Hence, IoT device management requires a solution for error reporting of the machine devices and gateways.

For an efficient and versatile error reporting mechanism, use of already existing systems rather than introducing a new protocol decreases the integration complexity. CoAP and LWM2M systems provide a great basis for building such an error reporting mechanism for IoT devices. Use of CoAP Observe/Notify scheme together with relevant objects and resources of LWMWM clients (e.g. device battery level resource, connectivity error resource etc.) is a great way of real-time error reporting for IoT devices in the already existing architecture i.e. device management interface can observe relevant resources on LWM2M client for instant error monitoring.

As a combined solution of Publish/Subscribe scheme and LWM2M objects, real-time error reporting is an important feature of an IoT device management interface.

2.5 Capillary Networks

In order to provide end-to-end IP connectivity for Capillary Networks (see Section 2.1.2) and constrained devices in the real world, a high-level network architecture has been designed combining the features discussed in Sections 2.2, 2.3 and 2.4. Capillary Networks is a proof of concept to connect billions of constrained devices to the network and to the cloud by providing features such as dynamic cloud resource allocation, smart network decisions, automatic configuration etc. It consists of three different domains: the Capillary Network, the connectivity domain and the data domain. Figure 2-10 presents a detailed view of CN architecture with additional functional views.

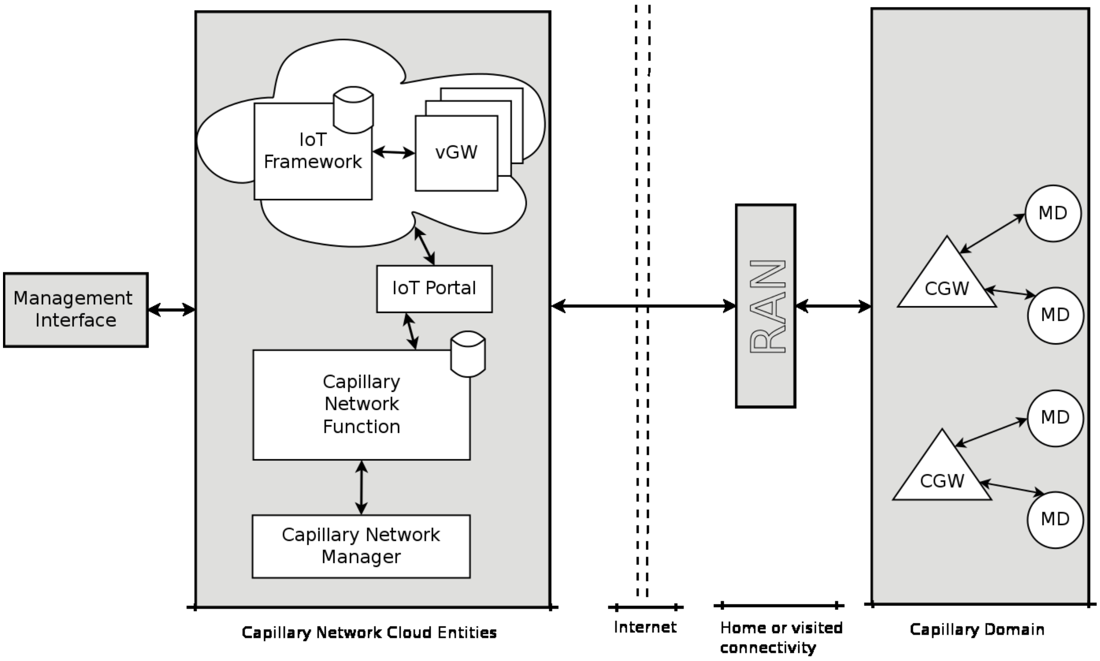

IoT Framework
vGW
IoT Portal
Management Interface
Capillary Network Function
Capillary Network Manager
RAN
CGW
MD
MD
CGW
MD
MD
Capillary Network Cloud Entities
Internet
Home or visited connectivity
Capillary Domain

Figure 2-10 Capillary Network Architecture Overview

From the Figure 2-10, the Capillary Networks architecture consists of the following components:

- **Machine Device (MD)**: Constrained devices such as sensors and actuators.
- **Capillary Gateway (CGW)**: The gateway connecting the machine device to the backhaul network.
- **Capillary Network Manager (CNM)**: Manual network management component for Capillary Network monitoring and configuration.
- **Capillary Network Function (CNF):** Automatic network management component for Capillary Network connectivity controlling and cloud resources monitoring.
- **IoT Portal:** Middleware component for CNF to control and initiate cloud resources.
- **Virtual Gateway (vGW):** Cloud resources created for each CGW by CNF i.e. virtualized instance in the cloud for the CGW.
- **IoT Framework:** Data storage for storing MD data readings and meta data about devices in the network.
- **Management Interface:** The management interface, which provides the network manager all the required tools for Capillary Networks management.

> This part of the Capillary Networks architecture forms the topic of this thesis. The management interface implementation was built on top of the existing CN architecture. It uses many of the existing CN components, updates some of them and defines new interfaces between CN entities.

Section 2.5.2 presents the more detailed explanations for each Capillary Networks component. Irrelevant components for this thesis, such as connectivity domain components or network security have been left out.

Before the CN components, we first discuss the features of the Capillary Networks in the next section.

2.5.1 Features of Capillary Networks

Creating end-to-end connectivity for the Capillary Networks and constrained devices is the main feature of the Capillary Networks but the technology presents more aspects to M2M communications. As the number of CNs and CGWs is expected to be very large and the location of these devices can be difficult to reach, CN focuses on automated management methods for CNs and MDs. This approach targets at simplifying the management by using automation but also utilizing cloud technologies together with CNs. Hence, the CN technology introduces the following features:

- **Automatic configuration:** The motivation for automatic configuration is to simplify network management and to make installing of new MDs and CGWs much easier. New MDs are connected to CGWs without additional, complex installation procedures. This includes configuration of IP addressing and routing, obtaining CGW prefixes from the network and assigning prefixes to MDs.
- **Dynamic middleware:** This feature enables to create virtual gateways on demand to provide middleware functions for connected CGWs and MDs. Hence, each CGW is assigned a virtualized instance in the cloud for this purpose. Cloud resources can dynamically be changed to adapt to the needs of the network.
- **Dynamic gateway selection:** The MD is indicated to connect to the optimal CGW based on various constraints and policies i.e. enabling constrained devices to switch CGW automatically based on the policies. This is an important feature as it brings this automation to the CN domain.
- **Data and storage:** In addition to network aspects, data gathered in IoT Framework from various MDs is processed and monitored in the CN technology. MD data readings are the main point of interest in data domain of the CN. Hence, the presentation of the data to the end user is an important aspect of the CN technology.
- **Security:** Authentication of CGWs and MDs is the main aspect of CN security along with signing the data readings of MDs. The authentication of CGWs is sim-card based using GBA (Generic Bootstrapping Architecture) procedure [41]. GBA authentication is performed between CNF and CGW. Hence, the generated session key after authentication is available both in CGW and in

the backend cloud in CNF for future encryption purposes between CGW and the cloud.

2.5.2 Capillary Networks Components

Machine Device (MD)

Machine devices (MD) are constrained devices in the capillary network that are connected to a Capillary Gateway via some short-range radio technology (e.g. IEEE 802.15.4 or Bluetooth Low Energy). Sensors and actuators are possible examples of MDs. For an MD to be part of a Capillary Network, it is assumed that MD has an IP stack, providing typically IPv6 addresses. This assures the end-to-end IP connectivity between the constrained device and other CN components.

MDs can act either as a client, in order to initiate communication or as a server in order to respond to a request. They can also take both client and server roles depending on how the CoAP and LWM2M implementations indicate.

The testbed for CN includes various sensor and actuator types acting as MDs. Both are implemented with Contiki on STMicroelectronics boards and are accessible via CoAP.

Capillary Gateway (CGW)

The main role of the capillary gateway is to connect the capillary network of MDs to the backhaul cellular network. Similar to MDs, CGWs also have an IP stack which consists of IPv6 addresses towards MDs and IPv4 or IPv6 addresses towards the backend haul. At the application layer, CGWs used CoAP towards the MDs and HTTP towards the backend.

Different from MDs, CGWs are more advanced devices, which can handle complex functions such as: performing CGW selection, storing a CoAP Resource Directory or a CoAP Mirror Server. The core functions of CGW include transferring control messages between MDs and backhaul, transferring MD data to backhaul and informing backhaul about the CN configuration.

The testbed for CN includes CGWs implemented using OpenWRT on Buffalo routers with 3G/4G connectivity. An additional Contiki based RPL (Routing Protocol for Low Power and Lossy Networks) root device is connected to CGW to provide IEEE 802.15.4 connectivity between CGW and MDs.

Capillary Network Function (CNF)

The Capillary Network Function (CNF) is a control function mainly designed for automatic network management such as managing the connectivity for the capillary network and controlling the instantiation of virtual gateway machines via the IoT Portal. Moreover, CNF applies the user-defined policies to CN management.

The communication between CNF and the network elements, such as CGW, IoT Portal and other middleware, are done via an HTTP REST interface. Similar HTTP

REST interfaces have been implemented to communicate between CNF and the management interface, which will be discussed in the upcoming sections.

For the testbed, CNF is implemented in the Internet rather than a closed network. CNF also enables to update battery and load constraints of CGWs virtually to demonstrate automatic gateway selection feature.

Capillary Network Manager (CNM)

The Capillary Network Manager (CNM) is yet another control function, which is mainly aimed at manual network management, monitoring and network configuration by providing required tools and services. The manager can set user-defined management rules/policies via CNM for CNs as well as monitor and update CN devices, parameters and counters.

Similar to CNF, the communication between CNM and other network nodes is done via an HTTP REST interface. As CNM is an important node for manual CN management, monitoring and configuration, several HTTP REST interfaces have been implemented to integrate CNM to the management interface.

IoT Portal

The IoT Portal is the network node between CNF and the cloud services. It provides an interface for CNF to create and control the configuration of virtual machines (i.e. vGWs) in the cloud. This node is not visible to the management interface and is under control of CNF.

Virtual Gateways (vGWs)

Virtual gateways are cloud resources automatically created by CNF for each physical CGW to provide middleware functions on demand in the cloud. vGWs can implement Mirror Proxies and Resource Directories, store meta data about CGWs or forward MD data to IoT Framework. The main advantage of vGWs is to provide CGW data on demand even when CGW is not reachable physically.

IoT Framework

IoT Framework is a data storage framework that stores data from MDs and provides services to manipulate the data for several other uses, such as semantics search and data analysis [42].

To provide a versatile and stable data storage for a large amount of MD data, IoT Framework architecture consists of four hierarchical entities:

- **Users:** Actual human users create user accounts and they are assigned a unique user id.
- **Resources:** Each resource represents one physical MD. A user can create many resources and the ownership of resources belongs to that user. Resources are given unique ids and can also store meta-data about MDs.
- **Streams:** Streams represent a separate data stream on each resource i.e. it can

also be referred as data streams. One resource can own several streams, for example a temperature sensor and a humidity sensor can belong to the same resource. However, in that case there would be two different streams for those sensor readings in IoT Framework. Each stream is assigned a unique ID upon creating, stream data is accessible via this id.

- **Datapoints:** Each stream consists of datapoints received from a resource (i.e. MD). Hence, each reading from an MD is stored in the IoT Framework as a datapoint. Datapoints are not accessible alone, they need to be retrieved using the Stream ID they are attached to. Figure 2-11 shows the overview of hierarchical IoT Framework entities.

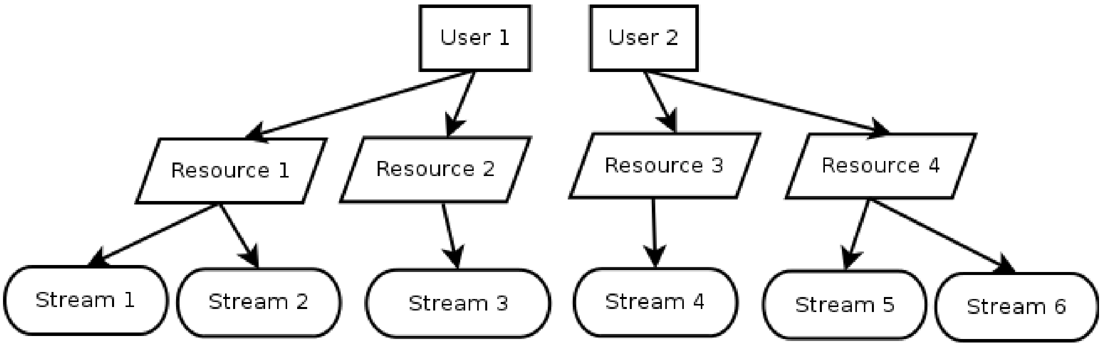


Figure 2-11: IoT Framework User, Resource and Streams

As depicted in Figure 2-11, each stream can belong to one resource and each resource can belong to only one user. This hierarchical design can be related to real life where one device (resource) belongs to only one user. But if the ID of the device is known, data readings of the device can be retrieved easily from the IoT Framework.

Since the IoT Framework users indicate the owners of the devices (MDs), the network manager is supposed to access the data of each user. Hence, network manager can be defined as a *super user* of the IoT Framework.

The IoT Framework uses Elasticsearch as the datastore. Elasticsearch provides a document like NoSQL datastore where the user can interact via REST API [43] (also see Section 2.3.1). Creating new documents requires a POST request containing a JSON object while a GET request also retrieves JSON object. All entities explained above are separate documents in Elasticsearch and Elasticsearch does not handle the relations between each of them. The entity simply holds the id of its owner in its document, different from traditional SQL primary key schemes.

The IoT Framework provides a rich API to create, read, update, delete and search users, resources and streams. Moreover, a publish/subscribe API for data points and full text search using Elasticsearch datastore [43] is also provided. Full text search support from Elasticsearch makes the IoT Framework streams and stream data easily reachable by understanding queries like “Temperature in Helsinki”. Semantic search queries enable the network manager to access and monitor the stream data clearly from all users i.e. the network manager has access to the data from all IoT Framework users. Hence, the IoT Framework APIs are extensively used in the management interface where MD data readings are presented to the end user (the network manager).

From the data security point of view, the data readings (as well as meta data of users, resources and streams) are not stored encrypted in the IoT Framework. The encryption can be performed only for transmission of the data from CGW to the cloud. Since CGWs are authenticated using GBA mechanism through CNF, each CGW retains a session key, which can be used for encrypting the data readings (as mentioned in Section 2.5.1). However, if the data gets encrypted in CGW, it is decrypted in the cloud so that search functionality is available in the IoT Framework.

2.6 Summary

In this chapter, the concept of Internet-of-Things (IoT) has been introduced. IoT is the technology to connect many of tiny devices to the network to create a versatile, horizontal and scalable infrastructure for creating diverse applications and uses from the connected world i.e. moving the real world to the IP-based digital world. End devices in IoT are generally constrained devices (e.g. low-power sensors), which result in different needs for the constrained networks such as connectivity of constrained devices, IoT-specific features of the network and the management or the use of constrained communication methods.

CoAP (Constrained Application Protocol) is an emerging application protocol specifically designed to work in constrained environments over UDP or SMS. CoAP also provides solutions for IoT-related challenges such as resource discovery or lightweight message transaction. It also provides specifications for Observe/Notify scheme to remove the need of data polling to decrease the network communication of the device.

LWM2M, however, is a specific device management protocol for constrained environments. Running over CoAP makes LWM2M easy to integrate with the constrained networks already implemented in CoAP. Moreover, object/resource design in LWM2M assures the management information to be in compact and easily reachable format.

The vast amount of data produced by the devices and the constrained networks requires two different approaches for data handling in IoT. For data communication with the device, a lightweight data format such as JSON is needed to minimize the size of the message. However, data storage in the cloud requires more features such as flexibility and scalability, which can be found in NoSQL solutions.

An IoT device management interface (portal) is required to wrap the entire network architecture, protocols, and communications; and to provide the end user with the relevant management features and data. Capillary Networks concept combines the mentioned specifications of an IoT network and hence, a management interface for the Capillary Networks infrastructure can demonstrate the IoT device management concept.

3 Requirements

In this chapter, we present the system requirements for the management interface. The chapter consists of two sections, which are divided to comply with the two main parts of the management interface:

- **Frontend**: The user interface visible to the end user.
- **Backend**: The server side that handles the logic and prepares the user interface with data.

This separation between the frontend and the backend is followed throughout the thesis since the requirements, design and implementation for each of them is quite distinct.

3.1 Frontend Requirements

Requirements for the frontend aims to present the features of the CN prototype (see 2.5.1) to the user in a modern, interactive and user-friendly way. Moreover, the frontend needs to provide more functionality to show the interface as an entire IoT device management platform, rather than just demonstrating the CN prototype. The requirements of the front end are presented in the following Table 3.1. While some of the requirements mentioned in the table are essential, some of them are targeted as additional features (not strictly essential) to the management interface. This difference between the requirements is depicted in "Importance" column of Table 3.1.

Table 3.1 Requirements for the Management Interface's Frontend

Feature	Requirement Explanation	Importance
Automatic configuration	The frontend needs to display automatic configuration of CN, MDs and CGWs. Hence, any new device (MD or CGW) or any change in the existing devices are required to be reflected to the end user. The user should perform minimum amount of extra steps to see these updates on the main page. Network topology changes need to be updated on the interface without affecting user interaction. Moreover, connection between MDs and CGWs is needed to be clear to the user.	Essential
Dynamic Middleware	The representation of dynamic middleware is required to be linked to each corresponding CGW (i.e. vGWs of each CGW should be indicated). The user should be able to get the information about virtual machines created as middleware in the cloud.	Essential

Dynamic Gateway Selection	It needs to be presented to the user on the main page. MDs changing CGWs should be easily detectable, should be updated instantly and should not affect user interaction with the interface. The interface needed to trigger the gateway selection process i.e. updating battery and load level constraints should be provided as well.	Essential
Capillary Networks Policies	The user should be able to control the policies for dynamic gateway selection. This is a core functionality of the manager to control the policies in CNs.	Essential
Machine Device Data Representation	As each MD will store data in the IoT Framework, the data from the IoT Framework is required to be assigned to the corresponding MD. Data needs to be shown in an easy and interactive way to the user, since its amount can be very large and hard to present in traditional ways.	Essential
Machine Device Data's Security Representation	Demonstrating security of CN prototype in the management interface is slightly less visible to the user as it is assumed that the authentication of CN devices is already handled in the CN architecture. However, the frontend should be able to provide information if each data point is sent to the IoT Framework encrypted or not.	Additional Feature
Search Functionality	The frontend should provide search functionality for searching through the IoT Framework's streams and resources. Stream data should be presented in a similar way, showing MD data readings.	Additional Feature
Meta Data Representation	The meta data of CGWs and MDs is required to be shown to the user to provide more information about these CN devices. In short, the user i.e. the capillary network manager should have access to as much information as possible.	Essential
Error Reporting	The frontend is required to present the errors or important messages received from CGWs or CN entities to the user instantly. One method to present these messages can be in form of notifications on the navigation panel. Similarly, logs and network counters from each CGW are to be shown on the interface for full network monitoring functionality.	Essential
Congestion Demonstration	The frontend is required to show a congestion demonstration scenario where tens of CGWs are simulated in the same proximity, using the same radio frequency and creating interference in the area. The demonstration presents the elimination of the interference created by the simulated CGWs. Frequencies of the CGWs are re-assigned automatically (self-healing). The frontend is required to reflect this type of self-healing of CGWs to the user.	Additional Feature

Anomaly Detection	Detected anomalies in MD data readings are needed be presented to the user. Management interface should give choice to apply anomaly detection algorithm on MD data or not. In case the anomaly detection is executed, anomalies should be visualized.	Additional Feature
LWM2M Server Integration	A separate screen is needed to demonstrate the communication with LWM2M server and the end user (e.g. reading resources from LWM2M server).	Additional Feature

Table 3.1 indicates that the frontend is required to contain several views to provide numerous functionalities to the end user. Since some of the requirements are marked "Essential", design and implementation of these requirements are analyzed more in detail. Requirements with "Additional Feature" level are designed and implemented as well. However, the full functionality may not be available in the frontend for these requirements. Evaluation of the requirements is done in Chapter 6.

3.2 Backend Requirements

Requirements for the management interface backend are determined so that all the features of the CN prototype, explained in Section 2.5.1 are presented to the end user with stability, versatility and reliability. Since the CN concept prototype consists of several components, the management interface backend requires to communicate with these components via several interfaces. It also needs to maintain the high-level management logic between CN middleware components. Hence, our management interface backend needs to be designed as a standalone application on top of CN components. The requirements of the backend implementation are presented in Table 3.2 below in the same format used for the frontend requirements in 3.1. Similar to frontend requirements in Table 3.1, some of the requirements in the table are marked essential and some of them are targeted as additional features (not strictly essential) to the management interface. This difference between the requirements is depicted in "Importance" column of Table 3.2.

Table 3.2 Requirements for the Managements Interface's Backend

Feature	Requirement Explanation	Importance
Automatic Configuration	The management interface backend should accept status updates from CNF at anytime to present automatic configuration of the CN devices to the user. These updates need to be processed and pushed to the frontend for the end user, in form of either IoT network topology changes, notifications, logs or network counters. Since real-time updates are important for network topology changes, network topology i.e. the CN device list can be pushed to the frontend at all times.	Essential

Dynamic Middleware	Information about dynamic middleware created for each CGW and MD needs to be retrieved from CNF and presented to the user in the frontend as also explained in 3.1.	Essential
Dynamic Gateway Selection	The backend is required to provide the communication between the frontend and CNF to perform dynamic gateway selection. For this purpose, battery and load constraints need to be forwarded to CNF for further processing. The results of dynamic gateway selection which is MDs changing CGWs i.e. CN topology changes are to be pushed to the frontend immediately as well.	Essential
Capillary Networks Policies	The backend should retrieve the CN policies from CNF and forward the CN policy updates from the frontend to CNF in case the policies are updated.	Essential
Machine Device Data Representation	The backend needs to support the IoT Framework APIs to be able to read, update, create, delete and search resources and streams. The large amount of MD data retrieved from IoT Framework also needs to be forwarded to the frontend. Backend implementation needs to format the data retrieved from the IoT Framework to create the required data format by the frontend design.	Essential
Machine Device Data's Security Representation	The backend assumes the network level security between CN nodes is already provided by CN middleware components and GBA [41]. Therefore, the backend is not required to implement CGW security. However, the backend is responsible of retrieving the information from the IoT Framework if the data points are transferred as encrypted or not (see Figure 5-2 item 3 and check Section 2.5.2).	Additional Feature
Search Functionality	The backend needs to have the feature to perform search queries on the IoT Framework users, resources, streams and data points. The query results needs to be provided to the frontend as well. (It is assumed that data on the IoT Framework is not encrypted to perform the search.)	Additional Feature
Meta Data Representation	Meta data information about CGWs and MDs needs to be fetched from each vGW and combined with other meta data information retrieved from the IoT Framework and CNF. The retrieved meta data by the backend are shown in infobubbles or in detailed information view of the CGW or MD.	Essential
Error Reporting	To provide the relevant data from the CN architecture about error reporting (e.g. list of notifications and logs from CGWs), the backend needs to harvest the data from the CN entities and forward it to the frontend.	Essential

Congestion Demo	To present the congestion demonstration, backend implementation should have access to congestion demo interface of CNF in terms of using REST APIs. Congestion parameters from the frontend are forwarded to CNF and the response from server, which includes many CGWs locations, is forwarded back to the frontend after iterating through it to format the response data.	Additional Feature
Anomaly Detection	Anomaly detection implementation needs to be triggered with the provided datapoint set when the end user prefers. However, anomaly detection was implemented in Matlab as a temporary solution. The interface between this Matlab program and the backend is required to be defined. (Matlab implementation is not meant for production.)	Additional Feature
LWM2M Server Integration	LWM2M server on the backend requires interaction with the end user. Hence, the backend implementation needs to read the input from LWM2M terminal window in the frontend and send this input to LWM2M server as a request. Similarly, the response should be forwarded to the frontend as well.	Additional Feature

Table 3.2 indicates that the backend is required to provide several functionalities to the frontend presentation. Similar to frontend requirements in 3.1, the requirements marked “Essential” are given more importance in design and implementation. Requirements with “Additional Feature” level are designed and implemented as well in the backend. However, the full functionality may not be available for these requirements. Evaluation of the backend requirements is done in Chapter 6.

