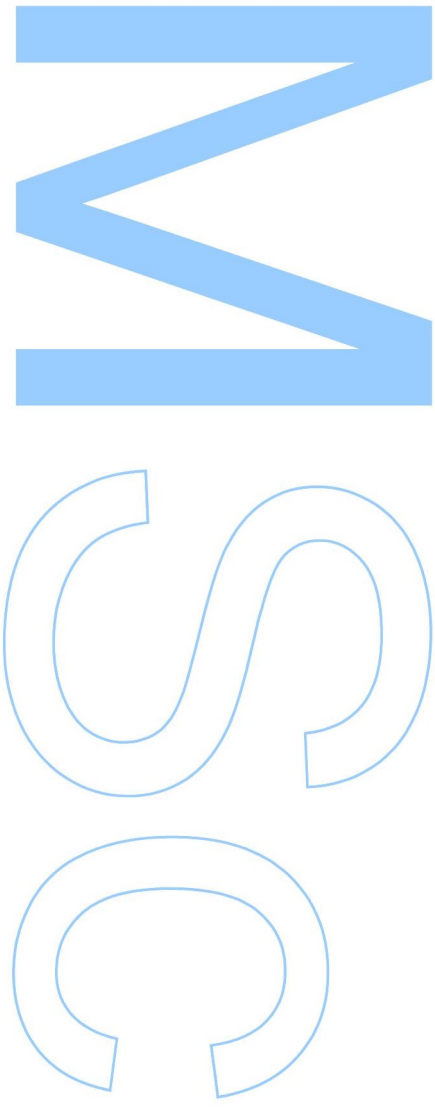
M2M Interoperability

António Filipe Carvalho Pinto
Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2016

Orientador
Ana Cristina Costa Aguiar, Professor Auxiliar, FEUP

Coorientador
Rui Pedro de Magalhães Claro Prior, Professor Auxiliar, FCUP



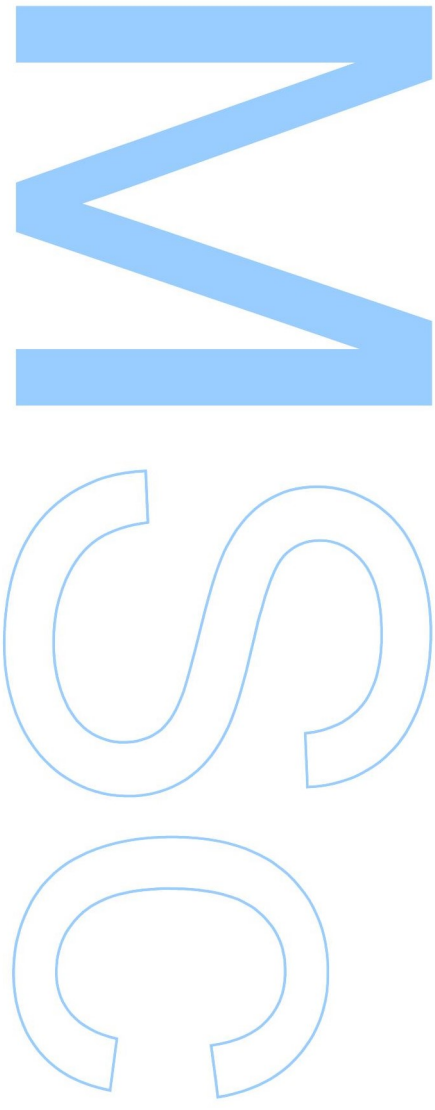
U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____



António Filipe Carvalho Pinto

M2M Interoperability

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
May 2016

António Filipe Carvalho Pinto

M2M Interoperability

Tese submetida à Faculdade de Ciências da
Universidade do Porto para obtenção do grau de Mestre
em Engenharia de Redes e Sistemas Informáticos

Orientador: Ana Cristina Costa Aguiar
Co-orientador: Rui Pedro de Magalhções Claro Prior

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
May 2016

Acknowledgments

First of all I want to express my gratitude to my supervisor Dra. Ana Aguiar for her patience, advises, and sharing of knowledge. I'm also very grateful to my co-supervisor Dr. Rui Prior and to Carlos Pereira for being always available to help me. I'm thankful as well to PT Inovação, that offered me the opportunity of this internship where I could practice and acquire experience in new technologies. I also want to thank my parents and my girlfriend for their advice and for always believing in me.

Abstract

Machine-to-Machine (M2M) communications are gradually becoming more important with the today's emergence of the Internet of Things (IoT). This type of communication allows the interaction between services and devices without human intervention. In this dissertation, we developed several M2M entities compliant with the ETSI M2M standard. The ETSI M2M standard provides detailed documents for the implementation of M2M communications and allows for interoperability between M2M implementations from different providers. In this dissertation, we analyse and improve the performance of an existing M2M service, developed for Android OS, which allows a smartphone to act as an M2M Gateway (GW). We also develop a Java library to ease the deployment of M2M Network Applications (NAs) and an Android library to help deploy M2M Gateway Applications (GAs). Finally, we present the results in an M2M mobile pilot to prove the robustness of the implementations and the simplicity of deployment and use of M2M ecosystems.

Resumo

As comunicações Máquina-a-Máquina (M2M) estão a tornar-se gradualmente mais importantes com a emergência da Internet of Things (IoT). Este tipo de comunicações permite a interação entre serviços e dispositivos sem intervenção humana. Nesta dissertação, desenvolvemos várias entidades M2M de acordo com o standard ETSI M2M. O ETSI M2M Standard disponibiliza documentação detalhada para implementação de comunicações M2M e permite interoperabilidade entre implementações M2M de diferentes fornecedores. Ao longo desta dissertação, analisamos e melhoramos a prestação de um serviço M2M existente, desenvolvido para o Android OS, que permite que um smartphone atue como uma M2M Gateway (GW). Também desenvolvemos uma biblioteca Java para facilitar o lançamento de M2M Network Applications (NA) e uma biblioteca Android para auxiliar a criação de M2M Gateway Applications (GA). Finalmente, apresentamos os resultados de um piloto M2M móvel para provar a robustez das implementações e a facilidade de criação e uso de ecossistemas M2M.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the recent years the number of networked devices has grown significantly and the number electronic devices connected to the Internet is bound to surpass the number of humans. Furthermore, besides computers, everyday objects will contain sensory capabilities, network conectivity and electronic components. This allows them to comunicate and exchange data with each other creating an interconnected network called Internet Of Things(IoT). As perspective, Gartner[12] predicts the existence of 6.4 billion connected things in 2016 and 20.8 billion by 2020 [11]. Cisco[1] estimates 11.6 billion connected devices and 30.6 exabytes of mobile data traffic monthly, in 2020[3].

The Internet Of Things expects the automation of communication between capable objects, replacing the requirement for Human-to-Human (H2H) or Human-to-Machine (H2M) communications, with Machine-to-Machine (M2M) communications. M2M communications allows for better scalability, efficiency and reduced expenses, since machines manage and cooperate with each other with minimal or no human intervention.

The major challenge is that the wide variety of devices and applications from different developers and companies have to communicate each other. They must be able to understand the protocols and communication language used by one another. This is specially problematic considering the large amount of proprietary protocols that can be created.

In order to enable the devices to communicate with each other seamlessly in an M2M environment, a great amount of efforts has been directed to the standardization of

[1] `http://www.cisco.com/`

M2M protocols. The efforts culminated in the M2M standard released by European Telecommunications Standards Institute (ETSI) [2], more recently partnered with OneM2M[3].

1.1 Motivation

The IoT has great potential to change daily life and improve life's quality. Small sensors can capture information, that otherwise would be undetected by an human being, anywhere at anytime and act autonomously to notify the user or a managing system. For example, alert the authorities about an house theft when the owner is away.

In order to interconnect various sensors, devices and services, the M2M communications will play an important role in providing interoperability. Nowadays, most devices are not inherently M2M capable. This can be a problem for the integration of new services, since we need specialized M2M capable devices. Another issue is the autonomy of smaller devices, which must use technologies with shorter communication range and be efficient in processing and memory usage.

One of the most promising use cases of M2M Communications lies in their integration in the smartphone market. Currently, smartphones are widely used. Most of them are equipped with a vast range of internal sensors and are capable of communicating with a large set of external devices through different communication technologies like Bluetooth, Wi-Fi, 3G and 4G. Smartphones are also very flexible, providing APIs for the development and deployment of new applications and services. Users can download and install all kinds of tools and services through online stores. Having such flexible system, it becomes much easier to add M2M capable applications. Smartphones' computing, communication and sensory capabilities keep growing. Being so, they represent a great opportunity to act as a Gateway for smaller devices with limited connection possibilities.

However, smartphones typically belong to private users who wish to have control over their devices. As such, a successful integration of M2M communications in the smartphone market depends on first obtaining their acceptance. Users' major concerns consist in the conservation of battery life, security and control over the device activities.

[2] http://www.etsi.org/
[3] http://www.onem2m.org/

In this dissertation we address M2M networks and interoperability possibilities, by improving and adding features to an existing M2M Gateway (GW) and developing other M2M entities, like the M2M Network Application (NA) and the M2M Gateway Application (GA).

1.2 Objectives

The work for this dissertation was developed in the scope of a project to develop an Mobile M2M GW, in collaboration with Instituto de Telecomunicações Porto[4] and Portugal Telecom Inovação[5]. Previously, an M2M GW and an M2M NA prototypes were developed for the Android OS [20]. This dissertation had the following objectives:

- Study the ETSI M2M standard and its implementation;
- Analyse the existing M2M software and its performance;
- Apply improvements and new features to the M2M software;
- Develop support for deployment of M2M nodes;
- Apply the software developed to simple real world cases.

1.3 Structure

This dissertation is structured as follows. In Chapter 2 we analyse the M2M technology. We make an overview of the most important aspects of the ETSI M2M standard, analyse the uses cases provided by ETSI to contextualize the practical uses and requirements of M2M networks. We also take a look at the existing implementations of the standard, their strategies and capabilities.

In Chapter 3 we make a top down overview of the developed systems and their interactions. We first exemplify an M2M ecosystem that can be created using the different M2M entities presented. Then we take a look at the main components of the developed systems, their purpose and interactions.

[4] `https://it.pt/ITSites/Index/5`
[5] https://telecom.pt/pt-pt/inovacao/Paginas/inovacao.aspx

In Chapter 4 we analyse the existing M2M GW that will be used as base for the development in this dissertation. We make an overview of the implementation choices and structure of the M2M GW. We also present an experiment to analyse and pinpoint major performance problems.

In Chapter 5 we present possibilities, problems and solutions found during the development phase. We go through the technical details of the improvements and features added to the existing software, as well as the process of development of the libraries.

In Chapter 6 we present a pilot deployed to demonstrate the functionality of the software developed in a real world scenario.

Finality, in Chapter 7 we identify the contributions and present the main conclusions of this work.

Chapter 2

State of Art

2.1 ETSI M2M Standard

In this section we will make an overview of the ETSI M2M standard, based on the functional architecture documentation[10].

2.1.1 High-Level Architecture

In a simplified manner, an M2M network is composed by M2M Service Capability Layers (SCL), M2M Applications and M2M Interworking Proxies (xIP). The SCLs provide M2M functions, exposed by a set of interfaces, which are used by M2M Applications to implement service logic. The xIPs are optional modules used to communicate with Non-ETSI M2M compliant entities and they can be an internal capability of an SCL or an M2M Application communicating with the SCL.

As can be seen in the Figure 2.1, the M2M entities can belong to two domains: Network Domain or Device/Gateway Domain. The Network Domain provides IP connectivity, interconnection with other networks and network services to the Device/Gateway Domain. It contains Network SCLs (NSCL), NAs and Network Interworking Proxies (NIP). The Device/Gateway Domain contains M2M Devices and GWs and provides the connectivity between them.The purpose of both M2M Devices and GWs is to run M2M Applications using an SCL. The M2M GWs also act as a network proxy between M2M Devices and the Network Domain. On the Device/Gateway Domain the following deployment cases are possible:

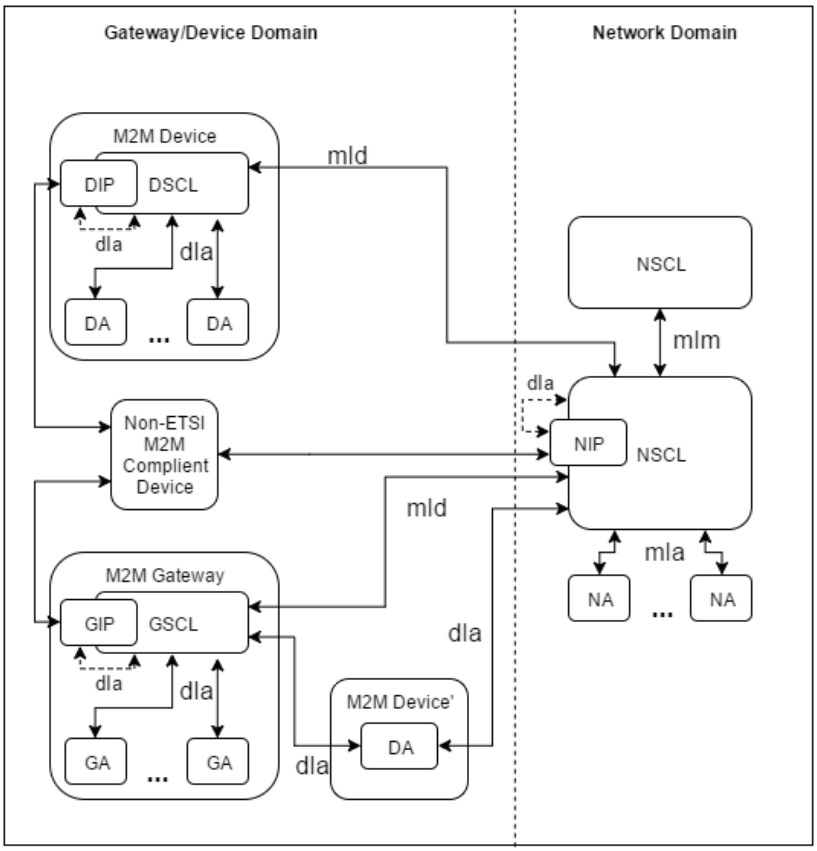


Figure 2.1: ETSI M2M standard high-level architecture diagram.

as a network proxy between M2M devices in a local network and the network domain.

- M2M Gateway
  An M2M GW contains a Gateway SCL (GSCL), GAs and optionally a Gateway Interworking Proxy (GIP).

- M2M Device
  An M2M Device contains a Device SCL (DSCL), Device Applications (DA) and optionally a Device Interworking Proxy (DIP).

- M2M Device'
  An M2M Device' is a simplified version of the M2M Device that only contains DAs. A Device' may use a M2M GW as Network Proxy to access the Network Domain or connect directly to the Network Domain.

- Non-ETSI M2M Compliant Device
  A Non-ETSI M2M Compliant Device does not have M2M capabilities of any kind

and is not considered an M2M entity; however, it can use the xIPs to connect to an SCL.

In order to establish communication between the M2M entities, they should provide one or more Reference Points. Reference Points are interfaces used to execute M2M operations. The Reference Points can be classified as:

- **mIa**
  Reference Point between the NA and NSCL that supports procedures like registration of the NA to the NSCL, Read/Write requests, Management Actions, Subscriptions and Notifications.

- **dIa**
  Reference Point between the GA/DA and the GSCL/DSCL/NSCL that enables mechanisms like registration of the DA to the DSCL, registration of the DA or GA to the GSCL, registration of the DA to the NSCL, Read/Write requests, Subscriptions and Notifications.

- **mId**
  Reference Point between SCLs that supports procedures like registration of the DSCL or GSCL to the NSCL, Read/Write requests, Management Actions, Subscriptions, Notifications and Security features.

- **mIm**
  Reference Point between NSCLs that supports procedures, across two different M2M Service Provider domains, like Read/Write requests Subscriptions and Notifications.

2.1.2 M2M Resource Structure

The exchange of information between M2M entities is made using M2M resources. The M2M resources are specified in the standard as the representation of M2M data that resides in an SCL and are structured as an hierarchical tree with parent-child relationships. The M2M data in each resource is stored in resource-specific attributes.

Since the M2M resources and their attributes are quite extensive, in the following lines we will only detail the most important ones, as depicted in Figure 2.2:

- **SclBase** resource is the root of the resource tree and contains all other M2M resources of the current SCL.
- **Scl** resource represents a remote SCL that is authorized to interact with the current SCL. This resource is created after a successful registration of the remote SCL on the host. It also contains a Point of Contact (PoC) attribute specifying the URI needed to communicate with the SCL.
- **Application** resource stores the information of an M2M Application registered on the current SCL.
- **Container** resource is used to exchange data between applications and SCLs, acting as a data buffer which eliminates the need for a direct connection between two entities.
- **ContentInstance** resource represents the data inside a container. The content of the data is considered opaque to the system, meaning it may be an image, text, encrypted message, etc.
- **Subscription** resource represents a request of an entity to be notified about modifications on the subscription's parent resource. It may also be used as a timed trigger of actions through the notifications sent in an expiration time event.
- **Collection** resource is an abstract representation of a set of resources of the same type. For example, a set of Application resources is stored under an Applications resource. Other examples of Collection resources are the following: SCLS resource, Containers resource, Subscriptions resource.
- **AccessRight** resource is a permission's representation associated to a resource. It is accessible to entities external to the current SCL, in order to control "who" can do "what" and secure privacy.

