UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale in Applicazioni Telematiche

A Lightweight and Secure Bootstrapping Mechanism for the Internet of Things

Anno Accademico 2014/2015

relatore
Ch.mo Prof. Simon Pietro Romano

correlatore
Ing. Jaime Jiménez Bolonio

candidato
Adriano Di Salvo
matr. M63/294

To Annarita.

«Every once in awhile we may fall on our face,
but we insist on doing what we wanna do.»
(Clifford Lee Burton)

Acknowledgements

At the end of this journey, there are many people I'd like to thank, people that have been fundamental to achieve this result.

First of all, many thanks to my thesis supervisor Professor Simon Pietro Romano, for giving me the opportunity of writing my master's thesis abroad in such a wonderful country, many thanks for his guidance and his advice and many thanks for his encouraging words.

I wish to thank my co-workers at Ericsson's NomadicLab, my supervisor Jaime, who gave me all the assistance I needed, guiding me through the research, design and development of the application, and also for spending some time reviewing this work; a special thanks goes to Roberto, for all the great time we spent together, especially during coffee breaks, and for all the little suggestions he gave me for my thesis; thanks to Jouni and Jan for their perfect organization; thanks to Inés, Mert, Miika, Tero, Nicklas, Petri, Jari, Ari, Oscar, Mohit, Benham, Vesa, Miljenko and Gonzalo, for helping me one way or another; thanks to all NomadicLab.

I wish to say a warm "thank you" to my family, for all the support they gave me; thanks to my father Alberto, my mother Anna and my sister Claudia; thanks to my grandparents, my cousins, my uncles; simply thanks to everyone.

Thanks to my friends for spending so much time, maybe too much, together during my studies. Thanks to my university mates Cristino, Enrico, Nello, Loredana, Simone, Silvio, Antonio, Alessandro, Mirko. Thanks to Jonathan, Altea, Carlo and Fabio for sharing a beer, or two. Thanks to my former music mates Mimmo, Ciro, Gaetano, Antonio and Fulvio for the wonderful time we had together, on the stage and outside it. Thanks to Giuditta and Antonio for the nice holidays time.

Last, but not the least, a very special thanks goes to Annarita. This work is for you, for your patience, your neverending support, your loving words. Even in the most difficult periods, there is always a light that never goes out: you. I love you.

Finally, thanks to you, as a reader, for spending your time for my work; it means a lot to me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Starting from the early '80s, *Machine-to-Machine* (M2M) has become a wide used communication paradigm in industry automation, telemetry applications and other data collection technologies that use both wired and wireless networks to share information. M2M allows simple, constrained devices, such as sensors, to communicate with each other, in order to perform monitoring and control of the environment.

M2M is considered an integral part of the *Internet of Things* (IoT), a relatively new concept that aims to connect any real world thing over the Internet, instead of just computers and mobile devices. In this vision, interconnected things become smart, in the sense that they are able to independently collect data, send it over the network to other devices or systems, where it is processed; the output produced can be ready to interpreted by a human user or can be sent to other machines.

In this scenario, it is fundamental to have a way to control and configure, automatically or with human intervention, deployed devices. Lightweight M2M is a management protocol that allows a server to manage several constrained devices. The strength of this protocol is the exceptional compactness of the messages, based on the *Constrained Application Protocol* (CoAP), that is a general purpose RESTful communication protocol developed for constrained environments.

In order to automate even more the deployment of the sensors, it has become necessary to provide a way to bootstrap a device automatically when turned on, and

in a secure way. The legacy security architectures, such as the Kerberos authentication protocol or the Generic Bootstrapping Architecture, are not suitable for a constrained environment, where power saving and lightweight communications are a fundamental requirement. Thus, it has become necessary to include a mechanism into LWM2M specification that allows a client to gather information about servers and security modes in a straightforward way.

The information gathered by a client must be universally understandable by a majority of device managers. In order to allow semantic interoperability, LWM2M uses a standard data model called LWM2M Objects. These objects constitute uniquely addressable resources that can be read, written or executed with simple requests. IPSO Alliance specify a set of general purpose objects, that can be used on sensor or controller devices to let them exhibit an universal interface that can be used by any server, without changing the underlying application logic.

1.2 Contribution

By the start of this master's thesis, no C implementation existed that fully conformed to the LWM2M Secure Bootstrap specification with DTLS. The contribution of this thesis includes the introduction of a LWM2M Bootstrap Server, containing information on the LWM2M Servers; this Server provides a Secure Bootstrap Service for LWM2M Clients, decoupling the Client-Server dependency; a justification for introducing a security mechanism in LWM2M protocol is also presented, together with the design, a working prototype and an evaluation of the performance of this prototype.

1.3 Thesis Organization

The thesis consists of 6 chapters. This introduction chapter has given an initial overview to the thesis itself, with the problem description and the contribution to the problem. Chapter 2 presents the background information related to this thesis. Chapter 3 gives the design process details and the choices to implement the solution presented in Chapter 4. In Chapter 5, we analyse the prototype implemented and we discuss the measured performances, comparing two different embedded devices. Finally, Chapter 6 contains the review of the thesis, summarizing the achievement of this work and providing suggestions for what could be the future work.

Chapter 2

Background

2.1 Internet of Things and Machine-to-Machine

The concept of Internet, meaning the network connecting several computers, has exponentially grown in scale in the last decades and has gave birth to a complex and diversified network, made of heterogeneous endpoint entities, such as computers and mobile devices; recently, it became necessary to evolve even more this concept, coming to a more general "Internet of Things" vision, in which the endpoints of this new network can be any real world object. Internet of Things is a new concept that is, in a way, the logical evolution of Machine-to-Machine [6].

Usually, M2M refers to a proprietary, vertical solution that provides connectivity between devices in a fixed installation [7, 8, 9]. M2M is based on the idea that connecting machine to form an information interchange network is a way to improve the performance of a complex system. Different electronic, communication and software technologies are combined to realize a M2M system. Innovations in network and communication technologies, improvements in electronics and advent of middleware for machines have enabled M2M to be practically deployed in various domains.

On the other hand, the Internet of Things encompasses a more horizontal and meaningful approach where vertical applications are pulled together to address the needs of multiple industry sectors. Proposed for the first time in 1999 by Kevin Ashton [10], IoT was born by the intuition that in the actual Internet architecture, all the information data comes from humans, by typing, pressing a record button, taking a digital picture or scanning a bar code; the purpose of the Internet of Things is to

automatize the process of collecting data and sending them across the network.

The components and the architectures for IoT are as much heterogeneous as possible [11], making it fundamental to focus on standardization of the interfaces to grant interoperability of several, diverse devices. The 'things' in the IoT (or the 'machines' in M2M) are physical entities whose identity, state (or the state of whose surroundings) is capable of being relayed to an internet-connected IT infrastructure. Almost anything can become a node in the Internet of Things, a concept that can be applied to a big number scenarios, such as: smart homes and domotics, healthcare, smart buildings, manufacturing, automotive and transport, supply chain, smart metering and grids, security and surveillance, military, and so on.

Many ICT companies estimate that the number of devices and objects connected to the Internet will be between 26 and 50 billion by 2020 [12, 13, 14]. The growth and convergence of processes, data, and things on the Internet will make networked connections more relevant and valuable than ever before. This led to the specification of a new class of network, the *Low Power and Lossy Network* (LLN), that has become very important in the characterization of the IoT traffic. A LLN is a network in which both the routers and their interconnect are constrained, have limited resources, high loss rates, low data rates, and instability [15]. This new trend brought the international internet community to elaborate new solutions for IoT routing, using the already existing IP stack, in a lightweight type of network called 6LoWPAN [16], a Wireless Personal Area Network that uses IPv6 and gives low power consumption.

2.2 Communication Patterns for IoT/M2M

In order to enable the Internet of Things, it can be possible to take advantage of many existing communication patterns, used in different protocols, in order to satisfy the special requirements needed in a constrained environment. In this section only two communication patterns will be analysed: the first important model is the Pub/Sub pattern, fundamental for information reporting coming from the devices; secondly, the REST architecture will be described, a way to simply address remote resources with an identifier, and no specification of the service. In the following section the protocols to enable IoT will be described.

2.2.1 Publish/Subscribe

The Publish/Subscribe model (often also referenced as Pub/Sub model) is an architectural design pattern which constitute an alternative to the classic request/response messaging model. In this new model, the entities are not directly communicating exchanging messages, but the communication is decoupled and entrusted to a third entity: the Broker.

In this model, described in Figure 2.1, there are three entities: Publishers (or data/event producers), Subscribers (or data/event consumers) and a Broker (or Dispatcher). When a Publisher generates a message or an event, it does not simply directly sends data to the Subscriber, but it notifies the Broker that a new message/event is being published; the Broker takes the responsibility of forwarding the message to the Subscribers that are interested in it.

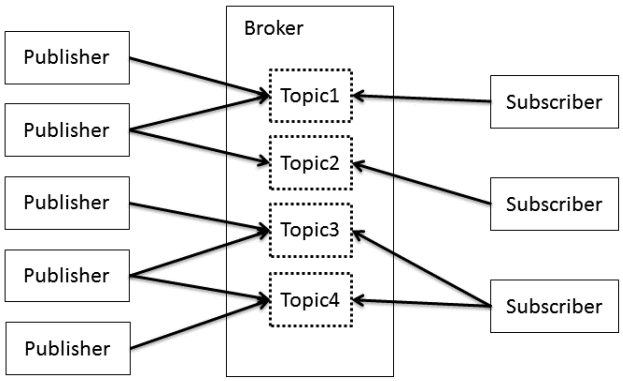

Figure 2.1: Publish/Subscribe Communication Model

Usually, each message belongs to a class, or contains a topic, that basically identifies the scope of the message itself. Publishing a message of a certain class can be translated into selective subscription by the consumers; for example, an entity could register to a broker, telling him to be interested only in a certain king of messages. When the broker receive a new publication from another entity, he forwards the message only to the subscriber really interested in the topic of the message.

There exists several Publish/Subscribe protocols, used in variegated fields, such as online stock quotes, internet games and sensor networks. A relevant protocol is the *Message Queue Telemetry Transport* (MQTT) [17], built to work with limited resources, or in a network with bandwidth limits; currently, MQTT is used in real world applications, such as Facebook Messenger or IECC Railway Signalling Control

System. For its lightweight characteristics, MQTT is particularly feasible to use in a constrained environment, such as IoT Device Management.