3.3 Summary

This chapter presents the requirements of the management interface to be implemented on Capillary Networks architecture. The requirements for frontend and backend of the interface are dependent on the Capillary Networks requirements and hence, requirements are classified accordingly. In addition to present Capillary Networks features, the interface is required to follow the basic expectations from the device management such as device monitoring, controlling, data presentation and error reporting. Chapter 4 discusses the system design to satisfy the requirements introduced in this chapter.

4 Design

In this chapter, we present the system design for the management interface using the requirements defined in Chapter 3. The chapter consists of two sections similar to Chapter 3:

- **Frontend**: Design of the user interface and graphics.
- **Backend**: Design of the backend server, its components and the communication with the Capillary Networks components.

We start the chapter with frontend design focusing on user interaction and continue with backend design introducing the backend server components of the management interface.

4.1 Frontend Design

User interface design of the management interface is based on the requirements mentioned above. The main idea behind the design is to create a responsive, modern and simple interface, which is easy to read and interact, concerning the large number of possible IoT devices. Hence, simplicity is the main driver of the user interface. This section discusses the graphical decisions made for the frontend design. The implementation of the frontend is discussed in Section 5.1 while the backend design to develop the frontend is presented in Section 4.2.

As the starting point of the user interface, the navigation panel of the interface is placed on top of the page. The navigation panel includes links to "Home", "Network Elements" and "Network Info" pages along with search box, notification panel and user information panel (see Figure 4-1).

To start with "Home" page of the interface, the requirement was to show MDs and CGWs to the user in a simple way. Instead of showing MDs and CGWs in some sort of table presentation, using a map was chosen with MDs and CGWs displayed on the map. The reasons why a map was chosen as the main page are:

- The current trend in IoT management systems is mainly aiming at showing the locations of IoT devices. This provides the network manager an easy overview of the devices on the map. Showing large number of devices in a simple way can also be performed by clustering the nearby devices on the map. The network manager can zoom in to see devices separately in the interested areas.
- Connections between MDs and CGWs are easier to visualize on the map than table representations.
- Some advanced devices are capable of using GPS nowadays. Hence, coordinate information of the devices is generally available. In case of low power IoT devices without GPS capabilities, the location of non-mobile devices can

be configured manually during installation.

- Map APIs provide very large choices to represent different types of data on the map i.e. visualization of data on the map is more user-friendly than tables.
- Web maps are interactive, that gives the network manager more options to monitor the network.
- In the future, location based analysis of IoT devices will be possible with the use of online maps and the gathering of information from these maps.

To show the connection between the CGW and MDs, a circle with the CGW in the middle and CGW's range as its diameter is drawn on the map. Connection between the CGW and MDs falling inside this circle is depicted as a line between CGW and MD icons. This way of presenting is aimed at making the connection visible on the map at the first glance of the user.

If an MD changes CGW as a result of automatic gateway selection, MD icon will be connected to the new gateway with a new line replacing the old line. The visual transition of MD changing gateway should be smooth on the map and should not interrupt the user behavior e.g. map freezing while updating CGW and MD connections on the browser is unacceptable.

Having a responsive home page with an online map introduces several functionalities on the map. Some of these functionalities include zooming into different locations quickly or providing CN, COMMUNE and LWM2M demonstration options to the user. For these two requirements, it is designed to have drop-down menus on the right top of the home page (on the map) named as "Locations" and "Options" (see Figure 4-1). The reason of selecting drop-down menus is to minimize the number of options visible on the map and to keep the map simple.

As the main target user of the interface is the network manager, the interface should provide a screen for the user to retrieve information about CGWs and MDs. For this purpose, CGW and MD icons are clicked individually to pop-up a short information bubble (summary information) on the icon. The user can access more detailed information about CGW or MD by clicking "More Information" button on the info bubble (see Figure 4-1). "More Information" button opens a floating window over the map where detailed meta-data about devices or MD data readings can be displayed. As the floating window space is large enough, "Logs" and "Counters" from CGW is added to the available space. The reason of designing the access to the device data in this way (digging in for details) is another goal to keep the main page simple.

MD data readings are shown on the floating window of the corresponding MD. The large number of datapoints of an MD requires a more elegant way to show the data than just datapoints i.e. not just table representation. Hence, datapoints are decided to be presented in a fully interactive graph where the users can zoom-in, zoom-out, reach each single datapoint's details via hovering and navigate through the graph. The graph also gives an insight of the whole datapoints in general to the user.

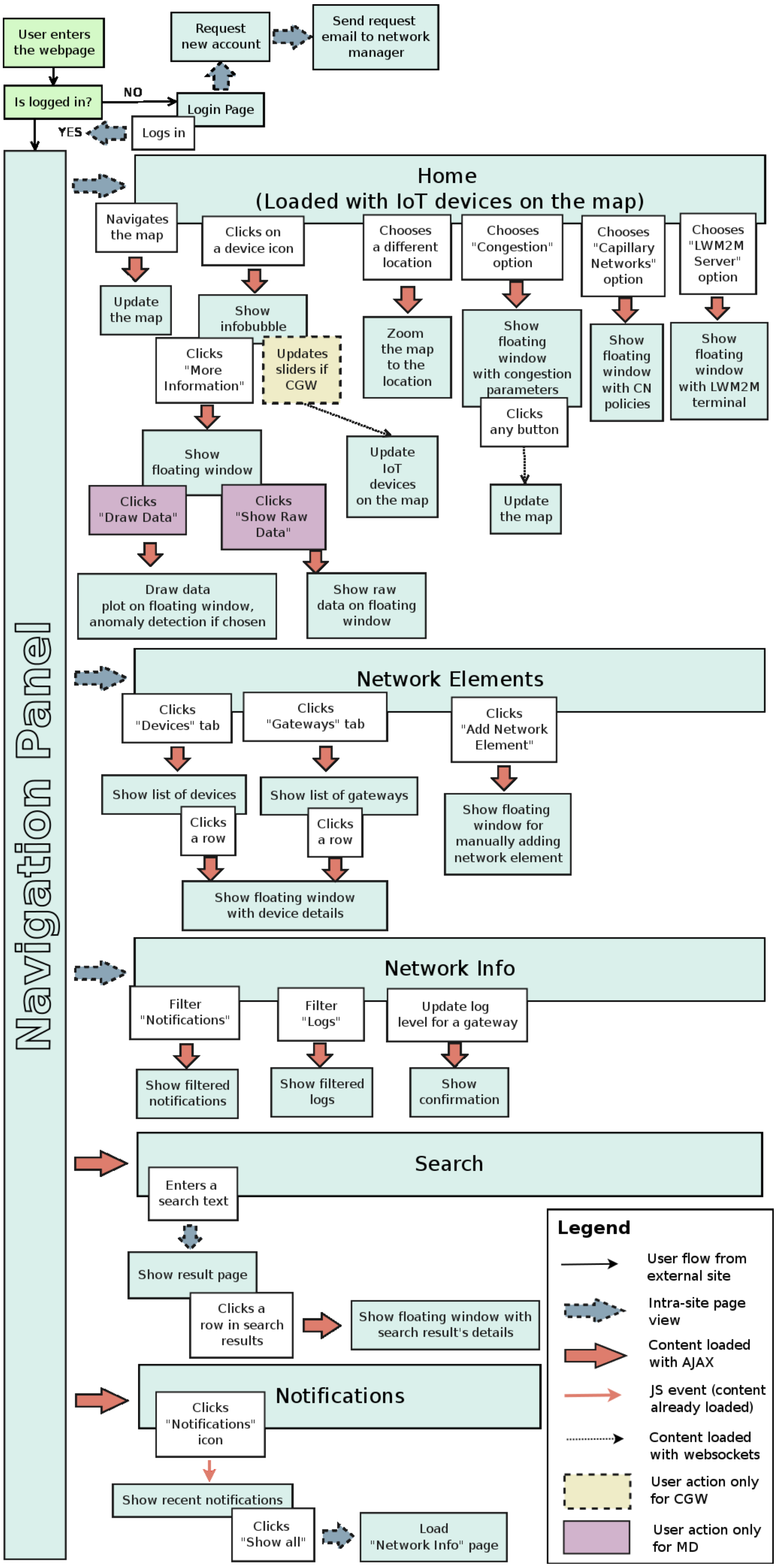

User enters the webpage
Is logged in?
NO
YES
Request new account
Send request email to network manager
Login Page
Logs in
Navigation Panel
Home
(Loaded with IoT devices on the map)
Navigates the map
Update the map
Clicks on a device icon
Show infobubble
Clicks "More Information"
Updates sliders if CGW
Show floating window
Clicks "Draw Data"
Clicks "Show Raw Data"
Draw data plot on floating window, anomaly detection if chosen
Show raw data on floating window
Chooses a different location
Zoom the map to the location
Update IoT devices on the map
Chooses "Congestion" option
Show floating window with congestion parameters
Clicks any button
Update the map
Chooses "Capillary Networks" option
Show floating window with CN policies
Chooses "LWM2M Server" option
Show floating window with LWM2M terminal
Network Elements
Clicks "Devices" tab
Show list of devices
Clicks a row
Clicks "Gateways" tab
Show list of gateways
Clicks a row
Show floating window with device details
Clicks "Add Network Element"
Show floating window for manually adding network element
Network Info
Filter "Notifications"
Show filtered notifications
Filter "Logs"
Show filtered logs
Update log level for a gateway
Show confirmation
Search
Enters a search text
Show result page
Clicks a row in search results
Show floating window with search result's details
Notifications
Clicks "Notifications" icon
Show recent notifications
Clicks "Show all"
Load "Network Info" page
Legend
User flow from external site
Intra-site page view
Content loaded with AJAX
JS event (content already loaded)
Content loaded with websockets
User action only for CGW
User action only for MD

Figure 4-1 User Interface Flowchart

The second link in the navigation tab is “Network Elements”. In this page, the idea is to list all the available CGWs and MDs in a table format with added search function. Although presenting devices on an online map uncovers many advantages such as interacting with the map or better visual representation, network manager should have the chance to monitor the devices as a list too. Adding table representation feature gives more freedom and options to the user, which is reason of this tab.

The third link in the navigation tab is “Network Info”. This page aims at providing the entire network monitoring information such as logs and notifications from CGWs to the user. The option to filter logs and notifications are also provided in case the network manager requires checking individual CGWs. The page is available to add more content concerning any network monitoring data.

The search box in the navigation tab is meant for any type of search the network manager wants to perform e.g. the users, data streams, device meta data etc. It is placed on the navigation panel for easy access.

Notification panel is used to inform the user immediately when a notification is received from a CGW, as part of error reporting feature. The number of unread notifications will be visible to the user and the user can expand the panel to access to notification details. The level of notification can be defined manually to control the number of notifications received. Notification panel is an important asset to the management interface as it provides real-time IoT network monitoring for the network manager.

4.2 Backend Design

In this section, we discuss the general overview of the backend design, which was created to satisfy both the backend requirements mentioned above and the frontend requirements. An overview of the backend design is shown in Figure 4-2.

The most important design aspect for the management interface backend architecture was to create a standalone solution on top of CN entities with minimal change in the CN architecture. The reasons behind a standalone solution are:

- There can be several different CN entities each dedicated to separate customers or different IoT projects. Hence, the management interface needs to be a modular solution to work with different CN instances with minimal changes i.e. different CN instances can import their own management interface easily.
- When the management interface is a standalone application, it can be located in an internal network in which the only public IP is assigned to the management interface. Securing the access to the interface and storing CN entities in an internal network provide a more secure environment i.e. the management interface is used as secured proxy between the public internet and CN entities.

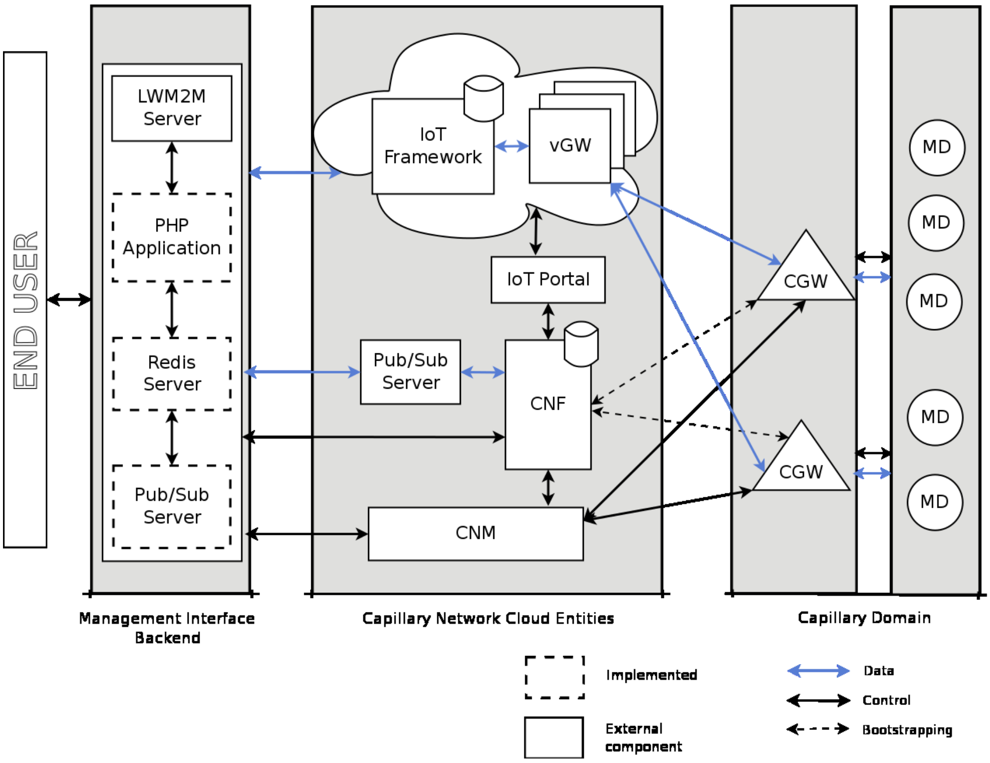


Figure 4-2 Management Interface Backend Architecture

As the communication between CN entities and the management interface is performed using REST APIs, the management interface can be installed anywhere on the network where it can access CN entities.

As depicted in Figure 4-2, the management interface backend is designed to consist of four main components:

- **PHP Application**: The main logic of the web server is supplied by this application. It receives the most of the user requests directly, retrieves required data from or update the required fields in the CN cloud entities and responds with the formatted user interface data. PHP application mainly provides the static pages in the user interface. Hence, it does not support real-time updates, which are performed by the Pub/Sub server. PHP application is explained in Section 5.2.1.
- **Pub/Sub Server**: To provide real-time updates on the user interface, a publish/subscribe server is needed in the backend since PHP is designed to provide only the static pages in the implementation. Pub/Sub server communicates with both the Pub/Sub server in the CN cloud and the PHP application. Pub/Sub server is discussed in Section 5.2.2.
- **Redis Server**: This component is needed to perform the data transfer between the PHP application and Pub/Sub server, even though they are installed on the same local machine. Redis server is presented in Section 5.2.3.
- **LWM2M Server**: To use LWM2M remote device management protocol on CN devices (see Section 2.2.5), the LWM2M server is to be installed on the

backend. Integrating LWM2M protocol to the management interface enables a new way to access constrained devices (e.g. MDs) directly from the interface. It is integrated via socket communication with the PHP application. LWM2M server integration is discussed in Section 5.2.4.

- **REST APIs**: Communication between management interface components and the CN entities is designed to be done mainly using REST APIs. Those REST APIs provide the integrity and the standardized communication between all the components, which creates an important part of the implementation. Moreover, this approach gives the chance of updating any entity source code in the system freely, with the only requirement of keeping the compliance to REST APIs. Hence, CN entities can be implemented as a black box concept and only provide the interfaces the management interface requires.

In Chapter 5, we discuss the implementation details of each management interface backend component in detailed in separate sections. Additionally, REST APIs between these components are also discussed in a separate section with several examples.

4.3 Summary

In this chapter, the frontend design flow of the interface is discussed first with the focus of keeping the interface as simple as possible. Moreover, the backend design to support the frontend functionalities and to communicate with the other Capillary Networks entities is introduced as well. The backend is designed as a wrapper, standalone application to present the Capillary Networks functionalities. The implementations of the frontend and backend, discussed in Chapter 5, are based on the design principles set in this chapter.

5 Implementation

In this chapter, we present the implementation steps of the management interface. The chapter starts with the implementation of the frontend of the interface and is followed by backend implementation details.

5.1 Frontend Implementation

As the backend implementation depends on the functionalities provided by the frontend design, we start the implementation discussion with the frontend section. Hence, in this section, details of the management interface frontend implementation are discussed. The section also focuses on user interaction with the frontend from IoT perspective.

Basic design aspects of the user interface were discussed in Section 4.1. To follow the design principle of simplicity, several tools were used in implementing the user interface such as HTML5 [44], CSS3 [45] and JavaScript [46].

To give the user interface a modern look, the website interface elements were built on top of the open source Bootstrap framework [47]. Bootstrap is a complete frontend framework with HTML, CSS and JS libraries providing HTML elements, CSS components and jQuery [48] libraries. To create the navigation tab, general page layouts, input elements such as textbox and buttons, tables, floating windows and the general website look, Bootstrap provided the HTML elements and CSS components that we used. Nevertheless, extensive changes were made on top of Bootstrap entities to create a unique look to the user interface and several extra features regarding dynamic and IoT-based user interaction.

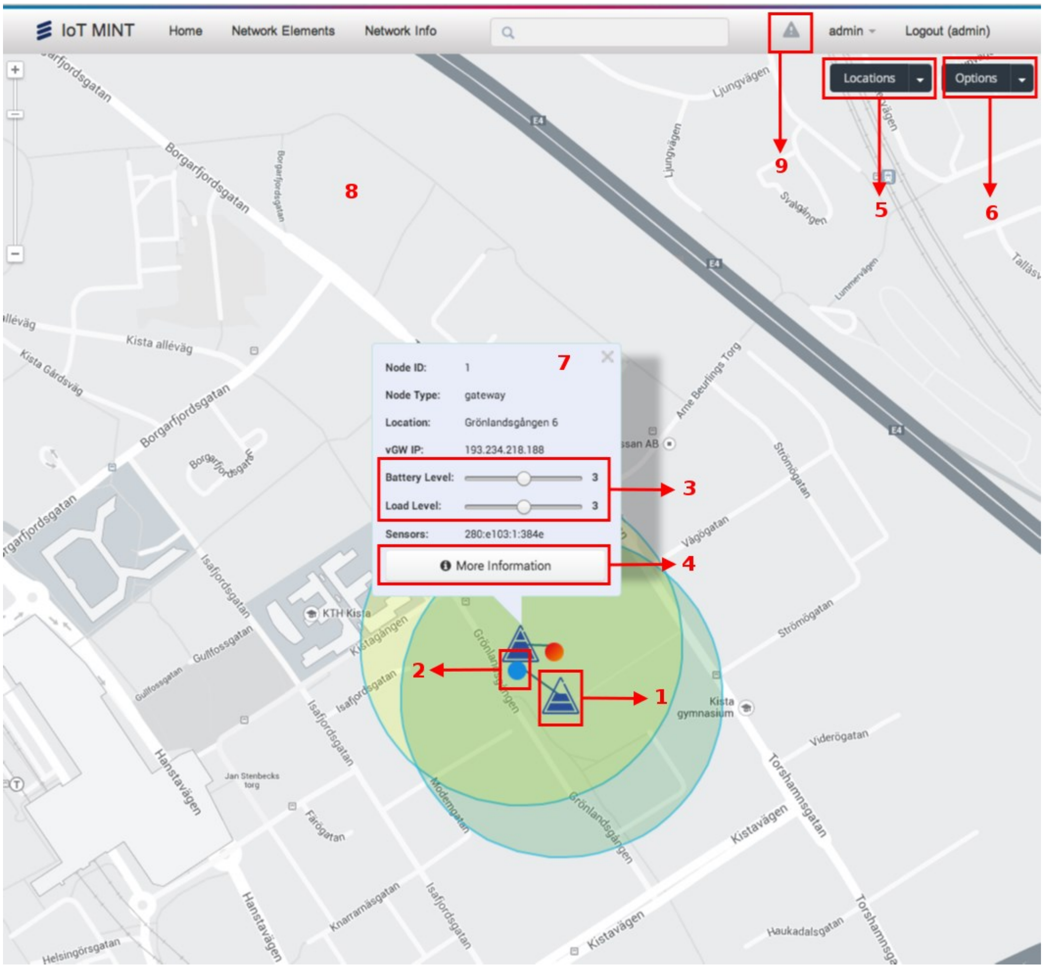
Extensive use of AJAX to retrieve information from the backend is one of the main results of aiming at an interactive user interface. Thus, great part of the data required in the frontend is retrieved from the backend by AJAX requests such as loading infobubbles, loading any floating window and its contents or updating congestion demonstration parameters as more is depicted in Figure 4-1. We aimed at increasing user experience by using AJAX requests.

In addition to AJAX, specific data sets, which are network topology (CN device list), notifications, logs and network counters, are retrieved from the backend via websockets [49]. To be more precise, the backend pushes these data sets to the frontend when the value changes. Updating battery/load constraints from the interface also uses websockets to send requests to the backend. More information about websocket implementation is presented in 5.2.2 Pub/Sub Server section in the backend implementation.

The home page of the user interface was designed to consist of an online map (see Section 4.1 and Figure 5-1 item 8). We decided to use Google Maps API [50] to build the interface upon, with the following reasons:

- Google Maps API is relatively simple to use.
- There are many available, free libraries built upon Google Maps API. They provide vast amount of functionalities.
- The service is responsive and fast in almost all browsers.