The set of attributes of each resource describe several characteristics of the resource. The most common attributes are:

- **accessRightID** contains the URI of an AccessRights resource that describes who is allowed to access the current resource;
- **creationTime** contains a timestamp specifying the creation of the current resource;

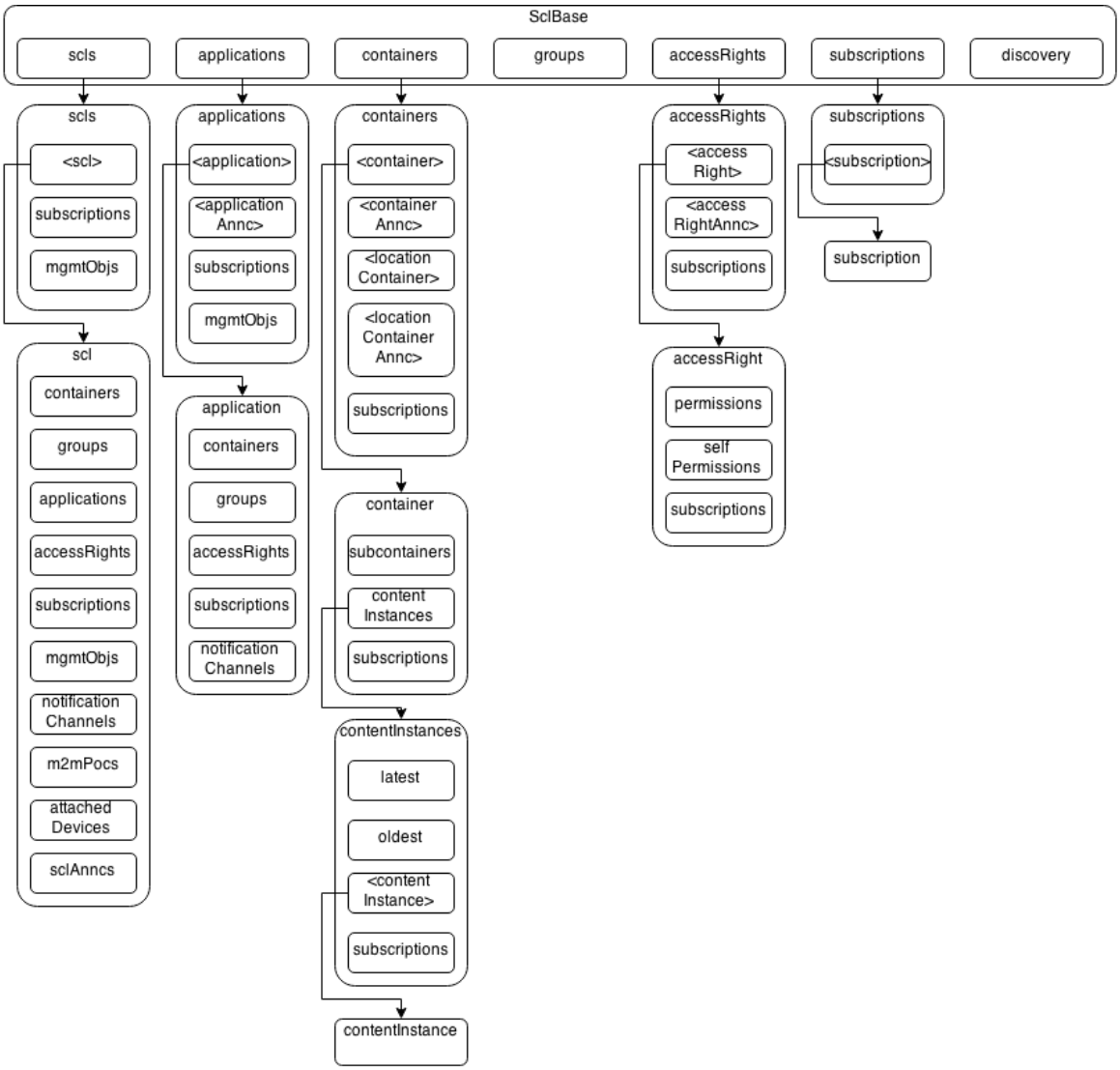


Figure 2.2: ETSI M2M standard resource structure.

- **expirationTime** contains a timestamp specifying when the current resource will be deleted;

- **lastModifiedTime** contains a timestamp specifying the last time the current resource was modified.

2.1.3 M2M Communications

In order to exchange resources between M2M entities, the ETSI M2M standard specifies procedures based on a RESTful architecture style. In this manner, the M2M

Applications and/or M2M SCL exchange representations of uniquely addressable resources that reside on a SCL. To be uniquely addressable, each resource has a identifier that is different from all sibling resources.

The RESTful architecture has a well know set of basic methods which act on resources, referred as CRUD methods: CREATE, RETRIEVE, UPDATE, DELETE.

- **CREATE** method creates a resource;
- **RETRIEVE** method reads a resource's content;
- **UPDATE** method writes the resource's content;
- **DELETE** method deletes a resource.

Additionally, the standard also specifies two extra methods, NOTIFY and EXECUTE. The first one is used notify any changes on a resource and it's mapped to RESTful architecture as an UPDATE method. The second one is used to execute a management command and it's also mapped to a UPDATE method, however the request doesn't contain a payload. To exchange requests and responses, the communication protocols supported by the ETSI M2M standard are RESTful, based on Hypertext Transfer Protocol (HTTP)[14] or Constrained Application Protocol (CoAP)[15].

In order to specify which resource is to be target, the request's URI should present the resource's path. For example, to target the Application *ApplicationID* inside the SCL *SclID* on host *host* by HTTP, the URI used for the request should be https://*host*/*SclID*/*ApplicationID*. As for the CRUD methods, they are transformed into HTTP or CoAP methods by mapping a CREATE to POST, a RETRIEVE to GET, a UPDATE to PUT. The result of the request can be the response or an acknowledgement of the request. The latter is used to implement asynchronous messaging by responding with the sent request as a result, and later, when available, send the requested data.

One important feature of M2M communications is the use of the publish-subscribe paradigm. The traditional model for communications is request-response, where an entity requests data and another entity responds with the data. This model makes sense in a H2M scenario, where data is retrieved sporadically and, in most cases, only once. However, in an M2M scenario data flows much more continuously, and is constantly updated. An interested entity would have to constantly request updates

in order to have the most recent data. This would require constant requests and responses that, potentially, would not even contain new data.

The publish-subscribe pattern allows an entity to make one request subscribing a publication from another entity. After a subscription is made, following updates to the publication will be notified to the subscriber. In an environment with thousands of devices, like an M2M scenario, this is a much more scalable communication model.

2.2 M2M Use Cases

The ETSI Fundation provides documentation for use cases of M2M Applications in eHealth[7], Connected Consumers[9], Automotive[8] and Smart Metering[5]. We will make an overview of this documentation, since they clarify some the practical uses and requirements of the M2M entities.

In eHealth, we can see several examples of eHealth M2M Applications like disease management, elderly monitoring and personal fitness. These cases assume the use of wearable sensors that capture patient data. Since this type of sensors usually use communications technologies with short range due to performance and battery limitations, the collected data must be aggregated and acted upon by another more capable device. In this instance the user should have a device acting as a M2M GW. The M2M GW may be fixed, like a PC, or mobile, like a cell phone.

In an eHealth scenario the exchange of data will most likely occur between a patient and a health care entity, like a clinician or an automated system of Electronic Health Record (EHR). Considering the medical nature of the data, the major concerns are the privacy, security and reliability of the transmitted data. In cases of disease management like hearth monitoring, delays in data forwarding may be life threatening. Other requirements of eHealth M2M Applications include handling M2M traffic with different priorities, real-time streaming communication and mobile connection.

The use cases presented for Connected Consumers include content sharing and device management and monitoring. Here we find cases where the M2M communications are on demand and do not need to send a continuous flow of critical data. The Connected Consumers documentation describes uses for M2M capable cameras that automatically upload photos to subscribed social networks or photoframes, download of ebooks from different providers by a M2M capable eBook reader and management of home appliances, inventory and surveillance data. In this scenarios the major

requirements are M2M management capabilities and different delivery mechanisms like periodical, scheduled and on-demand.

The automotive and transportation M2M applications involve stolen vehicle tracking, vehicle communications and electric vehicle charging. These applications can provide safety and security by remotely tracking and diagnosing vehicles. For example, the vehicle owner can receive notifications about the vehicle state and location. In order to do this, the vehicle's Telematic Control Unit (TCU) may act as a M2M GW for various in-built vehicle diagnostic sensors. The major requirements presented for automotive and transportation M2M applications are reliability and responsiveness, scheduled measurement delivery mechanisms and mobile communications while moving at high velocity.

The Smart Metering use cases have the main goal of enabling interoperability between utility meters. These meters aim to inform the consumers about their energy usage and provide monitoring tools to energy suppliers and distributors. In the scenario presented in the documents, various Smart Metering devices are connected to a device that acts as M2M GW communicating with a data center.

2.3 M2M Interoperability

Interoperability is arguably the main goal of the ETSI M2M standards. Previous M2M solutions existed before standardization, however, since these were proprietary and required specific hardware or software, they imposed limitations for the deployment of large networks with unique interconnecting devices. Thus, the ETSI M2M standard was created to unify M2M solutions so they can be seamlessly deployed and interconnected.

The ETSI foundation hosts an annual M2M Workshop where new M2M developments are presented. In conjunction with the workshop, there is an Interoperability Demonstration event, in which implementations of the ETSI M2M standard from different vendors test their mutual compatibility.

The event covers scenarios like eHealth, Intelligent Transportat System, Smart Metering, Home Automation, Entertainment, etc. It also includes integration with other tecnologies, like cloud, and presentation of SDK and high level APIs.

2.3.1 M2M Implementations

There are already M2M solutions based on the ETSI standard, provided by different companies. In this section we will overview the methods used by the most prominent companies, for the implementation and deployment of M2M systems.

2.3.1.1 Cocoon Project

The Cocoon[1] project by Open@Actility[2] is probably the most recurrent M2M solution in searches and references. This project provides the Object Network Gateway (ONG), which is an Open-Source implementation in JAVA/OSGi of the M2M GW.

The Open@Actility approach to the deployment of M2M GWs is to provide a jar file that executes the installation of the ONG on a remote system. The user must own a Host System that executes the file and a target system in which the ONG is installed. The target system must have a Linux OS, an SSH Server, ash or bash, DNS name Resolution and at least 12MB disk space. The management and configuration of the software is made by console commands provided by the ONG system.

In terms of M2M capabilities, the ONG provides an implementation of the M2M GSCL, execution of local GAs as OSGi Bundles, execution of NAs over the mId reference Point and drivers that implement GIPs for ZigBee and IPv6 over Low power Wireless Personal Area Networks(6LoWPAN) devices. The M2M resources are marshaled in XML format and exchanged through HTTP with RESTful operations.

2.3.1.2 OM2M Project

The OM2M[3] project is another emergent M2M solution based on the ETSI M2M standard, brought to us by The Eclipse Foundation[4]. It provides an Open Source Platform that can used to deploy NSCLs and GSCLs. Since it is composed by modules running on top of a OSGi layer, it is also extensible by plugins.

The OM2M can be obtained by downloading the compiled versions or by cloning and building the source code. The NSCL and GSCL can be configured by editing

[1] http://cocoon.actility.com/
[2] http://open.actility.com/
[3] http://www.eclipse.org/om2m/
[4] http://www.eclipse.org/

configuration files prior to their deployment. An OSGi console and a browser interface are also provided for managing plugins and visualizing data.

The M2M capabilities provided are the implementations of the NSCL and GSCL, execution GAs as plugins, various GIPs and NIPs for communication with vendor-specific technologies. The M2M resources are marshaled in XML format and exchanged through HTTP or CoAP with RESTful operations.

2.3.1.3 OpenMTC Plataform

The OpenMTC[5] is a M2M ETSI compliant middleware, used to integrate various devices and applications from different vertical domains with a standardized M2M open platform to aggregate, forward and manage collected data. The software contains the GSCL and NSCL components witch are extensible by OpenMTC's plugins.

This platform provides several communication protocols like HTTP, CoAP, Websocket, ZigBee, FS20 and Bluetooth. It also supports various hardware platforms like Android, Arduino and Raspberry Pi. The M2M resources can be marshaled in XML and JSON formats.

2.3.2 M2M Service APIs

When developing M2M services or applications based in the ETSI M2M standards, one concern is the necessity for a deep knowledge of the standards. Even though they facilitate the interoperability between different M2M service providers, their complexity and extension can turn away adoption of this technology by new developers. To counter this issue we can use APIs to create a layer of abstraction in the M2M interations.

For example, Asma Elmangoush et al. [4] propose a set of API's to interact with their M2M Communications Plataform, OpenMTC. The APIs proposed are divided in:

- **Network API**

  Covers CRUD operations to manage NAs. It has mostly operations to manage M2M Applications.

- **Device API**

[5] `http://www.openmtc.org/`

Covers CRUD operations to discover resources of devices and M2M GWs. It includes operations like application discovery, resource subscribing and group resources.

- **Data API**

  Covers CRUD operations to manage data. It has mostly operations to manage Containers.

In this case they prove the usefulness of the APIs by integrating a service broker with the M2M ecosystem.

Dmitry Namiot et al. [18] also propose an M2M Open API based on Web Intents. Web Intents are used to facilitate data and action requests interchange between a Web Client and a Web Service. Intents are also used in Android for a simplified inter-process communication. The authors state that it allows for a easy and extensible M2M Service design and propose the creation of a JavaScript Client Side API or M2M Browser applications.

2.4 Smartphones in an M2M scenario

So far we have seen a summary of the ETSI M2M standard. However, how does the standard applies to the mobile smartphone scenario? A smartphone by itself is Non-M2M ETSI complaint device but it does provides means to adapt to the M2M requirements.

Considering the deployment cases presented in the standard, for each case the smartphone should have the following capabilities:

- M2M Device'
  The smartphone runs one or more Android applications that act as M2M Device Applications by communicating with an NSCl or GSCL through the dIa reference point. This is the simplest implementation of M2M capabilities in a smartphone.

- M2M Device
  The smartphone runs one or more M2M Applications, like the M2M Device', and must also implement a local SCL. The local SCL must expose the mId and dIa reference points to communicate with SCL and M2M Applications, respectively.

Having a local SCL allows the smartphone to manage its own resources without constantly contacting a remote SCL. Since a mobile device has limited energy and, in most cases, limited network traffic, a local SCL improves the device's performance.

- M2M Gateway
  The M2M GW is very similar to the M2M Device. The smartphone runs M2M Applications, has a local SCL and exposes mId and dIa reference points. However, other M2M Devices can use the smartphone as a proxy to communicate with the NSCL.

As seen in the use cases provided by ETSI, the envisioned role for a smartphone is to act as a M2M GW. Since the smartphone can connect to a large set of devices, it is a capable mobile M2M GW. The major downside is its constrained resources since managing many M2M operations can impact the performance of the device.

Chapter 3

Support for M2M Interoperability

The mobile Gateway allows for integration of smartphone services and applications in an M2M environment. It is important to implement a feasible mobile Gateway, that can be accepted by a large variety of users who will use it in their devices.

The validation of the implementation requires the deployment of a pilot M2M environment containing the GSCL, NSCL, NAs and GAs. Considering the extent and complexity of the ETSI M2M standard and the different interpretations it can have in different scenarios, support tools to simplify the addition of other systems in an M2M Network also needs to be created.

In this chapter we give an overview of the different software components developed in their final state. The development details will be presented later at Chapter 5.

3.1 M2M Ecosystem Overview

We aim to create an M2M ecosystem similar to the one represented in the Figure 3.1, which shows a small M2M network composed by several M2M SCLs and Applications.

The NSCL provides the mId and mIa standardized interfaces. The mId is used to communicate with the two GSCLs and the mIa is used to communicate with the two NAs. Each GSCL is hosted in a different Android device and exposes the mId and dIa interfaces. The GSCL is managed by an Android service, called GW Service.

On the endpoints we can see the GAs and NAs. These components are likely to be created by different developers. Since the ETSI M2M standard is extensive and

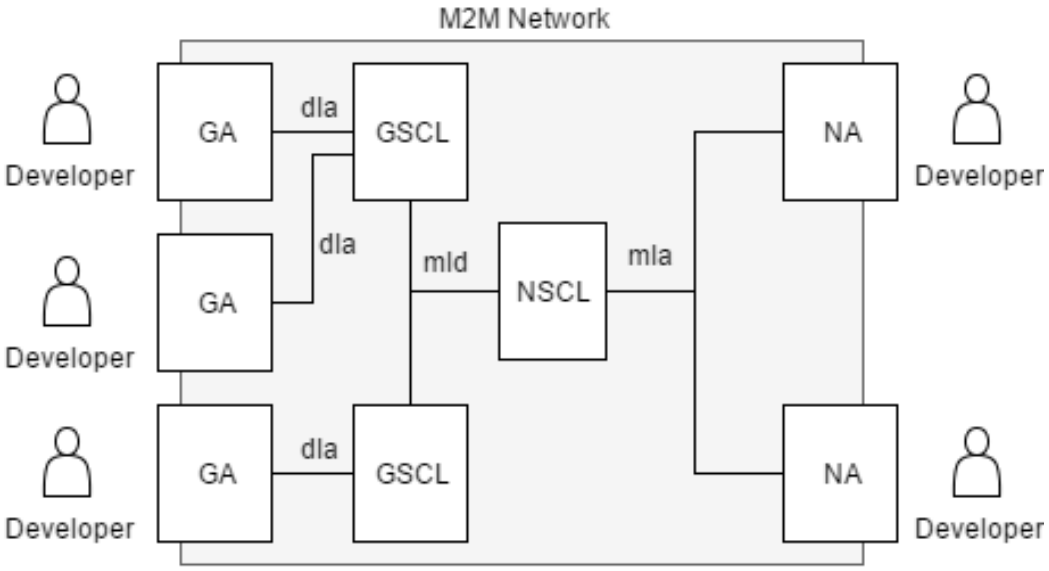


Figure 3.1: ETSI M2M high-level network architecture

complex, integrating M2M technologies in existing services can result in a large effort. To counter this, we wish to make the M2M network a black box that does not concern the developer by creating two Libraries: the GA Library (GALib) and NA Library (NALib). These libraries will hide the details of the ETSI standard behind a simpler API that provides methods for registration, publish-subscribe, resource management and communication with the hosting SCL.

The GALib aims to ease the creation M2M GAs on Android devices and is used to communicate with the GSCL in the GW Service through the dIa interface. The NA library is aimed at the development of M2M NAs and communicates with the NSCL using the mIa interface.

For a better perspective on the dynamics of these components, we will exemplify with the use case shown in Figure 3.2. Consider a smartphone user with the GW Service. The user can install an Android application to capture localization data. The application uses the GALib to communicate with the GW Service and be part of the M2M system as a GA. On the other side of the network, a Server uses the NALib to communicate with the NSCL and be part of the M2M system as an NA. The data captured on the GA is sent to the GW Service, that publishes it on the NSCL. The data becomes available for subscription by the NA. The NSCL will notify the NA with the incoming data, so the NA can process it. Now the NA, can for example, display the routes or the distance covered in a website for the user to see, or display the most popular routes in a city for a study. The NA can also publish the processed data

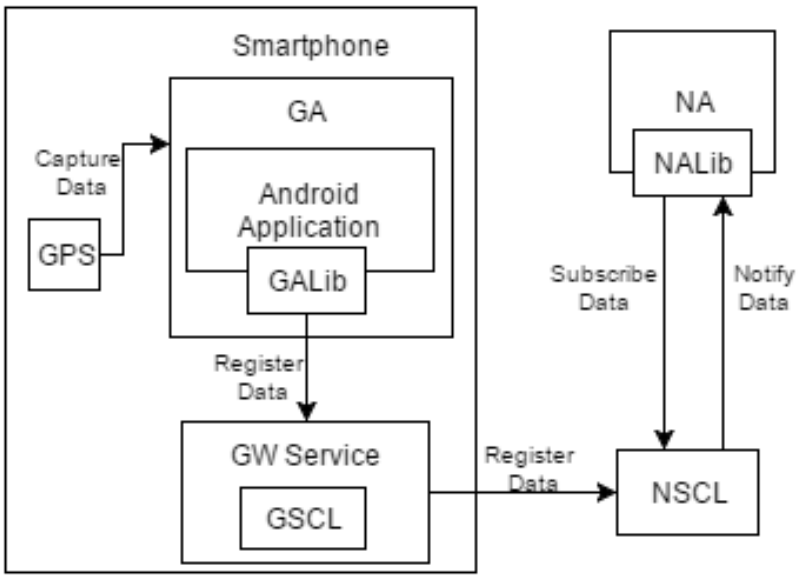


Figure 3.2: M2M Ecossystem example.

back in the NSCL to be used by other M2M entities. For example, the initial GA can subscribe the processed data and present the results to the user in the smartphone.