2.2.2 REST: REpresentational State Transfer

The *REpresentational State Transfer* (REST) [18] is a client-server architectural style for distributed hypermedia systems. REST refers to a set of principles of Network Architecture, which delineates how distributed resources are defined and addressed. Basically, REST is a way to describe simple interfaces in order to transmit data through a communication protocol without an optional layer (like SOAP) or the session management through cookies.

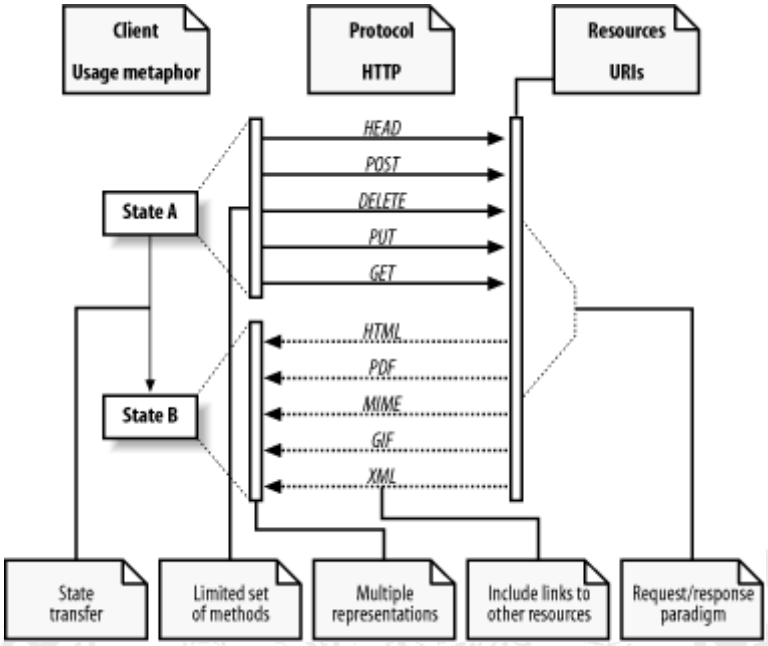

Figure 2.2: Example of a REST Architecture (Source: [1])

REST relies on few key principles in order to allow a better scalability and growth of the Web:

- The state of the application and its functionalities are divided into Web Resources; this defines basically a REST architecture as a ROA, a Resource Oriented Architecture, in contrast with the old SOA, Service Oriented Architecture, a model defined in the early '80s that is resulting too much weighty for modern, constrained applications.

- Each resource is unique and addressable with an universal syntax (URI - Uniform Resource Identifier) for use in the hypertext links.
- Every resource expose an uniform interface for state transfer between client and server; the interface consists of a constrained set of operations (CRUD - Create, Retrive, Update and Delete) and a constrained set of contents.
- The underlying protocol must be client-server, stateless, cacheable and layered.

The most important concept of the REST architecture is the existence of resources, which one can access through a global and unique identifier (URI). For accessing the resources, the components of a network communicates through a standard interface (e.g. HTTP [19]) and the exchange *representations* of these resources, that is basically the document containing the information. Many connectors (such as clients, servers, caches, tunnels, etc...) can reply to a request, but each connector doesn't need to know the previous history of other requests. Hence, an application can interact with a resource just knowing two things: the URI of the resource and the action requested on that resource. The application, however, needs to know also the data format of the information (the representation) given in the reply, typically an HTML, XML or JSON document.

2.3 Communication Protocols for IoT/M2M

In the last few years a new RESTful, simpler protocol was developed in order to face the heaviness of HTTP for IoT communication: the Constrained Application Protocol (CoAP). Using CoAP as transport protocol, a suitable device management protocol for IoT has been developed: *Lightweight M2M* (LWM2M); finally, to grant semantic interoperability new standards are being developed to be used as data model for different applications: those are called IPSO Smart Objects.

2.3.1 CoAP: Constrained Application Protocol

The *Constrained Application Protocol* (CoAP) is a RESTful application layer protocol specifically designed for constrained devices and constrained networks, defined by IETF's *Constrained RESTful Environment* (CoRE) working group [2]. Since CoAP

is based on a REST architecture, it helps to handle remote resources, identified by Universal Resource Identifiers (URIs), that can be manipulated using a subset of HTTP methods: GET, PUT, POST and DELETE. The four methods are chosen to match with the CRUD pattern for persistent storage.

The main goal of CoAP is to offer a simple and lightweight RESTful web protocol, designed for the resource constrained networks; CoAP is an asynchronous message exchange protocol, with UDP (and additionally DTLS for end-to-end security) binding, and also simple proxy and caching capabilities. One strength of CoAP is the low header overhead and parsing complexity: as seen in Figure 2.3, the header requires only 4 bytes to define basic information, such as Protocol Version, Message Type, Token Length (if present), Request Method / Response Code and Message ID (to avoid message duplication). The other optional field, are: Token, used for matching correlated messages, Options, used to carry metadata, and finally, preceded by a byte of "1", the Payload, that is the data to transmit.

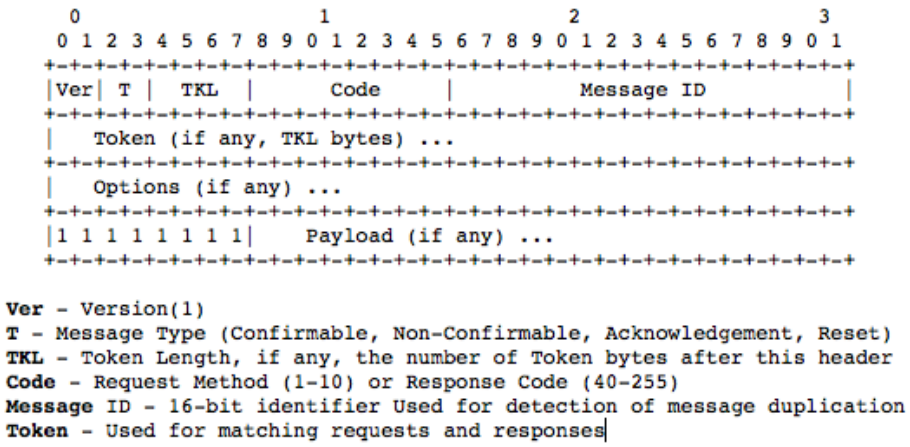

Figure 2.3: CoAP Header (Source: [2])

In CoAP, there are four types of message. *Confirmable Messages* (CON) are used to make a request that needs an *Acknowledge Message* (ACK) as a response; otherwise, *Non-confirmable Messages* (NON) are sent. Finally, *Reset Messages* (RST) are used to notify an endpoint that a CON message has been received but could not be processed. The need of CON and NON messages comes from the unreliability of the transport protocol used, that is mostly UDP; when TCP is used as transport layer protocol, it is possible to use just NON messages. Carried by those message types, there are several message methods/codes, for requests and responses. For requests, it is possible to use the following methods: GET (0.01), POST (0.02), PUT (0.03) and DELETE (0.04). For responses, there are three classes of message codes: Success (2.xx), Client Error

(4.xx) and Server Error (5.xx).

Along with CoAP, a Link Format has been defined in [20], to allow a constrained web server to describe hosted resources, their attributes and other relationships between links. The specification also includes a Resource Discovery mechanism, making discoverable which resources are offered by a CoAP endpoint; this is achieved introducing the well-known resource path, that allows a CoAP endpoint to retrieve all the public resources held by another endpoint. Web interfaces that a Resource Directory supports are being defined, in order for web servers to discover the RD and to register, maintain, lookup and remove resource descriptions [21]. The structure of CoAP URIs is:

$$coap://[hostAddress]:[portNumber]/resourcePath?queryString$$

Or, for resource discovery:

$$coap://[hostAddress]:[portNumber]/.well-known/core$$

Another important feature of CoAP is the Observe/Notify mechanism [22]. The mechanism is described in Figure 2.4; when a CoAP Client wants to observe a CoAP Server resource, it sends a simple CoAP GET Message on the resource, including the Observe Option and a random Token. From the moment the Server receives the request, for every value change in the resource, a notification message with the same Token is sent back to the Client, in an asynchronous and decoupled way. This scheme is a simplified version of the Pub/Sub pattern, where the communication is 1-to-1 and there isn't the intervention of a Broker; in fact, the CoAP Client acts as a Subscriber, subscribing himself to the Server during the "GET Observe" phase, and the CoAP Server acts as a Publisher (and, in part, also as a Broker) for every value change in the sensor, notifying value changes back to the Client.

This mechanism was introduced due to the application of CoAP, designed to be utilized in a constrained environment, such as a sensors network, where there's the need of a native notification mechanism in order to simplify the application logic of the endpoints, not forcing them to implement another different protocol, that will result in adding complexity to the system. To extend the Observe/Notify mechanism to a more generic 1-to-N case, a lightweight CoAP-PubSub Broker is being specified [23].

Finally, in order to facilitate connecting sensors to the internet, using CoAP, new media types have been defined in [24], called the Sensor Markup Language (SenML).

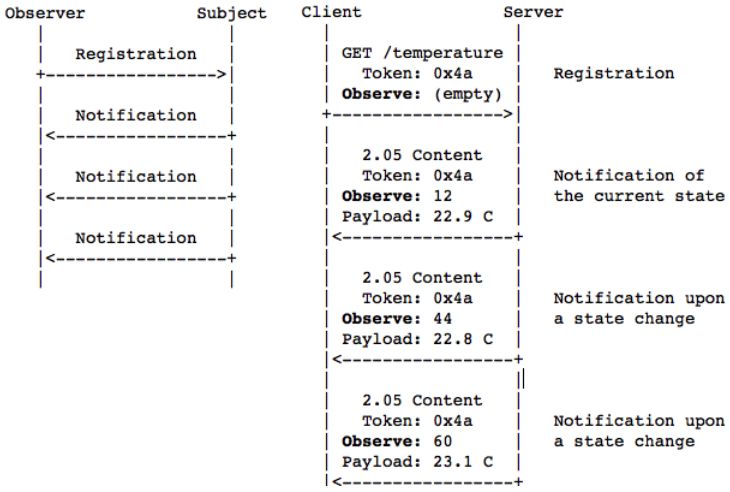

Figure 2.4: CoAP Observe Example (Source: [2])

SenML was designed to allow a sensor to compress information data in a standard format, using processors with very limited capabilities, and also a server to easily parse and process the information carried out through the network. Serializations for this data model are defined for JSON, XML and *Efficient XML Interchange* (EXI).