Google Maps API allows the developer to load custom icons on the map on desired locations. This feature of the API was used to locate CGWs and MDs on the map. Moreover, a circle around CGW showing its range and the line between MD and CGW showing their connection were drawn using another feature of the API. As the



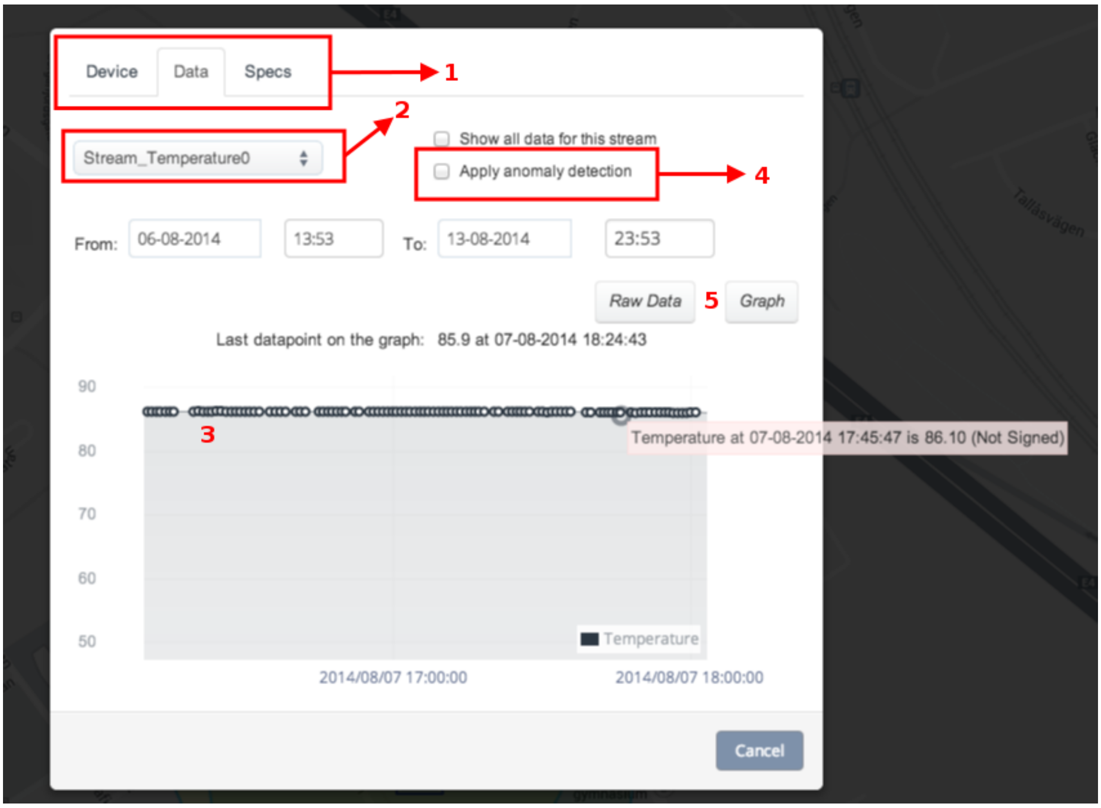
Node ID: 1
Node Type: gateway
Location: Grönlandsgången 6
vGW IP: 193.234.218.188
Battery Level: 3
Load Level: 3
Sensors: 280:e103:1:384e

Figure 5-1 Home Page of User Interface

interface is meant for large number of IoT devices on the map, a jQuery library called markerclusterer [51] was used to cluster the icons on the map as the user zooms out. Clustering the icons enhances the user interaction by reducing the number of visible icons. To provide the short information bubble on the icons (see Figure 5-1 item 1 and 2) when the user clicks on one, a separate open source jQuery library, infobubble [52] was used. The infobubble (see Figure 5-1 item 7) contained a summary of the device information with a “More Information” button in the bottom (see

Figure 5-1 item 4) and also for specific CGWs (determined by the backend), it contained “Battery” and “Load” sliders (see Figure 5-1 item 3). Using these sliders, the user is able to simulate the battery and load updates on CGW to trigger automatic gateway selection feature. The sliders are only meant for demonstration purposes changing battery and load levels of actual CGWs are not possible in the current testbed. The home page of the interface is shown in Figure 5-1.

When the user clicks “More Information” button on the infobubble, a floating window with different tabs appears on the map while darkening the background to highlight the new window (see Figure 5-2). The first tab in the window presents the detailed information about the device in a list form while the second tab presents data readings in case of an MD (see Figure 5-2 item 1). To present the datapoints, a jQuery plot library “Flot” [53] was used. Flot provides interactive, lightweight plots for large amount of data, which is required for IoT applications. An example plot drawn for a datapoint set is shown in Figure 5-2 item 3. In case the resource entry of MD in the IoT Framework has more streams attached to it, those streams are represented as in Figure 5-2 item 2. When the end user chooses to apply anomaly detection algorithm on the chosen datapoint set via ticking the choice (see Figure 5-2 item 4), detected anomalies are shown in red on Flot graph. The remaining area in Figure 5-2 item 1 can be used for logs from CGW, network counters from CGW and technical specifications of the device.


Stream_Temperature0

□ Show all data for this stream

□ Apply anomaly detection

From: 06-08-2014 13:53 To: 13-08-2014 23:53

Last datapoint on the graph: 85.9 at 07-08-2014 18:24:43

Temperature at 07-08-2014 17:45:47 is 86.10 (Not Signed)

Figure 5-2 Representation of Machine Device Data

For congested network demonstration, an option was added under “Options” drop-down menu (see Figure 5-1 item 6) to open a floating window when selected. This floating window includes several parameters for the congestion demonstration such as how many GWs to emulate, the location of GWs, which RAT the GWs use etc.

Another option was added under “Options” drop-down menu (see Figure 5-1 item 6), which is “Capillary Networks” to open a new floating window showing the gateway selection policies. This window presents the policies in JSON format. The policies on this window are editable using vkbeautify [54] JavaScript library.

LWM2M server option is also added under “Options” menu (see Figure 5-1 item 6). This option opens a new floating window, which is basically a terminal window to connect to LWM2M server. The user is allowed to send LWM2M commands to the server on this window to retrieve information about LWM2M resources.

In “Network Elements” tab on the navigation panel, CGWs and MDs are presented as two separate lists in two different tabs following the general look of the home page. The information showed on the table is identical to the information shown on the infobubble of an icon on the map.

In “Network Info” tab, notifications and logs from CGWs are listed in two different HTML tables following each other. This way, the difference between notifications and logs is highlighted. Placing notifications at top part states the importance of notifications on the interface.

As an important asset for real-time IoT network monitoring, the notification area on the navigation panel was implemented to stand out when a new notification is received i.e. the number of unread notifications is highlighted on the navigation panel and a small window opens under the navigation panel when the user wants to access the latest notifications (see Figure 5-1 item 9). This assures the important notifications to be visible to the user as soon as possible. To implement notification support, websockets were used which is discussed in the next section.

To realize the features discussed above, several JavaScript libraries were implemented such as device filtering and parsing, multiple selection of devices on the map, custom animations for markers, custom table presentation, floating window design and implementation, custom graph implementation etc. in addition to using some external JavaScript libraries. Figure 5-3 illustrates the implemented libraries for the frontend and the external libraries used. Some external libraries such as bootstrap or jQuery were used extensively throughout the entire interface. Therefore, such libraries are grayed out in the figure.

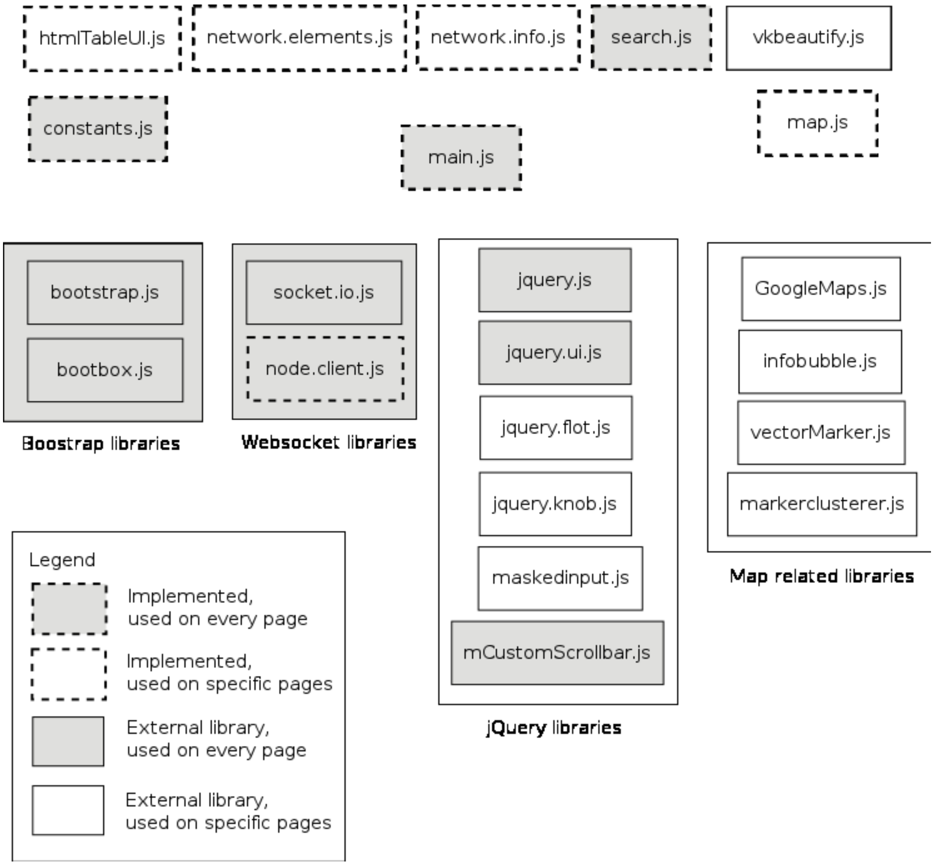


Figure 5-3 Frontend JavaScript Architecture

5.2 Backend Implementation

In this section, we introduce the backend implementation details of the management interface, which was built to present the data in the frontend as explained in previous section 5.1.

This section discusses all the backend components presented in 4.2 separately, explaining the relations between each of them and at the end, providing REST APIs used to integrate the backend components to the CN cloud entities.

5.2.1 PHP Application

The base of the backend implementation is built on a PHP [55] application. There are several reasons for PHP preference in the backend:

- PHP is an easy and fast scripting language.
- It is a mature language with few bugs and many available extensions.
- The documentation about PHP is quite vast.

The PHP application is designed to provide the main logic of the backend in which it coordinates the communication between CN entities (CNF, CNM, IoT Framework, vGWs), pub/sub servers and LWM2M server as well as providing the user interface with required data.

To implement the PHP application, a PHP framework was needed to build the application upon. The Yii Framework [56] was chosen for this purpose since it is a more versatile framework than the other alternatives (e.g. Zend, Fusebox, CakePHP etc.) and provides useful features.

Yii Framework is designed on Model-View-Controller (MVC) scheme, a commonly known architectural design pattern for creating web user interfaces [57]. MVC design patterns have the following features:

- Models are architectural components for representing the data structure.
- Views are the elements to create the user interface using the data from views, generally written in HTML or simple PHP.
- Controllers provide the logic between models, views and end user actions.

MVC pattern's main purpose is to separate logic from data structure and user interface, giving options such as code reusability and ease of code maintenance. As an example, the view can be updated freely without updating the controller or the model. In our case, the management interface is required to gather a vast amount of data it needs from third party components i.e. from CN entities located somewhere else in the network. Data is only available on external servers and needs to be fetched using HTTP REST APIs each time it is needed e.g. each time an end user makes a request. In our implementation, this data structure of the management interface resulted in controllers fetching the data by HTTP requests from CN entities. The retrieved data is forwarded to the views in each request. Because of this, the importance of the models in the backend was not high i.e. the number of model classes implemented was lower than expected. Although, there are still some models implemented for specific data resources (see Appendix A). Hence, controllers in the management interface are the most important elements in our PHP application to manage the communication and data flow between end user and CN entities.

Workflow of PHP Application

Figure 5-4 presents the workflow of PHP application and Yii Framework in a general view. The steps in this workflow are performed in the following order:

1. The user makes a request with a URL e.g. an AJAX GET request to retrieve detailed information of a CGW, event triggered by clicking "More Information" button on infobubble of CGW on the map. URL of such a request is "*http://www.iotmanagementinterface.com/site/detailedinfo? deviceId=5&deviceType=gateway&deviceVGWIP=111.111.111.111&resourceId=12345*"
2. Yii Framework component "Yii Application" validates that the URL is valid by using another framework component *urlManager*. Then, Yii application determines the controller and action for the given request i.e. each URL corresponds to a specific controller and an action (function) inside that controller. For the given example above, Yii application forwards the request and

given parameters to *DetailedInfo* action in *SiteController* class.

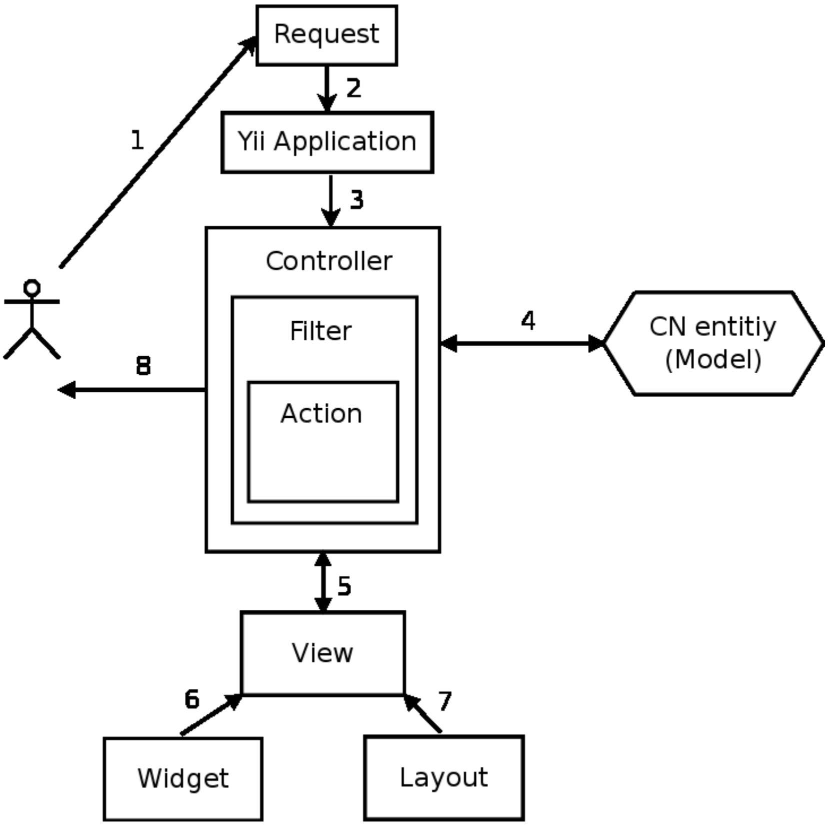


Figure 5-4 Workflow of PHP Application

3. *SiteController* class determines that *DetailedInfo* refers to *actionDetailedInfo* function in the class i.e. URL path */path/function* always refers to *actionSomeURL* in *PathController* in Yii Framework. The controller class first applies the filters e.g. access control filters if there is any. The function is performed only if the filters allow the function to be executed.
4. The action retrieves and harvests the detailed information of the requested CGW from relevant CN entities by making HTTP requests e.g. meta data stored in vGW and IoT framework.
5. The action formats the data retrieved from CN entities and renders the corresponding view e.g. *DetailedInfo* view for the given example.
6. The view can apply widgets on the data provided. Although, it is not a requirement by the framework.
7. The rendered view is encapsulated in the main page layout i.e. the response to the end user consists of an HTML page with the entire HTML headers. But in case of an AJAX request (the one in the example), encapsulation in the main page layout is not performed. Instead, the rendered view is returned directly

for AJAX requests such as some short HTML document without the headers.

8. The action returns the encapsulated view (not encapsulated in AJAX responses) to the user and completes the workflow of the request.

Controllers

To divide the workflow into logical components, implementation of controllers i.e. the separation of controller classes followed the frontend design depicted in Figure 4-1. Hence, five main controller classes were implemented:

- *SiteController*: It is responsible of every action performed on the home page (map) such as loading CGWs and MDs on the map, updating slider values, updating CN policies, showing detailed information of a device and many more. As mentioned in the workflow above, each of these functionalities correspond to a different *action* in *SiteController*.
- *ElementsController*: This controller is responsible of "Network Elements" page of the interface. The class retrieves the list of network elements from CNF and presents it to the user.
- *NetworkController*: This controller gathers the logs and notifications from CNM and CNF and presents it to the user. The user is capable of setting log levels or filtering notifications too.

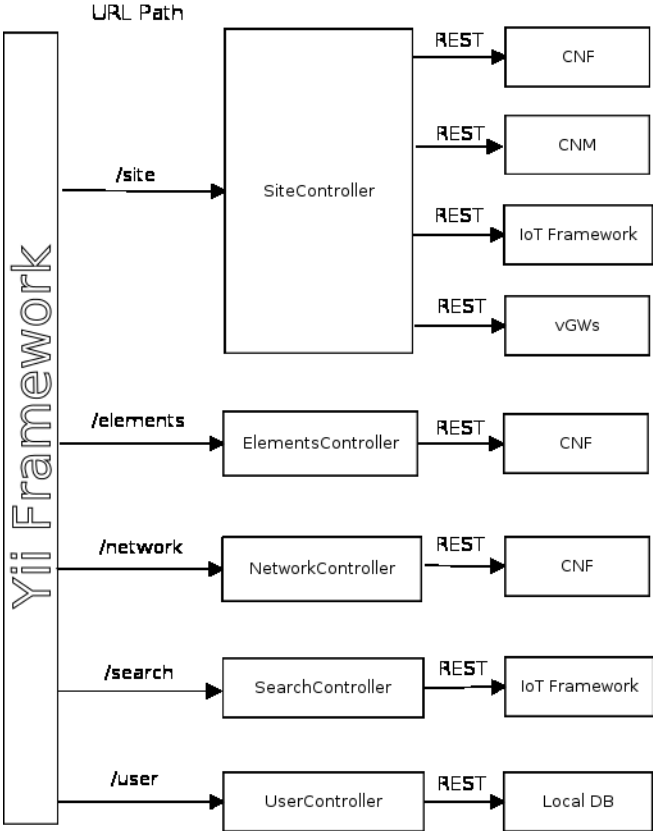


Figure 5-5 Controllers and Capillary Network Components

- *SearchController*: This controller searches IoT Framework with the given search query and returns the search results to the user on a separate page
- *UserController*: This controller basically controls the access to the user interface based on username/password credentials.

A full list of *actions* and helper functions for each controller is presented as UML diagrams in Appendix A. Moreover, almost each *action* in each controller communicates with at least one CN entity while executing its function due to the design and requirements of the backend. Hence, the actions need to use several HTTP REST APIs to perform the corresponding function. Several interfaces were implemented for the controllers to communicate with CN entities. These interfaces are discussed in “REST APIs” section.

The controllers listed above are not responsible to assure the data consistency in different CN entities. As in Figure 5-5, several components can access the CNF at the same time from different user accounts. However, it is the responsibility of the CNF to assure the data consistency since the management interface acts like a proxy between the network manager and the Capillary Networks cloud entities. Moreover, changes in the CNF are published to all the network managers’ frontends by Pub/Sub Server, which is discussed in Section 5.2.2.

Device List Data Format

To provide the list of capillary network devices, the PHP application needs to connect to CNF, retrieve the list of CGWs and MDs, parse the data, retrieve additional data from CNF for battery/load constraints of specific CGWs and create the formatted list (see Figure 5-8). Since the PHP application parses the device list data, this list is formatted so that it only contains the minimum required parameters for showing the devices on the map, which are:

- Capillary gateway IDs
- Capillary gateway GPS coordinates
- Under each capillary gateway entry, IDs of machines devices connected to that capillary gateway
- GPS coordinates of machine devices

Minimizing the data content of device list assures faster user experience in the frontend since JavaScript libraries perform better with shorter lists. An example device list is presented below:

```
[{ "node":"1", "devices":"",
"vGW":"193.234.218.188,2a00:1d50:2:1001:f816:3eff:fe94:4a3d",
"online":"True", "level":"5", "params":{"1": {"lat": 59.40442269, "lng":
17.95359135}} },

{ "node":"2", "devices":"280:e103:1:2a9c",
"vGW":"193.234.217.173,2001:14b8:400:131:f816:3eff:fe4f:de50",
"online":"True", "level":"0", "params":{"2": {"lat": 41.35342516, "lng":
2.13135839}, "280:e103:1:2a9c": {"lat": 41.35361845, "lng": 2.13043571}}]
```

Anomaly Detection

The PHP application retrieves MD datapoints from the IoT Framework via *SiteController* as depicted in Figure 5-5. Normally, the frontend presents datapoints as a data graph. But if the user wants to perform anomaly detection (see Section 2.4.4) on a given datapoint set, the set needs to be executed with the anomaly detection program. Since the mentioned program is a Matlab executable, the following temporary solution was found:

- Write datapoint set to a temporary file (e.g. json.txt) on the server (step 1 in Figure 5-6).
- Read from json.txt (step 2 in Figure 5-6) and run a python script to convert datapoint set from JSON to Matlab format. Output is written to another temporary file (e.g. matlab.txt) as step 3 in Figure 5-6.
- Run anomaly detection program with the formatted datapoint set (step 4 in Figure 5-6), output is written to another temporary file (e.g. output.txt) as step5 in Figure 5-6.
- The PHP application reads the new file, re-formats the data and creates a combined datapoint set with anomalous and normal data points (step 6 in Figure 5-6).

For this process, a new PHP component *AnomalyDetection* was implemented in PHP server. The workflow of this component explained above is presented in Figure 5-6.

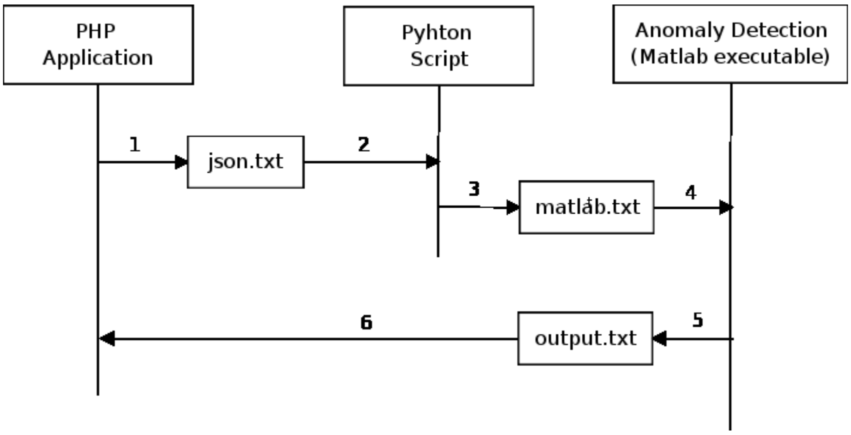


Figure 5-6 Workflow of Anomaly Detection Program

As stated above, this workflow is a temporary solution until the anomaly detection algorithm is imported from Matlab since it is not efficient in terms of overhead, latency and scalability.