In the following sections we will provide an overview of the functional architecture and capabilities of the GW Service, GALib and NALib.

3.2 GW Service Overview

The GW Service enables a smartphone to act as an M2M GW. Installing the service in a smartphone equips it with an SCL exposing M2M capabilities to other devices, a module to translate legacy devices communication to M2M communications and capabilities to run M2M applications. The M2M applications may be running on the smartphone as GAs or in other devices as DAs.

The GW Service is implemented for Android devices and runs as an Android service almost invisible for the smartphone user. The only interaction done through the graphical interface is to turn on and off. All other interactions are made by M2M Applications communicating with the GSCL.

The GW Service architecture is represented in Figure 3.3, and has the following modules and interactions:

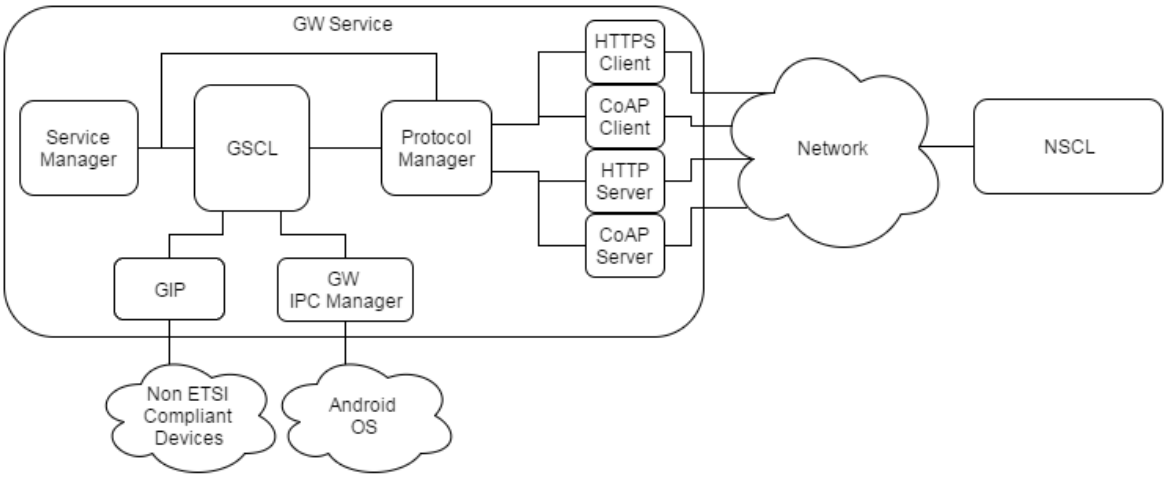


Figure 3.3: GW Service functional architecture.

Service Manager

Component that creates and manages the Android service. It is responsible for the setup, initiation and termination of all the other Components.

GSCL

The service's local SCL. Provides and processes the M2M functions and maintenance. It also stores and manages the local M2M resources. It can fulfill requests to create and retrieve local and remote resources, and it is responsible for cleaning expired resources.

Protocol Manager

Component that manages the communication protocols Clients and Servers. It is used by the GSCL to translate M2M operations to HTTP, CoAP or other protocols RESTful requests, and vice versa.

HTTP Client/Server

The communication's Client and Server used to exchange HTTP requests with other M2M entities.

CoAP Client/Server

The communication's Client and Server used to exchange CoAP requests with other M2M entities.

GIP

This module manages communication with legacy devices that do not support M2M operations. It was implemented as an internal capability of the GSCL.

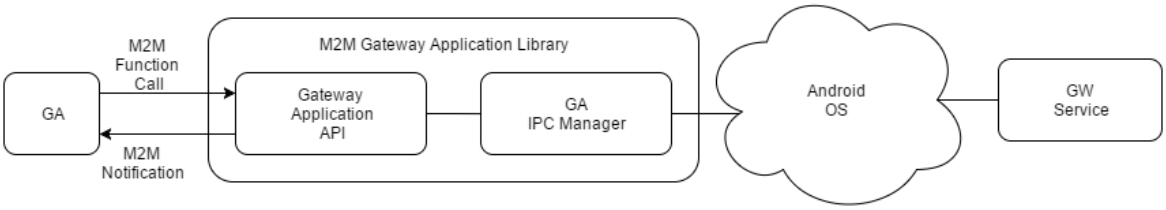


Figure 3.4: Gateway Application library architecture.

Currently, this module only supports Bluetooth devices, but it may support others in future, for example, ZigBee or Infrared devices.

GW IPC Manager

This module is used to communicate with GAs that reside in the same Android device, by using Inter Process Communication (IPC) capabilities.

3.3 GALib Overview

An M2M GA is an M2M Application that resides on a M2M GW and uses the GSCL. The GALib is an Android Library used to create Android applications that act as M2M GAs. The GALib requires the GW Service to be running to access its GSCL.

By allowing the development of Android GAs apart from the GW Service, we greatly extend the M2M capabilities of the smartphone. Instead of having the M2M Applications included in the GW Service and, therefore, developed only by the GW Service provider, we open the development of M2M applications to anyone with access to the GALib.

The GALib architecture is represented in Figure 3.4 and has the following Components:

Gateway Application API

Exposes a set of methods to setup the GA settings and make M2M requests to the GW Service. The developer creates an Android application and interacts with the M2M ecosystem through this module.

GA IPC Manager

Translates M2M operations to Android IPC requests. This module eliminates the necessity of network servers and clients for communication with the GSCL.

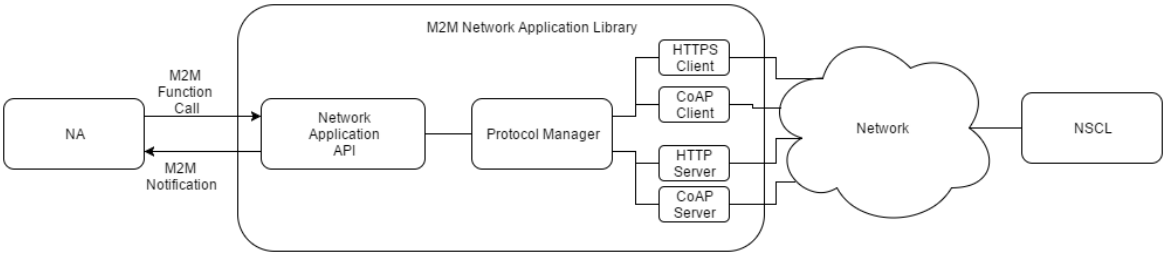


Figure 3.5: Network Application library architecture.

3.4 NALib Overview

An M2M NA is an M2M application that resides on the network domain and uses the NSCL. The NALib is a Java library used to create Java programs that act as NA applications by exposing a set of methods to interact with the M2M ecosystem. The NA is very similar to the GA however it connects directly to the NSCL and doesn't reside in the same device as his SCL.

The architecture, depicted in Figure 3.5, can be divided in following components:

Network Application API
: Exposes a set of methods to setup the NA and make M2M requests to the NSCL. The developer creates the GA and interacts with M2M ecosystem through this module.

Protocol Manager
: Component that manages the communication protocols Clients and Servers. It is used by the NA to translate M2M operations in HTTP, CoAP or other protocols RESTful requests and vice versa.

HTTP Client/Server
: The communication's Client and Server, used to exchange HTTP requests with other M2M entities. Similar to the GW Service HTTP Client/server.

CoAP Client/Server
: The communication's Client and Server, used to exchange CoAP requests with other M2M entities. Similar to the GW Service CoAP Client/server.

Chapter 4

Evaluation of an M2M Mobile Gateway

In this work we used an existing M2M GW as a starting point. For a close analysis of the previously developed software, in this chapter we present an overview of the M2M GW and the results of an experiment we conducted to identify possible performance problems. Pinpointing these problems allowed us to find and implement solutions to improve the GW service.

4.1 Mobile M2M Gateway Implementation

This section we will describe the components belonging to the initial M2M GW, the resource structure implemented and its work flow. We will also identify any implementation problems.

4.1.1 High Level Architecture

The Mobile M2M Gateway High-Level Architecture, depicted in Figure 4.1, can be divided in several different components:

- **Service Manager** is the component that implements an Android service, running on the background, and it is responsible for managing and maintaining the remaining components. It is also responsible for handling commands sent by a

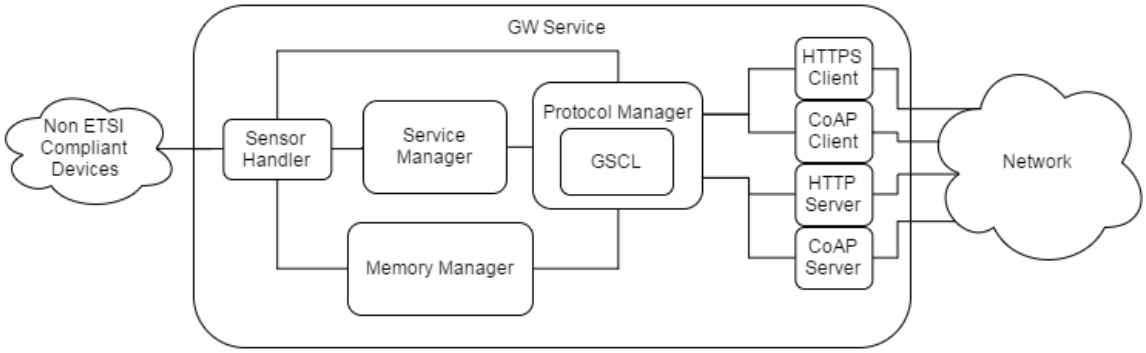


Figure 4.1: Mobile M2M Gateway high-level architecture.

Network Application.

- **Protocol Manager** is the component that handles all communications with the NSCL through HTTP and encapsulates the GSCL . Additionally, it is also responsible for marshaling the data to be sent and keeping track of the device's connectivity.

- **Sensor Handler** is the component responsible for searching the presence of sensors, manage their connection and communication procedures and sending the sensor data to the Memory Manager. The sensors may be external, accessible by Bluetooth, or internal, residing in the Android device itself.

- **Memory Manager** is the component which handles data buffers in memory and periodically forwards the collected data to the Protocol Manager.

- **GSCL** is the component responsible for maintaining the SCL resources in memory and/or storing them in a local database. This component resides inside the Protocol Manager.

This Architecture is different from that presented in Section 3.2. The differences are detailed in the implementation Section 5.1.1.

4.1.2 Resource Structure Mapping

In order to implement the GSCL, the Mobile Gateway mapped the SCL resources as seen on Figure 4.2. The resource mapping was validated in a Healthcare scenario with the following structure:

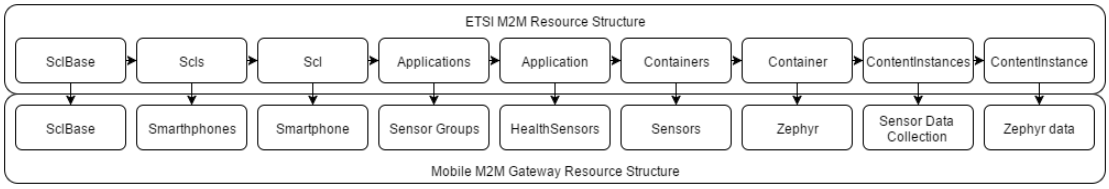


Figure 4.2: Mobile M2M Gateway Resource structure mapping for an healthcare scenario.

- Each SCL represents a patient smartphone;
- Sensors were categorized by types and each sensor group was mapped to an Application resource;
- Each sensor inside a sensor group is mapped to a Container resource.

Even though this approach works on a Healthcare scenario it does not follow all the standard's specifications. For example, the Application resource represents an M2M Application, like a DA, GA or NA, which runs the M2M service logic. Using the Application resource as a Sensor Group does not fulfil the purpose intended by the standard, as it diminishes the resource to a category representation.

4.1.3 Work Flow

The work flow may be divided into three distinct states. In this section we will detail the sequence of tasks performed by the M2M Gateway.

4.1.3.1 Bootstrap

The Bootstrap is the first state of the M2M GW, and comprises a set of tasks executed only when the application is turned on. The first task is to start all the components described in section 4.1.1 by triggering the Main Service. In the case of the Memory manager, the memory buffers and a GPS location service are initiated. In the Sensor Handler, a Bluetooth manager is initialized and a search for available external sensors is scheduled. Finally, the Protocol Manager initiates the GSCL and an HTTP Client and Server. The HTTP client has then the task to authenticate itself to the NSCL and retrieve a symmetric key to be used later for the TLS security protocol. In order to do this, the following steps are performed:

1. Load a temporary HTTP Client with the keystore and truststore provided inside the compiled program;
2. Request an TLS connection to the NSCL;
3. Perform the TLS Handshake Protocol between the Client and the NSCL;
4. Retrieve the symmetric key to a keystore;
5. Load the HTTP Client with the just created keystore and the truststore.

After the HTTP Client is ready, the second phase of the Bootstrap has the GSCL perform the following steps:

1. Register the SCL on the NSCL;
2. If the SCL is already registered, then retrieve it from the local database or from the NSCL;
3. Check whether the SCL is subscribed in the NSCL Applications resource and NSCL Application resources;
4. If a subscription is missing in the resources above, then perform the subscription..

When all subscriptions have been correctly checked,the Standby procedure starts. The subscriptions are used to enable other M2M entities to send commands to the Gateway, this is called actuation. If the NA, with id *appid*, wishes to send a command to the M2M GW, it only needs to create a Content Instance, containing the command, under the *m2m/applications/appId/containers/ACTIONS/contentInstances* resource. Since the M2M GW subscribed this resource, it will receive a notification with the NA command.

To keep the subscriptions updated, the M2M GW and the NSCL exchange of messages presented in the Figure 4.3. The Check Subscription Procedure in the figure requests one by one the subscriptions present in a certain resource, until a subscription referring the Gateway's current address is found. In case the last condition is not met, the Gateway creates a new subscription.

Even though this method works properly for a small use case, it is not scalable for a large number of devices. There are two major problems with this approach. The first one is that one command requested by one NA is sent to all the gateways. The other

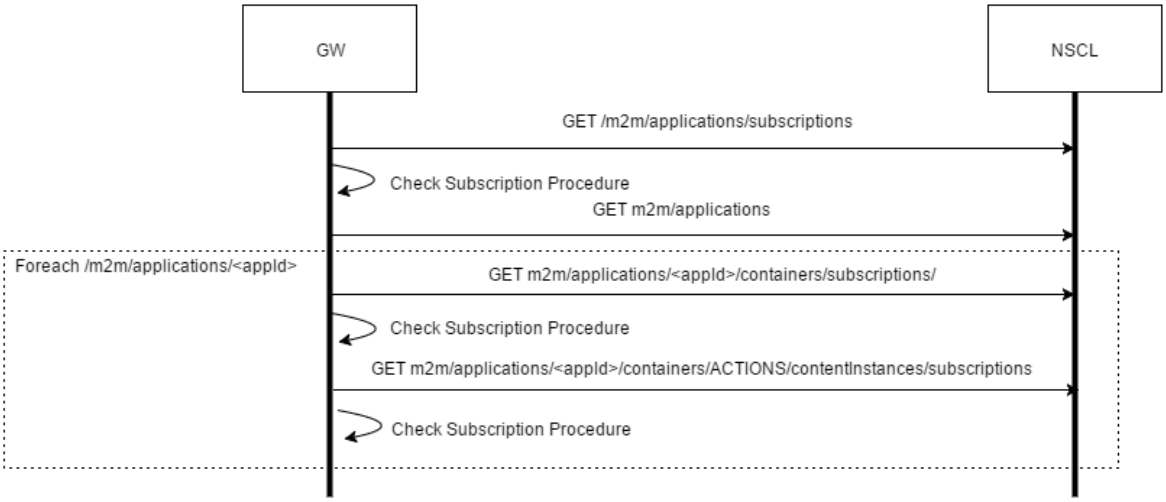


Figure 4.3: Subscriptions method for NA actuation.

one is that when the gateway starts it will make a number of requests that grows with the number of M2M devices in the M2M network. As an example, we will consider a set of m GW's and n NA's and, since the M2M GW is mobile and it will change its address frequently, we also consider that the Subscription Check Procedure will need to loop all existing subscriptions. The number of requests is when the M2M GW starts is:

- m requests for m2m/applications/subscritpions/subID;
- n requests for applications/appID/subscriptions;
- nm requests for applications/appID/subscriptions/subID;
- n requests for applications/appID/containers/subscriptions;
- nm requests for applications/appID/containers/subscriptions/subID;
- n requests for applications/appID/containers/ACTIONS/ contentInstances/subscriptions;
- nm requests for applications/appID/containers/ACTIONS/ contentInstances/subscriptions/subID.

Excluding requests that only happend once, we get:

$$numberOfRequests = m + 3n + 2mn$$

This means that using only 10 GW's and 10 NA's, it is necessary to make 240 requests when starting the M2M GW.

4.1.3.2 Standby

The Standby is the state in which the Gateway is not connected to any device. It is expected that the Gateway spends most of it's execution time in this state.

When in standby the Gateway has the responsibility to search for new sensors periodically, check sensor data in memory periodically, keep the track of the mobile device location, and maintain the HTTP Server and Client.

If the Sensor Handler finds a new sensor from which data will be sent, Gateway enters a Sensor Sending state.

4.1.3.3 Sending

The Sending state happens when the Gateway is sending sensor data. When using an external sensor the Sensor Handler must start a Bluetooth connection and, depending on the sensor, execute a handshake procedure with the sensor and starts receiving its data.

If the connection is successful the Gateway registers the sensor in the NSCL, initiates a data buffer and requests the sensor data. In case the buffer is full, a new ContentInstance containing the sensor data gathered is created in the NSCL; otherwise, even if the buffer never gets full, the Gateway will periodically send the data contained in the buffer.

The sensor sending state ends and returns to standby when the sensor is disconnected.

4.2 M2M Mobile Gateway Performance Analysis

In this section we present an experiment to analyse the performance of the existing gateway. With the analysis we hope to pinpoint any major problems, since we aim to have a large number of users adopting this technology.

4.2.1 Experiment Strategy

In order to analyse the performance of the Gateway application, we need to track the battery, Wi-Fi, CPU and RAM usage. Note that at the time of this experiments the android studio tools was not available for performance analysis. In some cases, since the Android API methods didn't offer the means to gather the desired data, it was necessary to create a periodic reading of some important Linux files in the Android OS core. The intervals between readings were made as small as possible, to make sure that the values gathered were relevant. However, the constant capture of data created a large delay in the Gateway application. With this problem in mind, we created a second standalone application to handle the periodic readings. This application is referred as the Andralyser and the inner analyser is referred as the Gwanalyzer.

The Andralyser also has the extra function of monitoring the device's overall battery usage, since once again there were no methods in the Android 4.4 (KitKat) API that allowed the analysis of the energy used by a single application. The solution found was to gather the data concerning the battery with the Gateway application on and off. The results were later compared to infer the battery life.

Additionally, to capture the packets in more detail, the Shark for Root[1] capture tool was used. The captured packets could later be viewed in programs, such as Wireshark.