2.3.2 OMA LWM2M: LightWeight M2M

Device Management is an industry term that denotes the possibility to control a set of mobile devices, normally deployed and working, collecting data from them and sending them remote commands. Device Management is granted by a set of hardware and software tools that allow to monitor and manage single elements in a network. Usually, running on a device there's a software module or component, acting as an interface for the device manager, called the agent. The agent provides useful information on the device, by keeping track of significant parameters. A device manager is responsible for querying agents in the network and also of receiving events from them if something of interest happens. This information can then be used to keep the devices up and running smoothly and also to take appropriate action if something bad occurs.

The Lightweight M2M protocol is a device management protocol specified by the *Open Mobile Alliance* (OMA) [3]. LWM2M can be a solution to improve the growth of the M2M market in several areas, from smart city management, energy management, to location tracking; as LWM2M provides an extensible data model, its use can be expanded into market areas that turn out to benefit the most from its design characteristics [25].

LWM2M is designed not only to run on constrained devices, but it has also many characteristics that make it a feasible solution also for larger systems, such as for network management (in alternative to SNMP [26]). In Figure 2.5 the LWM2M protocol stack is shown. LWM2M relies on CoAP on the application layer for the communication model; LWM2M provides a new semantic for CoAP messages and a new resource model to be used in several contexts. LWM2M could run in mainly two transport layer bindings, that are UDP (or TCP), with optional security given by DTLS (or TLS), and SMS.

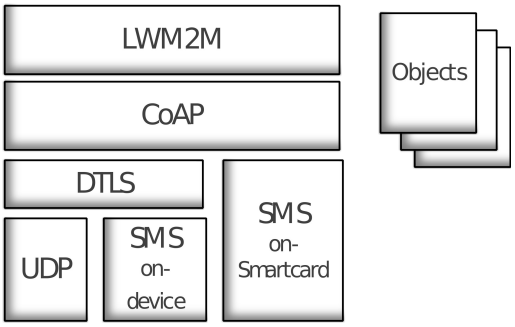

Figure 2.5: The LWM2M protocol stack (Source: [3])

Object and Resource Model

The LWM2M protocol defines a simple resource model where each piece of information made available by the LWM2M Client is a Resource. Resources are atomic entities logically organized into Objects. Resources and Objects have the capability to have multiple instances of the Resource or Object. An Object must be instantiated either by the LWM2M Server or the LWM2M Client before using the functionalities of an Object. After an Object Instance is created, the LWM2M Server can access that Object Instance and the Resources which belong to that Object Instance. A representation of the LWM2M data model is given in Figure 2.6.

Resources are defined per Object, and each Resource is given a unique identifier (*ResourceID*) within that Object. Each Object and Resource is defined to have one or more operations that it supports. A Resource may consist of multiple instances called a Resource Instance as defined in the Object specification. The LWM2M Server can send "Write" operation with JSON or TLV format to Resource to instantiate a Resource Instance. The LWM2M Client also has the capability to instantiate a Resource Instance. Resources are unambiguously addressed using the notation */ObjectID/ObjectInstance/ResourceID/ResourceInstance* (note that *ResourceInstance*

is not needed in case of a single instance resource).

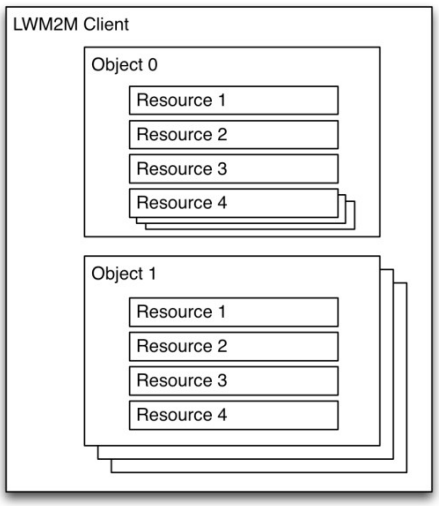

Figure 2.6: Relationship between LWM2M Client, Objects and Resources (Source: [3])

In the OMA LWM2M Technical Specification, a set of 8 objects is defined, that are:

- **LWM2M Security**: a mandatory object, containing security and keying material of a LWM2M Client to establish a connection with a LWM2M Server.
- **LWM2M Server**: this object contains information about a specific LWM2M Server; this object is mandatory to implement.
- **Access Control**: this object is used to check whether the LWM2M Server has access right for performing a operation.
- **Device**: a mandatory object, containing information about the device where the LWM2M Client runs; a LWM2M Server can query the informations through this object, that provides also reboot and factory reset functions.
- **Connectivity Monitoring**: object to monitor parameters related to network connectivity.
- **Firmware Update**: this object enables the management of firmware which is to be updated.
- **Location**: this object carries GPS information about the location of the device.
- **Connectivity Statistics**: this object enables the device to collect connectivity statistics, to be queried by a LWM2M Server.

Architecture and Interfaces

A strong point of LWM2M protocol is the fact that it has a direct mapping between CoAP message types and LWM2M data model, making it possible to extend it to several M2M scenarios already CoAP-compliant. Basically, LWM2M defines an application layer communication protocol between a LWM2M Server and a LWM2M Client, which is located in a LWM2M Device. A Client-Server architecture is depicted in Figure 2.7, where the LWM2M Device acts as a LWM2M Client and the M2M service, platform or application acts as the LWM2M Server.

As already said, LWM2M defines two transport protocols to be used below CoAP: UDP, as mandatory, and SMS bindings. Regarding security, *Datagram Transport Layer Security* (DTLS), described in Section 2.4.3, is used to provide a secure channel for the messages exchanged between Server and Client. DTLS security modes include *Pre-Shared Key* (PSK), *Raw Public Key* (RPK) (Section 2.4.4) or X.509 Certificates [27].

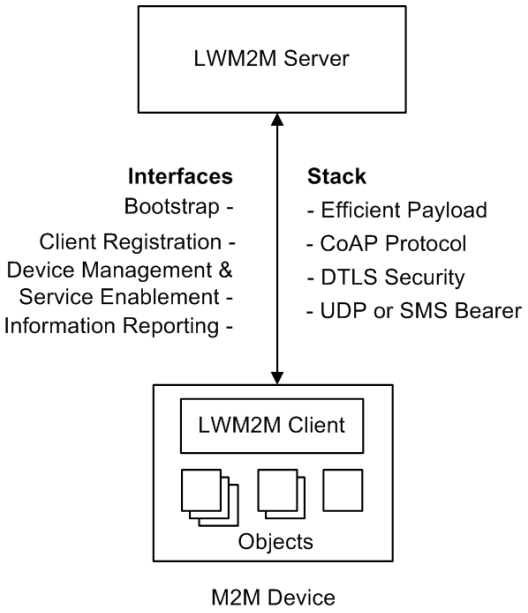

Figure 2.7: The overall architecture of the LWM2M protocol (Source: [3])

LWM2M defines four different interfaces to enable communication between Client and Server, as shown in Figure 2.8. The interfaces are:

- **Bootstrap**: used between a LWM2M Bootstrap Server and a LWM2M Client to provision the Client with initial information, to let it successfully register to one, ore more, associated LWM2M Server. The Bootstrap Server is a particular server,

globally reachable, that contains a Client-Server mapping to properly answer a bootstrap request from a Client.

- **Registration**: used by the LWM2M Client to register to one, or more, LWM2M Server, to provide information about the Client bindings, lifetime and the set of Objects supported client-side. A LWM2M registration is a soft state connection, meaning that has to be updated every once in awhile; otherwise, after the client lifetime has expired, the registration is deleted server-side.

- **Device Management**: used by the LWM2M Servers to access Object Instances and Resources available in the LWM2M Client. The operations defined are "CREATE", "READ", "WRITE", "EXECUTE", "DELETE", "WRITE ATTRIBUTE" and "DISCOVER".

- **Information Reporting**: allows the LWM2M Server to Observe [2] a Resource or Object Instance in the LWM2M Client. When a change occurs, the new value is sent to the LWM2M Server in a Notify message. An observation ends when a Cancel Observation message is sent by the Server.

All the operations of the interfaces are mapped to a particular CoAP message configuration. For example, the Bootstrap interface is enabled querying the "/bs" resource, same as the Registration interface with the "/rd" resource; Device Management interface operations are directly mapped to CoAP messages (GET, PUT, POST and DELETE), or particular combinations of them; finally, the Information Reporting interface is enabled using the "CoAP Observe" option.

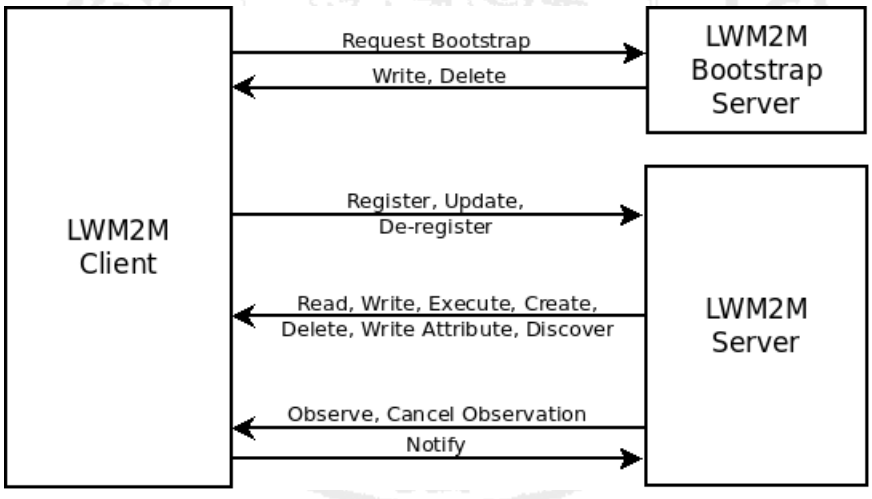

Figure 2.8: LWM2M interfaces

Bootstrap Interface

The Bootstrap Interface is used to provision essential information into the LWM2M Client to enable it to perform the operation *Register* with one or more LWM2M Servers. There are four bootstrap modes supported by the LWM2M architecture: Factory Bootstrap, Bootstrap from Smartcard, Client Initiated Bootstrap and Server Initiated Bootstrap.