5.2.2 Pub/Sub Server

As explained in the previous section, the scope of PHP application is to provide the user interface with data retrieved from several CN entities. However, one of the requirements for the backend implementation is to inform the user immediately of IoT network topology changes or notification/log updates received from CNF (see also Section 2.4.1). One possible option to achieve real-time updates on the user interface is to perform polling to the PHP application with short intervals. However, the polling approach is not user-friendly since each polling request can take long time and disrupt user's actions. Another option for real-time updates is to use websockets [49] and to initiate the communication to the end user from the server side. However, PHP works on a request-response based client-server scheme i.e. the communication needs to be initiated by the client. This fact indicates that PHP is not capable of performing real time interface updates initiated by the server. In other words, PHP cannot push any data to the end user. For this reason, another backend component was needed to perform server-initiated communication to the end user for real-time updates. This backend component is required to support clients subscribed to the backend and updates published to them when available. Hence, an additional pub/sub server was required in the backend implementation.

As our pub/sub server solution, we decided to use Node.js platform [58]. Node.js is an asynchronous event driven framework written in JavaScript and capable of handling very large number of connections in a scalable manner, which is an important asset for an IoT device management interface. A vast amount of Node clients can listen to a Node server for server-initiated communication i.e. for pushing data to the clients, a feature we used in the implementation.

Architecture of Node server requires Node clients to be implemented for the frontend so that the communication between Node server and clients can be initiated. Thus, an additional Node client was implemented in addition to the Node server. As depicted in Figure 5-7, a dedicated Node client registers to the Node server when each end user opens the management interface on his browser. The client subscribes to the server for the channels mentioned below, keeping the websocket open for any data pushed from the server. In addition to the Node client subscribing to the Node server, the Node server needs to subscribe to CN Pub/Sub server as well to receive data about the channels mentioned below. Hence, Node server acts like a proxy between CN Pub/Sub server and Node client, being both client and server in the same node and forwarding the pushed data from CN Pub/Sub server to Node client after formatting.

Implemented pub/sub server is required to support the following channels to be pushed to clients, which is received from CN Pub/Sub server:

- Network elements i.e. list of CGWs and MDs
- Notifications
- Logs
- Counters

As a design principle explained in backend requirements, the list of network elements will be loaded on the map only with the data received from pub/sub server i.e. there is no traditional HTTP request from the end user for list of network elements. The list is pushed to the frontend using only websockets. Pushing network elements to the client indicates that pub/sub server needs to retrieve the list of capillary devices. However, device list preparation was already implemented in PHP in earlier stages. As depicted in Figure 5-7 and Figure 5-8, the solution was found by using the Node server and the PHP server together. When the Node server receives *getDevices* command from the Node client (step 1.3 in Figure 5-7) over websocket in the initialization part (when the end user starts the management interface), the Node server sends a local HTTP AJAX request to the PHP application (step 1.4 in Figure 5-7). In our implementation, the Node server and the PHP application are installed on the same local machine, which results in a local HTTP request from the Node server to the PHP application. The PHP application retrieves network topology device list from CNF upon the request from the Node server. Moreover, the PHP application sends a separate request to CNF to retrieve battery/load constraint values to be shown as CN demonstration sliders on the frontend. PHP application parses these two responses from CNF, creates a combined list (see “Device List Data Format” sub-section in 5.2.1) and sends the response to the Node server (see Figure 5-8). The Node server publishes the data to the Node clients (step 2.1 in Figure 5-7). Afterwards, corresponding markers (device icons) are updated on the map by main.js JavaScript functions (steps 2.2 and 2.3 in Figure 5-7).

The initialization workflow of the interface is presented in Figure 5-7 as the green circle. Similar workflow is continued when there is an update received from CN Pub/Sub server by the Node server (steps 3.1, 4.1, 5.1, and 6.1 in Figure 5-7). This update can be any change in the network configuration or discovery of a new MD i.e. CN Pub/Sub server pushes any update from CN cloud to the Node server. In this case, the Node server pushes the data retrieved from PHP application directly to the Node client without waiting the Node client to send a request (steps 3.3, 4.2, 5.2, and 6.2 in Figure 5-7). In case of new notification, logs or counters from CN Pub/Sub server, the received data is published by the Node server and directly shown on the interface by the Node clients (steps 4.3, 5.3, and 6.3 in Figure 5-7).

Additionally, battery/load constraint updates are received from the frontend via websockets to Pub/Sub server. In this mode, the Node client pushes the updated constraint level to the Node server (step 7.2 in Figure 5-7). The Node server sends the received data to PHP application, to *actionUpdateCapillaryConstraint* function to further forward to CNF (step 7.3 in Figure 5-7). The Node server does not expect any response from *actionUpdateCapillaryConstraint* since CNF does not return any response to PHP application. In case network topology changes upon the constraint update, Pub/Sub server in CN cloud sends “network topology changed” notification to the Node server (step 3.1 in Figure 5-7). Therefore, the frontend is informed accordingly.

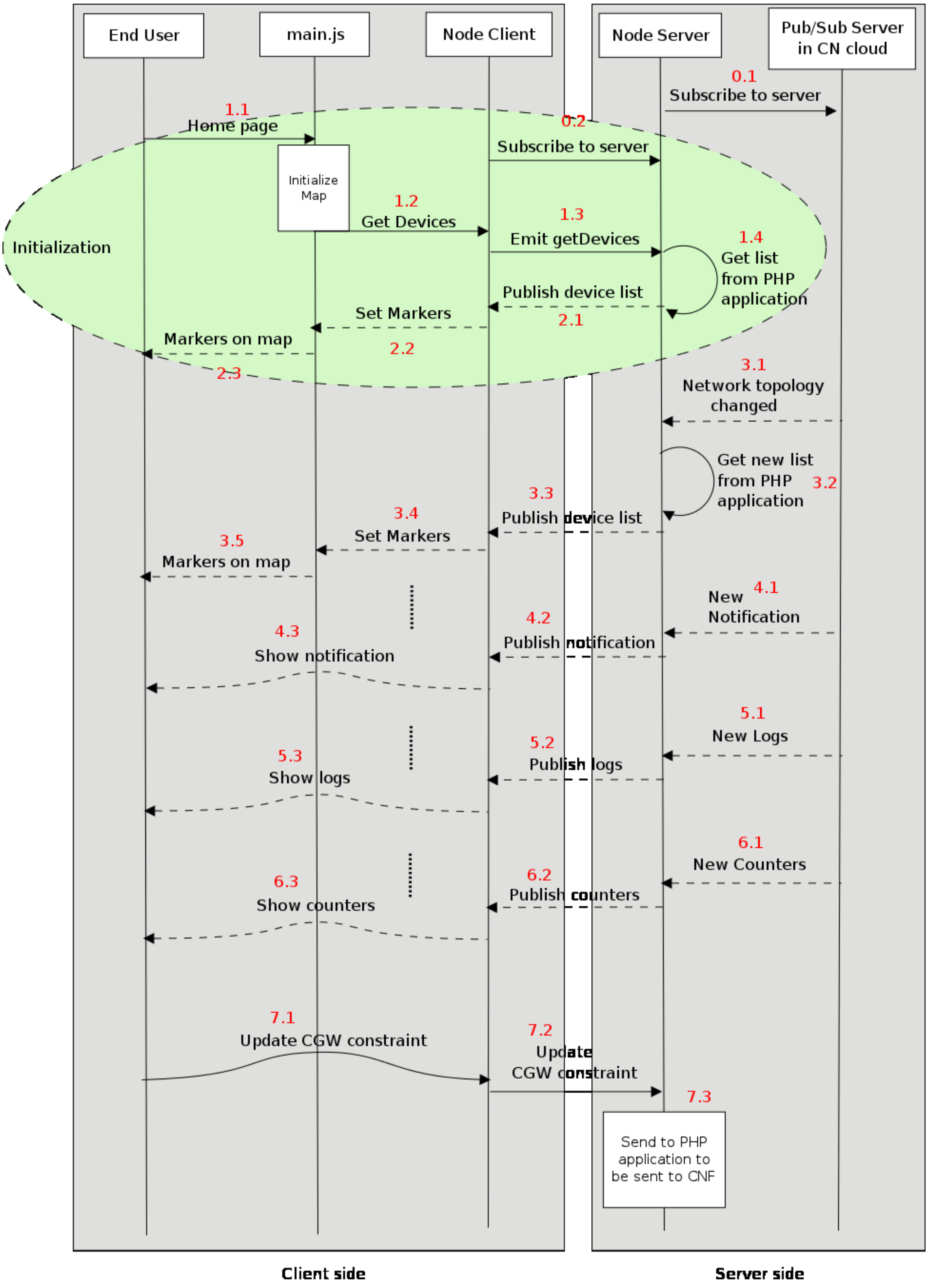


Figure 5-7 Sequence Diagram of Pub/Sub Server

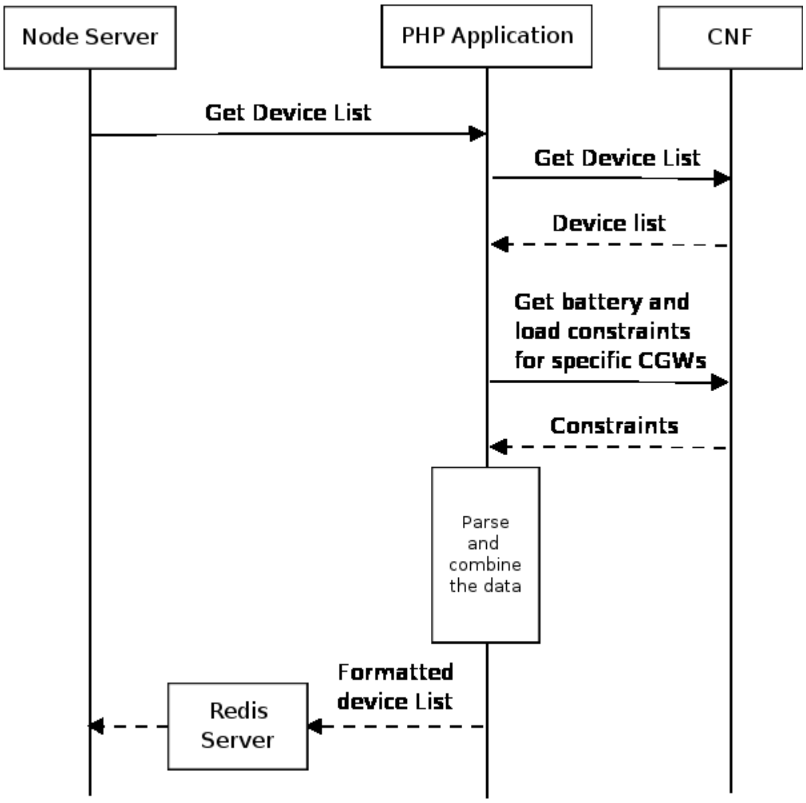


Figure 5-8 Sequence Diagram of Retrieving Device List from PHP Application

As presented in Figure 5-7 and explained earlier, Node server retrieves the list of IoT devices (network topology) from PHP application. Figure 5-8 represents this workflow as a sequence diagram. In the diagram, there is an additional component between PHP application's response and the Node server, which is called "Redis Server". The role of this component is explained in the next section.

5.2.3 Redis Server

Figure 5-8 indicates that there is a new component between PHP application and Node server, which intercepts the response from PHP application i.e. a Redis [59] server. Redis is an open-source, NoSQL, in-memory, queue-based, key-value store or data structure server (see Section 2.3.1). Being key value store does not make Redis a simple data store as keys can be sets, lists, bit arrays or basic strings. In-memory storage performs very fast compared to SQL databases to access the recent data [60]. The features to scale very large amount of data, to provide a very fast data store and to push/pop lists on the queue are the main reasons to choose Redis as our NoSQL solution.

The reason of having a Redis server lies in the communication necessity between PHP application and Node server. As discussed in earlier sections, the Node server sends a local HTTP AJAX request to the PHP application so that the PHP application can retrieve the data sets from CNF and sends back the formatted data to Node server. However, it was found out during implementation that the Node server could not receive the response from PHP application for this HTTP request. The reason behind this behavior is assumed to be the asynchronous nature of AJAX requests since the response is not ready yet when the Node server expects it. As a solution, an asynchronous data cache was needed to transfer the data prepared by PHP application to the Node server. We found out that in-memory Redis server is an excellent choice as a cache to transfer the data between two local servers.

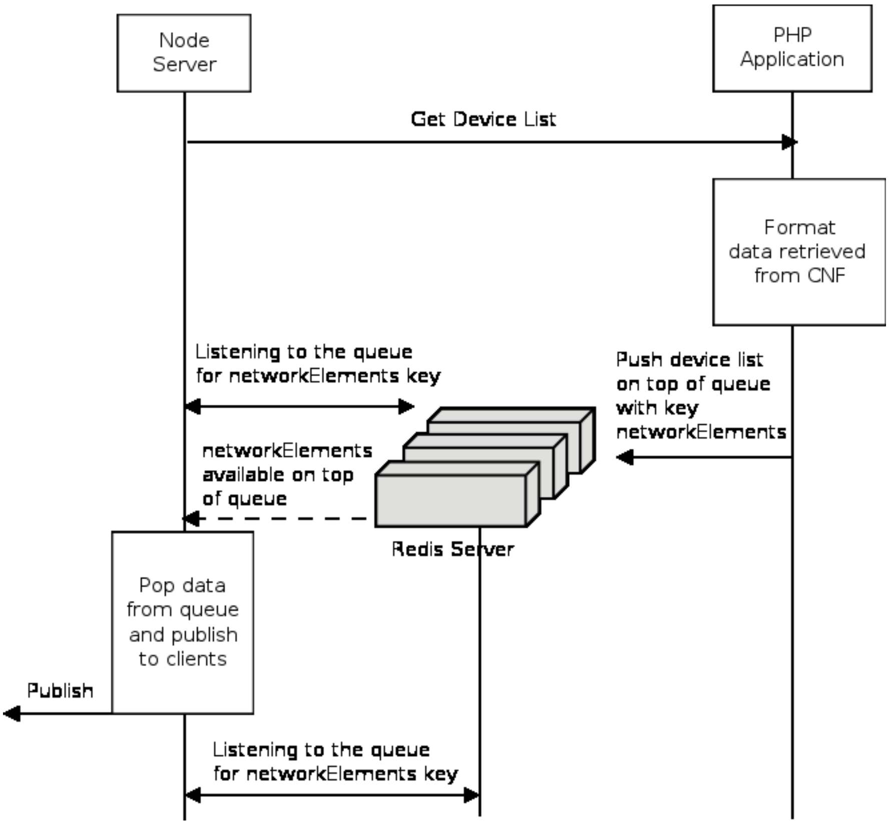

Node Server
PHP Application
Get Device List
Format data retrieved from CNF
Listening to the queue for networkElements key
Push device list on top of queue with key networkElements
networkElements available on top of queue
Redis Server
Pop data from queue and publish to clients
Publish
Listening to the queue for networkElements key

Figure 5-9 Workflow between Node Server, PHP application and Redis Server

In our implementation, PHP application prepares the IoT device list with the data sets retrieved from CNF. Then, PHP application pushes the prepared data to a Redis list with the key name *networkElements*. In Node server implementation, a function listens to this list on Redis and when a new data is available in the list queue with key *networkElements*, the function publishes the data to Node clients. Upon finishing the publishing, the function returns back to idle state to continue listening to the queue for the same list. The workflow is illustrated in Figure 5-9.

5.2.4 LWM2M Server Integration

The thesis is a combined project with another thesis about LWM2M implementation [61]. The implemented LWM2M server is combined with the management interface as an important addition to enable direct communication with LWM2M clients from the interface i.e. direct communication towards MDs.

Direct access to MDs e.g. sensors or actuators from the management interface is a novel solution using LWM2M and CoAP. With LWM2M server integration, the network manager can access LWM2M clients (i.e. actual MDs) directly from the management interface. The manager can send LWM2M commands from the interface to the LWM2M server, which each LWM2M client is connected to. The LWM2M server then forwards the response of the LWM2M client to the management interface. Some of the actions the manager can perform on MDs include reading LWM2M resources, updating LWM2M resources and firmware, retrieving errors directly from the MD or reading data values of the MD (see Section 2.2.5). This integration enables the manager to access each constrained device directly, which was not possible for constrained devices earlier. Hence, LWM2M server integration is meant to make the management interface following the latest technology in IoT device management.

The commands the network manager can send to the LWM2M server is retrieved from an interactive shell from the frontend of the management interface. The manager writes LWM2M commands on this shell (see Section 2.2.5 for LWM2M commands), which is then forwarded to LWM2M server by the PHP application. As an example, the following LWM2M command reads *resource ID /1024/10/1* from *client ID 0*. The response returns the status code, the URI and the content (see Section 2.2.5).

```
> read 0 /1024/10/1

{“status” : 205, “uri” : “/1024/10/1”, “content” : “20/10”}
```

As depicted in Figure 5-10, LWM2M server is installed on the same machine with PHP application as an implementation decision since the current prototype can be supported from a single LWM2M server. However, LWM2M server can be installed anywhere on the network in other implementations. LWM2M server only responds to CoAP messages i.e. it is not capable of understanding HTTP or any other protocol messages. For this reason, the communication between PHP application and LWM2M server is based on socket communication.

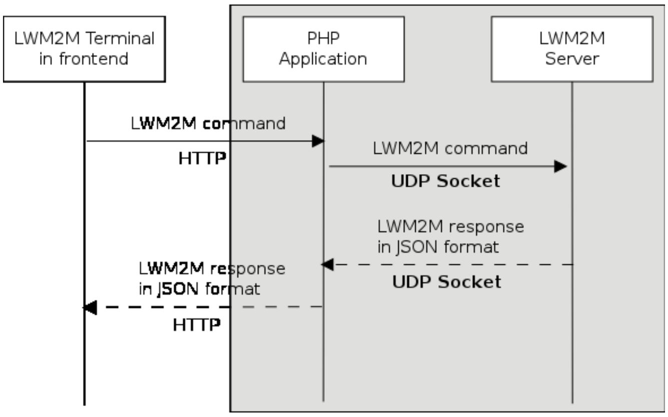
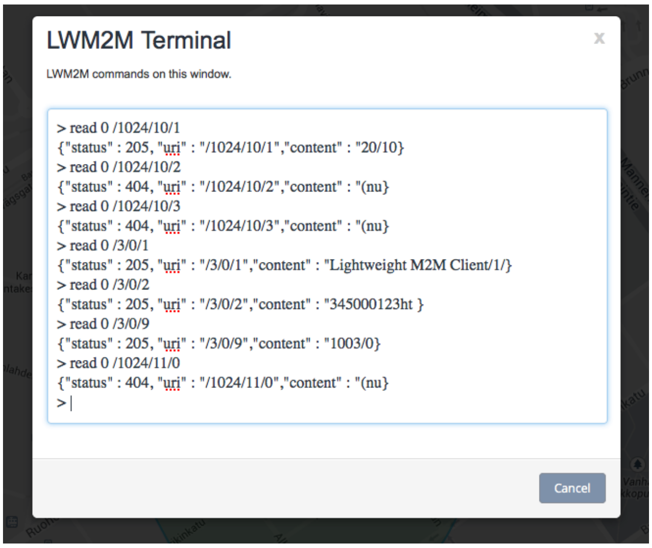

LWM2M Terminal in frontend
PHP Application
LWM2M Server
LWM2M command
HTTP
LWM2M command
UDP Socket
LWM2M response in JSON format
UDP Socket
LWM2M response in JSON format
HTTP

Figure 5-10 LWM2M and PHP application integration

When PHP application receives POST request from LWM2M terminal window with LWM2M command in the request body, the application sends this command to LWM2M server over a UDP socket. In the LWM2M server, the response to such requests was implemented to be in JSON format to be compatible with management interface. Hence, PHP application forwards the response format from LWM2M server directly to the end user in case there is no error from LWM2M server side. In practice, the management interface is a proxy between the end user and LWM2M server in charge of providing conversion from HTTP to UDP socket communication.

As an example, a screenshot from the interactive shell on the interface with several LWM2M commands is presented in Figure 5-11.



```
> read 0 /1024/10/1
{"status" : 205, "uri" : "/1024/10/1","content" : "20/10}
> read 0 /1024/10/2
{"status" : 404, "uri" : "/1024/10/2","content" : "(nu}
> read 0 /1024/10/3
{"status" : 404, "uri" : "/1024/10/3","content" : "(nu}
> read 0 /3/0/1
{"status" : 205, "uri" : "/3/0/1","content" : "Lightweight M2M Client/1/}
> read 0 /3/0/2
{"status" : 205, "uri" : "/3/0/2","content" : "345000123ht }
> read 0 /3/0/9
{"status" : 205, "uri" : "/3/0/9","content" : "1003/0}
> read 0 /1024/11/0
{"status" : 404, "uri" : "/1024/11/0","content" : "(nu}
> |
```

Figure 5-11 LWM2M example commands

5.2.5 REST APIs

In the previous sections, we discussed the extensive communication between PHP application and CN cloud entities. To establish this communication, we decided to implement REST APIs for requests to CN entities and to regulate the client's response from CN entities in JSON format. This way, the management interface i.e. the PHP application and CN entities can be located separately anywhere on the network and the communication between these two entities is maintained accordingly with REST APIs. Response format in JSON is an optimum solution for parsing the data efficiently in the PHP application (see Section 2.3.2).

Deciding which REST APIs to implement was based on backend and frontend requirements as well as constraints exposed by CN entities. Some APIs were already provided by e.g. CNF and IoT Framework, though most of the APIs required changes to comply with management interface requirements.

All updated/created APIs are presented below in separate tables for CNF, CNM, IoT Framework and vGW. However, the tables include only short explanations about each of them. The entire detailed REST API documentation is presented in Appendix B.

Table 5.1 REST APIs towards CNF

Management Interface Components using this API	Feature
PHP*: SiteController*	Get the full list of available CGWs and MDs
PHP*: SiteController*	Get the list of all available CGWs
PHP*: SiteController*	Get the list of all connected MDs for a CGW
PHP*: SiteController*	Get the vGW IP address for a CGW
PHP*: SiteController*	Get the online status for a CGW
PHP*: SiteController* PHP*: NetworkController*	Get the latest logs of a CGW
PHP*: NetworkController*	Get the most recent logs of all the CGWs
PHP*: NetworkController*	Get the logging level for a CGW
PHP*: NetworkController*	Set the logging level for a CGW
PHP*: NetworkController*	Get the latest notifications of a CGW

PHP: *NetworkController*	Get the most recent notifications for all the CGWs
PHP: *SiteController*	Get the latest network counters of a CGW
PHP: *SiteController*	Get the Capillary Networks policy
PHP: *SiteController*	Set the Capillary Networks policy
PHP: *SiteController*	Get the battery/load constraint value for a CGW
PHP: *SiteController*	Set the battery/load constraint value for a CGW

Table 5.2 REST APIs towards CNF used for demonstration

Management Interface Components using this API	Feature
PHP: *SiteController*	Get the list of gateways for congestion demonstration
PHP: *SiteController*	Emulate gateways for congestion demonstration
PHP: *SiteController*	Optimize the emulated gateways
PHP: *SiteController*	Clear the emulated gateways

Table 5.3 REST APIs towards CNM

Management Interface Components using this API	Feature
Demo use case: *LocationCollector*	Set GPS coordinates for a CGW or MD
Demo use case: *LocationCollector*	Set a configuration parameter
Demo use case: *LocationCollector*	Get the list of CNM status values

Table 5.4 REST APIs towards IoT Framework

Management Interface Components using this API	Feature
PHP*: SiteController* PHP*: SearchController*	Get the list of resources for a user
PHP*: SiteController* PHP*: SearchController*	Get a single resource entry with the given ID
PHP*: SiteController*	Get the list of streams for a user and a resource
PHP*: SiteController*	Get a single stream entry with the given ID
PHP*: SiteController* PHP*: SearchController*	Get the list of datapoints for a stream
PHP*: SiteController* PHP*: SearchController*	Get the filtered list of datapoints for a stream
PHP*: SearchController*	Search the IoT Framework with a query

Table 5.5 REST APIs towards vGW

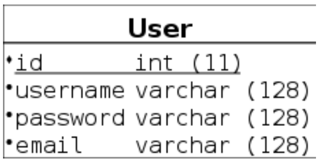
Management Interface Components using this API	Feature
PHP*: SiteController* PHP*: ElementsController*	Get a list of CGWs under the vGW
PHP*: SiteController* PHP*: ElementsController*	Get a list of MDs under the vGW

Table 5.6 APIs towards Pub/Sub server in Capillary Networks cloud

Management Interface Components using this API	Feature
Node server	Subscribe for network topology update
Node server	Subscribe for notifications
Node server	Subscribe for logs
Node server	Subscribe for network counters

5.2.6 Database

Stated several times earlier, the management interface retrieves all the data from external CN cloud entities via REST interfaces. Hence, the database in the management interface backend does not store any network or device related data. It is used to store only the username database of the management interface. Therefore, the database required is very simple, consists of only one table *user* as depicted in Figure 5-12 and was created in MySQL [62]. Yii Framework also automatically generates a separate database to store framework related variables. However, this database is out of discussion for this thesis.