4.2.1.1 Experiment Tools

- **Andralyser**

  When the Gateway Application starts, it sends an intent containing its process ID. When Andralyser receives the intent, it accesses the Linux files containing the desired information for the analysis and reads them periodically. The periodic reading has a rate of 100 samples per second and is in charge of capturing the Wi-Fi, CPU and RAM usage. Moreover, the Andralyser listens for changes in the battery state and register the values received through its intents. In all of the samples there is a capture timestamp associated for chronological organization purposes.

  If we suppose that the Gateway Application process Identification is *PID*, the file corresponding to the CPU statistics is located at the */proc/PID/stat* folder, the Wi-Fi statistics are located at the *proc/PID/net/dev* and RAM statistics

[1] `https://play.google.com/store/apps/details?id=lv.n3o.shark&hl=pt_PT`

are located at the */proc/PID/statm*.

- **Gwanalyser**

  The Gwanalyser is an inner class of the Gateway Application, initiated along side the Main Service component. After initiated, it can be accessed by any of the application's components to log events and resource usage values. Like the Andralyser, in all of the samples there is a capture timestamp associated for chronological organization.

- **Shark for Root**

  The Shark for Root application is a traffic sniffer based on tcpdump for android that supports capture on Wi-Fi and mobile networks. In order to be functional, the root access must be unlocked. After finishing the capture it saves a .pcap file which is compatible with programs like Wireshark.

The Android device used was the Google Nexus 5. When choosing the device for the experiment the requirement was that the device should be powerful. Otherwise the additional applications like Andralyser and Shark for Root would stop due to the limited resources in low end devices. This case was observed in another testing device, the Samsung Galaxy Europa GT-15500.

The device's specifications relevant for the experiment are:

- **Battery:** 2300 mAh;
- **CPU:** Qualcomm Snapdragon 800, 2,26 GHz;
- **Wi-Fi:** Dual-band (2.4G/5G) 802.11 a/b/g/n/ac;
- **Bluetooth:** 4.0;
- **RAM:** 2 GB;
- **Sensors:** GPS, Accelerometer.

4.2.1.2 Experiment Procedures

The procedures for the experiment were divided in four cases: GW standby, GW sending with external sensor, GW sending with internal sensor and GW turned off.

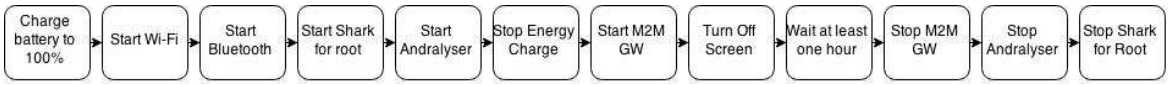


Figure 4.4: GW Standby experiment procedure.

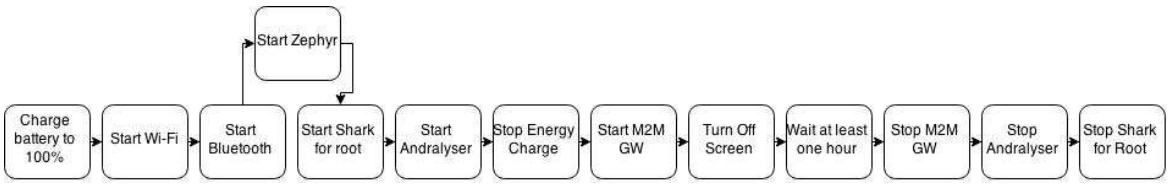


Figure 4.5: GW Sending with external sensor experiment procedure.

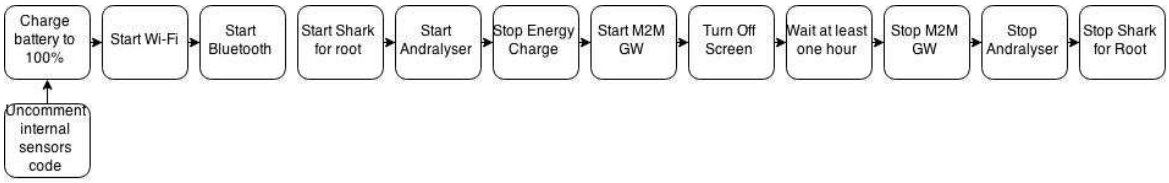


Figure 4.6: GW Sending with internal sensor experiment procedure.

The Bootstrap was not analysed since it is a one-time procedure making it is harder to analyse.

The standby case, seen in Figure 4.4, tested the state in which the M2M GW is turned on but is not connected to any sensor or sending data. The performance in this case should be optimized to the maximum to allow the M2M GW to be running on the background of the Android device without leaving a large footprint in the device's resources. The data originated from this block was gathered by running the application in an area without any sensor active.

The sending cases with internal sensors, seen in Figure 4.5, and external sensors, seen in Figure 4.6, were used to test the state in which the M2M GW is sending continuous data. For example, a weighing scale only sends one set of data to be forwarded, so it will not waste many resources. However, a Heart Rate Sensor, like a Zephyr, will send continuous sets of data, which can use too many resources for a mobile device. Using internal and external sensors in different cases allowed for a better analysis of the network and Bluetooth performance impact. Furthermore, each sending case was also divided into two different cases, with and without a Buffer, to study the performance impact of data aggregation. Using a buffer, the data was sent in intervals of 10 seconds; otherwise, the data was sent every second.

Finally, the GW turned off case, shown in Figure 4.7, serves as comparison by capturing

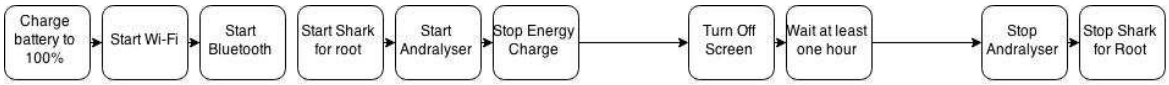


Figure 4.7: GW Off experiment procedure.

the normal smartphone's usage of resources.

In all cases the battery was first charged to 100% of the capacity and the Wi-Fi and bluethooth were turned on. After that, for the case using an external sensor, we started the zephyr. On the next step, in all cases the performance data capture tools, Andralyser and Shark for Root, were initiated and the battery charge stopped. In the GW off case, the screen is immediately turn off and the smartphone is left in standby. In the other cases the the M2M GW is started, the screen is turned off immediately and the application is left running. After at least 1 hour, the M2M GW, Andralyser and Shark for Root were stopped and the performance data retrieved. For each case the procedure was repeated 3 times.

4.2.2 Experiment Results and Analysis

In this section we analyse the data obtained in the experiment for energy consumption and network, CPU and memory usage.

4.2.2.1 Battery Life

On Figure 4.9 we find the approximated time, in hours, it took for the battery to go from 100% to 10% of its capacity. We refer to this time as the smartphone's battery life. To avoid the large amount of time needed to deplete the battery from 100% to 10% in all the experiment's cases, we assumed the battery depletion is linear. We validated this assumption with the results, seen in Figure 4.8, obtained by another team in the scope of Sense My City project[17]. The figure depicts the battery percentage at each second during a full battery depletion. Since the battery depletion is linear, we registered the time interval of the first and last drop of the battery level and we used a linear model to extrapolate the results to calculate how much time it would take for the battery to reach 10% of its capacity.

Starting with the GW Off values, we can see that the device has 115 hours of battery life. This value is reduced to 45 hours when we leave the Gateway in standby, which is a large drop in battery duration. The reason may have been the fact the service

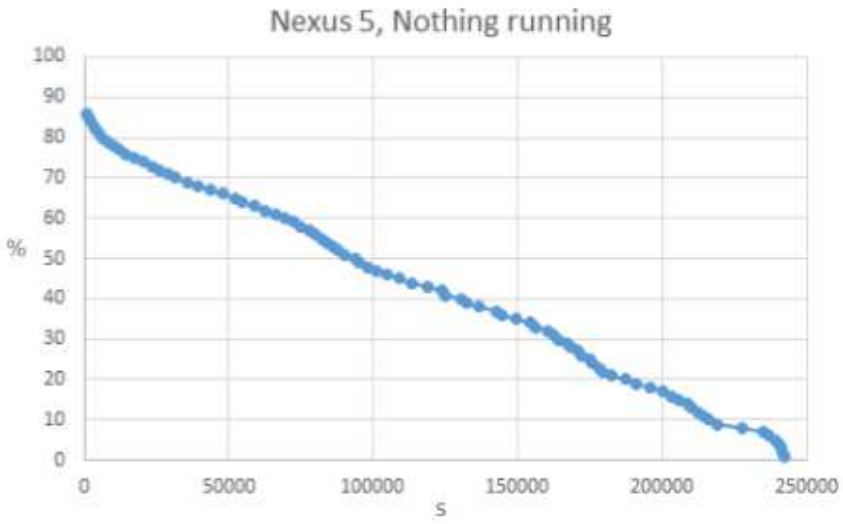


Figure 4.8: Battery Depletion on Nexus 5 with nothing running [17].

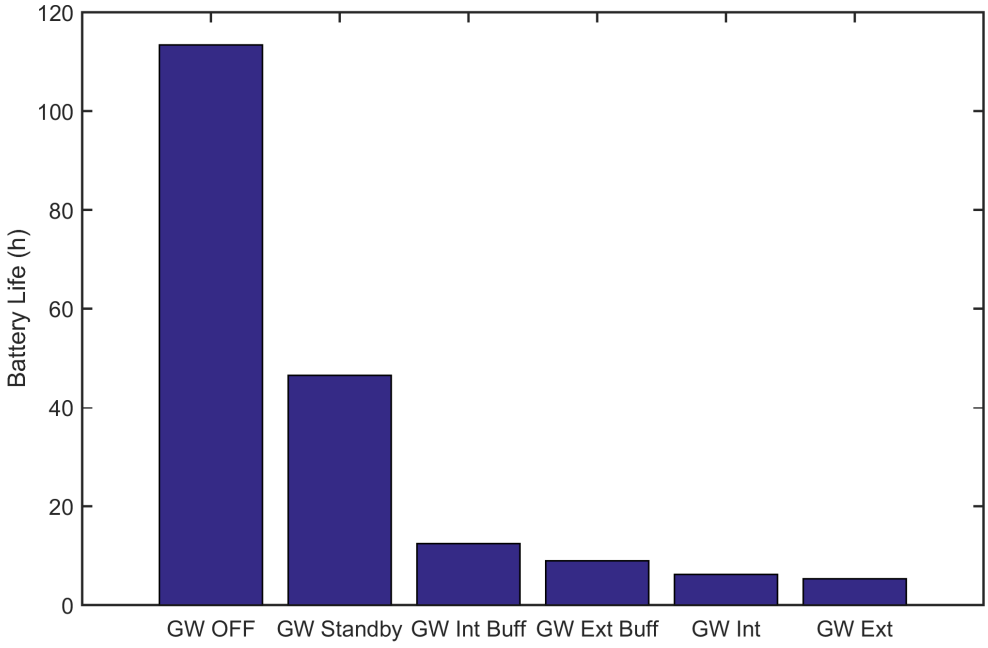


Figure 4.9: Battery Life.

is constantly running on background and requires the existence of an HTTP server listening for requests, a client securely connected to the NSCL, a periodic search for Bluetooth sensors and the constant use of location services.

When comparing the battery life with the Network usage, let us first look at the cases of GW Standby and GW Internal Sensor Sending, in which Bluetooth is not used. Since the major difference between these cases is the processing and transmission of data, we can assume that the battery variation comes from either the CPU and Memory or Network usage.

If we analyse the Table 4.1 and Figure 4.14, some variations are visible in memory and CPU usage. We cannot consider memory usage to have great variation, but the CPU Time usage is 2 times larger for the buffered case and 10 times larger for the unbuffered case. The increase in CPU time is obviously explained by the constant reading, parsing, marshalling, storing and sending of data. However, the CPU time usage still does not reach 1%, so we cannot attribute the battery depletion to the processor, since the simple *top* command in the android shell uses about 2% of the CPU time. We then consider that the difference of battery life is due to the network usage.

In the cases in which an external sensor is used, it is possible to see the Bluetooth footprint, since, in relation to the internal sensor cases, the only change in the experiment was the usage of Bluetooth to communicate with the sensors. The difference in values is smaller than the one caused by the network, but it still is a considerable drop and it may be a target for optimization.

4.2.2.2 Network Usage

The next important field to study is the network usage, since it is known for its high depletion of battery. As we can see in figures 4.10 and 4.11, the values for the GW Off case are very small and mostly related to the android services, so we will ignore the extra network used by the device alone. In the case of the GW Standby there is a small increase in network usage which is related to a periodic TLS handshake from the HTTP Client.

Regarding the sensor sending cases, the upload reaches higher values than the download which is normal when using the M2M ETSI standard. In the cases of data sending, the response generated by the NSCL always contains the payload sent. Even though these messages are required by the M2M ETSI standard, they are not used by the Gateway in a meaningful away and present a great source of overhead in the upload usage.

Another interesting aspect that can be observed is the impact a buffer has in the

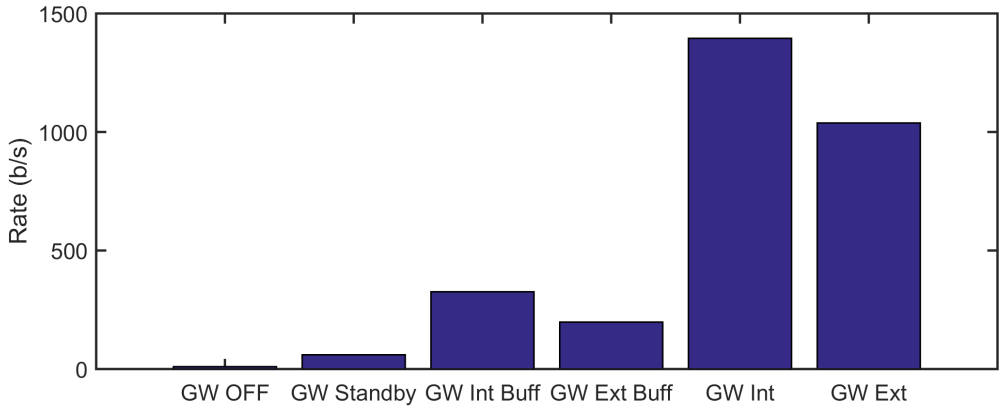


Figure 4.10: Incoming network usage.

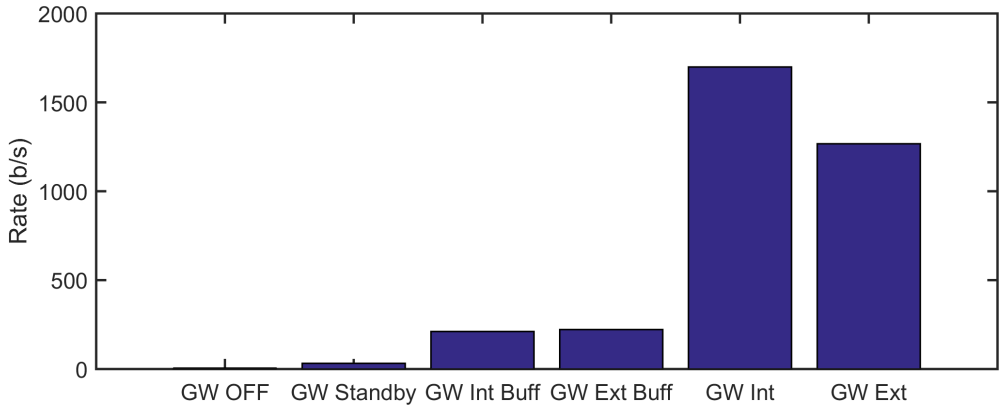


Figure 4.11: Outgoing network usage.

network usage. When the buffer is active, the quantity of bytes sent and received is highly reduced. The main reason for this difference consists in overhead caused by the data encapsulation. For a better understanding, in the Figure 4.12, the bytes per second exchanged were divided in JSON and HTTP overhead. Furthermore, the last figure also represents the number of bytes containing sensor data encoded in base 64, which are extracted from the bodies. The difference between encoded and decoded data in base 64 is always a constant overhead of approximately 33%.

Comparing the columns from each experience, the relation between sensor data and encapsulation bytes is easily visible. On the non buffered cases, most of the bytes are used in JSON and HTTP encapsulation. On the other hand, if we analyse the buffered cases, it is noticeable that, in relation to the encapsulation bytes, a higher percentage of data bytes is exchanged. We may also notice that the quantity of data base 64 bytes sent are smaller for the buffered cases. This is due to the fact that the encoded sensor

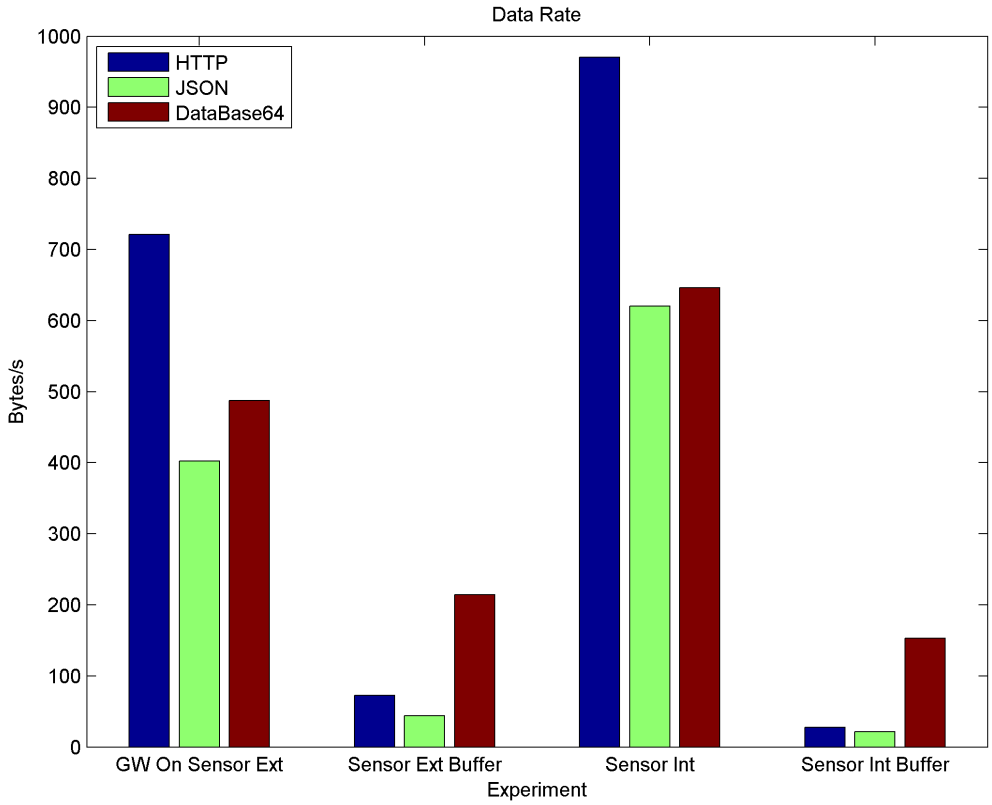


Figure 4.12: Data encapsulation overhead.

data also contains another layer of JSON which is sent repeatedly in every message, causing overhead. The Figure 4.13 exemplifies the encapsulation scheme.