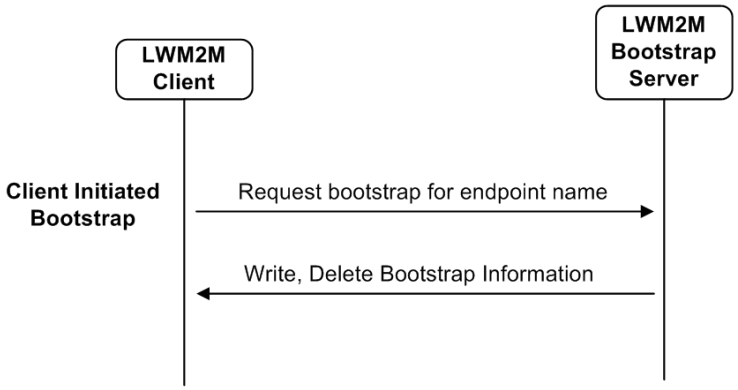

Figure 2.9: Procedure of Client Initiated Bootstrap (Source: [3])

We are interested in the Client Initiated Bootstrap, which message flow is described in Figure 2.9. In this scenario, the client contains already all the informations about the Bootstrap Server, such as URI, Security Mode and any Security Keys, that were pre-provisioned by the manufacturer, and sends a Bootstrap Request to it. The Bootstrap Server replies with the information about the LWM2M Server assigned to the Client (by checking his Endpoint Name).

The operation of sending the information is mapped with a LWM2M WRITE operation in the Bootstrap interface; the targets of the Write operation are new (or existing) instances of Security Object and Server Object. The Bootstrap Server sends a Write request to the Client, conveying in the message payload the content of the Objects in a standard format (usually TLV, or JSON). Optionally, the Bootstrap Server can choose to Delete previous instances of Security and Server Objects. With the Object Instances, the LWM2M Client should be able to perform a Register operation to the new LWM2M Server and continue with the Device Management session.

2.3.3 IPSO Smart Objects

The IP Smart Objects (IPSO) Alliance is a forum formed on September 2008 by 25 founding companies to promote the *Internet Protocol* (IP) as the network technology for connecting Smart Objects around the world [11]. Once that was considered achieved, IPSO Alliance sets a new target to push for semantic interoperability on higher layers; this new objective led to the creation of the IPSO Objects.

IPSO Alliance provides a common design pattern, an object model, that can effectively use CoAP protocol to provide high level interoperability between Smart Object devices and connected software applications on other devices and services. The common object model is based on the LWM2M specification for constrained device management. The object model from LWM2M is reused to define application level IPSO Smart Objects. This enables the OMA Name Authority (OMNA) to be used to register new objects, and enables existing LWM2M compliant device libraries and server software to be used as an infrastructure for IPSO Smart Objects.

IPSO Smart Objects are not fixed objects like in LWM2M; they represent much more a specified collection of reusable resources that has a well-known object ID and which represents a particular type of physical sensor, actuator, connected object or other data source [5]. The reusable resources, which make up the Smart Object, represent static and dynamic properties of the connected physical object and the embedded software contained therein. IPSO Alliance defines a set of IPSO Smart Objects, which conform to the LWM2M Object Model, and which can be used as data objects, or web objects, to represent common sensors, actuators, and data sources. In Table 2.1 there's a list of the basic and gateway IPSO Smart Objects.

2.4 Secure Bootstrap

Devices in the Internet of Things usually are small constrained sensors or actuators that operate autonomously, without user interaction and, in many cases, even without a direct user interface in the form of input devices or displays. This is a design choice, as the devices will not usually need human intervention while operational; for this reason, before a device can connect itself to the network and its services, it needs to be aware of them. To generalize as much as possible the manufacturing process, the information pre-provisioned within the devices must be only a basic set of settings and

Object	Object ID	Multiple Instances?
IPSO Gateway System	1022	No
IPSO Gateway Firewall Rule	1023	Yes
IPSO Gateway Fixed Interface	1024	Yes
IPSO Gateway Wireless Interface	1025	Yes
IPSO Digital Input	3200	Yes
IPSO Digital Output	3201	Yes
IPSO Analogue Input	3202	Yes
IPSO Analogue Output	3203	Yes
IPSO Generic Sensor	3300	Yes
IPSO Illuminance Sensor	3301	Yes
IPSO Presence Sensor	3302	Yes
IPSO Temperature Sensor	3303	Yes
IPSO Humidity Sensor	3304	Yes
IPSO Power Measurement	3305	Yes
IPSO Actuation	3306	Yes
IPSO Set Point	3308	Yes
IPSO Load Control	3310	Yes
IPSO Light Control	3311	Yes
IPSO Power Control	3312	Yes
IPSO Accelerometer	3313	Yes
IPSO Magnetometer	3314	Yes
IPSO Barometer	3315	Yes

Table 2.1: IPSO Smart Objects defined in [5]

credentials [28].

When a new device has to be deployed, there's the need to provision it with initial information for connectivity and to enable a specific service; one simple way could be to perform a manual configuration of the device, but in a constrained environment it is better to automate this process to minimize human intervention. To do that, one possible solution is to authenticate the device to a bootstrap server to get information about the service enablers in the network. In this scenario, the credentials of the device should be manufacturer-provided, and must be globally unique; a way to achieve this, is to add a manufacturer identifier as part of the device identifier (e.g. urn:dev:lmf:5555).

However, this kind of architecture has many security leaks: when data flows in the network in plaintext form, it is very easy to steal information through packet sniffing and impersonate each endpoint with a man-in-the-middle attack, using the stolen credentials. Thus, there's the need to provide device authentication and data encryption in the communication between client and server. One way to do that is to provision into the device a shared secret with the bootstrap server to encrypt data traffic; this solution, however, adds criticality to the distribution of the secret: if anyhow an attacker manages to uncover the secret, the device becomes automatically useless, since it is not possible to burn in it a new shared secret. Another security approach is

to rely on asymmetric cryptography; in this case a public/private key pair is installed into the device. A full certificate installation into the device will add complexity and resource consumption into the device.

For these reasons, usually a Raw Public Key solution is adopted: in this case, there's the need to validate the asymmetric key pair using an out-of-band mechanism. During provisioning, the identifier of the device is collected, for example, by reading a barcode on the outside of the device or by obtaining a pre-compiled list of the identifiers. The identifier is then installed in the corresponding endpoint, for example, an M2M data collection server.

2.4.1 Kerberos

The Kerberos protocol [29] is a network authentication protocol that allows several endpoints to communicate over an insecure network, granting user authentication and data encryption. Kerberos prevents network attacks, such as packet interception or replay attacks, and ensures integrity of the data. Kerberos relies on a symmetric cryptography scheme and requires a trusted third party, that guarantees the identities of the various actors.

Kerberos is based on the Needham-Schroeder protocol [30]. It utilizes a trusted third party for centralizing the key distribution, called *Key Distribution Center* (KDC), that consists of two logically separated entities: the *Authentication Server* (AS) and the *Ticket Granting Server* (TGS). Kerberos utilizes "tickets", that are used to prove users' identities. The AS keeps a database of secret keys; each entity on the network shares a secret key with the AS. The knowledge of the key serves to prove the identity of an entity. To enable the communication between two entities, Kerberos generates a session key, that is used by the two endpoints to communicate.

The protocol is composed of three phases, described in Figure 2.10 and 2.11; a total of four entities participate in the communication: "Alice" represents the Client starting the communication, "Bob" represents the Service Server and the KDC is composed of a AS and a TGS. The phases are:

1. First, the Client authenticates itself sending a login in plaintext to the AS through the network. The reply, encrypted with the shared secret between the Client and the AS, which is called as $K_{A,AS}$, contains a session key for use between the Client and the TGS, which is called as $K_{A,TGS}$, and a ticket encrypted using a shared

secret between AS and TGS, so that only TGS can decipher the information.

2. Next, the Client uses the session key to authenticate itself to the TGS sending the ticket, the identity of the recipient of the message (the Service Server) and finally an encrypted timestamp t (with $K_{A,TGS}$), for time validity check. The TGS replies with a session key between the Client and the Service Server $K_{A,B}$, encrypted with $K_{A,TGS}$, and a ticket that only the Service Server can decrypt, since it has been encrypted using a shared secret between the Service Server and the TGS, $K_{B,TGS}$.

3. Finally, the Client contacts the Service Server, sending it the ticket and a timestamp t encrypted with the session key $K_{A,B}$. The Service Server replies with an encrypted version of $t + 1$ to prove its identity to the Client, being able of decrypt the ticket from TGS, recover the session key $K_{A,B}$ and decrypt the timestamp; after then, a secure session has been established and a secure communication is granted between the Client and the Service Server.

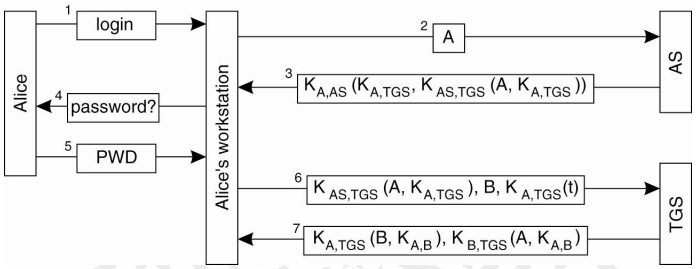

Figure 2.10: Kerberos Protocol user authentication (Source: [4])

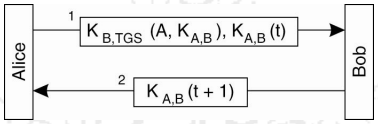

Figure 2.11: Kerberos Protocol service authorization (Source: [4])

Kerberos Protocol results to be a strong authentication protocol, but it doesn't provide authorization mechanism nor denial of service prevention; also, security of the KDC is critical, since it keeps a database of all the credentials of the principals.

2.4.2 GBA: Generic Bootstrapping Architecture

Another way to authenticate an endpoint over the network is using the *Generic Bootstrapping Architecture* (GBA) [31]. Defined in the 3GPP, the main scope of GBA is to

provide authentication and bootstrapping session credentials between a device and a mobile operator.

When a device wants to use a HTTP based service that requires authentication, it will get a HTTP 401 Unauthorized from the server. If the service is GBA capable, this will also be indicated in the response message. The server is in GBA called a *Network Application Function* (NAF). This message triggers the device to perform security bootstrapping with a server, the *Bootstrapping Server Function* (BSF). The device performs a HTTP digest based *Authentication and Key Agreement* (AKA) protocol run with the BSF. The BSF gets an authentication vector from the *Home Subscriber Server* (HSS) to authenticate the device.

The device and the BSF mutually authenticate each other and generate a GBA master key K_s. In addition, the BSF communicates a *Bootstrapping Transaction Identifier* (B-TID) to the device. From the master key K_s, the device can generate a NAF specific key K_{sNAF} and use it to perform HTTP digest authentication through the NAF, along with the B-TID generated through the GBA bootstrapping process. When the NAF gets the authentication response in the form of the digest, it can not yet authenticate the device as it does not have the session key K_{sNAF}; based on the provided B-TID, it identifies the operator with which the device has bootstrapped and queries the operator about the session key.