User

•id	int (11)
•username	varchar (128)
•password	varchar (128)
•email	varchar (128)

Figure 5-12 Database table

The table consists of only four columns with id (primary key), username, password and email, all of the values are stored in plain text. A new row in the table is created manually by the management interface admin when a new user requests username. To control user access to the management interface, *UserController* class in PHP application uses the rows username and password in this table to grant access to a user in the login page.

5.3 Demo Use Case

In this section, we present an industry use case for demonstration, which targets at presenting CN concept, architecture and working principles along with the management interface in reality. Although the implemented use case demonstration is a permanent setup in Ericsson premises in Jorvas at the time of the thesis writing, the setup has been demonstrated in several internal Ericsson meetings and external events too.

The use case aims at demonstrating several features of CN prototype in the demo setup:

- Automatic configuration and discovery of MDs
- Self-load balancing of CGWs
- Gateway selection for MDs
- Middleware & service creation
- Storage, representation and filtering of data

To create a realistic environment, the focus of the use case is based on an industrial harbor with several cranes. Use case defines a train track stretching to harbor to load the cargo ships with the containers carried by the train. The harbor area is covered by CGWs to allow sensors on the containers to connect to the network i.e. MDs are located in the containers and connected to CN backend via CGWs in the harbor area. The network manager monitors the devices on the harbor area on the management interface, is notified of any possible errors via notifications on the interface and can perform load/battery constraint changes on CGWs in the area.

Putting the use case into practice, we decided to build up the environment using remote-controlled Lego® trains and other miniature decoration elements. The environment we set up is presented in Figure 5-13. In this setup, two virtual CGWs are assumed to be available on two cranes in the harbor, items 1 and 2 in Figure 5-13. In fact, actual CGWs are located behind the setup. However, actual MDs are placed inside the train containers (items 3 and 4 Figure 5-13), measuring the temperature in the setup room. While the train (item 3 in Figure 5-13) is moved along the track, containers are placed on the carriages moving with the train or containers can be left in the harbor area as well. Blue line from A to B indicates the range of both CGWs.

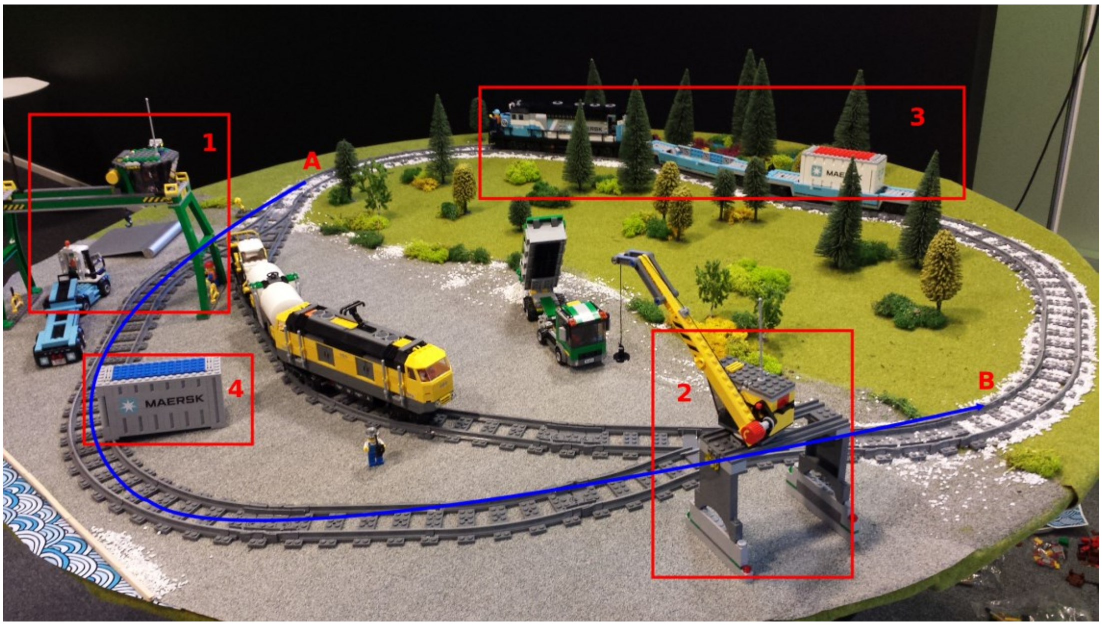


Figure 5-13 Demo Environment

The table on which the setup is set has only a diameter of around 1.5 meters. Therefore, the movement of MDs with the train is not possible to be detected by CGWs, since the radio link quality does not differ much in that range. To solve this problem i.e. detecting the location of MDs on the setup, we decided to track the containers with a camera and to apply an image recognition program on a computer available in the demo setup. Image recognition program tracks a given color in the camera video while filtering out all the other colors. Hence, we placed two distinct colors on the containers, red and blue which none of our decoration elements have. Placing the camera over the setup to cover the entire train track allowed us to track red and blue containers (item 3 and 4 in Figure 5-13). In other words, camera and image recognition program combination works as a satellite for the demonstration.

ColourTracker image recognition program was found online as an open source Java project in [63]. The program interface allows to change saturation, hue and intensity levels of the image so that a distinct color tone is determined and tracked. It detects the largest area with the given color values on the image. However, the program returns the detected color's coordinates in the image frame e.g. coordinates in a frame of 320x240 pixels image. These coordinates have to be translated into GPS coordinates so that GPS coordinate updates of MDs are sent to CNM periodically via *cmd=set_coordinates* command. Therefore, a new Java program *LocationCollector* was implemented using *ColourTracker* as the main reference. Within *LocationCollector*, GPS coordinates of MDs are calculated using the output of image recognition program, calculated bearing of the tracked color pattern and distance of the tracked color pattern to the center of the image, by using the Haversine formula [64] as depicted below. Bearing of the tracked color pattern refers to the angular coordinate of the pattern on the image.

$$bearing = (2 * pi) - \arctan(\frac{y - CAMERA\_CENTER\_Y\_IN\_PIXELS}{x - CAMERA\_CENTER\_X\_IN\_PIXELS})$$

$$\begin{aligned}
&distanceToCameraCenter \\
&\qquad = sqrt((x - CAMERA\_CENTER\_X\_IN\_PIXELS)^2 \\
&\qquad + (y - CAMERA\_CENTER\_Y\_IN\_PIXELS)^2) * DISTANCE\_CONSTANT
\end{aligned}$$

$$\begin{aligned}
latitude = \arcsin\Bigg(& sin(CAMERA\_CENTER\_LAT\_RADIANS) * cos\left(\frac{distanceToCameraCenter}{EARTH\_RADIUS}\right) \\
& + cos(CAMERA\_CENTER\_LAT\_RADIANS) \\
& * sin\left(\frac{distanceToCameraCenter}{EARTH\_RADIUS}\right) * \cos(bearing)\Bigg)
\end{aligned}$$

$$\begin{aligned}
&longitude \\
&= CAMERA\_CENTER\_LON\_RADIANS \\
&+ \arctan(\frac{\left(bearing * \sin\left(\frac{distanceToCameraCenter}{EARTH\_RADIUS}\right) * cos(CAMERA\_CENTER\_LAT\_RADIANS)\right)}{\cos\left(\frac{distanceToCameraCenter}{EARTH\_RADIUS}\right) - \sin(CAMERA\_CENTER\_LAT\_RADIANS) * \sin(latitude)})
\end{aligned}$$

CNM receives GPS coordinate updates of MDs from *LocationCollector* and determines the connectivity of each MD to the appropriate CGW. Therefore, network topology is centrally controlled from CNM. MDs moving with remote controlled train are depicted on the management interface with the same movement i.e. location changes of MDs are illustrated on the management interface map. The demo setup allows to demonstrate CN features in the following ways:

- When the container on the train gets out of range of both CGWs i.e. outside the blue line from A to B in Figure 5-13, that specific MD is not shown on the management interface. Similarly, containers (MDs) reaching the range of a CGW are connected to that CGW and become visible on the interface i.e. while the containers are somewhere on the line between A and B.
- Turned-off MDs can be placed in the range of a CGW and once the MD is turned, CGW automatically discovers and configures that MD. Hence, MD becomes visible on the map somewhere on the blue line.
- When an MD resides in the intersection of both CGWs, the end user can update load constraints of one CGW. CNM re-connects the MD to the CGW with lower load level automatically and this transition is shown on the interface. This case is applicable for item 4 in Figure 5-13 as it resides in the intersection of both CGWs.
- Similar to above, CNM connects the MD residing in the intersection of both CGWs to the most suitable CGW depending on CN policies in use.
- The demonstrator can present the middleware entities (vGWs) created for each CGW from the management interface.
- Data readings of MDs can be shown as a plot or raw data from the detailed information window of MD, which can be reached via clicking MD icon on the map. Since the sensors actually measure the room temperature, real-time room temperature is shown on the interface.

This demonstration shows the entire Capillary Networks idea from the constrained device (MD) up to the management interface and cloud entities in an active and interesting way. Hence, all the components of the Capillary Networks are presented concurrently. Moreover, the interactive nature of the demonstration setup presents the concept in action and allows even the non-technical people to follow the demonstration with ease.

5.4 Summary

The frontend implementation composes of use of Bootstrap framework with focus on the map view on the home page. Apart from HTML and CSS, JavaScript is used extensively to create a dynamic frontend.

The backend implementation is based on a PHP application which communicates with Capillary Networks entities via REST APIs to retrieve required management information. Moreover, a Node.js publish/subscribe application and a Redis server support the PHP application in terms of dynamic, real-time management commands in addition to a LWM2M shell integration for LWM2M commands.

6 Measurements and Evaluation

After explaining the implementation details of the management interface in the previous chapter, we evaluate the interface itself and discuss the performed experiments in this section. The purpose of the measurements performed on the experiments is to find the limits of the interface, considering that it was designed to handle a very large amount of IoT devices. Hence, obtained measurements present a rough view over the system performance.

The chapter begins with the evaluation of the requirements discussed in Chapter 3 and how the implemented solution complies with the requirements. Section 6.2 presents the test results and the analysis of the website (the interface itself) performance measurements in several use cases. Finally, performance of anomaly detection on constrained device data readings is discussed in Section 6.3.

6.1 Requirements Evaluation

In this chapter, we evaluate how the final interface satisfies the requirements discussed in Chapter 3. Requirements were classified into different importance levels and hence, the evaluation is based on the importance level of each requirement. This evaluation only focuses on whether the requirement is implemented and if so, with what kind of features i.e. performance analysis of the implementation is not included in this section. Following sections discuss different requirements for both frontend and backend.

6.1.1 Frontend Requirements Evaluation

Each requirement presented in Table 3.1 in Section 3.1 is evaluated separately in Table 6.1 below.

Table 6.1 Frontend Requirements Evaluation

Feature	Requirement Evaluation	Importance	Satisfied?
Automatic configuration	The network topology changes (including new device appearing) are automatically pushed to the frontend via Websocket and updated on the map real-time without any distraction to the user i.e. no freezing of the interface at any time and no extra steps needed from the user for new network configuration retrieval.	Essential	Yes

Dynamic Middleware	This is achieved by showing the dynamic middleware addresses (vGW IP addresses) on the meta-data information of each gateway and device.	Essential	Yes
Dynamic Gateway Selection	Similar to automatic configuration, dynamic gateway selection changes are pushed using Websockets to the frontend and updated on the map automatically. The use can trigger the GW selection by the provided battery and load sliders.	Essential	Yes
Capillary Networks Policies	With an interactive floating window, the user can retrieve the capillary network policies in raw JSON format and update on the same page.	Essential	Yes
Machine Device Data Representation	The MD data is represented in either graph or raw format in the floating window of an MD icon on the map. The graph is interactive and supports several features (zoom in/out, individual data point retrieval etc.) for a better user experience. Raw data is available as an extra feature.	Essential	Yes
Machine Device Data's Security Representation	The interface shows in the MD data graph shows individually if each datapoint was sent encrypted to the IoT Framework from the MD. However, the capillary networks architecture does not use any encryption of datapoints at the time of research. Hence, all data points show "Not signed/encrypted" and this requirement is partially satisfied.	Additional Feature	Partially
Search Functionality	The frontend has search functionality implemented but the search can only be performed in IoT Framework streams, not for resources. Found stream data is presented similar to showing MD data readings.	Additional Feature	Partially
Meta Data Representation	Meta data of CGWs and MDs are presented in two different ways: First, a summary in the infobubble for quick overview and then, a detailed view in a floating window.	Essential	Yes
Error Reporting	The frontend shows the important errors and critical updates from GWs as notification on the main page. Logs and network counters are available as well in detailed information window of a GW. Notifications, logs and network counters are sent to the frontend via Websockets, enabling real-time monitoring.	Essential	Yes

Congestion Demonstration	The user can initiate the congestion demonstration via a separate window and can input the desired attribute values.	Additional Feature	Yes
Anomaly Detection	Anomaly detection can be optionally applied on the MD data in the MD data graph window i.e. the default data graph does not trigger anomaly detection. If executed, anomalies are visualized as red on the graph. However, the functionality of anomaly detection is limited to only predefined data sets, making this requirement not fully satisfied.	Additional Feature	Partially
LWM2M Server Integration	A separate screen is implemented as a LWM2M terminal (shell) window on the interface, in which the user can communicate with the LWM2M server. However, the functionality of the server is limited and the frontend requires a better representation of LWM2M server integration than a simple shell window.	Additional Feature	Partially

Table 6.1 indicates that the all of the “Essential” requirements are implemented and fully functional on the management interface frontend. Though, some of the “Additional Features” are partially implemented since neither the current architecture supports the full functionality nor the external sources supported the features. All things considered, the frontend implementation can be considered to satisfy the requirements in general.

6.1.2 Backend Requirements Evaluation

Backend requirements presented in Table 3.2 in Section 3.2 are evaluated individually below in Table 6.2.

Table 6.2 Backend Requirements Evaluation

Feature	Requirement Evaluation	Importance	Satisfied?
Automatic Configuration	The use of a Pub/Sub node.js server in the backend enables the automatic information push to the user in case of any updates such as network topology change, logs, notification or network counters. This Pub/Sub server is subscribed to CNF and hence, CNF updates (e.g. new MD) are immediately pushed to the end user.	Essential	Yes
Dynamic Middleware	The backend retrieves the middleware (vGW IP) created for each GW from CNF and attaches it to the meta-data of the CGW. Hence, the middleware info is shown along with the meta-data.	Essential	Yes

Dynamic Gateway Selection	The backend forwards the battery and load constraint updates to CNF. Gateway selection logic in CNF might change the network topology if the requirements are met. If updated, the new network topology is pushed to the user via Pub/Sub server.	Essential	Yes
Capillary Networks Policies	The policies are retrieved from CNF and any updates made by the user are also forwarded to CNF.	Essential	Yes
Machine Device Data Representation	PHP application coordinates the communication with the IoT Framework, retrieves the MD data and formats it if needed. However, the backend only supports read and search functionalities of IoT Framework since update/create/delete are performed by either CNM or CNF only. MD registration/de-registration is done in CNM or CNF as well and that's why this requirement is partially satisfied.	Essential	Partially
Machine Device Data's Security Representation	Since the Capillary Networks architecture at the moment of research does not encrypt the MD data from CGW to the cloud but only authenticates the CGW using GBA, the backend does not receive any information about MD data security. Hence, it only shows "Not Signed/Encrypted" for each data point in MD data readings.	Additional Feature	Partially
Search Functionality	The backend supports the search functionality in IoT Framework only for streams. Hence, searching for resources or users has not been implemented.	Additional Feature	Partially
Meta Data Representation	Detailed meta data information about CGW and MDs is collected from 3 different sources: CNF, vGW and IoT Framework. The backend parses these data and creates the combined meta data to be sent to the user.	Essential	Yes
Error Reporting	Pub/Sub server in CNF informs the Pub/Sub server in the management interface backend about any errors and the backend forwards the data in real-time to the user in terms of notification. This assures real-time network monitoring.	Essential	Yes
Congestion Demo	The backend uses the REST API towards CNF to simulate a congested network in a given area with the given amount of CGWs. The results are then forwarded to the user.	Additional Feature	Yes

Anomaly Detection	The backend uses the application written in Matlab to perform anomaly detection for a given set of MD data readings. The interface between PHP application and this Matlab program is a temporary local file. However, writing to and reading from the file is slow and the anomaly detection algorithm supports only few datasets as the valid input, which makes this feature not fully functional for every MD dataset.	Additional Feature	Partially
LWM2M Server Integration	LWM2M commands are read from the LWM2M shell on the frontend. PHP application forwards these commands to the LWM2M server and retrieves the response back. However, LWM2M server at the time of research is not fully functional and the integration of LWM2M server and the management interface needs to be updated from just a shell window.	Additional Feature	Partially

Table 6.2 indicates that most of the "Essential" requirements are implemented in the backend. Partially satisfied "Essential" features lack the functionality from the related entities and to be included in the future. Moreover, some "Additional Features" are also partially implemented since neither the current architecture supports the full functionality nor the external sources supported the features. The evaluation denotes that the backend implementation satisfies the requirements in a general perspective.

6.2 The Interface Performance Analysis

The management interface is designed to manage very large number of IoT devices which are spread over different, wide geographical areas. The interface also supports real-time updates from the network topology for viable device management. However, the actual number of devices the interface can handle and the limits of the underlying architecture remain unclear. In this section, several tests are performed to clarify the ambiguities about the management interface performance.

Testing Methodology

The testing was aimed at creating different amount of simulated CGWs and MDs in the Capillary Networks architecture and showing them on the management interface. The method for this purpose included using the *set_devices* API towards CNM in a separate script to inform CNM periodically about the CGWs and devices. CNM informs the CNF about the new network elements and the Pub/Sub server in CNF notifies the Pub/Sub server in the management interface. Then, the devices are loaded on the browser. Thus, the logic of the script simulates the same steps as if an actual CGW and sensors were actually deployed. The logic of the script can be summarized as follows:

1. Register a CGW in CNF.
2. Set coordinates for the CGW in CNM (*cmd=set_coord*).

3. Generate random IDs for random number (between 1 and 5) of sensors. We assume that the maximum five sensors connects to the GW.
4. Set coordinates for these sensors in CNM (*cmd=set_coord*).
5. Set connectivity of the sensors to the CGW in CNM (*cmd=set_devices*).
6. Keep connectivity alive by periodically informing CNM by a separate HTTP request for each GW.
7. Delete the CGWs from CNF after connectivity in CNM times out i.e. the script is terminated.

This method tests the performance of both the backend and the frontend. Hence, the measurements during the test were obtained both from the backend using Apache access logs and from the frontend using Google Chrome Version 38.0.2125.101 and integrated website inspection tool [65].

The testing was performed by increasing the number of simulated CGWs from 10 to 10000, each time by 10 times. Each CGW is assigned 4 sensors connected to it so the total number of CGWs and MDs on the system is 5*CGWs i.e. reaching maximum of 50000 at the final step of the test.

Results

Table 6.3, Table 6.4 and Table 6.5 below present the HTTP and WebSocket traffic received by the browser after the page is reloaded and the script is being executed. Files refer to total number of files received, KB refers to total length of the files and the time is the loading time of the entire files in milliseconds.

Table 6.3 Measurement Results of HTTP Requests for HTML, JS-Core and JS-Google Maps

	HTTP								
GWs	HTML			JS – Core			JS – Google Maps		
	Files	KB	Time (ms)	Files	KB	Time (ms)	Files	KB	Time (ms)
10	1	2.7	211	34	425	621	14	158	466
100	1	2.7	177	34	424	623	14	158	470
1000	1	2.7	180	34	424	621	14	158	472
10000	1	2.7	203	34	424	642	14	158	478

Table 6.4 Measurement Results of HTTP Requests for CSS and Images

	HTTP					
GWs	CSS			Images		
	Files	KB	Time (ms)	Files	KB	Time (ms)
10	17	60.1	476	93	630	1060
100	17	60.3	481	93	635	1070
1000	17	60.1	477	93	635	1068
10000	17	60.1	486	93	635	1084

Table 6.5 Measurement Results from WebSockets and Browser Memory Use

GWs	WebSocket				Browser Memory Use	
	Number of Frames	First Frame (KB)	Last Frame (KB)	Frame Update Interval (sec)	Memory Use Rate	In Total (MB)
10	11	2.18	5.96	2.2	1%	40
100	152	2.18	16.19	3.9	10%	380
1000	1870	2.18	88.32	6.8	26%	1000
10000	-	2.18	-	11	45%	1700

Table 6.3 and Table 6.4 show that the size and the loading time of the page elements (HTML, JS, CSS and images) do not change with the increasing number of GWs, which is an expected, obvious result since the same page is loaded each time. However, increasing the number of GWs and associated sensors (4 sensors for each GW) dramatically effect the management interface performance as seen in Table 6.5.

Table 6.5 presents that the frame content received by the browser increases together with the number of GWs and sensors. Though, frame update interval i.e. the interval between the device list updates observed on the frontend increases as well. It is also observed that the memory use of the browser grows significantly, which makes the browser unresponsive for the case with 10000 GWs and 40000 sensors.

One of the reasons behind this worse performance with the increased number of simulated GWs is the design of the communication between CNF and the management interface. Each time the CNF receives *any* update on the network, it notifies the management interface to retrieve the *entire* device list again. In the test script, connectivity requests for each GW are sent separately to CNM for each GW. Hence, the network is updated every time when CNM receives a connectivity update for a GW. This results in the management interface performing immense amount of consecutive HTTP requests to CNF to retrieve the entire device list, parsing each response and pushing the data via the WebSocket to the browser.