Considering the impact of the encapsulation overhead exemplified in this section we believe that an effort to reduce the size and frequency of headers would improve the network usage as well as the battery life. Besides the headers, there is also an overhead added by ACK packets implemented by the TCP layer. Finally, another improvement could be to re-evaluate the necessity of the repetitive response messages imposed by the ETSI M2M standard.

4.2.2.3 CPU Usage

In Section 4.2.2.1 we saw that CPU usage is not significant, but there is a curious behaviour with the process faults values in Figure 4.15. The values presented represent the sum of major and minor faults from the process and its children. Most of the faults are major and originated from the process's children which means that they are caused

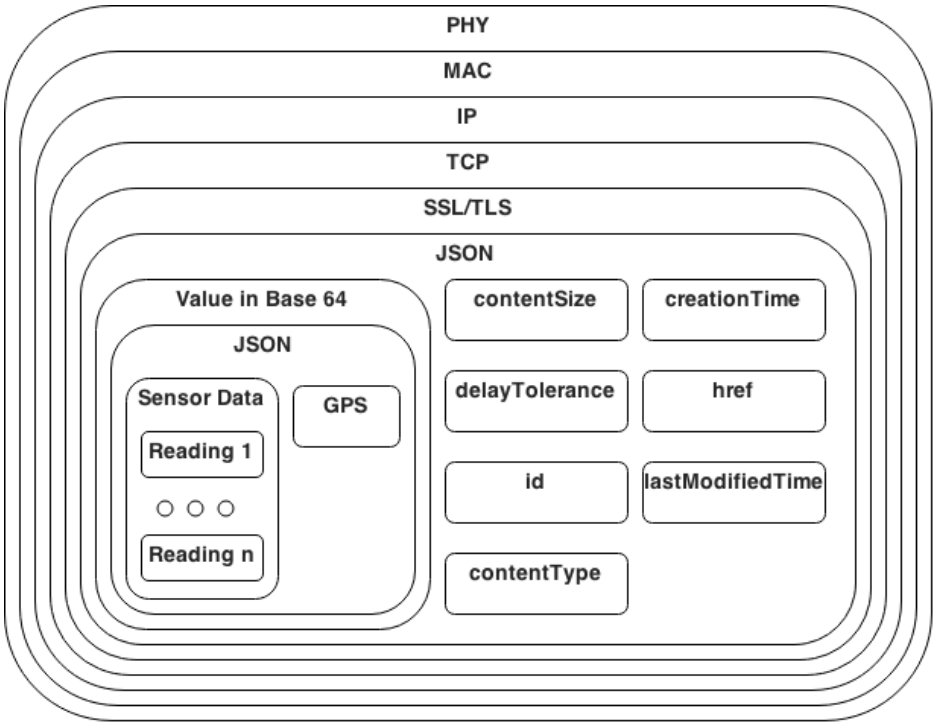


Figure 4.13: Data encapsulation example.

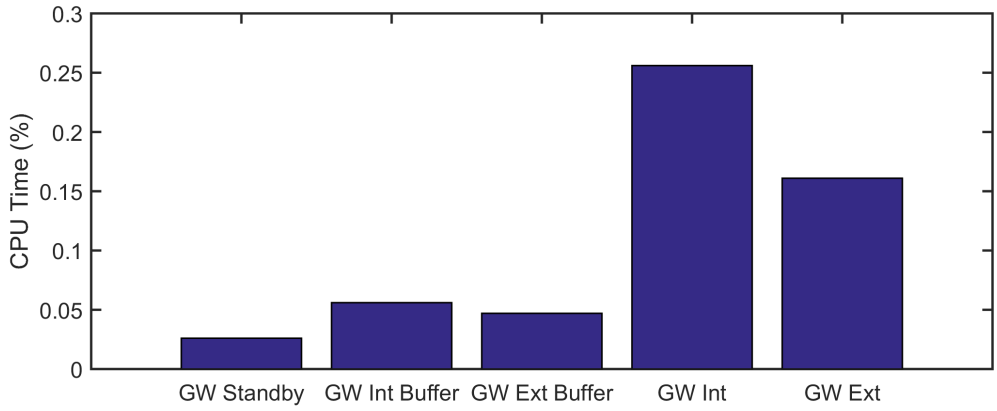


Figure 4.14: CPU Time usage.

by the children trying to access data that is not mapped in the physical memory.

We can see that the number of faults per hour grows for the cases in which there is data sending with buffer and that it grows even larger when a buffer is not used. Since the major difference between these cases is the presence of the buffer, we can assume that the faults are caused by the data sending. When data is sent almost all threads exchange messages through their queues, so the problem may be related to high rate of messages exchange between the threads.

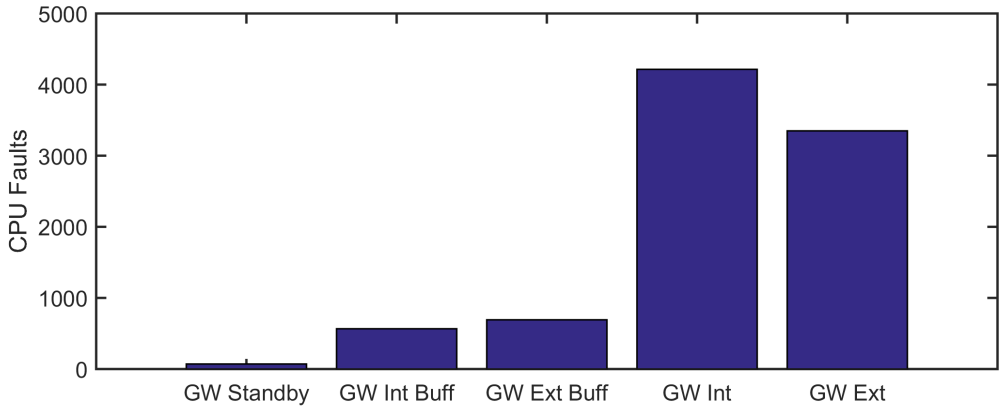


Figure 4.15: CPU Faults.

Experience	Total	Resident	Shared	Data/stack
GW Standby	884069	41722	11294	22757
GW External Sensor Sending	887036	42219	11528	25724
GW External Sensor Buffer Sending	886285	41945	11414	24973
GW Internal Sensor Sending	885473	42425	11435	24161
GW Internal Sensor Buffer Sending	884786	42012	11405	23474

Table 4.1: Memory usage.

4.2.2.4 Memory Usage

In Table 4.1 we find the mean values for the memory usage. The values concerning the total column demonstrate a large memory footprint. However these values only represent the physical and virtual memory reserved, that is, not all of it is being used. Then, the most important values to be aware are those in the resident column, which represent the physical memory occupied.

The values captured in the experiment are quite constant. All sending cases use about 42 Mb of memory and the standby case uses 41 Mb. The variation is not large and the values are consistent with a normal android service. For example, in the smartphone used in this experiment, the Google calendar service uses about 880 Mb of total memory and 36 Mb of resident memory.

4.2.3 Experiment Conclusions

In the implementation overview we found that the actuation relies on a series of subscriptions that are not scalable and can rapidly grow in an environment with a large number of devices. We also found some inconsistencies with the M2M resource mapping that will require a rework that impacts the registration and actuation procedures.

During the measurements, it was clear the network communications performance requires more attention. For example, the implementation of CoAP and DTLS could be a great improvement, since they are targeted at constrained devices. Moreover we can also reduce the exchanged data by reducing the messages overhead or eliminate irrelevant data received from the NSCL. In the case of memory and CPU usage, the behaviour was normal, aside from some variation in page faults. The battery consumption can be improved by optimizing the overall mechanisms and procedures of the M2M GW.

Chapter 5

Interoperable M2M Ecossystem Development

Considering the results obtained in the last chapter we implemented some changes to improve the GW Service. We also developed the two libraries discussed in Chapter 3 to deploy NAs and GAs, NALib and GALib.

In this chapter, we will describe problems and solutions found during the development and the implementation choices of the GW Service, GALib and NALib.

5.1 Gateway Service

In this section we will describe the modifications made to the GW service, intended to solve some of the problems identified in last chapter's experiment. We are mainly concerned with the resource mapping and actuation, but we also take into account other changes to improve performance.

5.1.1 GW Service Architecture

As stated before, the initial architecture from Figure 4.1 was altered to the architecture in Figure 3.3.

The main differences are the following:

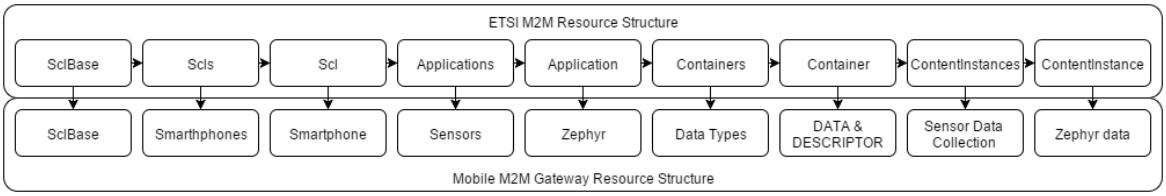


Figure 5.1: Final resource mapping.

- The Sensor Handler was replaced with the GIP Component. Their function is similar, however the sensor handler would only connect to a limited set of Bluetooth devices that needed specific support in the GW Service. The GIP allows the GW Service to connect to any Bluetooth device compatible with Android devices. Support for other communication technologies can also be added.
- In the most recent architecture, the GSCL was separated from the Protocol Manager. This allows the GSCL to act independently from the communication protocol being used. The GSCL simply delegates the management of network communications to the protocol manager.
- In the newer version, the Memory Manager was merged with the GIP instead of running on its own thread. Since the Memory Manager was only used to buffer data from sensors, this task was moved to the GIP.

The changes made to the architecture allow for clearer separation of modules and easier improvement and implementation of new features.

5.1.2 Resource Mapping

To better fit the resources‘ properties and purposes, it is necessary to rework their mapping, discussed in Section 4.1.2. For example, the Application resource has the Application Point of Contact (APoC) attribute[10, 84], which may contain a communication address to access an external device, e.g. a ZigBee device. We decided that the Applications resources would represent external sensors available to the M2M GW and GAs using the M2M GW. The resultant resource mapping can be seen in Figure 5.1 for a specific device.

As we observe, the major difference is that the sensors, which were mapped to the Container resource, are now mapped to the Application resource. This was imple-

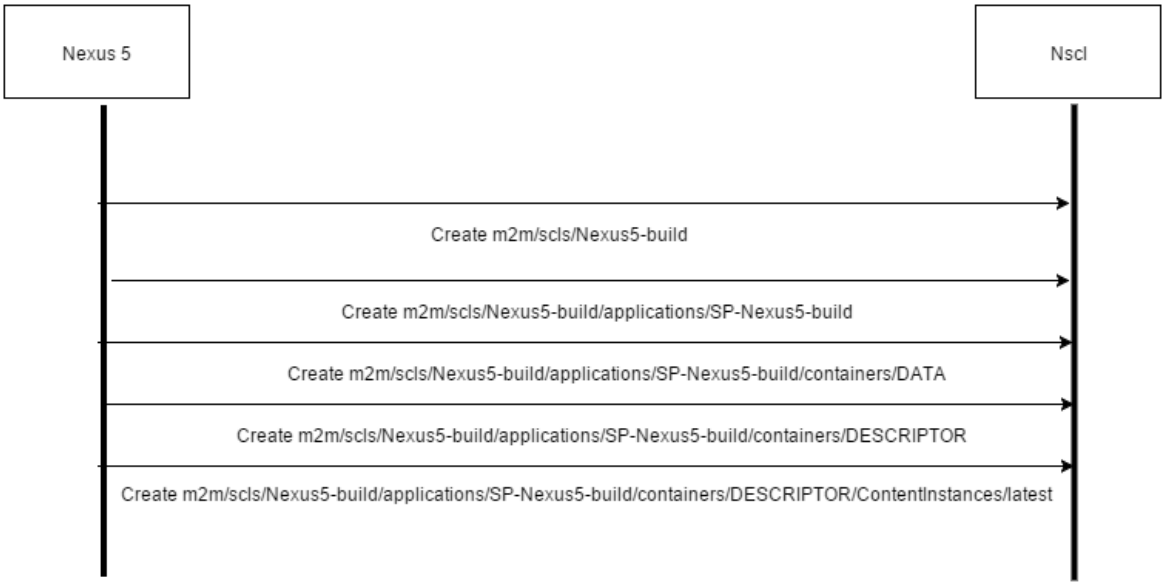


Figure 5.2: Registration process.

mented because most sensors will use the GIP, which is also a GA. Since we want to distinguish the different sensors connected by the GIP, we register them as different applications. Furthermore, the smartphone should also be mapped as an Application resource, due to its sensor-like capabilities, like Bluetooth device discovery.

Inside each M2M Application there are two containers: DATA and DESCRIPTOR. The DATA Container contains the data captured by the M2M Application and the DESCRIPTOR Container contains information concerning the application. This strategy is inspired by the M2M implementation of Actility's Cocoon project [1] and on the ETSI Technical Report [6] .

In order to accommodate these changes, the registration process was also modified to the procedure depicted in Figure 5.2.

When the M2M GW device is registered in the NSCL, it is necessary to create an SCL and an M2M Application representing the smartphone. Since each M2M Application should have the containers DATA and DESCRIPTOR , they are also immediately created in the registration Process. Finally, the Gateway publishes a Content Instance in the DESCRIPTOR Container, describing the smartphone Characteristics and capabilities. The importance of the new resource mapping and registration process will become clearer in the next section, since we make use of the DATA and DESCRIPTOR Containers and the M2M Application APoC attribute.

5.1.3 Actuation on the M2M GW

It was necessary to implement an Actuation procedure that required fewer messages in a large scale M2M network and that was able to target one single M2M GW device. To solve this problem, three approaches were explored, based on the MgmtObjs resources[10, 104], the Remote SCL resource's access and the Retargeting mechanism[10, 238].

5.1.3.1 MgmtObjs based approach

The MgmtObjs is a collection of <MgmtObj> and <MgmtCmd> described by the ETSI M2M standard as resources which hold management data and commands.

This resource may be located in branches of the SCLS, <SCL>, Applications and <AttachedDevice> resources, providing a considerable flexibility for the implementation of the Actuation mechanism.

Regarding the <MgmtObj> and <MgmtCmd> resources, the second is the most suitable resource for the intended purposes, since its mechanisms enable an NA to trigger management commands or RPCs defined in the BBF TR-069 document[19]. In addition, it contains useful attributes like the following:

- **cmdType** - used to identify a command;
- **execReqArgs** - used to specify arguments;
- **execEnable** - used to execute a command or to specify an URI to execute a command.

The <MgmtCmd> resource also contains a collection <execInstance> resource, each one representing an ongoing instance of an execution request. This resource provides atributes to describe the execution state and result, and to cancel an execution. Furthermore, the requester can subscribe an <execInstance> to be notified of the execution results. Based on the ETSI M2M standard guidelines, we created a procedure for the Actuation on a M2M GW, depicted in Figure 5.3. This procedure can be described by the following steps:

1. The GSCL creates a <mgmtCmd> resource which represents an available command;

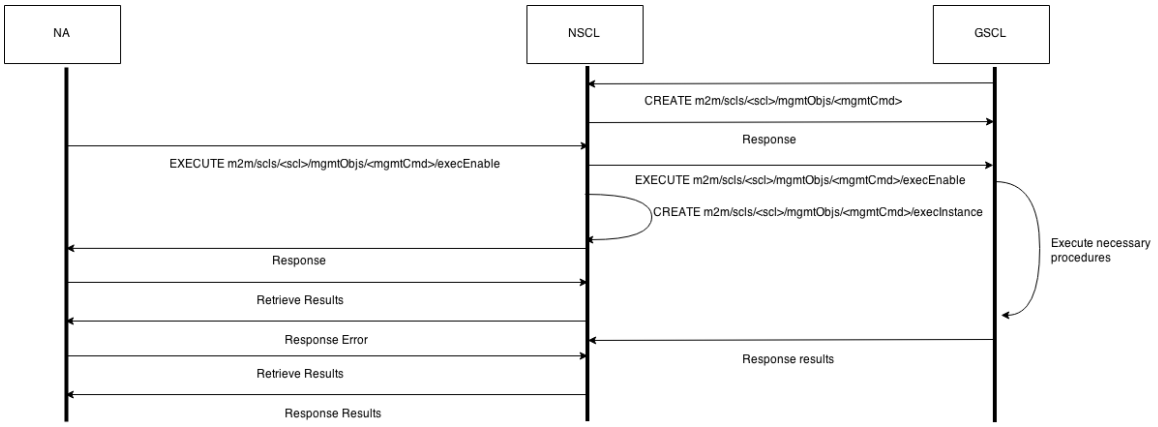


Figure 5.3: MgmtObjs actuation procedure.

2. The NA requests an execution targeted to the desired <mgmtCmd> resource;

3. The NSCL processes the request, forwards it to the GSCL, creates an <execInstance> and responds to the NA execution request;

4. The NA subscribes the new <execInstance>;

5. The GSCL processes the request, executes the necessary procedures and updates the <execInstance> on the NSCL;

6. The NSCL notifies the NA with the updated <execInstance > containing the results.

5.1.3.2 Remote SCL resource's access based approach

In order to have a functional actuation on an M2M GW, the NA must in some way be able to contact the targeted M2M GW. The most straightforward method found consisted in using the ETSI specification to access resources on different SCLs. The standard states that an M2M Application may access remote resources with zero to three SCL hops. Then, using the one hop case, if the an M2M entity requests a GSCL resource which is not present in the NSCL, the NSCL will forward the request to the GSCL, so that it can respond to the NA itself.

However, the GSCL must have a customized interpretation of the received requests that is not defined in the standard. This does not require that the standardized request interpretation be ignored, but rather extended. As such, when the GSCL receives a

request, it will try to resolve the request in the standardized way. If it is not able to do so, it will then interpret the request as an actuation command.

5.1.3.3 Retargeting based approach

The Retargeting is defined in the ETSI M2M standard as a mechanism to enable an SCL to route messages to an M2M Application. This mechanism is achieved by using two attributes of the Application resource: APoC and Application Point of Contact Paths (APoCPaths). The APoC attribute contains an URI that can be used to contact the registered application and the APoCPaths contains a list of paths allowed for use, in retargeting.

A Retargeting procedure is then triggered if the target resource, in a received request, is not present in the receiving SCL and the target URI matches or is prefixed by the combination of the registered Application resource path and one of the aPoCPaths. When this happens, the registered Application resource path, in the targeted URI, is substituted by the Application's aPoC and the request is forwarded.

In the scope of the actuation, the various actions available in an M2M Application can be listed as retargeting paths in the aPoCPaths attribute. When an M2M entity makes a request for one of the aPoCPaths, the NSCL should retarget the request to the Gateway Device. The Gateway then interpreters the request and executes the desired action.