The BSF can locate the bootstrapping context and generate the matching session key K_{sNAF} from the master key K_s; the BSF then returns the session key to the NAF. This requires that the connection between the NAF and the BSF is protected and that the BSF knows and trusts the NAF. The GBA specification leaves the security implementation details of this interface open. The flow of GBA operations is showed in Figure 2.12.

There are many applications for GBA, for example a "Secure Access in IP Multimedia Services using Generic Bootstrapping Architecture (GBA) for 3G & Beyond Mobile Networks" in [32]. The strength of GBA over a *Public Key Infrastructure* (PKI) is that the interactions between the user and the service are very few, making it a very low cost solution; the credentials are stored inside the SIM card, and are automatically submitted by the device when the bootstrap process starts: the user doesn't have to provide the credentials manually. Moreover, it is very easy to integrate the authentication method into terminals and service providers, as it is based on HTTP's well known *Digest Access Authentication* [33].

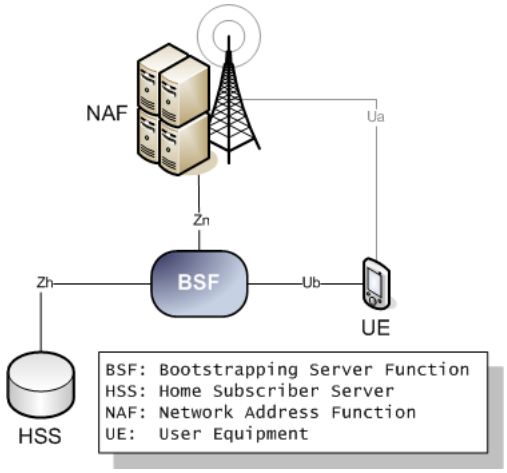

Figure 2.12: The Generic Bootstrapping Architecture

2.4.3 DTLS: Datagram Transport Layer Security

In constrained environments, one of the most used transport layer protocols is UDP, that results to be a logical choice due to its lightweight communication feature, low overhead, and so on. In order to secure UDP channel, the internet community had to introduce a new protocol, that is the UDP equivalent for TLS, which was designed for TCP protocol, and it's positioned between application and transport layers.

The DTLS protocol, as it is at its final 1.2 version [34], provides communications privacy for datagram protocols. The protocol allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery. The DTLS protocol is based on the *Transport Layer Security* (TLS) protocol and provides equivalent security guarantees. TLS [35] has three subprotocols that are used to allow peers to agree upon security parameters for the record layer, to authenticate themselves, to instantiate negotiated security parameters, and to report error conditions to each other. The Handshake Protocol is responsible for negotiating a session and for creating security parameters for use by the application when protecting data.

The main difference with TLS comes from the unreliability of the Datagram Channel, that translates in the possibility of packet losses and reordering. TLS Handshake Protocol breaks in the occurrence of such problems, in a way depicted following:

1. Since the integrity check of the (N+1)-th received packet depends on the sequence number of the N-th received packet, in the case of packet loss the integrity check will fail, breaking the TLS communication.

2. TLS Handshake Protocol assumes that handshake messages are delivered reliably and breaks if those messages are lost.

In order to face these issues, DTLS uses a simple retransmit timer to handle packet loss and sequence numbering to face message reordering; with this mechanism, it is possible to have a DTLS handshake process even when using a UDP channel. The DTLS Handshake Protocol uses the following messages to negotiate security parameters between end-to-end applications:

- **Hello Messages**: The hello phase messages are used to exchange security enhancement capabilities between the client and server. The possible Hello Messages are basically Client Hello and Server Hello, containing the Protocol Version, the SessionID, the CipherSuites supported by the endpoint and finally a cookie, introduced in DTLS to prevent Denial of Service attacks.
- **Server/Client Certificate**: This message conveys the server's/client's certificate chain to the client/server. In case of Client Certificate, the server will use it when verifying the CertificateVerify message (when the client authentication is based on signing) or calculating the premaster secret (for non-ephemeral Diffie-Hellman).
- **Server/Client Key Exchange Messages**: This message conveys cryptographic information to allow the client/server to communicate the premaster secret: a Diffie-Hellman public key with which the client/server can complete a key exchange (with the result being the premaster secret) or a public key for some other algorithm.
- **ChangeCipherSpec**: This optional message is sent by both the client and server to notify the receiving party that subsequent records will be protected under the just-negotiated CipherSpec and keys. The ChangeCipherSpec message is normally sent at the end of the DTLS handshake.
- **Finished**: The Finished message is the first one protected with the just negotiated algorithms, keys, and secrets. Once a side has sent its Finished message and received and validated the Finished message from its peer, it may begin to send and receive application data over the connection.

With DTLS, it is possible to establish a secure channel through only 6 message flights. Each flight is a group of contiguously sent handshake messages, as shown in Figure 2.13. Being utilized in constrained environments, many efforts to compress even more the DTLS Record and Handshake headers have been made [36].

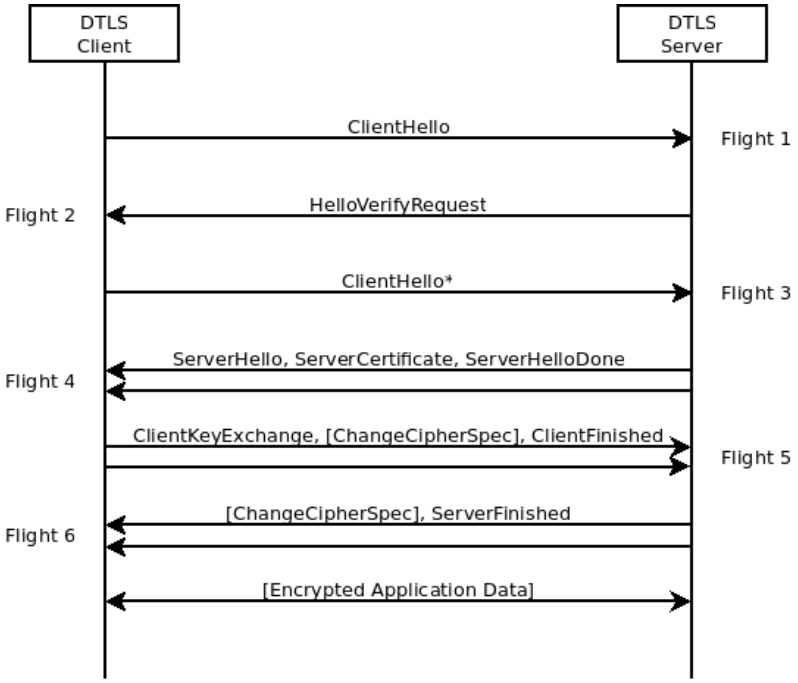

Figure 2.13: DTLS Handshake Protocol flights

2.4.4 RPK: Raw Public Key

As stated before, one of the most attractive choice for security is the Raw Public Key Encryption. The strongest point of this approach is avoiding transmitting the entire X.509 Certificate, containing a huge amount of data, not suitable for IoT environment; moreover, an entire certificate is yet much more complicated to be parsed. Raw Public Key approach, instead, grants a minimum overhead to the constrained device; this mechanism, however, only provides authentication when an out-of-band mechanism is also used to bind the public key to the entity presenting the key. In [37] two TLS/DTLS certificate extensions are defined, to support Raw Public Key Encryption for end-to-end security.

In the field of Public Key cryptography, one of the most famous Public Key Cryptosystem was RSA [38, 39]. Based on number theory, RSA strength was based on the difficulty to factorize and the easiness to multiply numbers; in fact, in order to uncover a private key, it is necessary to find out which pair of prime numbers factorize the public key. For large numbers, however, this process becomes computationally challenging. As the computation power of modern calculators grows year by year, it's necessary to extend the dimensions of the key in order to avoid security breaking; this is not a sustainable situation for mobile and low-powered devices that have limited computational power. The gap between the difficulty of factoring large numbers and

multiplying large numbers is shrinking as the number (i.e. the key's bit length) gets larger.

All this means is that RSA is not the ideal system for the future of cryptography. New cryptographic algorithms were developed during the past decades, not any more based on number factorization, but on elliptic curves [40]. The mathematical problem underlying this new approach is the elliptic curve discrete logarithm; this means that for numbers of the same size, solving elliptic curve discrete logarithms is significantly harder than factoring. Since a more computationally intensive hard problem means a stronger cryptographic system, it follows that elliptic curve cryptosystems needs shorter keys to provide the same security level than RSA and Diffie-Hellman key exchange algorithm [41, 42]. For instance, an ECC system using a key size of 160 bits has the same strength of an RSA system using 2048 bit long keys.

During the years many cryptographic algorithms based on elliptic curves were developed [43], such as *Elliptic Curve Diffie-Hellman* (ECDH) for key exchange. Among them, a suite has been chosen to be the most appropriate to work in a constrained environment, that is TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8 [44, 2, 45].

Chapter 3

Design

The goal of this thesis is to show how the LWM2M protocol could be extended adding end-to-end security through a secure transport protocol; moreover, to automatize the process of deploying new devices, a Bootstrap Server is needed to convey all the information needed by the Client to find its Server. During the thesis research, it has been found necessary to focus on aspects such as secure bootstrapping, asymmetric cryptography and semantic interoperability. In Figure 3.1 a typical M2M scenario is showed, where there are multiple M2M devices, which will support the LWM2M protocol running a LWM2M Client. The manager of the M2M devices is located in the Internet and runs at least one LWM2M Server; a bootstrapping service is also located in the Internet and runs a single LWM2M Bootstrap Server instance. The manager and the M2M devices are connected by gateways, which are transparent in the LWM2M layer. A very similar approach is being used in Capillary Networks project [46].

In a typical scenario, the LWM2M Bootstrap Server needs to be deployed in a fixed installation, e.g. in the "Home Connectivity" domain as an alternative to GBA (Section 2.4.2). A LWM2M Bootstrap Server is a node that stores information about Clients and Servers, in order to point a device to the right device management service. It is responsible of answering to bootstrapping requests coming from LWM2M Clients, erasing information about LWM2M Bootstrapping Servers and rewriting data about LWM2M Servers on the device.