From the frontend side, the WebSocket receives frequent device list updates in different frames, which triggers several JavaScript functions. However, the functions are incapable of handling this amount of frequent updates, which is caused by a possible memory leak as shown in Figure 6-1.

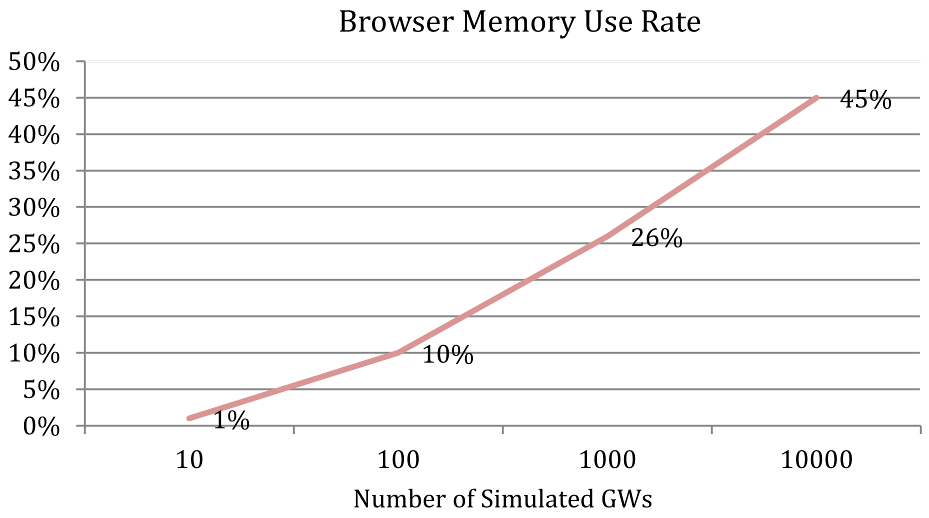


Figure 6-1 Browser Memory Use Rate

The experiment reveals that increasing the time interval to send updates to CNM in the test script to 5 seconds clearly improves the performance of the management interface, shown in Table 6.6.

Table 6.6 Measurement Results from WebSockets with latency and Browser Memory Use

GWs	WebSocket				Browser Memory Use	
	Number of Frames	First Frame (KB)	Last Frame (KB)	Frame Update Interval (sec)	Memory Use Rate	In Total (MB)
10	11	2.18	5.96	5	1%	40
100	144	2.18	16.19	5	5%	140
1000	1830	2.18	44.32	5	16%	500
10000	19256	2.18	128.64	5	27%	1050

Table 6.6 clarifies that the management interface is capable of handling large amount of devices if the device list update rate is increased to 5 seconds or above. However, to decrease the 5 seconds latency and have a real-time update capability with large number of devices, several changes both in Capillary Networks architecture and the management interface backend/frontend are needed, for instance:

- The device list retrieval from CNF to the management interface should be redesigned i.e. the device list is pushed to the management interface directly within the notification message without the interface making a separate POST request to CNF for retrieving the list.
- CNF should return the same format of device list that the management interface uses. This eliminates the need of data parsing in the interface backend and decreases latency.
- Instead of the entire device list, which might consist of 1000s of GWs and sensors unchanged, data fragments for only the GWs and sensors which have been updated should be transmitted. This significantly decreases the message size on the WebSocket frames.
- Similar to fragmented data, CNF should send notifications only for the updated GWs and sensors, instead of the entire network.

Conclusions

The measurements show that the management interface is capable of handling large amount of devices provided that the device list updates have a minimum of 5 seconds latency. Trying to have real-time updates with many devices, i.e. 10000 CGWs and 40000 sensors, result in the interface being unresponsive. This provides that the device retrieval design from CNF to the management interface is not optimal for frequent updates and to achieve real-time updates, the architecture between CNM, CNF and the management interface should be re-designed.

6.3 Anomaly Detection Performance Analysis

Section 5.2.1 Anomaly Detection discusses the integration of anomaly detection to the management interface as a proof of concept. However, the anomaly detection program performs rather slowly with the current implementation and the integration. In this section, we perform tests to measure the performance of the anomaly detection program with the management interface.

Testing Methodology

For these measurements, the management interface device data graph window is used. Anomaly detection algorithm is executed on different amount of data points starting from 100 and increasing up to 5000. The response time is measured for each try from Apache access logs in the management interface backend.

Results

Table 6.7 below shows the results of applying the anomaly detection program to a given number of data points. The time spent for executing the program increases with the number of given data points, as expected. However, the backend takes more than 11 seconds (11275 ms) to respond to the anomaly detection of 5000 data points, which is a significantly large latency.

The latency value proves that the anomaly detection program written in Matlab is not fast enough for processing large amount of data sets. Moreover, using a temporary file for communicating between the PHP application and the Matlab program increases the latency even more.

Table 6.7 Measurements from Anomaly Detection Test

	HTTP		Browser Memory Use	
Number of data points	Size of received data set (KB)	Time spent (ms)	Memory Use Rate	In Total (MB)
100	1.12	4024	1%	40
1000	4.5	4800	2%	75
1500	6.5	5180	2%	75
2500	9.4	6092	2%	75
5000	18.95	11275	3%	100

Decreasing the latency can be achieved by both providing the anomaly detection program in a faster programming language (e.g. python) and integrating it to PHP application natively, rather than using a temporary folder.

Regarding the browser memory use, increased number of data points has insignificant effect on the memory use at the user end as seen in Table 6.7.

Conclusions

The measurements prove that the current anomaly detection program integration experiences significant latency when the number of data points increases. Implementing the program in faster language and integrating it natively with the PHP application can decrease the response time notably.

7 Conclusions

In this thesis, we have presented a Web-based, standalone device management interface for IoT networks and devices running over the Capillary Networks architecture. We first introduced the IoT concept, which consists of connecting real life devices to the network to enable horizontal application development using the data collected by the devices. However, the expected large number and resource constraints of the devices have exposed the device management problem in IoT networks, which we discussed during the thesis.

Communication protocols specifically designed for constrained networks, which are CoAP and LWM2M have been introduced along with the protocols used to integrate the management interface to the existing Internet, which are HTTP and WebSocket. The advantage of CoAP over HTTP has been pointed out to be more suitable for constrained devices with reduced overhead, use of UDP, short message size and support of Observe/Notify mechanism. LWM2M, which runs on CoAP, completed the IoT device management stack by being a lightweight, object-based management protocol suitable for constrained devices. Combining HTTP and WebSocket over CoAP and LWM2M has enabled the management interface to be a Web-based service.

The constrained nature of the IoT networks required the analysis of IoT-specific features such as Publish/Subscribe, aggregation, anomaly detection, prioritization and error detection. It was discussed that Publish/Subscribe scheme can be extensively used in IoT network management for real-time management and monitoring and hence, it was included in the implementation as a core feature.

The requirements of the management interface were based on the Capillary Networks architecture, common network management features and additionally IoT-specific features. The design of both the frontend and the backend of the interface were prepared to satisfy the requirements and to tackle the legacy problems introduced by the Capillary Networks architecture. Moreover, the implementation of the interface was done using the proposed design by combining several Web technologies and frameworks. Moreover, the implemented interface required to be in communication with many Capillary Networks entities through the use of extended REST APIs.

The measurements retrieved from the management interface tests indicated the limits of the system. The tests showed that the interface did not perform satisfactorily with frequently updated, high number of devices, leading the interface to get considerably slow for the end user. The reason for this was found to be the architecture of Capillary Networks and the integration of the management interface on top of it. Solution for this problem obviously requires the architectural update of Capillary Networks and redefining the features of the entities in Capillary Networks cloud for better integration with the management interface.

The research topic and the implementation presented in the thesis offer yet another way of device management in IoT networks and can be extended to include new features, protocols, different types of devices and gateways along with the integration of cloud resource management. Hence, the objective was to present the initial, fundamental concept for IoT device management for future networks.

Future Work

The management interface implemented for this thesis can be developed further for a better system performance. The ideas are presented below.

- **Redesigning the Capillary Networks Architecture:** The tests discussed in Chapter 6 shows that the communication between Capillary Networks entities and the management interface are not optimized for large-scale networks. The architecture can be redesigned to tackle the problems such as real-time network update and to remove the legacy network entities or communication paths.
- **Additional Use of Node.js:** The increased use of Node.js, possibly replacing the PHP application, can introduce better performance on the interface and on the real-time management. Node.js is capable of handling more concurrent processes and can be integrated to the cloud instances as a simpler process than PHP.
- **Device List Fragmentation:** The current architecture forwards the entire list of devices to the end user when a single update is observed in the network. Instead, fragmented device list including only the updated part of the network should be sent to decrease the message list format and to optimize the server processing.
- **Integration of Additional LWM2M Features:** Although a preliminary version of a LWM2M Server is integrated to the management interface, integration of actual LWM2M Clients with the interface and adding support for LWM2M features, such as subscribing to LWM2M clients, would ensure better representation of the end-to-end communication with the client and the server.
- **Support of Multi-Tenancy:** Current Capillary Networks and management interface architecture do not support multi-tenancy of CGWs and sensors in different levels of ownership. Adding this support can introduce use cases which are closer to the industry level examples. However, the architecture needs to be updated heavily to achieve multi-tenancy.

Bibliography

[1] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A Survey," *Computer Networks*, no. 54, pp. 2787-2805, June 2010.

[2] Jan Höller et al., *From Machine-to-Machine to the Internet of Things: Introduction to a New Age of Intelligence*, 1st ed. Oxford, The UK: Academic Press, 2014.

[3] M. Alam, R.H. Nielsen, and N.R. Prasad, "The evolution of M2M into IoT," in *Communications and Networking (BlackSeaCom), 2013 First International Black Sea Conference on*, Batumi, 2013, pp. 112-115.

[4] Lu Tan, East China Normal Univ., Shanghai, China Comput. Sci. & Technol. Dept., and Neng Wang, "Future internet: The Internet of Things," in *Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference on*, vol. 5, Chengdu, 2010, pp. 376-380.

[5] Ericsson, More Than 50 Billion Connected Devices , February 2011.

[6] Ericsson, "Low Energy IoT Capillary Networks," Internal Report 2013.

[7] COMMUNE Web site. [Online]. http://projects.celtic-initiative.org/commune/commune.html

[8] R. Fielding et al., Hypertext transfer protocol–HTTP/1.1., June 1999, https://www.ietf.org/rfc/rfc2616.txt.

[9] Z. Shelby, ARM, K. Hartke, C. Bormann, and Universitat Bremen TZI, The Constrained Application Protocol (CoAP), June 2014, http://tools.ietf.org/html/rfc7252.

[10] Open Mobile Alliance, OMA Lightweight Machine to Machine Device Management (OMA-DM), December 2013, OMA Lightweight M2M v1.0.

[11] Brian Lavoie and Henrik Frystyk Nielsen. (1999, May) Web Characterization Terminology & Definitions Sheet. [Online]. http://www.w3.org/1999/05/WCA-terms/

[12] T. Berners-Lee, CERN, L. Masinter, Xerox Corporation, and M. McCahill, Uniform Resource Locators (URL), December 1994, https://www.ietf.org/rfc/rfc1738.txt.

[13] Roy Thomas Fielding, Architectural Styles and the Design of Network-based Software Architectures, 2000.

[14] Hongjun Li, "RESTful Web Service Frameworks in Java," in *Signal Processing, Communications and Computing (ICSPCC), IEEE International Conference*,

Xi'an, 2011, pp. 1-4.

[15] L. Richardson and S. Ruby, *RESTful Web Services*.: O'Reilly & Associates, May 2007.

[16] F. Belqasmi, R. Glitho, and Chunyan Fu, "RESTful web services for service provisioning in next-generation networks: a survey ," *Communications Magazine, IEEE* , vol. 49, no. 12, pp. 66-73, December 2011.

[17] I. Fette, Google Inc., A. Melkinov, and Isode Ltd., The WebSocket Protocol, December 2011, http://tools.ietf.org/html/rfc6455.

[18] V. Pimentel and B.G. Nickerson, "Communicating and Displaying Real-Time Data with WebSocket," *Internet Computing, IEEE*, vol. 16, no. 4, May 2012.

[19] Ian Hickson and Google Inc. (2014, June) The WebSocket API. [Online]. http://dev.w3.org/html5/websockets/

[20] G. Montenegro et al., Transmission of IPv6 Packets over IEEE 802.15.4 Networks, September 2007, http://tools.ietf.org/html/rfc4944.

[21] T. Winter et al., RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks, March 2012, https://tools.ietf.org/html/rfc6550.

[22] Walter Colitti, Kris Steenhaut, Niccolo De Caro, Bogdan Buta, and Virgil Dobrota, "Evaluation of Constrained Application Protocol for Wireless Sensor Networks," in *Local & Metropolitan Area Networks (LANMAN)*, Chapel Hill, 2011, pp. 1-6.

[23] Carsten Bormann, Angelo Castellani, and Zach Shelby, "CoAP: An Application Protocol for Billions of Tiny Internet Nodes," *IEEE Internet Computing*, vol. 16, no. 2, pp. 62-67, March 2012.

[24] K. Hartke and Universitaet Bremen TZI, Observing Resources in CoAP, June 2014, http://tools.ietf.org/html/draft-ietf-core-observe-14.

[25] (2014, September) IPSO Alliance. [Online]. http://www.ipso-alliance.org/

[26] V.N. Gudivada, Marshall Univ., Huntington, WV, USA Weisburg Div. of Comput. Sci., D. Rao, and V.V. Raghavan, "NoSQL Systems for Big Data Management ," in *Services (SERVICES), 2014 IEEE World Congress on* , Anchorage , 2014, pp. 190-197.

[27] W. Naheman, Xinjiang Univ., Urumqi, China Coll. of Resources & Environ. Sci., and Jianxin Wei, "Review of NoSQL databases and performance testing on HBase ," in *Mechatronic Sciences, Electric Engineering and Computer (MEC), Proceedings 2013 International Conference on* , 2013, pp. 2304-2309.

[28] S. Sakr, Sydney, NSW, Australia Univ. of New South Wales & Nat. ICT Australia (NICTA), A. Liu, D.M. Batista, and M. Alomari, "A Survey of Large Scale Data Management Approaches in Cloud Environments ," *Communications Surveys & Tutorials, IEEE* , vol. 13, no. 3, pp. 311-336, April 2011.

[29] Guoxi Wang, Tongji Univ., Shanghai, China Sch. of Software Eng., and Jianfeng Tang, "The NoSQL Principles and Basic Application of Cassandra Model ," in *Computer Science & Service System (CSSS), 2012 International Conference on* , Nanjing , 2012, pp. 1332-1335.

[30] T. Bray and Google Inc., The JavaScript Object Notation (JSON) Data Interchange Format, March 2014, http://tools.ietf.org/html/rfc7159.

[31] G. Xylomenos et al., "A Survey of Information-Centric Networking Research," *Communications Surveys & Tutorials, IEEE* , vol. 16, no. 2, pp. 1024-1049, July 2013.

[32] B. Dannewitz, C. Ahlgren, C. Imbrenda, D. Kutscher, and B. Ohlman, "A survey of information-centric networking," *Communications Magazine, IEEE* , vol. 50, no. 7, pp. 26-36, July 2012.

[33] I.P. Zarko, K. Pripuzic, M. Serrano, and M. Hauswirth, "IoT data management methods and optimisation algorithms for mobile publish/subscribe services in cloud environments," in *Networks and Communications (EuCNC), 2014 European Conference on*, Bologna, 2014, pp. 1-5.

[34] D. Clement, C. Michel, and B. Cyrille, "Wireless Sensor Network Cloud services: Towards a partial delegation," in *Smart Communications in Network Technologies (SaCoNeT), 2014 International Conference on*, Vilanova i la Geltru, 2014, pp. 1-6.

[35] S. Ben Fredj, M. Boussard, D. Kofman, and L. Noirie, "A Scalable IoT Service Search Based on Clustering and Aggregation," in *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*, Beijing, 2013, pp. 403-410.

[36] S.D.T. Kelly, N.K. Suryadevara, and S.C. Mukhopadhyay, "Towards the Implementation of IoT for Environmental Condition Monitoring in Homes," *Sensors Journal, IEEE*, vol. 13, no. 10, pp. 3846-3853, May 2013.

[37] F. McAtee, S. Narayanan, and G.G. Xie, "Performance analysis of message prioritization in delay tolerant networks," in *Military Communications Conference, IEEE*, Orlando, 2012, pp. 1-6.

[38] G.D. Troxel and L. Poplawski Ma, "Secure Network Attribution and Prioritization: A Coordinated Architecture for Critical Infrastructure," in *Military Communications Conference, IEEE*, San Diego, 2013, pp. 226-230.

[39] Weiyu Zhang, Qingbo Yang, and Yushui Geng, "A Survey of Anomaly Detection Methods in Networks," in *Computer Network and Multimedia Technology, 2009. CNMT 2009. International Symposium on*, Wuhan, 2009, pp. 1-3.

[40] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly Detection for Discrete Sequences: A Survey," *Knowledge and Data Engineering, IEEE Transactions*

on, vol. 24, no. 5, pp. 823-839, November 2010.

[41] 3GPP TS 33.220 - Generic Authentication Architecture (GAA), Generic Bootstrapping Architecture (GBA).

[42] Project CS 2013 by Uppsala University and Ericsson Research. (2014, March) IoT-Framework Engine. [Online]. https://github.com/projectcs13/sensor-cloud

[43] Elasticsearch BV. (2014, March) Elasticsearch. [Online]. http://www.elasticsearch.org/

[44] W3C. (2014, March) HTML5. [Online]. http://www.w3.org/TR/html5/

[45] W3C. (2014, March) CSS. [Online]. http://www.w3.org/Style/CSS/

[46] Mozilla. (2014, March) Javascript. [Online]. https://developer.mozilla.org/en/docs/Web/JavaScript

[47] Bootstrap. Bootstrap Framework. [Online]. http://getbootstrap.com/

[48] The jQuery Foundation. jQuery. [Online]. http://jquery.com/

[49] WebSocket. [Online]. https://www.websocket.org/

[50] Google Inc. (2014, March) Google Maps. [Online]. https://developers.google.com/maps

[51] Markerclusterer. [Online]. http://google-maps-utility-library-v3.googlecode.com/svn/trunk/markerclusterer/

[52] Infobubble. [Online]. http://google-maps-utility-library-v3.googlecode.com/svn/trunk/infobubble/

[53] Flot. [Online]. http://www.flotcharts.org/

[54] Vkbeautify. [Online]. https://code.google.com/p/vkbeautify/

[55] PHP: Hypertext Processor. [Online]. http://php.net/

[56] Yii PHP Framework. [Online]. http://www.yiiframework.com/

[57] Avraham Leff and James T Rayleigh, "Web-application development using the Model/View/Controller design pattern," in *Enterprise Distributed Object Computing Conference, 2001. EDOC '01. Proceedings. Fifth IEEE International*, Seattle, 2001, pp. 118-127.

[58] Node.js Project. [Online]. http://nodejs.org/

[59] Redis. [Online]. http://redis.io/

[60] How fast is Redis? [Online]. http://redis.io/topics/benchmarks

[61] Domenico D'ambrosio, A Group Communication Service for Lightweight M2M (LWM2M), 2014, Università degli Studi di Napoli 'Federico II'.

[62] MySQL. [Online]. http://www.mysql.com/

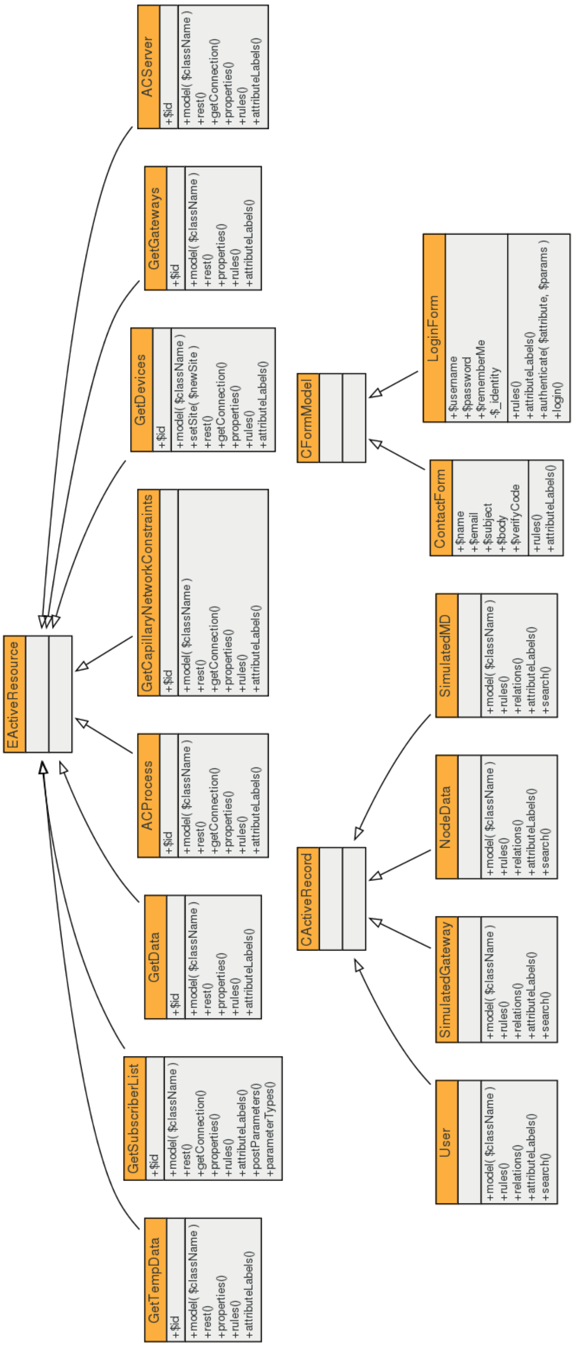
[63] Java Colour Tracker. [Online]. http://www.uk-dave.com/projects/misc/java-colour-tracker/

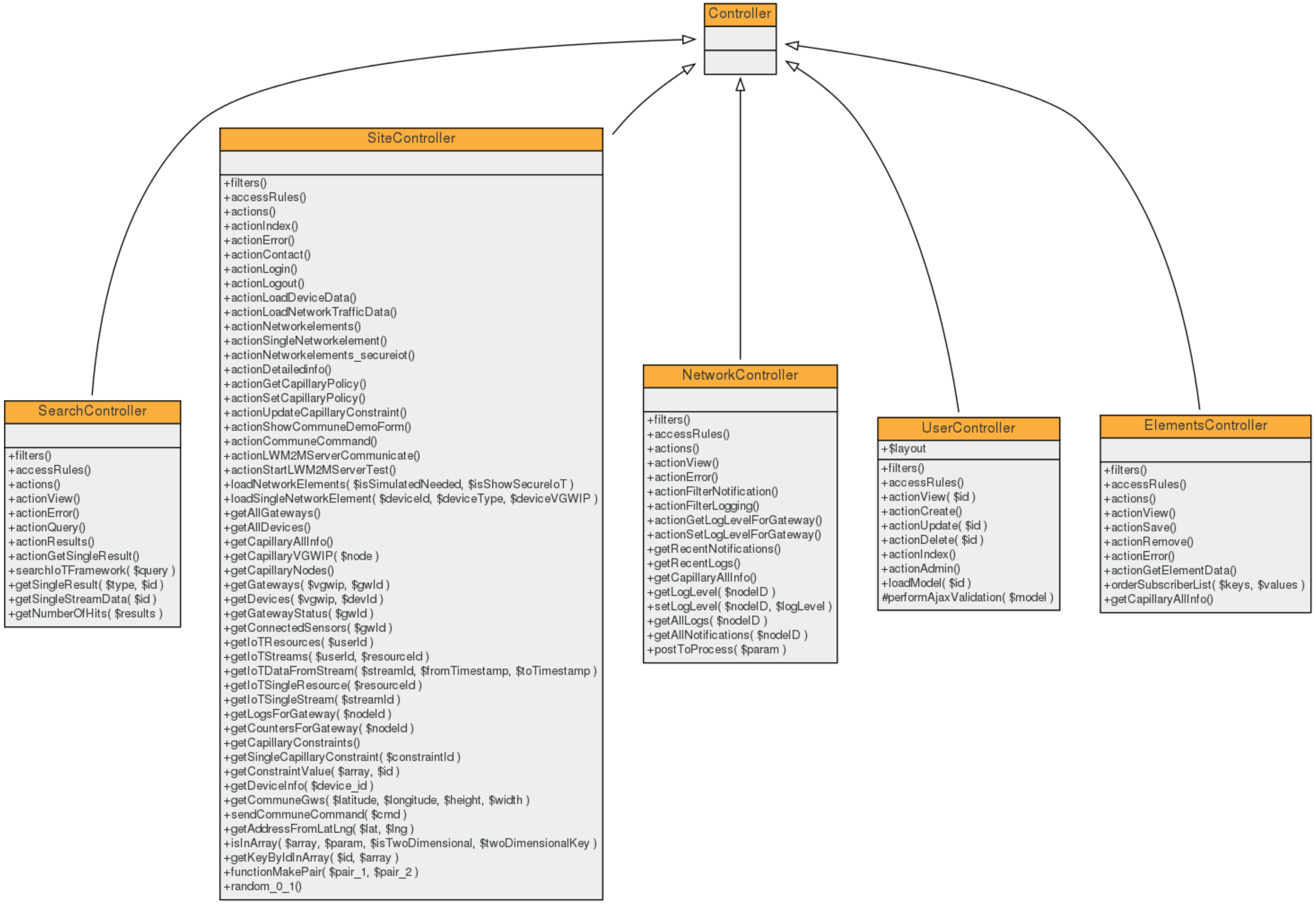
[64] C. C. Robusto, "The Cosine-Haversine Formula," *The American Mathematical Monthly*, vol. 64, no. 1, pp. 38-40, January 1957.