Figure 5.4, shows an example of an M2M Application registration, prepared for retargeting, and the outcome of three different requests. The example has the following steps:

1. The M2M GW creates a new Appplication "appId";
2. The Application has the *APoC* attribute targeting the M2M GW's URI and Port and two APoCPaths;
3. The M2M GW creates a Container "contId1";
4. Outcome one:
    - The NA requests the Container "contId1";
    - The NSCL has the resource;
    - The NSCL responds with the resource;

5. Outcome two:
    - The NA requests the Container "contId2";
    - The NSCL doesn't have the resource;
    - The NSCL doesn't find a match in the APoCPaths;
    - The NSCL responds with an error;
6. Outcome three:
    - The NA requests the APoCPath "allowed/retarget/path1/anything/else";
    - The NSCL doesn't have the resource;
    - The NSCL finds a match in the APoCPaths;
    - The NSCL forwards the request, replacing the prefix with the M2M GW URI in APoC;
    - The M2M GW interprets the request and sends the result to the NSCL;
    - The NSCL responds the NA with the result received.

As in the case of the Remote SCL resources access, the GSCL also needs a customized interpretation of the received requests. The used method is the same: if it is not able to resolve the request normally, it will interpret the request as an actuation command.

5.1.4 Actuation Implementation

To implement the Actuation procedure, the most favorable options are Retargeting and Remote SCL resource's access. However, the first option was created to provide access to M2M Application resources, so we implemented it on the GA, which we will see later. For the actuation on a M2M GW we will use the second option.

One of the benefits of the actuation consists, for example, in eliminating the need for the automatic Bluetooth searches and sensor data captures. As such, to explain the actuation process, in the following lines we will exemplify a use case scenario to capture data from a generic Bluetooth device. The entity requesting the data is a generic Network Application.

The first requirement is to register the Gateway and provide other M2M nodes the information about the smartphone M2M Application capabilities. In order to do so, the registration follows the steps in section 5.1.2 and uses the DESCRIPTOR

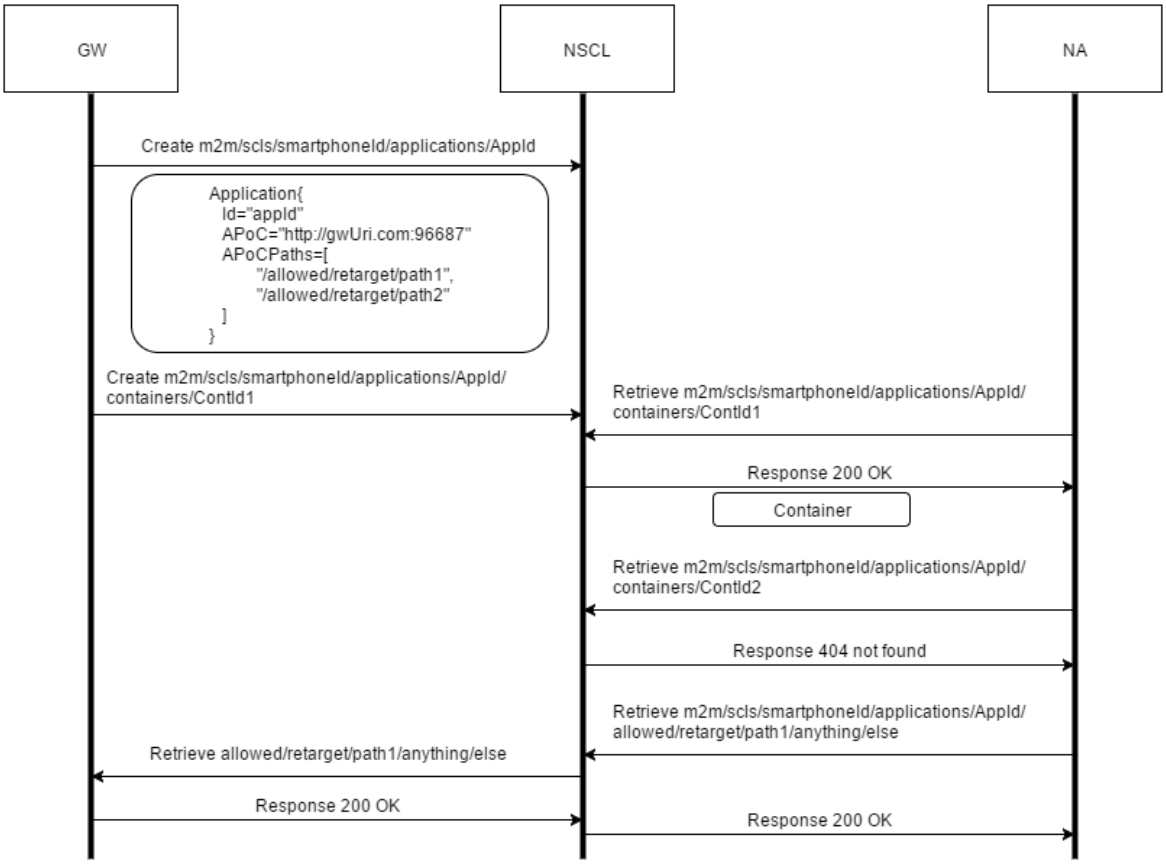


Figure 5.4: Retargeting actuation procedure.

Container of the smartphone M2M Application to publish a Content Instance whose content describes the supported actuation commands, as shown in Figure 5.5.

Other M2M nodes can then retrieve the latest Content Instance or subscribe the DESCRIPTOR Container to acquire the information about the available actuation commands. In this case the commands are BTPAIR and BTSEARCH.

The next requirement, is to discover the Bluetooth devices in the smartphone surroundings. In order to do that, the following steps are executed, as depicted in Figure 5.6:

1. The NA retrieves the latest Content Instance in the Container DESCRIPTOR, to obtain the available commands;
2. The NA subscribes the DATA Container, to be notified of the commands results;
3. The NA sends a BTSEARCH command by targetting the request to an inexistent resource;

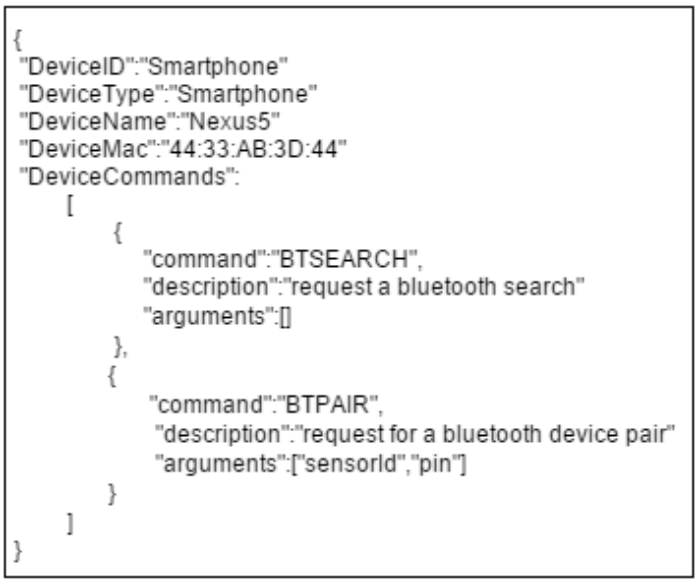


Figure 5.5: Content Instance content describing the smartphone Application capabilities.

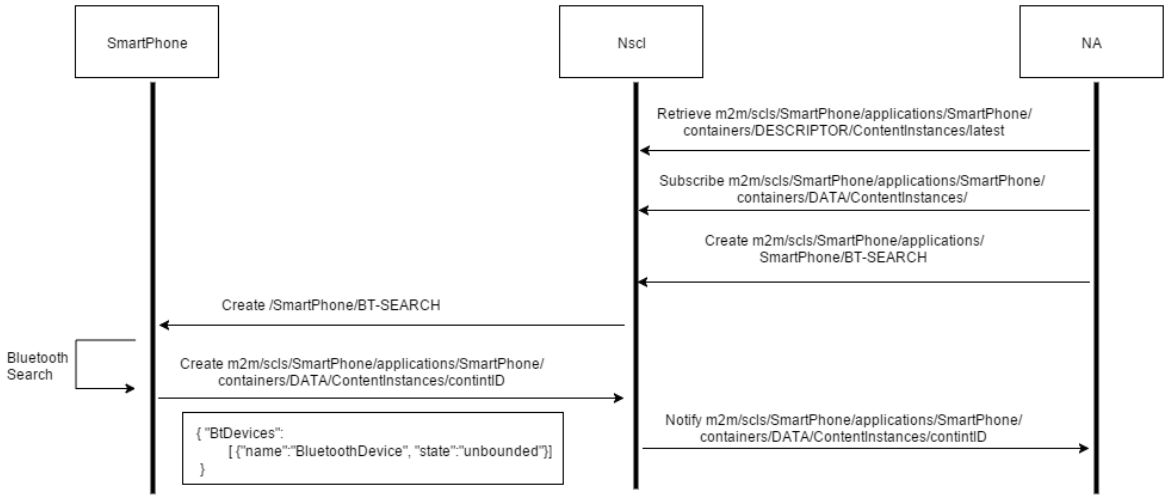


Figure 5.6: Actuation message exchange of a bluetooth search.

4. The NSCL does not recognize the resource and forwards the request to the Gateway;

5. The GSCL does not recognize the resource and interprets the request as a command;

6. The Bluetooth search is performed on the smartphone;

7. The M2M GW creates a Content Instance in the DATA container listing the

Figure 5.7: Actuation message exchange of a bluetooth device pair.

available devices and their paired state;

8. The NSCL notifies the NA with the search result.

At this point, the devices found are not paired. When the NA is notified with the search results, it can select from the list the device to be paired with the smartphone. The process for pairing a device, represented in Figure 5.7, consists on the following steps:

1. The NA sends a BTPAIR command, with the arguments *sensorId* and *pin* as a query string, by targeting the request to an inexistent resource;
2. The NSCL does not recognize the resource and forwards the request to the Gateway;
3. The GSCL does not recognize the resource and interprets the request as command;
4. The Bluetooth device pairing is performed on the smartphone;
5. The Gateway registers the sensor Application containing the containers DATA and DESCRIPTOR;
6. The Gateway creates a Content Instance, describing the Sensor Application properties and available actuation commands, in the Container DESCRIPTOR;

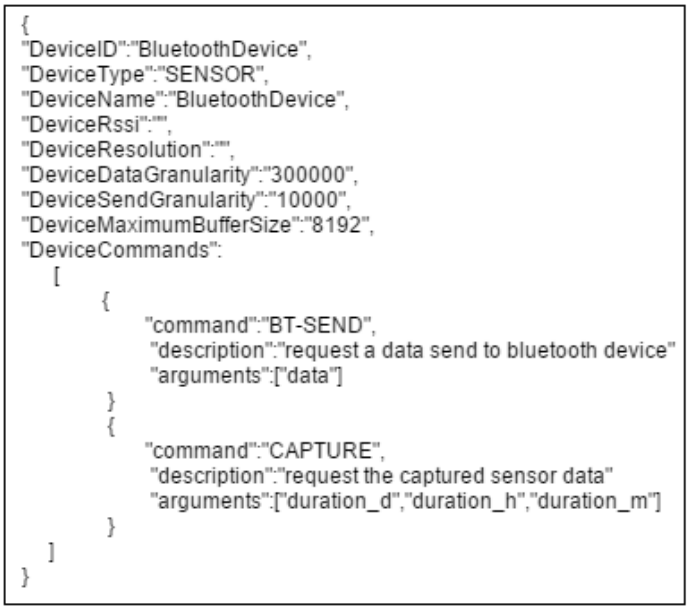


Figure 5.8: Content Instance content describing the bluetooth device Application capabilities.

7. The Sensor Application provides the commands BTSEND, BTCAPTURE and CONFIG;

8. The M2M GW creates a Content Instance in the smartphone DATA container, updating pair state of the available devices list;

9. The NSCL notifies the NA with the updated available devices list.

The final requirement is that the NA informs the Gateway about its interest in the Bluetooth data. At this time, it needs to use the commands provided by the Bluetooth device M2M Application. As seen before, it can retrieve the latest Content Instance, depicted in Figure 5.8, or subscribe the DESCRIPTOR Container of an M2M Application, to acquire the information about the available actuation commands. Then, the process to receive the Bluetooth data , depicted in 5.9,has the following steps:

1. The NA retrieves the latest Content Instance in the Bluetooth device Container DESCRIPTOR, to obtain the available commands;

2. The NA subscribes the Bluetooth device DATA Container, to be notified of the commands results;

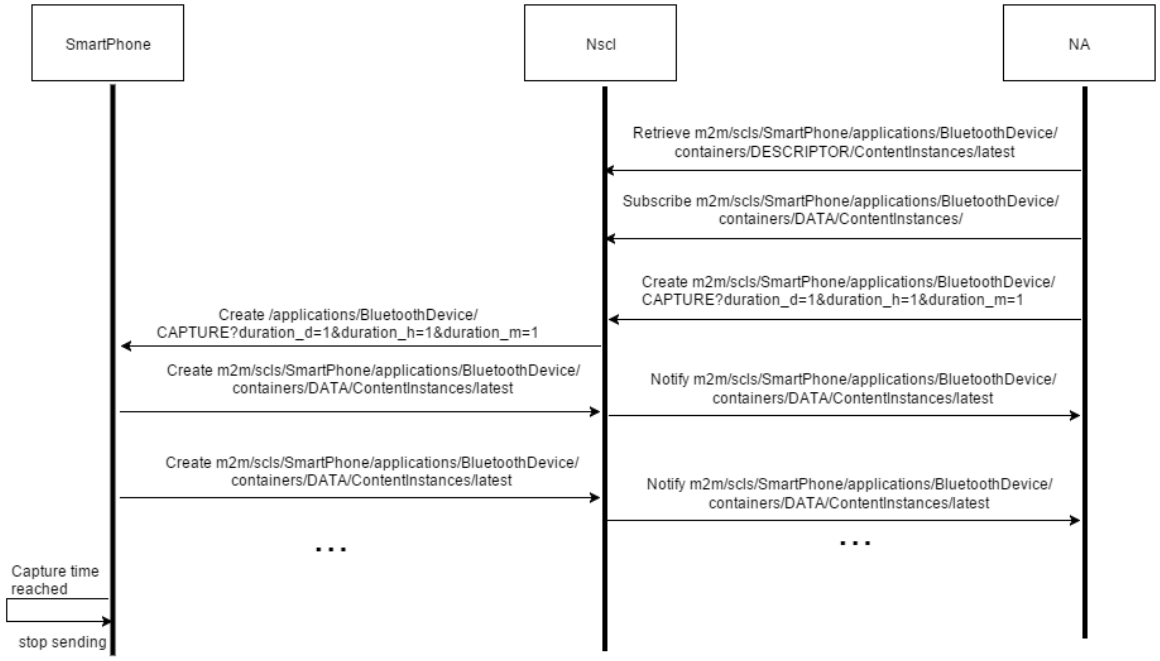


Figure 5.9: Bluetooth device capture.

3. The NA sends a BTCAPTURE command, with the arguments /textitduration_d, *duration_h*, /textitduration_m as a query string, by targetting the request to an inexistent resource;

4. The NSCL does not recognize the resource and forwards the request to the Gateway;

5. The GSCL does not recognize the resource and interprets the request as command;

6. Whenever the sensor is available, the gateway captures and sends its data, for the time interval of duration_d days, duration_h hours and duration_m minutes;

7. The NSCL notifies the NA with the captured data.

5.1.5 M2M GW accessibility

The GW Service will probably be running inside a private network when using the WiFi connection, so it can not be contacted unless the user configures a public IP or maps the router ports. However, the user may not have the knowledge or permissions

to do this. In order to solve this issue, three options were found: UPnP[1], NAT-PMP[2] and Long Polling.

UPnP and NAT-PMP are two protocols that allow the remote configuration of port mapping and port forwarding in a routing device. Routers distributed by network vendors usually support one of these protocols. Most support UPnP since it is an older protocol.

In order to provide accessibility to the GW Service, during the configuration of the HTTP Client and Server the Protocol Manager attempts first to setup the UPnP. If the UPnP is not available, the GW Service proceeds to setup the NAT-PMP. If both Port Mapping protocols fail, the M2M GW should use Long Polling.

5.1.6 Other minor changes

We added other minor changes to the GW Service to improve its performance. We present them next.

NSCL Responses

When creating resources the NSCL sent the resource created in the response. This feature was removed, since it is optional in the ETSI M2M standard. This change greatly reduces the incoming traffic.

Bluetooth Raw Data

Bluetooth data was processed and encapsulated in JSON. Now the Bluetooth data is sent in raw bytes as it is captured, delegating the data processing to the data subscriber. This reduces the JSON overhead in the outgoing traffic and the processing performed in the M2M GW. Furthermore, this also allows the M2M GW to connect to any Bluetooth device with Android support and gather their data without any prior device specific preparation. However, this approach can increase the load of the subscriber, so we assume the subscriber is a an NA capable of handling the extra processing time.

Local Resources

[1] `http://upnp.org/`

When creating and retrieving resources the M2M GW would always contact the NSCL. In the current version, the M2M GW tries to minimize the network communications whenever possible, checking for local resources before contacting the NSCL.

Redundant Threads

Some threads were considered unnecessary since they were mostly idle. Usually those threads would only forward messages. They would receive messages from one thread and send them to another thread. These redundant threads were removed.

Data Compression

In order to improve the Outgoing Network Usage we also added a compression feature. However, compressing resources sent would make them unrecognisable to other M2M entities and, therefore, not M2M compliant. So we could only use compression in the data inside the Content Instance resources. To compress and decompress we used the Deflater and Inflater classes, respectively, provided by Java.

5.2 Gateway Application Library

In this section, we describe the implementation problems and solutions found during the development of the library used to create Gateway Applications in an Android environment, the GALib. The purpose of the GALib is to provide easy means to develop Android M2M Applications.

We will focus mainly in the implementation strategies of the dIa Interface and actuation. A guide can be found in Appendix B.

5.2.1 dIa Interface

The standard states that this component communicates with a GSCL through the dIa reference point, using a RESTful architecture style. A GA to GSCL interaction can be easily made, using the already existent HTTP server in the GSCL, which interpreters

M2M RESTful requests. Since the GA is located in the same device as the GSCL, it only needs to create an HTTP client and send messages to the GSCL port on the localhost. However, the GSCL also needs to be able to contact the GA. Thus, if a smartphone user has five Android applications acting as GA, the device will have six HTTP servers running. This can create an unnecessary load in the smartphone, so other approaches were considered using the Android IPC capabilities. There are two options provided by the Android system itself: Intents and Binders.

5.2.1.1 Intents Method

The Android Intents are messaging objects, used to request actions across application components. They are mainly used to start activities and services and deliver system event broadcasts. There are two types of intents: Explicit and Implicit. The first is used to target the action to a component contained in the application. The second does not specify the component, so it can declare a general action to be handle by any other application in the system.