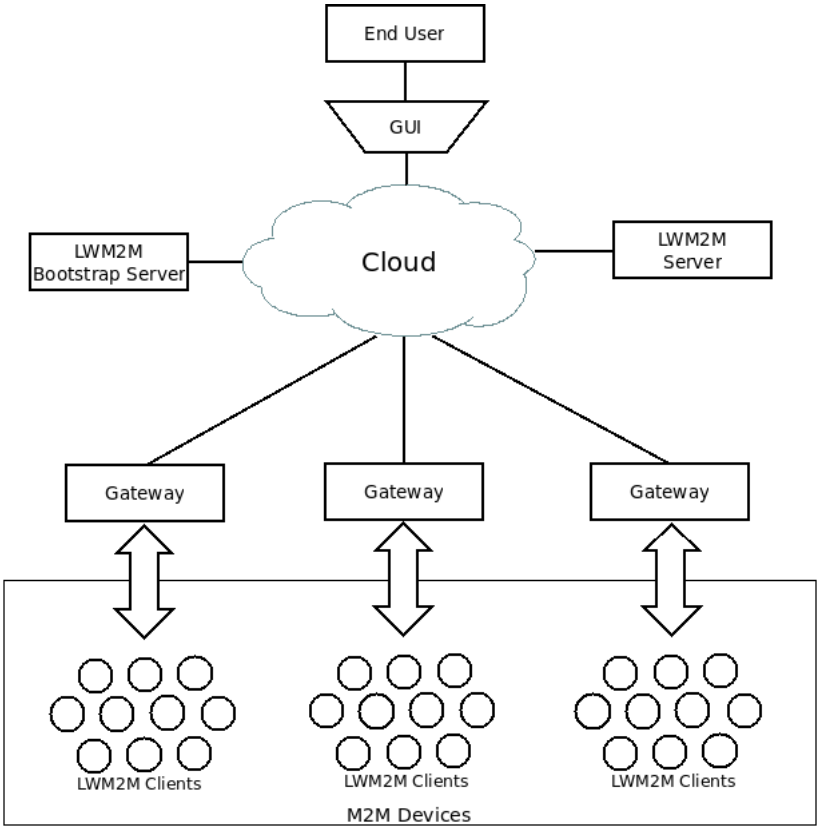

Figure 3.1: M2M Architecture

3.1 Motivation and Use Cases

LWM2M is a very simple device management protocol, suitable for constrained devices. For this reason, it has to provide a native bootstrapping mechanism, compliant with the lightness of the protocol, and give a wide range security support at the application layer. This thesis work aims to implement the bootstrap interface of OMA standard, adding also a security layer, for the following reasons:

- **Device Bootstrap**: LWM2M Clients are assumed to be run in constrained devices; usually, these devices are small sensors provided by a manufacturer, responsible of pre-provisioning all the information needed by the device to be deployed. Hence, devices should not be manually configured; instead, a bootstrapping procedure is needed, in order to simplify the deployment of a large number of devices. LWM2M Clients cannot point directly to a specific LWM2M Server, since it is not possible to forecast to which server it will be associated. In order to generalize the manufacturing process, a LWM2M Client must point to a LWM2M Bootstrap Server, ideally an entity that resides in a fixed installation and it's glob-

ally reachable and addressable. When a device is bought, a new association in the LWM2M Bootstrap Server must be added, in order to point the relative LWM2M Client to the right LWM2M Server.

- **Security mechanisms**: if the standard bootstrapping and registration procedure for a new device is executed in plaintext communication, it becomes very easy for a malicious user to sniff packet or impersonate a fake client, e.g. sending a fake temperature value by playback (and IP spoofing) attack. LWM2M, using UDP as transport layer protocol, can be secured using DTLS with three options: Pre-Shared Key, Raw Public Key and X.509 Certificate. Raw Public Key mode constitutes a good trade-off between resource consumption and security robustness. Moreover, using asymmetric encryption opens the possibility of implementing an authentication mechanism, based on identity verification through Public Key hash (i.e. using SHA-256, or its truncated version).

- **Semantic Interoperability**: as part of the work, it could be interesting to add semantic interoperability in the application being developed; while LWM2M standard objects are meant to be used for device management, there is the need to address sensors' values as resources readable by any type of server. IPSO Smart Objects are used by the application for that purpose; for example, a temperature sensor attached to a device would expose a standard IPSO Object, with ID 3303 and 5700 as Resource ID to read sensor's current value. With this specification, every LWM2M Server receiving a registration request from a LWM2M Client will know that on the device there is a temperature sensor and that the temperature value can be read by asking (CoAP GET) for resource /3303/0/5700.

- **Web User Interface**: in order to simplify the management of multiple devices for a human administrator, a *Web User Interface* (WUI) for Capillary Network has already been developed [47]. LWM2M Server and LWM2M Bootstrap Server need to exhibit an interface to communicate with a generic Web Server. The WUI provides the possibility to show devices on a map, and interact with them performing read, write and execute operations on the resources.

The following Use Case summarizes well the scope of this thesis work:

Name Adding a new device.

Brief Description A user wants to add a new out-of-the-box device to his sensor network.

Actors There are three actors:

- The final user, that is both the owner of the device and the device manager.
- The device, running an instance of the device management agent (the LWM2M Client).
- The server, running instances of LWM2M Server, LWM2M Bootstrap Server and the Web Server for the User Interface.

Preconditions The device needs to have information about the Bootstrap Server (IP address and Security Modes supported); the device needs also to have a valid Raw Public Key pair and a QR Code to be scanned by the user. The keys and the QR code are pre-provisioned into the device during the manufacturing process.

Basic Flow This is the basic flow for the use case:

1. The user activates the out-of-band public key validation mechanism.
2. The device is turned on by the user.
3. LWM2M Client establish a DTLS session with the LWM2M Bootstrap Server.
4. LWM2M Client sends a bootstrap request to the LWM2M Bootstrap Server.
5. LWM2M Bootstrap Server sends a notification to the Web Server.
6. The user approves the bootstrapping request.
7. LWM2M Bootstrap Server bootstraps the LWM2M Client.
8. LWM2M Client establish a DTLS session with the LWM2M Server.
9. LWM2M Client registers to the Device Management Service.
10. LWM2M Server sends a notification to the Web Server.
11. The Web Server shows the device on the map. The user can interact with the device.

Post Conditions The device must be correctly registered to the LWM2M Server, and it must be possible to perform operations on it, such as Write, Read and Execute, from the Web User Interface.

When a user wants to deploy a new device, he must firstly activate the out-of-band mechanism to validate the device's public key. The QR code printed onto the device is scanned by the user using a smartphone, that is for example running a LWM2M App connected with the Web Server. The notification of this operation is received by the

Web Server and it's showed on the User Interface. After the notification appears on the screen, the user can turn on the device; at this point, the LWM2M Client application on the device should start automatically, getting internet connectivity and starting a bootstrapping session. The LWM2M Client first starts a DTLS session with the LWM2M Bootstrap Server, negotiation all the supported security parameters; then it sends a bootstrap request using LWM2M protocol. The Bootstrap Server replies with an ACK code and then erases Security and Server Objects at the Client (containing informations about the Bootstrap Server). When it receives a bootstrapping request, the LWM2M Bootstrap Server communicates through the Web Server interface, sending information about the device (i.e. the client's identifier). The user must check if the information at the User Interface matches; if the user approves the new client, the Web Server sends a command to the LWM2M Bootstrap Server. Then, the LWM2M Bootstrap Server performs a Write operation on the LWM2M Client, writing new Security and Server Objects. At this point, the LWM2M Client sends a registration request to the LWM2M Server; the LWM2M Server approves the request and sends a notification to the Web Server, showing the new device on the map. This is a generic use case and could be applied both for devices (such as sensors) and gateways.

3.2 Objective and Requirements

The main objective of this thesis work is to enable device management for IoT in a secure way, automatically. Information exchanged during the bootstrap phase, and all the management session, should be protected. The application should account for interoperability issues with other implementation of LWM2M protocol (diverse clients communicating with diverse servers).

Functional Requirements

The functional requirement for the application to be developed should be:

1. **Automatic bootstrap**: LWM2M Client and LWM2M Bootstrap Server should perform an automatic bootstrap procedure, as the Client starts.
2. **Selective bootstrap**: the user of the application should be able, optionally, to select which Client is to be bootstrapped, and which is not.

3. **Semantic interoperability**: different implementation of LWM2M Server should be able to read Client's objects, understanding their meaning; different implementation of LWM2M Client must be able to bootstrap and register to the device management application, exhibiting a common interface.

4. **Randomly generated key pairs**: each time a LWM2M Client is bootstrapped, a new key pair has to be randomly generated by the LWM2M Bootstrap Server; optionally, key pair information of LWM2M Server and LWM2M Bootstrap Server must be stored in a configuration file.

5. **Trust anchor for Raw Public Key**: the way CoAP is using Raw Public Key requires no certificate, so the key pair is not validated by any third party trust anchor; the public key must be bound to the LWM2M Client using an out-of-band validation mechanism.

6. **Protection against network attacks**: the bootstrap functionality should provide robustness against well-known network attacks, such as information eavesdropping, session hijacking, denial of service or man-in-the-middle attack.

7. **Robust bootstrapping**: the bootstrap functionality should account for failure or sleeping node situations, as well as packet loss or connectivity breaks.

Security Requirements

In a network communication, it is paramount to protect the information that is sent through the network. Many security mechanisms have been developed during past years. The most common mean to prevent information disclosure or identity theft is to encrypt data using asymmetric cryptography. During this thesis work, it has been found necessary to add security capabilities to the IoT prototype. For this reason, the security requirements for the application are:

1. **Confidentiality**: the assurance that information is not disclosed to unauthorized individuals, programs or processes.

2. **Integrity**: information must be accurate, complete and not altered by unauthorized modification; this must be guaranteed when data are stored and when they are transmitted.

3. **Non-repudiation**: message enciphered with private key came from someone who knew it.

4. **Authenticity**: only the owner of the private key knows it, so data enciphered with private key must have been generated by the owner.

5. **Availability**: most information needs to be accessible and available to users when it is requested so that they can carry out tasks and fulfil their responsibilities.

3.3 Architecture

The architecture of the application is based on the aforementioned requirements; the implementation is a distributed application for device management based on LWM2M 1.0 technical specification [3]. The architecture is described in Figure 3.2. We are trying to build an application on top of DTLS security layer; the application transport protocol used is CoAP; the device management protocol is LWM2M. In addition to this, IPSO Smart Objects (Section 2.3.3) are used for data model, as requested by the specific application. The lower layers are UDP for transport, IPv6 for routing and WiFi (IEEE 802.11) [48] or IEEE 802.15.4 [49] as datalink and physical layer.