[65] Google Chrome Browser, , http://www.google.com/chrome/.

Appendix A

UML Diagrams of Backend Implementation




Controller
SiteController
+filters()
+accessRules()
+actions()
+actionIndex()
+actionError()
+actionContact()
+actionLogin()
+actionLogout()
+actionLoadDeviceData()
+actionLoadNetworkTrafficData()
+actionNetworkelements()
+actionSingleNetworkelement()
+actionNetworkelements_secureiot()
+actionDetailedinfo()
+actionGetCapillaryPolicy()
+actionSetCapillaryPolicy()
+actionUpdateCapillaryConstraint()
+actionShowCommuneDemoForm()
+actionCommuneCommand()
+actionLWM2MServerCommunicate()
+actionStartLWM2MServerTest()
+loadNetworkElements($isSimulatedNeeded, $isShowSecureIoT)
+loadSingleNetworkElement($deviceId, $deviceType, $deviceVGWIP)
+getAllGateways()
+getAllDevices()
+getCapillaryAllInfo()
+getCapillaryVGWIP($node)
+getCapillaryNodes()
+getGateways($vgwip, $gwId)
+getDevices($vgwip, $devId)
+getGatewayStatus($gwId)
+getConnectedSensors($gwId)
+getIoTResources($userId)
+getIoTStreams($userId, $resourceId)
+getIoTDataFromStream($streamId, $fromTimestamp, $toTimestamp)
+getIoTSingleResource($resourceId)
+getIoTSingleStream($streamId)
+getLogsForGateway($nodeId)
+getCountersForGateway($nodeId)
+getCapillaryConstraints()
+getSingleCapillaryConstraint($constraintId)
+getConstraintValue($array, $id)
+getDeviceInfo($device_id)
+getCommuneGws($latitude, $longitude, $height, $width)
+sendCommuneCommand($cmd)
+getAddressFromLatLng($lat, $lng)
+isInArray($array, $param, $isTwoDimensional, $twoDimensionalKey)
+getKeyByIdInArray($id, $array)
+functionMakePair($pair_1, $pair_2)
+random_0_1()
SearchController
+filters()
+accessRules()
+actions()
+actionView()
+actionError()
+actionQuery()
+actionResults()
+actionGetSingleResult()
+searchIoTFramework($query)
+getSingleResult($type, $id)
+getSingleStreamData($id)
+getNumberOfHits($results)
NetworkController
+filters()
+accessRules()
+actions()
+actionView()
+actionError()
+actionFilterNotification()
+actionFilterLogging()
+actionGetLogLevelForGateway()
+actionSetLogLevelForGateway()
+getRecentNotifications()
+getRecentLogs()
+getCapillaryAllInfo()
+getLogLevel($nodeID)
+setLogLevel($nodeID, $logLevel)
+getAllLogs($nodeID)
+getAllNotifications($nodeID)
+postToProcess($param)
UserController
+$layout
+filters()
+accessRules()
+actionView($id)
+actionCreate()
+actionUpdate($id)
+actionDelete($id)
+actionIndex()
+actionAdmin()
+loadModel($id)
#performAjaxValidation($model)
ElementsController
+filters()
+accessRules()
+actions()
+actionView()
+actionSave()
+actionRemove()
+actionError()
+actionGetElementData()
+orderSubscriberList($keys, $values)
+getCapillaryAllInfo()

Appendix B

REST APIs

In this Appendix, we present the full list of REST APIs used in the implementation of the management interface. The APIs are grouped to five as REST APIs towards CNF, Demonstration, CNM, IoT Framework and vGW. Each API entry is explained in detail and supported request/response examples.

REST APIs Towards CNF

The APIs presented in this section are used by the management interface components (see Chapter 5) for communication with the Capillary Network Function (CNF).

Get the full list of available CGWs and MDs

Verb	URI	Description
POST	/process.php	Get the full list of available CGWs and MDs

Normal Response Code(s): 200

Error Response Code(s): unauthorized (401), badRequest (400)

Response Format: JSON

This interface gets the full list of available CGWs and MDs with additional info such as connected MDs to CGW, vGW IP, log level, GPS coordinates and online status. If the operation fails, an HTML error code is returned with an error message in output.

POST Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
cmd	Body	String	get_info	Yes	Yes

Example request body

```
cmd=get_info
```

Example response body

```
[{"node":"2","devices":"","vGW":
"193.234.217.173,2001:14b8:400:131:f816:3eff: fe4f:de50", "online":"True",
"level":"0", "params":{"2": {"lat": 41.35342516, "lng": 2.13135839},
"280:e103:1:2a9c": {"lat": 41.35361845, "lng": 2.13043571}} }]
```

Get the list of all available CGWs

Verb	URI	Description
POST	/process.php	Get the list of all available CGWs

Normal Response Code(s): 200

Error Response Code(s): unauthorized (401), badRequest (400)

Response Format: JSON

This interface gets the list of all available CGWs. If the operation fails, an HTML error code is returned with an error message in output.

POST Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
cmd	Body	String	nodes	Yes	Yes

Example request body

```
cmd=get_nodes
```

Example response body

```
["1", "2", "2911", "3", "4", "5"]
```

Get the list of all connected MDs for a CGW

Verb	URI	Description
POST	/process.php	Get the list of all connected MDs for a CGW

Normal Response Code(s): 200

Error Response Code(s): unauthorized (401), badRequest (400)

Response Format: JSON

This interface gets the list of all connected devices (MDs) to the given CGWs. The output includes GPS coordinates of the listed MDs as well. If the operation fails, an HTML error code is returned with an error message in output.

POST Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
cmd	Body	String	get_devices	Yes	Yes
node	Body	String	Node identifier	Yes	Yes

Example request body

```
cmd=get_devices&node=2
```

Example response body

```
{ "node":"2", "devices":"", "params":{"2": {"lat": 41.35342516, "lng":
2.13135839}, "280:e103:1:2a9c": {"lat": 41.35361845, "lng": 2.13043571}} }
```

Get the vGW IP address for a CGW

Verb	URI	Description
POST	/process.php	Get the vGW IP address for a CGW

Normal Response Code(s): 200

Error Response Code(s): unauthorized (401), badRequest (400)

Response Format: JSON

This interface gets the vGW IP address for the given CGW. The output includes both IPv4 and IPv6 addresses for the given CGW's vGW instance. If the operation fails, an HTML error code is returned with an error message in output.

POST Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
cmd	Body	String	get_vgw_ip	Yes	Yes
node	Body	String	Node identifier	Yes	Yes

Example request body

```
cmd=get_vgw_ip&node=1
```

Example response body

```
{"node":"1","ip4":"193.234.218.188", "ip6":"2a00:1d50:2:1001:
f816:3eff:fe94:4a3d" }
```

Get the online status for a CGW

Verb	**URI**	**Description**
POST	/process.php	Get the online status for a CGW

Normal Response Code(s): 200

Error Response Code(s): unauthorized (401), badRequest (400)

Response Format: JSON

This interface gets the online status for the given CGW. The output includes the CGW ID and the online status of the CGW, which can be “Online” or “Offline”. If the operation fails, an HTML error code is returned with an error message in output.

POST Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
cmd	Body	String	get_node_status	Yes	Yes

Name	Style	Type	Description/Value	Required	Case-Sensitive
node	Body	String	Node identifier	Yes	Yes

Example request body

```
cmd=get_node_status&node=1
```

Example response body

```
{"node":"1", "status":"Online" }
```

Get the latest logs of a CGW

Verb	URI	Description
POST	/process.php	Get the latest logs of a CGW

Normal Response Code(s): 200

Error Response Code(s): unauthorized (401), badRequest (400)

Response Format: JSON

This interface gets the latest N number of logs stored on CNF for the given CGW. N is a number determined by CNF and is not defined in this API. The output includes the list of log entries. Each log entry consists of log entry ID, log level and the log text. If the operation fails, an HTML error code is returned with an error message in output.

POST Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
cmd	Body	String	get_log	Yes	Yes
node	Body	String	Node identifier	Yes	Yes

Example request body

```
cmd=get_log&node=1
```

Example response body

```
[{ "id":"1694774", "level":"4", "entry":"2014-04-23 07:36:40 WARNING: GW 1
disconnected from the network (failed 3 times)" }, { "id":"1694775", "lev-
el":"5", "entry":"2014-04-23 07:37:17 GW 1 connected to the network" }, {
"id":"1694776", "level":"5", "entry":"2014-04-23 07:37:26 Device
280:e103:1:3829 (type temperature) connected to GW 1" }]
```

Get the most recent logs of all the CGWs

Verb	URI	Description
POST	/process.php	Get the most recent logs of all the CGWs

Normal Response Code(s): 200

Error Response Code(s): unauthorized (401), badRequest (400)

Response Format: JSON

This interface gets the most recent K logs stored on CNF for all the CGWs. K is a number determined by CNF and is not defined in this API. The output includes the list of log entries. Each log entry consists of log entry ID, CGW ID, log level and the log text. If the operation fails, an HTML error code is returned with an error message in output.

POST Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
cmd	Body	String	get_log	Yes	Yes
node	Body	String	common	Yes	Yes

Example request body

```
cmd=get_log&node=common
```

Example response body

```
[ { "id":"3395092", "node":"1", "level":"4", "entry":"2014-04-23 07:36:40
WARNING: GW 1 disconnected from the network (failed 3 times)" }, {
"id":"3395093", "node":"1", "level":"5", "entry":"2014-04-23 07:37:17 GW 1
connected to the network" }, { "id":"3395094", "node":"1", "level":"5", "en-
try":"2014-04-23 07:37:26 Device 280:e103:1:3829 (type temperature) connected
to GW 1" }]
```

Get the logging level for a CGW

Verb	URI	Description
POST	/process.php	Get the logging level for a CGW

Normal Response Code(s): 200

Error Response Code(s): unauthorized (401), badRequest (400)

Response Format: JSON

This interface gets the logging level for the given CGW. The output returns the CGW ID and the logging level. If the operation fails, an HTML error code is returned with an error message in output.

POST Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
cmd	Body	String	get_logging_level	Yes	Yes
node	Body	String	Node identifier	Yes	Yes

Example request body

```
cmd=get_logging_level&node=1
```

Example response body

```
{ "node":"1", "level":"5" }
```

Set the logging level for a CGW

Verb	URI	Description
POST	/process.php	Set the logging level for a CGW

Normal Response Code(s): 200

Error Response Code(s): unauthorized (401), badRequest (400)

Response Format: Plain Text

This interface sets the logging level of the given CGW to the given logging level value. The output returns a plain text if the operation succeeds with response code 200. If the operation fails, an HTML error code is returned with an error message in output. Hence, the operation success is determined using the response code.

POST Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
cmd	Body	String	set_logging_level	Yes	Yes
node	Body	String	Node identifier	Yes	Yes
level	Body	Integer	New log level value	Yes	No

Example request body

```
cmd=set_logging_level&node=1&level=4
```

Example response body

```
Success.
```

Get the latest notifications of a CGW

Verb	URI	Description
POST	/process.php	Get the latest notifications of a CGW

Normal Response Code(s): 200

Error Response Code(s): unauthorized (401), badRequest (400)

Response Format: JSON

This interface gets the latest N number of notifications stored on CNF for the given CGW. N is a number determined by CNF and is not defined in this API. The output includes the list of notification entries. Each notification entry consists of notification entry ID and the notification text. If the operation fails, an HTML error code is returned with an error message in output.

POST Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
cmd	Body	String	get_notification	Yes	Yes

Name	Style	Type	Description/Value	Required	Case-Sensitive
node	Body	String	Node identifier	Yes	Yes

Example request body

```
cmd=get_notification&node=2
```

Example response body

```
[ { "id":"1", "entry":"2014-04-10 11:34:05 GW 2 started" }, { "id":"2", "entry":"2014-04-10 11:34:05 GW 2 connected to the network" }, { "id":"3", "entry":"2014-04-10 12:40:53 GW 2 started" }, { "id":"4", "entry":"2014-04-10 12:43:18 GW 2 started" }, { "id":"5", "entry":"2014-04-10 12:43:19 GW 2 connected to the network" }, { "id":"6", "entry":"2014-04-10 12:50:30 GW 2 started" }, { "id":"7", "entry":"2014-04-10 12:50:32 Root device of GW 2 is connected" }]
```

Get the most recent notifications for all the CGWs

Verb	URI	Description
POST	/process.php	Get the most recent notifications for all the CGWs

Normal Response Code(s): 200

Error Response Code(s): unauthorized (401), badRequest (400)

Response Format: JSON

This interface gets the most recent K notifications stored on CNF for all the CGWs. K is a number determined by CNF and is not defined in this API. The output includes the list of notification entries. Each notification entry consists of notification entry ID, CGW ID and the notification text. If the operation fails, an HTML error code is returned with an error message in output.

POST Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
cmd	Body	String	get_notification	Yes	Yes
node	Body	String	common	Yes	Yes

Example request body

```
cmd=get_notification&node=common
```

Example response body

```
[ { "id":"63098", "node":"1", "entry":"2014-04-23 07:44:19 GW 1 connected to
the network" }, { "id":"63099", "node":"1", "entry":"2014-04-23 07:49:40 GW 1
connected to the network" }, { "id":"63100", "node":"1", "entry":"2014-04-23
07:50:19 WARNING: GW 1 disconnected from the network (failed 3 times)" }, {
"id":"63101", "node":"1", "entry":"2014-04-23 07:50:59 WARNING: GW 1 discon-
nected from the network (failed 3 times)" }]
```

Get the latest network counters of a CGW

Verb	URI	Description
POST	/process.php	Get the latest network counters of a CGW

Normal Response Code(s): 200

Error Response Code(s): unauthorized (401), badRequest (400)

Response Format: JSON

This interface gets the latest network counter values for the given CGW. The output includes a list of available network counters in key-value format, the key as the counter name and the value as the counter value. The availability of which counters to return depends on the CNF implementation and is out of the scope of this API. If the operation fails, an HTML error code is returned with an error message in output.

POST Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
cmd	Body	String	get_counters	Yes	Yes
node	Body	String	Node identifier	Yes	Yes

Example request body

```
cmd=get_counters&node=1
```

Example response body

```
{"constraints from":"configuration file","constraints in
use":"{\"1\":{\"battery\":5,\"load\":2},\"3\":{\"battery\":3,\"load\":1},
,\"2\":{\"battery\":4,\"load\":2}}","devices":"0","devices de-
nied":"0","devices offline":"0","devices right gw":"0" ,"devices wrong
gw":"0", "master":"True", "network access failed":"94 times","network access
failure rate":"55.294117647059 % (mean)"}
```

Get the Capillary Networks policy

Verb	URI	Description
GET	/access.controller/m-nf?cmd=getPolicy	Get the Capillary Networks policy

Normal Response Code(s): 200

Error Response Code(s): internalServerError (405)

Response Format: JSON

This interface gets the current Capillary Networks policy content stored on CNF. The output includes the policy in JSON. If the operation fails, an HTML error code is returned with an error message in output.

GET Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
cmd	URI	String	getPolicy	Yes	Yes

Example request body

```
http://0.0.0.0:8080/access.controller/m-nf?cmd=getPolicy
```

Example response body

```
{ "xxalarm": { "load": -1 }, "default": { "battery": 2, "load": -2, "connec-
tions": -1 }, "xxxdefault": { "battery": 1, "load": -1 }, "xxdefault": {
"connections": -1 } }
```

Set the Capillary Networks policy

Verb	URI	Description
GET	/access.controller/m-nf?cmd=setPolicy&policy=X	Set the Capillary Networks policy

Normal Response Code(s): 200

Error Response Code(s): internalServerError (405)

Response Format: JSON

This interface sets the current Capillary Networks policy with the given policy value. The given policy value is required to be in valid JSON format. The output includes a JSON object. If the “status” attribute in the returned JSON object is set to “ok”, it means that the policy is applied successfully. If the operation fails, an HTML error code is returned with an error message in output.

GET Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
cmd	URI	String	getPolicy	Yes	Yes
policy	URI	JSON	New policy value	Yes	Yes

Example request body

```
http://0.0.0.0:8080/access.controller/m-nf?cmd=setPolicy
&policy={"xxalarm":{"load": -5 }}
```

Example response body

```
{ "status": "ok” }
```

Get the battery/load constraint value for a CGW

Verb	URI	Description
GET	/access.controller/m-nf or /access.controller/m-nf?cmd=getConstraints or /access.controller/m-nf?cmd=getConstraints&id=X	Get current battery/load constraint values

Normal Response Code(s): 200

Error Response Code(s): internalServerError (405)

Response Format: JSON

This interface gets the current battery/load constraint values. These constraint values may not be available for all available CGWs. If the operation is performed without “*id*” parameter, the output returns the entire list of CGWs with the values of battery and load constraints. To perform the operation with “*id*” parameter, the parameter must be defined in this way:

id=NODE_ID+””+CONSTRAINT_ID

NODE_ID refers to the ID of CGW (e.g. 1 or 4) while *CONSTRAINT_ID* refers to the identifier for battery as 1 and load as 2. E.g. “*id=11*” refers to battery constraint of CGW with ID 1. The output to the request with “*id*” parameter returns the value of the constraint in JSON.

If the operation fails, an HTML error code is returned with an error message in output.

GET Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
cmd	URI	String	getConstraints	No	Yes
id	URI	Integer	Constraint Identifier	No	No

Example request body

```
http://0.0.0.0:8080/access.controller/m-nf?cmd=getConstraints
```

Example response body

```
{"3":{"battery":1,"load":1},"2":{"battery":1,"load":1},"1":{"battery":1,
"load":1},"5":{"battery":1,"load":1},"4":{"battery":1,"load":1}}
```

Example request body

```
http://0.0.0.0:8080/access.controller/m-nf?cmd=getConstraints&id=11
```

Example response body

```
{"11":3}
```

Set the battery/load constraint value for a CGW

Verb	URI	Description
GET	/access.controller/m-nf?cmd=setConstraint &id=X&value=Y	Set current battery/load constraint values

Normal Response Code(s): 200

Error Response Code(s): internalServerError (405)

Response Format: JSON

This interface sets the given battery/load constraint id to the given value. To perform the operation with "*id*" parameter, the parameter must be defined in this way:

id=NODE_ID+””+CONSTRAINT_ID

NODE_ID refers to the ID of CGW (e.g. 1 or 4) while *CONSTRAINT_ID* refers to the identifier for battery as 1 and load as 2. E.g. "*id=11*" refers to battery constraint of CGW with ID 1. The operation sets the given value to this defined ID. The output includes a JSON object. If the "status" attribute in the returned JSON object is set to "0", it means that the constraint value is updated successfully. If the attribute is set to "-1", there is an error while updating the constraint. If the operation fails, an HTML error code is returned with an error message in output.

GET Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
cmd	URI	String	setConstraint	Yes	Yes
id	URI	Integer	Constraint Identifier	Yes	No
value	URI	Integer	New value: Currently accepted values are from 1 to 5.	Yes	No

Example request body

```
http://0.0.0.0:8080/access.controller/m-nf?cmd=setConstraint&id=11&value=5
```

Example response body

```
{"status":0}
```

REST APIs towards CNF used for Demonstration

Get the list of gateways for congestion demonstration

Verb	URI	Description
GET	/access.controller/emulation? cmd=gws&lat=X&lon=Y&h=H&w=W	Set current battery/load constraint values

Normal Response Code(s): 200

Error Response Code(s): badRequest (400), internalServerError (405)

Response Format: JSON

This interface gets the list of gateways, which are already simulated before and stored on CNF. The output is a list of gateway entries available in the rectangle centered at (*lat*, *lon*) with the height of *h* and the width of *w*, the parameters defined in the request. A gateway entry in the response list consists of gateway's following properties: RAT type (wifi, xbee or IEEE 802.15.4), latitude, longitude, impi (the owner of the gateway) and channel. If the operation fails, an HTML error code is returned with an error message in output.

GET Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
cmd	URI	String	gws	Yes	Yes
lat	URI	Float	Center latitude, in decimal format	Yes	No
lon	URI	Float	Center longitude, in decimal format	Yes	No
h	URI	Float	Height, in km	Yes	No
w	URI	Float	Width, in km	Yes	No

Example request body

```
http://0.0.0.0:8080/access.controller/emulation?cmd=gws&lat=60.171888&lon=
24.950799&h=1&w=1
```

Example response body

```
{"Gateways":[{"ratType":"IEEE
802.15.4","longitude":24.948774,"impi":"generated","latitude":60.170097,"cha
nnel":"25"},{"ratType":"IEEE
802.15.4","longitude":24.956118,"impi":"generated","latitude":60.168793,"cha
nnel":"25"}]}
```

Emulate gateways for congestion demonstration

Verb	URI	Description
POST	/access.controller/emulation	Emulate gateways for congestion demonstration

Normal Response Code(s): 200

Error Response Code(s): badRequest (400), internalServerError (405)

Response Format: Plain Text

This interface emulates the given number of gateways in the given rectangle area. The rectangle area is centered at the given (*lat*, *lon*) point with the height of given *h* and the width of given *w*. The generated gateways use the given RAT (Radio Access Type) on the same channel. The output returns plain text with response code 200 if the operation succeeds. If the operation fails, an HTML error code is returned with an error message in output.