In order to use Intents to implement the dIa reference point, a broadcaster and receiver must be implemented in both the GSCL and GA. The intent's *action* attribute will describe the M2M operation to be performed and the *extras* attribute will contain any objects relevant to the operation. An example of the registration process can be seen in the Figure 5.10, with the following steps:

- Booth GA and GSCL have a Broadcast Listener setup to filter dIa requests and responses;
- The GA marshals an Object representing an M2M Application;
- The GA broadcasts an intent with the Action corresponding to an M2M Application creation request and the marshaled Application resource in the Extras;
- The Android System searches all applications and finds a matching intent filter in the GSCL service;
- The GSCL receives the intent, extracts the Application resource and starts application creation procedure;
- The GSCL sends to the GA an Intent with the operation results;

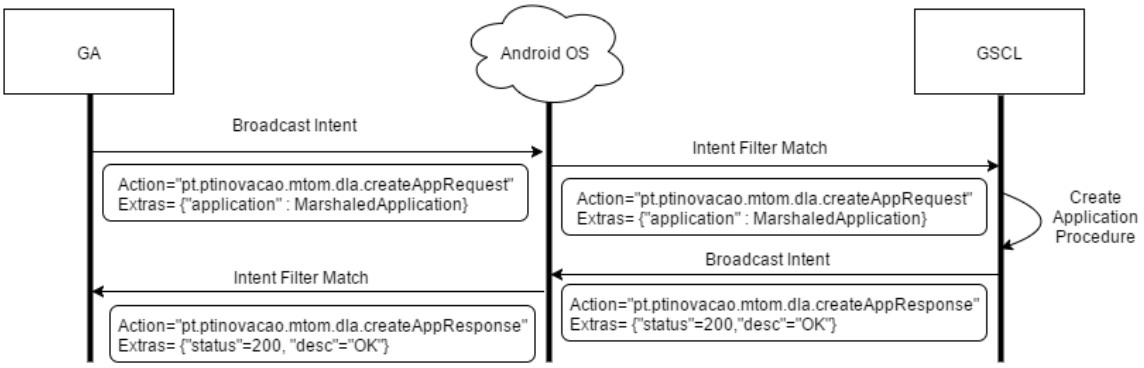


Figure 5.10: Intent based dIa implementation.

- The results are described by an HTTP/CoAP status code integer and description string.

5.2.1.2 Binders method

The Android API provides means to create a communication channel between an application component and an android service. This type of services are called Bound Services and act as servers in a Client-Server relationship with other components.

There are three options to bind to a service: extending a Binder, using a Messenger and creating an Android Interface Definition Language (AIDL). The first option is used when the service runs in the same process as the binding component, so it does not apply to GA/GSCL communication channel. The other two options are more adequate.

The AIDL provides means of decomposing objects into primitives that the operating system can understand and send them across processes. This method allows the creation of a programming interface, defined by an *.aidl* file. This file is used by the sdk tools which automatically creates an abstract class to handle the IPC calls. The *.aidl* file must be present in the service and in the binding component.

The Messenger is a class provided by the Android API and it is based in the AIDL structure. The major difference is that the AIDL requires the service to be able to handle multiple requests at the same time and the messenger has a queue of requests to handle one at a time. To use this method it is only required that a message handler is specified when the Messenger is created.

5.2.1.3 dIa Implementation

To decide which method to use, between the Android Binders and Intents, we consulted the poster by Hsieh et al. [13], where performance of the Android IPC is evaluated by latency, memory footprint and CPU usage. In all three cases the use of Intents has a similar performance for small payloads but it quickly becomes much worse as the payload size grows.

The poster also evaluates the use of SQL-like Content Provider, which could be implemented by having a SQLite database shared between the GAs and the GSCL. However, this method doesn't apply easily to the dIa implementation, due to the lack of synchronization or notification mechanisms.

For this implementation, the chosen method was the Messenger. The major factor for this decision to the fact that the Android API Guides recommend Messenger over AIDL. Moreover, the GA would also need to be multi-threading capable and thread-safe, increasing the complexity of the GA library integration. Since there is no evident necessity for simultaneous request handling, using the Messenger maintains the dIa simplicity and functionally.

The Messenger shares data by sending Message objects. To adapt this object to the dIa Reference Point, the following attributes are used:

- *what* - integer describing the M2M operation to be executed;
- *replyTo* - messenger used by the GSCL to contact the GA;
- *data* - contains an android *Bundle*, with data relevant to the operation.

Depending on the situation, the *Bundle* may contain the following data:

- **source** - An unique identifier representing the Android application, which allows several M2M Applications to be mapped to one single Android application and Messenger;
- **status** - The request results, represented by an HTTP/CoAP status code Integer;
- **desc** - The request results described in an human readable string;

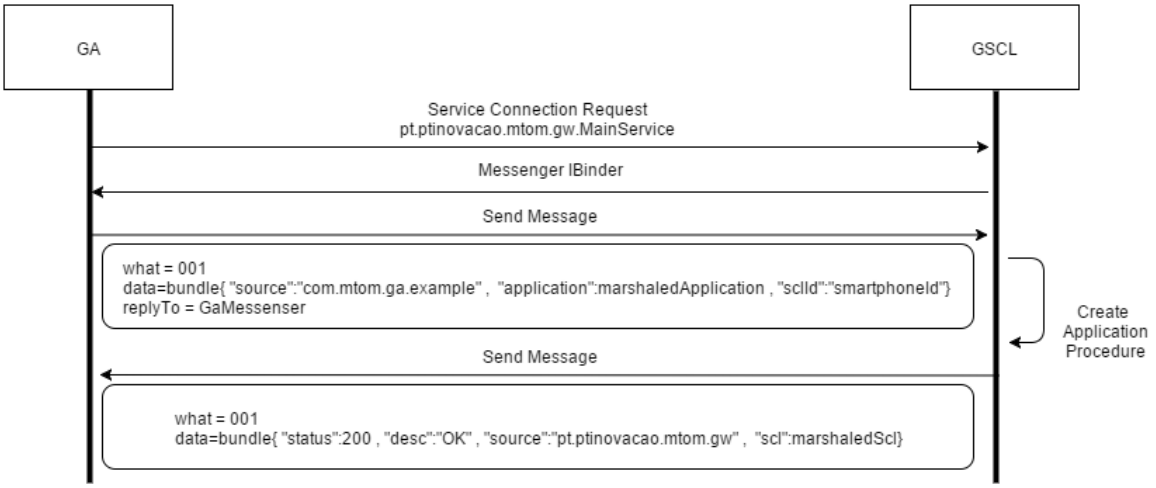


Figure 5.11: Messenger Binder based dIa implementation.

- **scl/application/container** - The resource being created or retrieved, marshaled as string;
- **sclId/applicationId/containerId** - The resources identifiers in the path of the targeted resource.

In Figure 5.11 is represented an example of the GA registration on the GSCL service, with the following steps:

1. The GA requests a Service Connection to the GSCL service;
2. If the GSCL service is running, it returns a Messenger IBinder;
3. The GA receives the IBinder and creates a Messenger connection Channel to the GSCL;
4. The GA sends a message corresponding to the M2M Application creation request;
5. The GSCL starts the application creation procedure;
6. The GSCL sends to the GA a message with the operation results.

5.2.2 Actuation on the GA

Another important feature is that other M2M nodes are able to contact the GA. In other words, it is necessary to extend the Actuation discussed in section 5.1.3 to enable it to reach a GA.

Fortunately, the solution is already discussed in the Retargeting method, in section 5.1.3.3. Since the purpose of Retargeting is to route M2M messages to M2M Applications, it aligns perfectly with this problem.

To use the retargeting in this dIa implementation, only one minor change has to be made: when the Applications' APoC property is the localhost, then the dIa through IPC is used. When the GSCL receives a request, the following steps are taken:

- The GSCL receives a request targeted to an Application resource;
- When the targeted resource exists:
  - The GSCL responds with resource;
- When the targeted resource does not exist:
  - The GSCL responds with an error;
- When the targeted resource matches an APoCPath:
  - If Application APoC is the localhost:
    - Use IPC;
  - Application APoC is not the localhost:
    - Use HTTP Client.

5.3 Network Application Library

The other library developed, the NALib, is used to deploy NAs. The library was implemented in Java, to target stationary devices. In the ETSI M2M standard, the NA is connected to an NSCL through the mIa reference point. Unlike the GAs created with the GA Library, the NA will not be running on the same device as the hosting SCL. So, in this case, we used the HTTP protocol for the communications between

the NALib and the NSCL, similar to the communications between the GW and the NSCL.

The library exposes a set of methods that can be used to trigger M2M requests and manage the NA settings. When a method is called, the library uses the Protocol Manager to create an M2M request in a RESTful format and send it through the available communication protocols. The Protocol Manager manages the communication clients and servers, currently only with support for the HTTP protocol. A guide for the NALib can be found in the Appendix C.

5.4 Tools

In order to implement the GW Service and Libraries, some external tools were used to help the development. In this section we will describe the tools used to this end.

HttpComponents

The HttpComponents[2] project is a toolset of Java components, provided by the Apache Software Foundation[3]. This toolset offers means for creating and managing HTTP client and server applications. It also supports other associated protocols like TLS. The tools are mainly aimed at Java applications, but the project also provides support for Android OS in some releases.

In our development phase, we used HttpComponents to implement the HTTP clients and servers and the TLS authentication, present in the GW Service and NALib.

Jackson

The Jackson[4] is a Java library, also supported in the Android OS, that provides tools to process JSON data. We used Jackson, in the GW Service and NALib, to process Java objects representing M2M resources into JSON and vice versa.

Tomp2p

[2] `https://hc.apache.org/`
[3] `http://apache.org/`
[4] `https://github.com/FasterXML/jackson`

The TomP2P[5] is a library with a set of tools for deployment of Peer-to-Peer (P2P) networks. One of this tools provides means to set up automatic port forwarding using UPNP or NAT-PMP. We used this library to provide accessibility to the GW Service by using UPNP or NAT-PMP.

[5] http://tomp2p.net/

Chapter 6

Proof of Concept

In this chapter we present scenarios where the M2M system was integrated with other projects. A mobile eHealth pilot was proposed to evaluate the end-to-end latency in an IoT environment[16]. The pilot required the deployment of several M2M nodes which make use of the M2M entities developed in the scope of this dissertation. We will use the mobile eHealth pilot as proof of concept. The experiments in Chapter 4 were not repeated due to the NSCL being heavily used by other projects, slowing down the operations. Therefore, the conditions of the first experiment could not be recreated.

6.1 EHealth Scenario

The eHealth scenario consists on a set of users advised to have their daily routine remotely monitored by a healthcare entity. The users put on a wearable Bluetooth device that monitors their heart rate and mobility and start an application on their smartphone every morning. The application is stopped and the device is removed every evening.

The data captured by the wearable device is aggregated by the smartphone's application. Periodically, the application sends the data to a Data Processor system. To send the data, the smartphone has a constant Internet connection using a cellular data plan.

On the Data Processor system, relevant physical indicators are extracted from the data received and are marshaled in a format recognizable by an EHR service. The

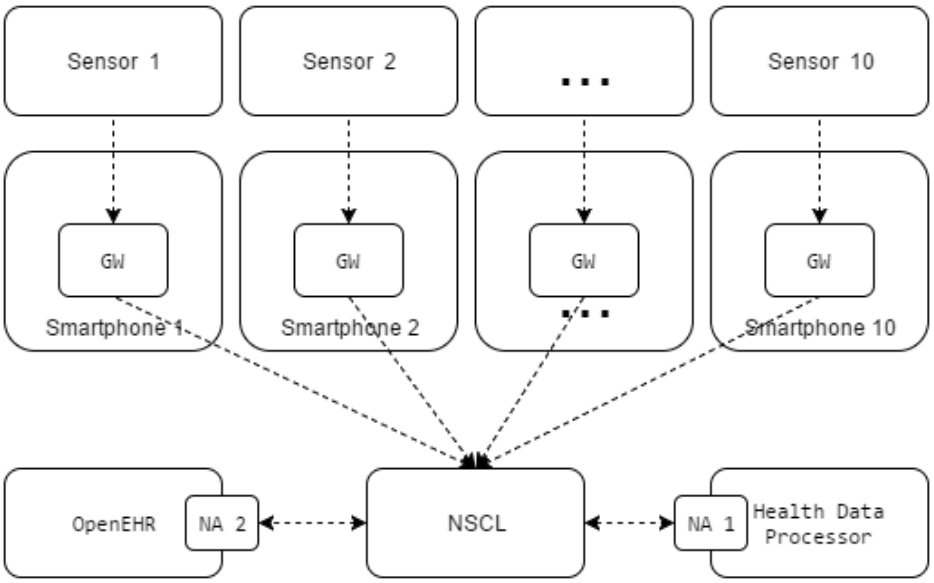


Figure 6.1: M2M pilot ecosystem.

marshaled data is then sent to an EHR provided by a Healthcare entity. The EHR is then able to store and display the processed health data to the interested parties.

6.2 Pilot Deployment

As stated before, this scenario was composed by several M2M nodes that create an ETSI M2M ecosystem, represented in Figure 6.1. The only non-M2M ETSI compliant nodes were the wearable devices. As such, they were interconnected with the M2M network by a GIP. The users' smartphones contained the GW Service, thus the smartphones acted as M2M proxies, providing a GIP, GSCL and connection to the M2M network.

The EHR service was hosted at CINTESIS, FMUP, and followed the OpenEHR[1] non-proprietary standard architecture provided by the OpenEHR Fundation. This standard aims to enable interoperability and openness in eHealth. It also defines models that specify how clinical data should be stored, designated archetypes.

The data captured by the M2M GWs was sent as bluethooth raw bytes. The Data Processor was created to parse and format that data, since the EHR service did not have such capabilities.

[1] `http://www.openehr.org/`

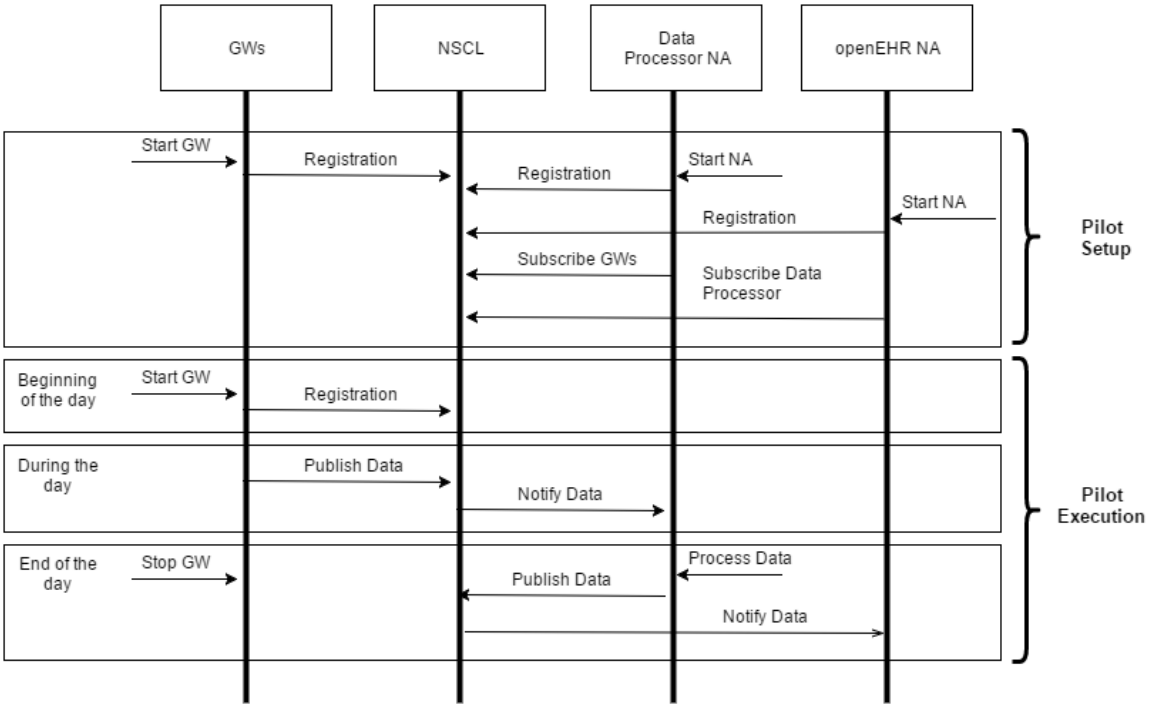


Figure 6.2: Pilot procedures.

Both the EHR Service and Data Processor interacted with the M2M ecossystem as an NA, using the NALib. The M2M GWs and the NAs communicated with each other by publishing and subscribing resources in the NSCL. As such, the NSCL, provided by PT Inovação and hosted at FEUP's servers, acted as a broker for the communications between the different entities.

6.3 Pilot Setup

The pilot was set to run for 3 weeks with the participation of 10 volunteers, acting as patients. Each volunteer was provided with a Moto g2 smartphone with the Android 4.4.4 OS and a Zephyr HxM BT sensor. They should use the Zephyr connected to the smartphone's GW Service at least from Monday to Friday, and preferably for 8 hours a day. The procedures can be separated into two phases, as seen in Figure 6.2.

The setup phase occurred, as a preparation, before delivering the devices to the volunteers. All the M2M GWs were started, in conjunction with the Zephyr devices, to trigger the registration procedure and to make their resource tree available on the NSCL.

Next, booth NAs were also started and registered in the NSCL. In the Data Processor NA registration, a DATA container was created to latter be used for data publication. After the registration, the Data Processor NA subscribed, in the NSCL, the DATA containers belonging to the Zephyr M2M Applications under each one of the smartphones SCLs. The OpenEHR NA subscribed, in the NSCl, to the DATA container under the Data Processor M2M Application.

The execution phase occurred repeatedly every day during the pilot. At the beginning of each day, all users should equip their Zephyr and start the GW Service in their smartphones. When starting the GW Service, the in-built GIP searched for Bluetooth devices. If the Zephyr device was found, the GIP would connect to it automatically.

During each day, each M2M GW continually captured heart rate, distance and speed measurements from the Zephyr, and published the data periodically in intervals of 1 minute. With each publication, the NSCL would notify the Data Processor, that parsed and stored the incoming data.

At the end of each day, the users turned off the M2M GW and stopped publishing data. The Data Processor processed the data of each user and published it in the NSCL as a JSON object in accordance with an EHR archetype compatible with the EHR Service. The NSCL then notified the EHR Service with the processed data for storage, display and analysis.

In order to obtain the performance data, the M2M GWs were monitored mainly in their network usage. We used a method similar to that presented in Section 4.2.

6.4 Results

During the pilot, the M2M GWs were collecting and sending data for 479 hours making a total of 22759 publications. The total amount of sensor data collected reached 430 MB; however, data was compressed before being sent by M2M GWs, amounting only to 43.3 MB in total sensor data sent. Considering the HTTP encapsulation size, the M2M GWs, in total, transmitted 65.6 MB and received 6.3 MB of data.

In the scope of this dissertation, the major concern in this pilot was the reliability and ease of deployment of the M2M network. In the case of the EHR service, the integration was made by a developer not familiar or trained with M2M technologies. The developer was only informed about the library capabilities and the M2M resource

tree, to be able to locate the desired resources. The integration suffered only from minor problems related to the system in which it was running, e.g., system security blocking communication ports.