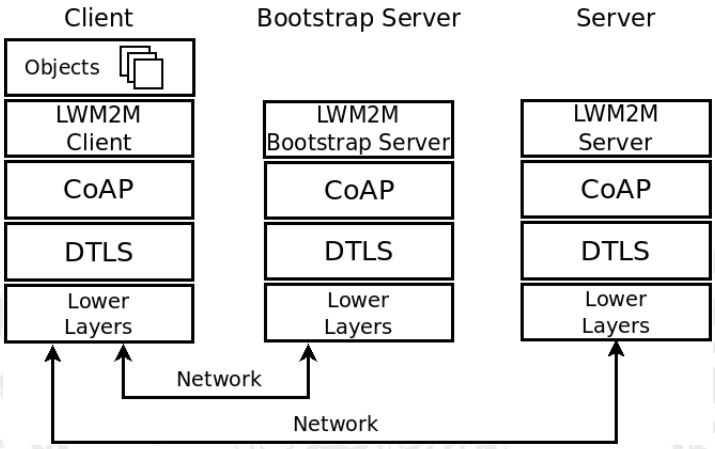

Figure 3.2: LWM2M Design Stack

The LWM2M Bootstrap Server is an entity whose only function is the configuration of LWM2M Clients. In a generic LWM2M architecture, there are several entities of each kind; an example is showed in Figure 3.3: each LWM2M Client at startup points at a LWM2M Bootstrap Server instance. The instance pointed at by a specific Client depends on the data pre-provisioned into the device during the manufacturing process. For this reason, a LWM2M Bootstrap Server is a critical endpoint of an IoT network: it must have an high availability, a fixed IP address, and must be reachable from almost every point of the network. In case a LWM2M Bootstrap Server stops running and

cannot be restored, it is necessary to change the information of the Client (i.e. through a firmware update issued by the manufacturer itself).

Once a LWM2M Client is configured correctly, it will be able to register to one or more LWM2M Servers: there are no constraints to the number of Servers to which a Client could be connected to simultaneously. Hypothetically, a LWM2M Bootstrap Server could write information about several LWM2M Servers in a Client, allowing many Clients to connect to more than one Server forming an IoT device management network.

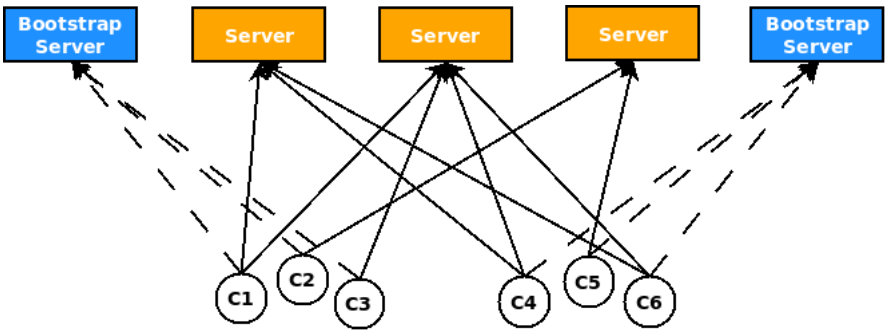

Figure 3.3: LWM2M Bootstrap Server function

Since a Client could be registered to several Servers, it is necessary to implement a simple authorization mechanism on the resources. LWM2M standard provides a very simple *Access Control List* (ACL) mechanism, implemented through a particular LWM2M Object; each instance of this object is connected to a particular instance of another object, and stores information on whether an action on that object instance could be performed or not by the Server. On every request, the LWM2M Client must check the ACL and act accordingly. Other security features can be achieved with the support of the DTLS layer. DTLS provides some security mechanisms, adding an handshake phase in which two endpoints can negotiate the security parameters needed for further communication. Once the handshake is complete, all the LWM2M messages are encapsulated into DTLS application data packets.

During the design of the application, several choices were made. The protocol used for device management, LWM2M, was chosen for its lightness compare to other device management protocols; it is possible to have a wide range of management features using only the small header of CoAP and a very simple data model for storing information. Moreover, the native bootstrapping mechanism provides an easy way to provision a new device and make it part of the network without using other procedures, external to LWM2M.

For the security layer, several alternatives were analysed. The protocol that best fit the requirements of our application was DTLS, being a simple security protocol based on UDP. DTLS has proven to be the best choice, since it provides confidentiality, integrity and authentication to the endpoints with a relatively small overhead, compared to the alternatives.

3.3.1 Secure Bootstrap

To enable secure bootstrapping, it has been necessary to follow the LWM2M technical specification; in particular, the Bootstrap Interface has been implemented. In a client initiated bootstrap, the LWM2M Client asks the LWM2M Bootstrap Server, through a CoAP POST message, for a specific resource (/bs), along with a query string filled with the urn of the endpoint. A bootstrap request URI appears in the form:

$$coaps://www.example.com/bs?ep = urn:dev:lmf:5555$$

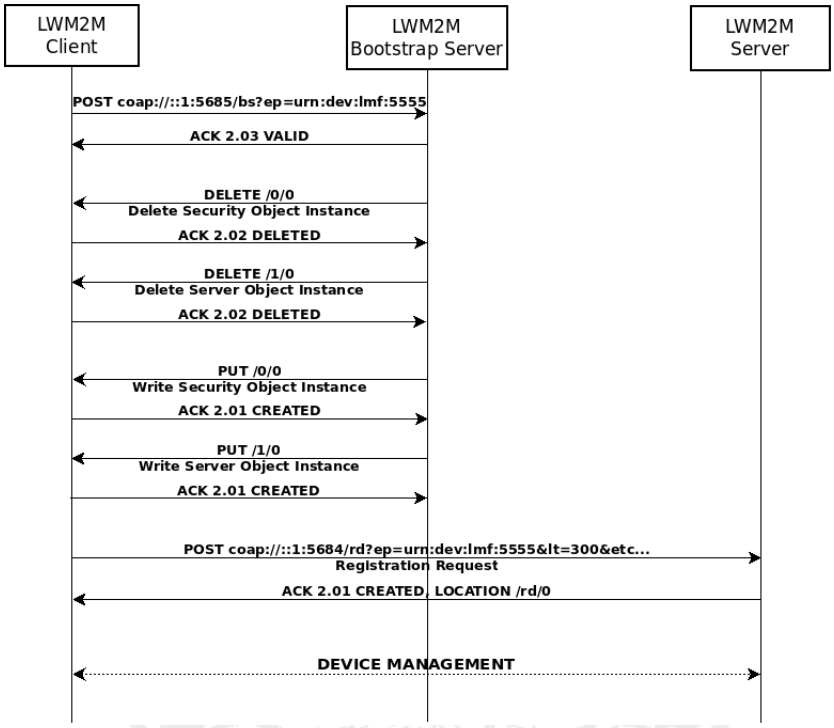

Figure 3.4: Bootstrap message flow without security

The bootstrap message flow is described in Figure 3.4. In this configuration, communication is made in plaintext, without the additional DTLS layer. After the bootstrapping request, the LWM2M Bootstrap Server sends a 2.03 Valid ack message,

to notify the Client that the request has arrived (to avoid packet retransmissions, or other bootstrapping solutions to be initiated by the client). At this point, the LWM2M Bootstrap Server should perform write and delete operations at the Client, in order to change the LWM2M Server accounts onto the device; if the Bootstrap Server account is not deleted in a determined timeout, the information is purged by the client.

Once the LWM2M Bootstrap Server has performed the WRITE operations, effectively creating two new object instances, the Client can register to the LWM2M Server. This is done by sending a CoAP POST message to another specific resource (/rd), with a query string with several information, such as client endpoint, registration lifetime or LWM2M version. The LWM2M Server accepts the registration and replies with a 2.01 Created ack with Location Option; this option gives a reference to be used client-side for every Registration Interface operation (i.e. Registration Update and De-Registration). Once the registration is complete, the device management session can begin.

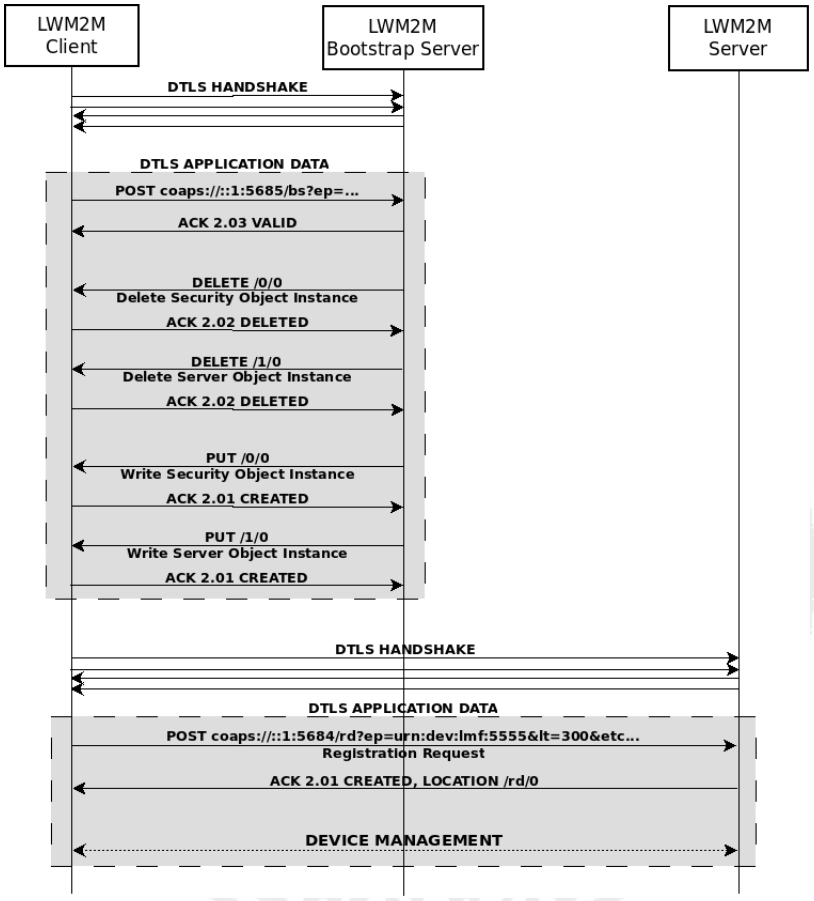

Figure 3.5: Bootstrap message flow with DTLS

The LWM2M Client endpoint name could be assigned in many ways. Since it is

used by the Bootstrap Server to assign the proper LWM2M Server to the client, it is necessary to bind it securely to the LWM2M Client. One way to achieve this, is to use the endpoint name as a Secure ID; to obtain a Secure ID, it could be possible to calculate the hash function of the client public key, using the truncated SHA algorithm, known as SHA-120. This algorithm consists in calculating a 256-bits hash key with SHA, as usual, and then truncate it to 120 bits to limit packet size and resource consumption.