POST Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
cmd	Body	String	generate	Yes	Yes
lat	Body	Float	Center latitude, in decimal format	Yes	No
lon	Body	Float	Center longitude, in decimal format	Yes	No
h	Body	Float	Height, in km	Yes	No
w	Body	Float	Width, in km	Yes	No
gws	Body	Integer	Number of GWs to emulate	Yes	No
rats	Body	String	The RAT types the emulated GWs will use, currently accepted: xbee, wifi, zwave.	Yes	Yes

Example request body

```
cmd=generate&lat=60.171888&lon=24.950799&h=1&w=1&gws=100&rats=xbee
```

Example response body

```
Done!
```

Optimize the emulated gateways

Verb	URI	Description
GET	/access.controller/emulation?cmd=optimize	Optimize the emulated gateways

Normal Response Code(s): 200

Error Response Code(s): badRequest (400), internalServerError (405)

Response Format: Plain text

This interface sends a command to optimize the frequency channels of the generated gateways i.e. triggers the self-healing process. The output returns plain text with response code 200 if the operation succeeds. If the operation fails, an HTML error code is returned with an error message in output.

GET Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
cmd	URI	String	optimize	Yes	Yes

Example request body

```
http://0.0.0.0:8080/access.controller/emulate?cmd=optimize
```

Example response body

```
Done!
```

Clear the emulated gateways

Verb	URI	Description
GET	/access.controller/emulation?cmd=clear	Clear the emulated gateways

Normal Response Code(s): 200

Error Response Code(s): badRequest (400), internalServerError (405)

Response Format: Plain text

This interface clears the generated gateways from CNF. The output returns plain text with response code 200 if the operation succeeds. If the operation fails, an HTML error code is returned with an error message in output.

GET Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
cmd	URI	String	clear	Yes	Yes

Example request body

```
http://0.0.0.0:8080/access.controller/emulate?cmd=clear
```

Example response body

```
Done!
```

REST APIs Towards CNM

The APIs presented in this section are used by the management interface components (see Chapter 5) and the demo use case (see Chapter 5.3) for communication with the Capillary Network Manager (CNM).

Set GPS coordinates for a CGW or MD

Verb	URI	Description
GET	?cmd=set_coordinates&node=X&lat=Y&lng=Z or ?cmd=set_coordinates&device=X&lat=Y&lng=Z	Set GPS coordinates for a CGW or MD

Normal Response Code(s): 200

Error Response Code(s): badRequest (400), internalServerError (405)

Response Format: JSON

This interface sets the GPS coordinates of the given CGW or MD. For the CGW GPS update, parameter *node* is used while parameter *device* is used for MD GPS update. If the operation is successful, the output returns a JSON object with the new latitude and longitude values. If the operation fails, an HTML error code is returned with an error message in output.

GET Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
cmd	URI	String	set_coordinates	Yes	Yes
node	URI	String	CGW ID, if CGW is updated	No	Yes
device	URI	String	MD ID, if MD is updated	No	Yes
lat	URI	Float	Latitude, in decimal format	Yes	No
lng	URI	Float	Longitude, in decimal format	Yes	No

Example request body

```
http://0.0.0.0:8080?cmd=set_coordinates&node=5&lat=60.1662157&lng=24.932549

OR

http://0.0.0.0:8080?cmd=set_coordinates&device=280:e103:1:3829&lat=60.166215
7&lng=24.932549
```

Example response body

```
{"lat": "60.1662157", "lng": "24.932549"}
```

Set a configuration parameter

Verb	URI	Description
GET	?cmd=set_config	Set configuration parameter

Normal Response Code(s): 200

Error Response Code(s): badRequest (400), internalServerError (405)

Response Format: JSON

This interface sets one or more configuration parameters for the CN. Any parameter (singular or multiple) can be updated separately or together from the list below. If the operation is successful, the output returns a JSON object with all of the configuration parameters in a list i.e. if the request only has *cmd=set_config* parameter, it just returns all the configuration parameters and their values. If the operation fails, an HTML error code is returned with an error message in output.

GET Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
cmd	URI	String	set_config	Yes	Yes
timer_query_policy	URI	Float	Interval to query policy from CNF (default 10), in seconds	No	Yes
timer_query_constraints	URI	Float	Interval to query constraints from CNF (default 2), in seconds	No	Yes
timer_post_devices	URI	Float	Interval to post device list to CNF (default 0.1), in seconds	No	Yes
timer_force_post_devices	URI	Float	Interval to force an extra post of device list to CNF (default 60), in seconds	No	Yes
timer_expire_gw_granuality	URI	Float	Interval to check for expired gateway registrations (default 2), in seconds	No	Yes
timer_expire_gw	URI	Float	Expiration time for gateway registrations (default 30), in	No	Yes

Name	Style	Type	Description/Value	Required	Case-Sensitive
			seconds		
timer_expire_dev_granuality	URI	Float	Interval to check for expired gateway registrations (default 2), in seconds	No	Yes
timer_expire_dev	URI	Float	Expiration time for device registrations (default 30), in seconds	No	Yes
urlpolicy	URI	String	URL to obtain policy from CNF	No	Yes
urlconstraints	URI	String	URL to obtain constraints from CNF	No	Yes
urldevicelist	URI	String	URL to post device lists to CNF	No	Yes
address_order	URI	Boolean	Use address order instead of join order (default false)	No	Yes
port	URI	Integer	Port number of REST server (default 80)	No	Yes
connectivity_radius	URI	Float	Radio coverage radius for determining device connectivity based on location from gateway (0=disable, default 0.2), in km	No	Yes
coordinate_treshold	URI	Float	Minimum distance of change before updating coordinates (0=disable, default 0.005), in km	No	Yes
postcleaning	URI	Integer	Method to clean the device lists before posting to CNF	No	Yes

Name	Style	Type	Description/Value	Required	Case-Sensitive
			(default 2)		
debug	URI	String	Comma-separated list of topics to print debug information on	No	Yes

Example request body

```
http://0.0.0.0:8080? cmd=set_config&timer_post_devices=0.1&timer_force_post_
devices=60
```

Example response body

```
{"urldevicelist":
"http://[2a00:1d50:2:1001:f816:3eff:fea8:fc81]/leiot/process.php", "post-
cleaning": 2, "timer_expire_dev_granuality": 1, "timer_query_constraints":
2, "timer_expire_gw": 5, "timer_force_post_devices": 60, "tim-
er_expire_gw_granuality": 2, "urlpolicy":
"http://[2a00:1d50:2:1001:f816:3eff:fea8:fc81]: 8080/access.controller/m-
nf?cmd=getPolicy", "debug": "cnm,cgw,cnf", "urlconstraints":
"http://[2a00:1d50:2:1001:f816:3eff:fea8: fc81]:8080/access.controller/m-
nf", "port": 80, "timer_post_devices": 2.0, "forceipv4": false, "ad-
dress_order": false, "connectivity_radius": 999999999999.0, "coordi-
nate_treshold": 0.005, "timer_query_policy": 10, "timer_expire_dev": 5}
```

Get the list of CNM status values

Verb	URI	Description
GET	?cmd=get_status	Get the list of all status counters

Normal Response Code(s): 200

Error Response Code(s): badRequest (400), internalServerError (405)

Response Format: JSON

This interface gets the full list of status values of CNM. If the operation is successful, the output returns a JSON object in key-value format, where the key refers to status name (description) and the value refers to counter value. However, the status list returned depends on CNM implementation and available status values in CNM. If the operation fails, an HTML error code is returned with an error message in output.

GET Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
cmd	URI	String	get_status	Yes	Yes

Example request body

```
http://0.0.0.0:8080?cmd=get_status
```

Example response body

```
{ "connectivity": "1->280:e103:1:3829 3->280:e103:1:3829 ", "constraints":
"{"1": {"battery": 3, "load": 3}", "constraints changed": "Tue Jun 10
06:56:27 2014", "dev280:e103:1:3829 coord":
"21.001008804052326:52.17633998406954", "dev280:e103:1:384e coord":
"21.001008804052326:52.17633998406954", "gw1 activations": 1, "gw1 coord":
"21.001001:52.176614", "gw1 devs connectivity": "280:e103:1:3829", "gw1 devs
from gw": "280:e103:1:3829", "gw1 devs selection": "280:e103:1:3829", "gw1
devs to cnf": "280:e103:1:3829", "gw1 last update": "Wed Jun 11 14:26:33
2014", "gw1 status": "active",  "policy": "{"default": {"battery": 2,
"load": -2, "connections": -1}, }", "policy changed": "Tue Jun 10 06:56:27
2014", "received get_status": 23, "received set_config": 1, "received
set_coordinates": 255, "received set_devices": 1680, "received total": 1965,
"received unknown": 6, "selection": "280:e103:1:3829->GW1", "selection
changes": 1, "sent get constraints": 56455, "sent get policy": 11291, "sent
set_devices": 66, "sent total": 67812 }
```

REST APIs Towards IoT Framework

The APIs presented in this section are used by the management interface components (see Chapter 5) for communication with the IoT Framework. None of the APIs in this section includes any GET parameter since the data is retrieved using a path model.

Get the list of resources for a user

Verb	URI	Description
GET	/users/USER_ID/resources	Get the list of resources for a user

Normal Response Code(s): 200

Error Response Code(s): badRequest (400), internalServerError (405)

Response Format: JSON

This interface gets the list of resources, which belong to the given user ID in the path. If the operation is successful, the output is the list of resource entries. Each resource entry in the response consists of the values presented in Resource Parameters below. If the operation fails, an HTML error code is returned with an error message in output.

GET Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
-	-	-	-	-	-

Resource Parameters

Name	Type	Description
active	Boolean	Online status of the resource
creation_date	String	The date when the resource was created
description	String	Text description for the resource
id	String	Unique identifier for the resource
location	String	Latitude and longitude of the resource, in decimal format
manufacturer	String	Manufacturer information of the resource
model	String	Model information of the resource
name	String	Name of the resource
polling_freq	Integer	Polling frequency of the sensor from IoT Framework, in seconds
serial	Integer	Serial number of the resource
tags	String	Text tags for the resource, used in search
type	String	Type of the resource

Name	Type	Description
uri	String	URI of the resource if available
user_id	String	Unique user identifier of the resource, to which the resource belongs.

Example request body

```
http://0.0.0.0/users/PbN2Zf5UR2a7KFHK_WSMww/resources
```

Example response body

```
{"hits":[{"active":"true","creation_date":"2013-12-23","description":"CPU
load at home lap-
top","id":"BGEKy_27RHya6b2jCCnLKw","location":"60.165943,24.942425","manufac
turer":"Ericsson","model":"JAIME03","name":"Resource2","polling_freq":"30","
serial":"2300002","tags":"cpu, Helsinki, Simulation","type":"CPU","uri":"No
Uri","user_id":"PbN2Zf5UR2a7KFHK_WSMww"}]}
```

Get a single resource entry with the given ID

Verb	URI	Description
GET	/resources/RESOURCE_ID	Get a single resource entry with the given ID

Normal Response Code(s): 200

Error Response Code(s): badRequest (400), internalServerError (405)

Response Format: JSON

This interface gets a single resource entry, which corresponds to the given resource ID in the path. If the operation is successful, the output is one resource entry. If the operation fails, an HTML error code is returned with an error message in output.

GET Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
-	-	-	-	-	-

Example request body

```
http://0.0.0.0/resources/BGEKy_27RHya6b2jCCnLKw
```

Example response body

```
{"active":"true","creation_date":"2013-12-23","description":"CPU load at
home aptop","id":"BGEKy_27RHya6b2jCCnLKw","location":"60.165943,
24.942425","make":"23","manufacturer":"Ericsson","model":"MERT03","name":"Re
source2","polling_freq":"30","serial":"2300002","tags":"cpu, Helsinki, Simu-
lation","type":"CPU","uri":"No Uri","user_id":"PbN2Zf5UR2a7KFHK_WSMww"}
```

Get the list of streams for a user and a resource

Verb	URI	Description
GET	/users/USER_ID/resources/RESOURCE_ID/streams	Get the list of streams for a user and a resource

Normal Response Code(s): 200

Error Response Code(s): badRequest (400), internalServerError (405)

Response Format: JSON

This interface gets the list of streams, which belong to the given user ID and to the resource ID in the path. If the operation is successful, the output is the list of stream entries. Each stream entry in the response consists of the values presented in Stream Parameters below. If the operation fails, an HTML error code is returned with an error message in output.

GET Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
-	-	-	-	-	-

Stream Parameters

Name	Type	Description
creation_date	String	The date when the stream was created
id	String	Unique identifier for the stream

Name	Type	Description
location	String	Latitude and longitude of the stream, in decimal format
name	String	Name of the stream
type	String	Type of the stream
resource_id	String	Unique resource identifier of the stream, to which the stream belongs.
user_ranking	Float	User ranking value of the stream

Example request body

```
http://0.0.0.0/users/PbN2Zf5UR2a7KFHK_WSMww/resources/msWhHwijRrCQOVz0K6hi9w
/streams
```

Example response body

```
{"hits":[{"creation_date":"20140124T15","id":"7TZLf92hQ4GTe8ZENcCMag","locat
ion":"60.145843,24.942435","name":"ANMLY1","resource_id":"msWhHwijRrCQOVz0K6
hi9w","type":"Anomally1","user_ranking":5}]}
```

Get a single stream entry with the given ID

Verb	URI	Description
GET	/streams/STREAM_ID	Get a single stream entry with the given ID

Normal Response Code(s): 200

Error Response Code(s): badRequest (400), internalServerError (405)

Response Format: JSON

This interface gets a single stream entry, which corresponds to the given stream ID in the path. If the operation is successful, the output is one stream entry. If the operation fails, an HTML error code is returned with an error message in output.

GET Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
-	-	-	-	-	-

Example request body

```
http://0.0.0.0/streams/7TZLf92hQ4GTe8ZENcCMag
```

Example response body

```
{"creation_date":"20140124T15","id":"7TZLf92hQ4GTe8ZENcCMag","location":"60.145843
,24.942435","name":"ANMLY1","resource_id":"msWhHwijRrCQOVz0K6hi9w","type":"A
nomally1","user_ranking":5}
```

Get the list of datapoints for a stream

Verb	URI	Description
GET	/streams/STREAM_ID/data	Get the full list of datapoints for a stream

Normal Response Code(s): 200

Error Response Code(s): badRequest (400), internalServerError (405)

Response Format: JSON

This interface gets the full list of datapoints, which belong to the given stream ID. If the operation is successful, the output is the full list of datapoint entries (this list can be very large). Each datapoint entry in the response consists of the values presented in Datapoint Parameters below. If the operation fails, an HTML error code is returned with an error message in output.

GET Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
-	-	-	-	-	-

Datapoint Parameters

Name	Type	Description
id	String	Unique identifier for the datapoint
signature	String	Text to identify if the datapoint was transferred encrypted to the IoT Framework or not
streamid	String	Unique stream identifier of the datapoint, to which the datapoint belongs.
timestamp	String	Timestamp of the datapoint
value	String	Value of the datapoint

Example request body

```
http://0.0.0.0/streams/griJCeK-QpWAzp7eNjt9Kg/data
```

Example response body

```
{"hits":[{"id":"zY6ES_yTT9GKRU6rP2ZNyw","signature":"(Signed - Veri-
fied)","streamid":"griJCeK-
QpWAzp7eNjt9Kg","timestamp":"20131210T090635.0000","value":"27.83203125"},{"
id":"k035UU_0QtqE4zE-gdbETw","signature":"(Signed - Veri-
fied)","streamid":"griJCeK-
QpWAzp7eNjt9Kg","timestamp":"20131210T091441.0000","value":"30.76171875"}]}
```

Get the filtered list of datapoints for a stream

Verb	URI	Description
GET	/streams/STREAM_ID/data/_search?timeStampFrom=X&timeStampTo=Y	Get the filtered list of datapoints for a stream

Normal Response Code(s): 200

Error Response Code(s): badRequest (400), internalServerError (405)

Response Format: JSON

This interface gets the filtered list of datapoints, which belong to the given stream ID. Datapoints are filtered according to the given dates and times i.e. from *timeStampFrom* to *timeStampTo*. If the operation is successful, the output is the filtered list of datapoint entries (this list can be very large). If the operation fails, an HTML error code is returned with an error message in output.

GET Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
timeStampFrom	URI	String	Starting timestamp value, in the format of YYYYMMDDTHHMMSS.0000	Yes	Yes
timeStampTo	URI	String	Ending timestamp value, in the format of YYYYMMDDTHHMMSS.0000	Yes	Yes

Example request body

```
http://0.0.0.0/streams/griJCeK-
QpWAzp7eNjt9Kg/data/_search?timeStampFrom=20131217T160120.0000&timeStampTo=2
0131217T160321.0000
```

Example response body

```
{"hits":[{"id":"jTCwpVDMQBqM2BF-AiayrA","signature":"(Signed - Veri-
fied)","streamid":"griJCeK-
QpWAzp7eNjt9Kg","timestamp":"20131217T160120.0000","value":"25.390625"},{"id
":"22F5960gTT-KSR3I7frl-g","signature":"(Signed - Veri-
fied)","streamid":"griJCeK-
QpWAzp7eNjt9Kg","timestamp":"20131217T160321.0000","value":"25.390625"}]}
```

Search the IoT Framework with a query

Verb	URI	Description
POST	/_search	Search the IoT Framework with a query

Normal Response Code(s): 200

Error Response Code(s): badRequest (400), internalServerError (405)

Response Format: JSON

This interface executes a search in the IoT Framework with the given search query. If the operation is successful, the output is a list of search results. If the operation fails, an HTML error code is returned with an error message in output.

POST Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
query	Body	String	The string to search for	Yes	Yes

Example request body

```
{ "query" : {"query_string" : {"query" : "Jorvas"}}}
```

Example response body

```
{"streams":{"took":4,"timed_out":false,"_shards":{"total":5,"successful":5,"
failed":0},"hits":{"total":0,"max_score":null,"hits":[]}}, "us-
ers":{"took":2,"timed_out":false,"_shards":{"total":5,"successful":5,"failed
":0},"hits":{"total":0,"max_score":null,"hits":[]}}}
```

REST APIs towards vGW

The APIs presented in this section are used by the management interface components (see Chapter 5) for communication with vGW instances.

Get a list of CGWs under the vGW

Verb	URI	Description
GET	/dh/getGateways.php	Get a list of CGWs under the vGW

Normal Response Code(s): 200

Error Response Code(s): badRequest (400), internalServerError (405)

Response Format: JSON

This interface gets a list of available CGWs under the vGW. If the operation is successful, the output is a list of CGW entries. If the operation fails, an HTML error code is returned with an error message in output.

GET Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
-	-	-	-	-	-

Example request body

```
http://0.0.0.0/dh/getGateways.php
```

Example response body

```
{"Gateways":[{"id":"1","address":"2001:14b8:100:354::2","lastseen":"Mon Aug
18 15:39:03 EEST 2014","latitude":"59.40442269","longitude":"17.95359135",
"imsi":"116101114111641","dev_type":"cg","meta":"hardware: Buffalo WZR-HP-
AG300H; operating system: OpenWRT AA; software: Ericsson Research Capillary
Gateway; short range radio: 802.15.4; wide range radio: 4G; description:
Capillary Gateway 1","userId":"4WlKN3g-TJGA5bzP-OQc6g"}]}
```

Get a list of MDs under the vGW

Verb	URI	Description
GET	/dh/ getDevices.php?dev=X	Get a list of MDs under the vGW

Normal Response Code(s): 200

Error Response Code(s): badRequest (400), internalServerError (405)

Response Format: JSON

This interface gets a list of available MDs under the vGW. If the operation is successful, the output is a list of MD entries. If the operation fails, an HTML error code is returned with an error message in output.

GET Parameters

Name	Style	Type	Description/Value	Required	Case-Sensitive
dev	URI	String	MD ID	No	Yes

Example request body

```
http://0.0.0.0/dh/getDevices.php?dev=280:e103:1:3829
```

Example response body

```
{"Devic-
es":[{"gw":"1","id":"280:e103:1:3829","service":"sensor","interface":"Temper
ature0","latitude":"59.40420426","longitude":"17.95348406","meta":"brand:
STMicroelectronics; model: MBxxx (STM32W108CC); mac: 00-80-e1-03-00-01-38-
29; short range radio: 802.15.4; signing: false; description: Contiki Sensor
TEMP; color: Blue; unit: C","resourceId":"gD5QM-
EHRlm0U4DJgKyfdg","streamId":"KoMuC6etQjupvOpuWPMHOg"}]}
```

REST APIs towards Pub/Sub Server in Capillary Networks Cloud

The APIs presented in this section are used by the management interface components (see Chapter 5) for communication with the Pub/Sub server in the Capillary Networks cloud. As the communication with the Pub/Sub server is not done over HTTP but rather in websockets, the information is presented different in this section.

Subscribe for network topology update

Protocol	Subscription Channel	Description
Websockets	/messages	Update on the network topology

This interface is the subscription channel to the Pub/Sub server in the CN cloud for network topology update. When there is any update on the network topology (new MD, MD changes CGW etc), the CN Pub/Sub server publishes on this channel. The published message is not the new list of devices but rather just a warning from the CN cloud.

Example publish content

```
“update”
```

Subscribe for notifications

Protocol	Subscription Channel	Description
Websockets	/notifications	New notification

This interface is the subscription channel to the Pub/Sub server in the CN cloud for new notification.

When there is any new notification from CGWs, the CN Pub/Sub server publishes the new content on this channel.

Example publish content

```
“{ "id":"63098", "node":"1", "entry":"2014-04-23 07:44:19 GW 1 connected to
the network" }”
```

Subscribe for logs

Protocol	Subscription Channel	Description
Websockets	/logs	New log

This interface is the subscription channel to the Pub/Sub server in the CN cloud for new logs. When there is any new log from CGWs, the CN Pub/Sub server publishes the new content on this channel.

Example publish content

```
“{ "id":"1694774", "level":"4", "entry":"2014-04-23 07:36:40 WARNING: GW 1
disconnected from the network (failed 3 times)" }”
```

Subscribe for network counters

Protocol	Subscription Channel	Description
Websockets	/counters	New network counters

This interface is the subscription channel to the Pub/Sub server in the CN cloud for new network counters. When there is any new counter value from CGWs, the CN Pub/Sub server publishes the new content (the entire counter list) on this channel.

Example publish content

```
“{"constraints from":"configuration file","constraints in
use":"{\"1\":{\"battery\":5,\"load\":2},\"3\":{\"battery\":3,\"load\":1},
,\"2\":{\"battery\":4,\"load\":2}}","devices":"0","devices de-
nied":"0","devices offline":"0","devices right gw":"0" ,"devices wrong
gw":"0", "master":"True", "network access failed":"94 times","network access
failure rate":"55.294117647059 % (mean)"}”
```