In the case of GW Service users, there were reports of some sensor device malfunctioning, unrelated to the M2M service itself, in which the Zephyr was not charging correctly. The GW Service was also easy to use since it had reduced human intervention. It facilitated the participation of the volunteers, only requiring them to turn the service on and off.

Chapter 7

Conclusions

Throughout this dissertation we studied the internetworking and applications of the M2M communications based on the ETSI M2M standard. The main objective was to produce and recreate an operational and interoperable M2M network containing different M2M entities. In order to achieve this goal, we made the following contributions:

- We analyzed the ETSI M2M standard, the use cases provided by ETSI and the implementations of other developers.
- We analyzed the implementation of an existing M2M GW and design an experiment to pinpoint performance issues.
- To improve the M2M GW, we updated the the resource mapping and registration procedures, formulated a new method for actuation, added accessibility possibilities with port forwarding, and applied other smaller changes.
- We developed an Android library to ease the deployment of GAs in smartphones equipped with the GW Service. In this library, we implemented a dIa interface though IPC, to make it more viable to have several GAs.
- We developed a Java library to ease the deployment of NAs.
- We deployed a pilot using several M2M GWs and two NAs. All elements worked as intended during the pilot and were easily used by users and developers not acquainted with M2M technologies.

The NALib was also used in other projects with different developers. Since September 2015, it is being used to publish data from UrbanSense[1] sensors and by two partners to subscribe that data. It was also used, in the scope of a dissertation at FEUP, for a Smart Cities API benchmarking.

Some features were not implemented, as was the case of the CoAP and DTLS. These protocols would have improved the communication efficiency. However, the NSCL did not have support for them and we did not find a library that supported both protocols for android devices. Other feature that was not implemented was the long polling, mostly due to time constraints, but also because we did not find an efficient strategy to implement it. In any case, the M2M GW already supports UPnP and NAT-PMP, which are supported by most routers. ssss Finally, there is the matter of security which was not fully accomplished. As of now, the implemented HTTP clients support TLS; however, the servers do not, meaning that incoming messages are readable by everyone. We were also not able to implement the security provided by using the AccessRights resources. Using AccessRights permission to perform operations on M2M resources within an SCL can be granted or denied to an M2M Entity.

Overall, the ETSI M2M standard can be integrated in several applications, and it can be adapted to different use cases and requirements. It can be implemented in all kinds of programmable devices, like smartphones, and it is not dependent on the OS or programming languages. However, to have this kind of flexibility, the standard specifications are abstract and complex. In our implementation, for example, there are still resources and properties that are unexplored. In conclusion, the ETSI M2M standard has many capabilities and opens up a lot of possibilities for future technologies and services.

[1] `http://futurecities.up.pt/site/hybrid-sensor-networking-testbed/`

Appendix A

Acronyms

6LoWPAN	IPv6 over Low power Wireless Personal Area Networks
AIDL	Android Interface Definition Language
APoC	Application Point of Contact
APoCPaths	Application Point of Contact Paths
CoAP	Constrained Application Protocol
DA	Device Application
DIP	Device Interworking Proxy
DSCL	Device Service Capability Layer
EHR	Electronic Health Record
ETSI	European Telecommunications Standards Institute
GA	Gateway Application
GALib	GA Library
GIP	Gateway Interworking Proxy
GSCL	Gateway Service Capability Layer
GW	Gateway
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
IPC	Inter Process Communication
H2M	Human-to-Machine
H2H	Human-to-Human
M2M	Machine-to-Machine
NA	Network Application
NALib	NA Library
NIP	Network Interworking Proxy

NSCL	Network Service Capability Layer
SCL	Service Capability Layer
OGN	Object Network Gateway
P2P	Peer-to-Peer
PoC	Point of Contact
TCU	Telematic Control Unit
xIP	Interworking Proxy

Appendix B

GALib guide

The GA library provides a simple way to create ETSI M2M Applications in Android.The only M2M knowledge needed, is to get acquainted with the resource structure, simplified in Figure 2.2. This is important to be able to target the desired resources, respecting the ETSI M2M Structure. The Android device must also have GW Service running when using the Application.

The main methods of the GA Library can be found in the *GatewayApplication* class. To get an instance of the *GatewayApplication*, use the method *getInstance()*. The properties seen in table B.1 are accessible through the classes' *get* and *set* methods.

Property	**Type**	**Description**
scl	Scl	M2M SCL resource to which the GA is connected
objectMapper	ObjectMapper	Used to map resources to JSON and vice-versa
requestHandler	RequestHandler	Handles incoming requests or notifications

Table B.1: GatewayApplication Class Properties.

The GA may be initialized by using the methodinit(Application app,Context context,RequestHandler requestHandler). Notice that we need to specify an Application, the Android application Context, and a RequestHandler.

The Application object will be used to register the GA. An example of creating an Application is:

```
Application application=new Application();
//define the Unique Identifier of the GA
application.setAppId("ExampleApp");
```

```
//define where the GA server can be contacted. If it is set to localhost,
    the Gw Service server will be used.
application.setAPoC("localhost");
//define when the NA will expire
application.setExpirationTime("2020-12-31T23:59:59.000Z");
```

The RequestHandler will be used to handle the messages that arrive at the GA. This is an abstract class so we have to override its methods. An example of creating a NotificationHandler is:

```
RequestHandler requestHandler = new RequestHandler() {

    @Override
    // method called when the GA is ready for use
    public void onRegistrationComplete(int status, String description, Scl
        scl) {
      //log results
      Log.i(TAG,"Registration Complete: "+status+" "+description);
   }
    @Override
    // method called when a request for the GA arrives.
    public void handleRequest(String target, String payload) {
      Log.i(TAG,"request received..." );
        // log target
      Log.i(TAG,target);
        // log payload
        Log.i(TAG,payload);
   }
}
```

The GA initialization connects to the GW Service and automatically registers the new Application on the GW and NSCL. Now that the M2M Gateway Application is running, we can use a set of methods to perform M2M operations. In the following lines, Some examples of the most important operations are presented.

Containers hold several instances of content. An example of a Container creation is:

```
Container container=new Container();
//define the Unique Identifier of the Container
container.setId(contId);
```

```
//define the expiration time of the Container
container.setExpirationTime("2020-12-31T23:59:59.000Z");
//define the maximum number of data instances in the Container
container.setMaxNrOfInstances((long) 10000000);
//define the maximum Byte size of the Container
container.setMaxByteSize((long) 100000000);
//create the container inside Application "appId" and the SCL of the Gateway
    Service
GwResponse gwResponse=gatewayApplication.CreateContainer(container,
    gatewayApplication.scl.getSclId(), "appId");
//log results
Log.i(TAG,gwResponse.resource);
Log.i(TAG,gwResponse.status);
Log.i(TAG,gwResponse.description);
```

ContentInstances hold one instance of content inside the container. An example of a ContentInstance creation is:

```
//create Content Instance inside the Container "ExampleCont", inside
    Application "ExampleApp" on the SCL of the Gateway Service
GwResponse gwResponse=gatewayApplication.CreateContentInstance("example of
    content for content instance",
    gatewayApplication.scl.getSclId(),"ExampleApp","ExampleCont");

//log results
Log.i(TAG,gwResponse.resource);
Log.i(TAG,gwResponse.status);
Log.i(TAG,gwResponse.description);
```

Subscriptions are used when we wish to be notified about resources alterations. We only need to subscribe a resource once, unless the expiration time is reached. An example of a Subscription creation is:

```
Subscription subscription = new Subscription();
//define the expiration time of the Subscription
subscription.setExpirationTime("2020-12-31T23:59:59.000Z");
//define Subscription type
subscription.setSubscriptionType(SubscriptionType.ASYNCHRONOUS);
//define URI to be notified
```

```
subscription.setContact("localhost");
//Subscribe ContentInstances inside the Container "ExampleCont", inside
    Application "ExampleApp"on the SCL of the Gateway Service.
 GwResponse gwResponse=gatewayApplication.CreateSubscription(subscription,
     gatewayApplication.scl.getSclId(), "ExampleApp", "ExampleCont", true,
     true, true);
//log results
Log.i(TAG,gwResponse.resource);
Log.i(TAG,gwResponse.status);
Log.i(TAG,gwResponse.description);
```

We can also retrieve resources. To retrieve all the containers in an Application we may use:

```

//Retrieve Containers in the Application "ExampleApp"
GwResponse gwResponse=
    gatewayApplication.RetrieveContainers(gatewayApplication.scl.getSclId(),
    "ExampleApp");
List<ReferenceToNamedResource> conts= ((Containers)gwResponse.resource).
    getContainerCollection().getNamedReferences();
//for each reference, obtain the container
for(ReferenceToNamedResource ref:conts){
   //print results
   Log.i(TAG,"Container id "+ref.getId()+" ref " +ref.getValue());
   // retrieve the Container
   Container container=networkApplication.retrieveContainer(
       "/m2m/applications/ExampleApp/containers/" + ref.getId());
   //log results
   Log.i(TAG,ref.getId());
    Log.i(TAG,ref.getValue());

}
```

To retrieve all the Content Instances in a Container we may use:

```

//retrieve the ContentInstances resource in Application "ExampleApp" and
    container "ExampleCont".
gwResponse=gatewayApplication.RetrieveContentInstances(
```

```
    gatewayApplication.scl.getSclId(), "ExampleApp", "ExampleCont");
List<ContentInstance> contints = ((ContentInstances)gwResponse.resource).
    getContentInstanceCollection().getContentInstances();
//For each ContentInstance
 for(ContentInstance contint:contints){
   //log result
   Log.i(TAG," ContentInstance id:"+ref.getId()+"
       content:"+ref.getContent().getValue());
}
```

To stop the GA use the method *finish()*.

Appendix C

NALib guide

The NA Library provides a simple way to create and use an ETSI M2M Application. The only M2M knowledge needed, is to get acquainted with the resource structure, simplified in Figure 2.2. This is important to be able to target the desired resources, respecting the ETSI M2M Structure.

The main methods of the NA Library can be found in the *NetworkApplication* class. To get an instance of the *NetworkApplication*, use the method *getInstance()*. After getting an instance of the class, we may want to use the default configurations or customize some. The properties seen in table C.1 are accessible through the classes' *get* and *set* methods.

After setting our preferences, the NA may be initialized by using the method init(Application application, NotificationHandler notificationHandler, int naPort). Notice that we need to specify an Application and NotificationHandler objects and the port in which the NA can listen to notifications.

The first will be used to register the NA. An example of creating an Application is:

```
Application application=new Application();
//define the Unique Identifier of the NA
application.setAppId("ExampleApp");
//define where the NA server can be contacted
application.setAPoC("http://mobilelab.fe.up.pt:9090");
//define when the NA will expire
application.setExpirationTime("2020-12-31T23:59:59.000Z");
```

Property	Type	Description
application	application	M2M Application resource representing the NA
nsclProtocol	Protocols	Defines which protocol used in the NA Client
nsclUri	String	Defines the URI to contact NSCL
nsclPort	int	Defines the Port to contact NSCL
naProtocol	Protocols	Defines which protocol used in the NA Server
m2mServiceBootStrapURL	String	Defines the URI used to obtain the TLS keys
debugMode	Boolean	Defines if debug output is active
keystore	InputStream	Define the pre-shared keystore
truststore	InputStream	Define the pre-shared truststore

Table C.1: NetworkApplication Class Properties.

The second will be used to handle the messages that arrive at the NA server. This is an abstract class so we have to override its methods. An example of creating a NotificationHandler is:

```
NotificationHandler notificationHandler=new NotificationHandler() {
   @Override
   public Boolean handleNotification(NsclMessage nsclMessage) {
      System.out.println("Received a notification from NSCL");
        //print the request target URI
      System.out.println("Target:" + nsclMessage.getTarget());
        //print the request payload
      System.out.println("Payload:" + nsclMessage.getPayload());
      return true;
   }
};
```

The NA initialization, automatically register a new Application on the NSCl and starts the NA Server and Client. Now that the M2M Network Application is running, we can use a set of methods to perform M2M operations. In the following lines, Some examples of the most important operations are presented.

Containers hold several instances of content. An example of a Container creation is:

```
Container container = new Container();
//define the Unique Identifier of the Container
container.setId("ExampleCont");
//define the expiration time of the Container
container.setExpirationTime("2020-12-31T23:59:59.000Z");
//define the maximum number of data instances in the Container
container.setMaxNrOfInstances((long) 10000000);
//define the maximum Byte size of the Container
container.setMaxByteSize((long) 100000000);
//create container inside Application "ExampleApp"
nsclMessage = networkAplication.create(container,
    "/m2m/applications/ExampleApp/containers/");
//print nscl response
System.out.println("Response: status "+nsclMessage.getStatus()+" payload
    "+nsclMessage.getPayload());
```

ContentInstances hold one instance of content inside the container. An example of a ContentInstance creation is:

```
ContentInstance contentInstance = new ContentInstance();
//Create the Content held by the Content Instance
Content content = new Content();
//Set the Content data
content.setValue("Example content to send".getBytes());
//Set the Content type
content.setContentType("plain/text");
//Add the Content to the ContentInstance and state his size
contentInstance.setContent(content);
contentInstance.setContentSize((long)content.getValue().length);
//create Content Instance inside the Container "ExampleCont", inside
    Application "ExampleApp"
nsclMessage = networkAplication.create(contentInstance,
    "/m2m/applications/ExampleApp/containers/ExampleCont/contentInstances/");
//print nscl response
System.out.println("Response: status "+nsclMessage.getStatus()+" payload
    "+nsclMessage.getPayload());
```

Subscriptions are used when we wish to be notified about resources alterations. We

only need to subscribe a resource once, unless the expiration time is reached. An example of a Subscription creation is:

```
Subscription subscription = new Subscription();
//define the expiration time of the Subscription
subscription.setExpirationTime("2020-12-31T23:59:59.000Z");
//define Subscription type
subscription.setSubscriptionType(SubscriptionType.ASYNCHRONOUS);
//define URI to be notified
subscription.setContact("https://mobilelab.fe.up.pt:9090");
//Subscribe all ContentInstances inside the Container "ExampleCont", inside
    Application "ExampleApp"
nsclMessage = networkAplication.create(subscription,
    "/m2m/applications/ExampleApp/containers/ExampleCont/
    contentInstances/subscriptions/");
//print nscl response
System.out.println("Response: status "+nsclMessage.getStatus()+" payload
    "+nsclMessage.getPayload());
```

The notifications arrive encapsulated in multiple layers of JSON and Base64. To easily extract the Content Instances from the NSCLMessage, we may use:

```
@Override
public Boolean handleNotification(NsclMessage nsclMessage) {

    // Extract a list of strings representing the data contained in the
        ContentInstances
   String[] contentInstances =
       NaUtils.extractContentInstanceContent(nsclMessage.getPayload());
   return true;
}
```

We can also retrieve resources. To retrieve all the containers in an Application we may use:

```
//Retrieve Containers in the Application "ExampleApp"
Containers containers = networkApplication.retrieveContainers(
    "/m2m/applications/ExampleApp/containers/");
//for each reference, obtain the container
for (ReferenceToNamedResource ref:
```

```
   containers.getContainerCollection().getNamedReferences()) {
   //print results
   System.out.println("Container id "+ref.getId()+" ref " +ref.getValue());
   // retrieve the Container
   Container container=networkApplication.retrieveContainer(
       "/m2m/applications/ExampleApp/containers/" + ref.getId());
   //print results
   System.out.println("Response ID "+ container.getId()+" Max number of
       instances "+container.getMaxNrOfInstances());
}
```

To retrieve all the Content Instances in a Container we may use:

```
//retrieve the ContentInstances resource in Application "ExampleApp" and
    container "ExampleCont".
ContentInstances contentInstances =
    networkApplication.retrieveContentInstances(
    "/m2m/applications/ExampleApp/containers/ExampleCont/contentInstances/");
//For each ContentInstance
for (ContentInstance ref:
    contentInstances.getContentInstanceCollection().getContentInstances()) {
   //print result
   System.out.println(" ContentInstance id:"+ref.getId()+"
       content:"+ref.getContent().getValue());
}
```

To stop the NA use the method *end()*.

References

[1] Actility SA. User Guide: ZigBee ETSI M2M REST Interface. 2013.

[2] M. Cheshire, S.Krochmal. Rfc 6886 - nat port mapping protocol (nat-pmp). `https://tools.ietf.org/html/rfc6886`, 2016-1-20.

[3] Cisco. Cisco visual networking index: Global mobile data traffic forecast update, 2015-2020 white paper. `http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html`, 2016.

[4] Asma Elmangoush, Thomas Magedanz, Alexander Blotny, and Niklas Blum. Design of RESTful APIs for M2M services. *2012 16th International Conference on Intelligence in Next Generation Networks*, pages 50–56, 2012.

[5] ETSI. ETSI TR 102 691 - V1.1.1 - Machine-to-Machine communications (M2M); Smart Metering Use Cases. 1:1–49, 2010.

[6] ETSI. ETSI TR 102 966 - V0.9.3 - Machine to Machine Communications (M2M); Interworking between the M2M Architecture and M2M Area Network technologies. Technical report, 2013.

[7] ETSI. Machine-to-Machine Communications (M2M): Use Cases of M2M applications for eHealth. 1:1–29, 2013.

[8] ETSI. TR 102 898 - V1.1.1 - Machine to Machine communications (M2M); Use cases of Automotive Application in M2M capable networks. 1:1–15, 2013.

[9] ETSI. Use Cases of M2M applications for Connected Consumer. 1:1–19, 2013.

[10] European Telecommunications Standards Institute. Etsi ts 102 690 Machine-to-Machine communications (M2M); Functional architecture. 1:1–332, 2013.

[11] Gartner. Gartner says 6.4 billion connected. `http://www.gartner.com/newsroom/id/3165317`, 2016.

[12] Gartner. Technology research | gartner inc. `http://www.gartner.com/technology/home.jsp`, 2016.

[13] Cheng-kang Hsieh, Hossein Falaki, Nithya Ramanathan, Hongsuda Tangmunarunkit, and Deborah Estrin. Performance Evaluation of Android IPC for Continuous Sensing Applications IPC Requirements of Continuous Sensing Applications.

[14] IETF. Rfc 2616. `https://datatracker.ietf.org/doc/rfc2616/`, 2014.

[15] IETF. Rfc 7252. `https://datatracker.ietf.org/doc/rfc7252/`, 2014.

[16] Carlos Pereira, António Pinto, Duarte Ferreira, and Ana Aguiar. Experimental characterisation of iot service composition latency using a mobile ehealth pilot. *Submitted to IEEE Transctions on Network and Service Management*, pages 1–13.

[17] João G. P. Rodrigues, Ana Aguiar, and João Barros. SenseMyCity: Crowdsourcing an Urban Sensor. `https://arxiv.org/pdf/1412.2070v1.pdf`, pages 1–10, 2014.

[18] Manfred Sneps-Sneppe and Dmitry Namiot. About M2M standards and their possible extensions. *2012 2nd Baltic Congress on Future Internet Communications*, pages 187–193, apr 2012.

[19] The Broadband Forum. TR-069 CPE WAN Management Protocol. (November), 2013.

[20] Ricardo Jorge Travanca Morgado. Mobile Healthcare on a M2M Mobile System. Master's thesis, FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO, 2014.