In order to perform a client initiated bootstrap, a LWM2M Client needs to have stored locally a LWM2M Bootstrap Server account; this account corresponds to a pair of a LWM2M Security Object instance and a LWM2M Server Object instance. The Security Object [3] is a mandatory object that contains useful information about a generic LWM2M Server; this server could be a Bootstrap Server, if Resource 1 is properly set to true. A LWM2M Client uses this object to initially communicate with the server, using the Security Mode describe in Resource 2, and optionally the security keys stored in Resources 3, 4 and 5. The LWM2M Server Object [3] contains other information; the two instances are matched using the "Unique Server ID" as a sort of foreign key, recalling databases' terminology.

In order to add security capabilities, LWM2M protocol can be encapsulated into DTLS messages; DTLS can add many new features in the end-to-end communication, such as message encryption or digital signature. The idea is to wrap every CoAP message in a DTLS Application Data packet, a special DTLS message that is part of the DTLS Record Protocol. The Record Protocol takes messages to be transmitted, coming from upper layers (i.e. CoAP), fragments the data into manageable blocks, optionally compresses the data, applies a *Message Authentication Code* (MAC), encrypts, and transmits the result. The received data is decrypted, verified, decompressed, reassembled, and then delivered to higher-level clients. In Figure 3.5 is shown the bootstrap message flow with DTLS; the detail of the DTLS Handshake Protocol is showed in Figure 3.6.

It should be noted that the DTLS encapsulation adds some information in each CoAP message, incrementing the size in byte of the packets: in particular, for each CoAP message a total amount of 29 bytes is added, of which 13 represents the Record Protocol header and 16 are used for the MAC.

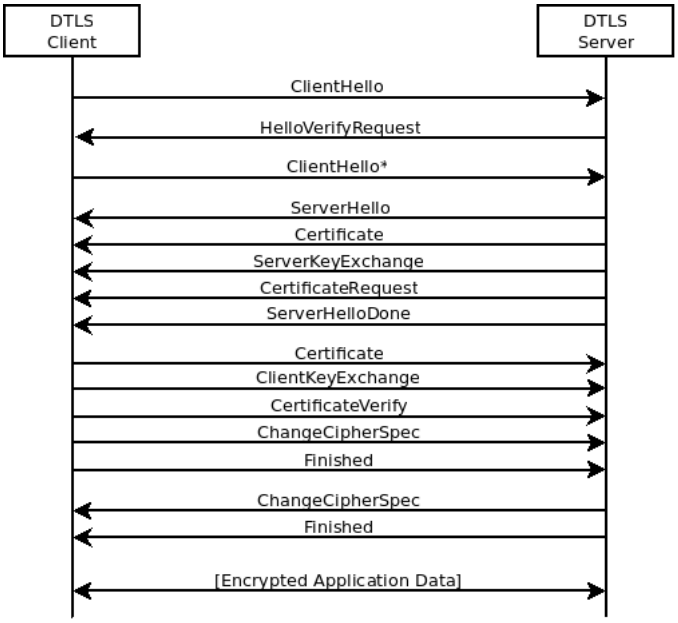

Figure 3.6: Detailed DTLS Handshake Protocol

Public Key Validation

In Raw Public Key mode, client and server only exchange key pairs without a complete certificate; for this reason, there is no mutual authentication. Without a secure binding between identifier and key, the protocol will be vulnerable to Man-in-the- Middle attacks. In particular, the server is not able to validate the public key of a specified client; so, there is the need to bind a particular raw public key to the client that is using it. The most common approach is to use an out-of-band mechanism to validate this key, through a trusted third party.

One proposal is to use the *DNS-based Authentication of Named Entities* (DANE) protocol [51]. DANE is a protocol to allow certificates to be bound to DNS names using *Domain Name System Security Extensions* (DNSSEC); it has been proposed as a way to authenticate TLS client and server entities without a certificate authority (CA). DANE enables the administrator of a domain name to certify the keys used in that domain's TLS clients or servers by storing them in the *Domain Name System* (DNS). DANE needs DNS records to be signed with DNSSEC.

The solution adopted for this thesis work is more suitable for constrained devices; the idea is to use an out-of-band mechanism to validate the device's raw public key using a QR code. When a user wants to deploy a new device, and he needs to bootstrap it, he performs a preliminary operation: he scans a QR code printed on the device (or

on the box) with a scanner connected to his user interface (i.e. a smartphone); in the QR code should be stored information about the device's raw public key, or its SHA-256 hash, that appears on his service management interface and then the device can be switched on. This information exchange can be made using another RESTful protocol that is not CoAP, i.e. HTTP is the most convenient protocol. When the device tries to bootstrap/register to the interface, his Secure ID is communicated to the LWM2M Servers; if the two informations matches, the raw public key of the device is bounded to the device itself.

3.3.2 IPSO Smart Objects

Once the device management is enabled and secured, the application-specific layer can be designed. The application we are interested in is the monitoring of a sensor network; let's imagine a generic scenario in which there are several sensor devices, each of which is running a LWM2M Client instance and is connected to a single (or multiple) LWM2M Server. The Server needs to read sensor values from the devices, deployed for instance in a Smart House, and act accordingly to the read values.

In order to make an easier design of the Server application, it is necessary that all devices use a common interface. For example, if there were three temperature sensors, each of which with different resources for temperature readings, it would not be so easy for a developer to build an application that allows a user to monitor the three temperature values. Moreover, if a new temperature sensor is deployed (or an old one is substituted) the whole Server application should be rewritten.

IPSO Smart Objects are a set of LWM2M Objects that have fixed Object IDs and reusable Resources, which makes it convenient for writing IoT Applications on top of LWM2M. In the previous example, each temperature sensor will show the same IPSO 3303 interface, which makes it easier to fetch the temperature value of each device, even when new ones are deployed: in other words, each LWM2M Client will have the /3303/0/5700 resource, corresponding to a temperature value (5700) of an instance (0) of a temperature sensor (3303). So it possible to fetch the temperature value of a generic sensor by sending a CoAP GET packet for the following path:

$$coaps://deviceX.example.com/3303/0/5700$$

where "deviceX.example.com" could also be the IP address and port of the device.

Object Definition

Name	Object ID	Instances	Object URN
IPSO Generic Sensor	3300	Multiple	urn:oma:lwm2m:ext:3300

Resource Definitions

ID	Name	Op	Mandatory	Type	Description
5700	Sensor Value	R	Mandatory	Float	Last or Current Measured Value from the Sensor.
5701	Units	R	Optional	String	Measurement Units Definition.
5601	Min Measured Value	R	Optional	Float	The minimum value measured by the sensor since power ON or reset.
5602	Max Measured Value	R	Optional	Float	The maximum value measured by the sensor since power ON or reset.
5603	Min Range Value	R	Optional	Float	The minimum value that can be measured by the sensor.
5604	Max Range value	R	Optional	Float	The maximum value that can be measured by the sensor.
5605	Reset Min and Max Measured Values	E	Optional	Opaque	Reset the Min and Max Measured Values to Current Value.
5750	Application Type	RW	Optional	String	The application type of the sensor or actuator as a string.
5751	Sensor Type	R	Optional	String	The type of the sensor.

Table 3.1: IPSO Generic Sensor Object (Source: [5])

The full set of IPSO Smart Objects has been added to the project, client-side. In particular, for this thesis three objects has been found to be fundamental:

1. IPSO 3300: Generic Sensor (Table 3.1), to simulate a generic sensor that is changing its value on a random time.
2. IPSO 3301: Illuminance [5].
3. IPSO 3303: Temperature [5].

Using IPSO Objects it is possible to monitor sensors (simulated or not) through LWM2M Information Reporting Interface; thanks to the Observe/Notify paradigm of LWM2M, it is possible for a Server to receive sensor updates coming from the Client asynchronously: the Client decides how often to update the value of the sensor and sends a Notify message to the Server. It is also possible for a LWM2M Server to chose to observe only value changes under determined conditions, e.g. if the temperature sensor value raises above a certain threshold.

For the Observe/Notify mechanism, it is important to mention the effect of *Network Address Translation* (NAT) on the application; if the LWM2M Client is not sending any notification to the LWM2M Server for a certain amount of time, it could happen that the association in the NAT table between the two endpoints is erased, making it

impossible for the LWM2M Server to send any other requests to the LWM2M Client. For these reason, the application is easiest used in IP environments where devices are directly routable (i.e. IPv6). No other solutions were addressed by this thesis work, such as the use of STUN, TURN or ICE protocols.

3.3.3 Device Management User Interface

As part of this thesis work, the LWM2M Server and the LWM2M Bootstrap Server needed to be integrated into a Web Server, developed during another thesis about device management [47]; this work was developed for the Capillary Network architecture, and as a part of it there is also a LWM2M communication.

In this thesis, we are trying to integrate both LWM2M Server and LWM2M Bootstrap Server into a Web Server that shows a Web User Interface for device management; the UI consists of a map where the devices are deployed and showed, so that the user can select one (or a group) of them and perform read/write/execute operations in a very simple way.

Another important feature, is the selective bootstrap; through the user interface, an user may decide whether to accept or not a bootstrap request coming from a device. This way, the out-of-band mechanism for Raw Public Key validation described in the previous section may be implemented easily.

Knowing these requirements, the design could be made: LWM2M Server and LWM2M Bootstrap Server should be two separate entities that show another interface to the Web Server, in addition to the LWM2M one. The communication between the servers should be bidirectional, in the sense that LWM2M Servers should notify the Web Server when a LWM2M Client tries to bootstrap or register and the Web Server should be able to perform operations on them, sending commands. The generic architecture we have implemented in this thesis work is shown in Figure 3.7.

3.4 Summary

In this chapter a full design of a basic device management service has been presented. Although the main focus of this thesis is the bootstrapping architecture, a device management user interface has been described as well. In Figure 3.8, the full stack for

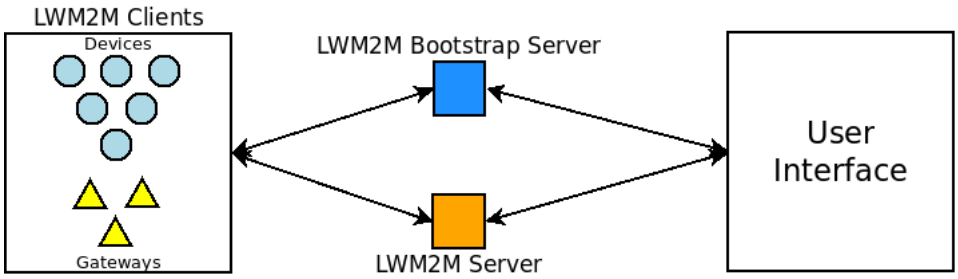

Figure 3.7: The complete bootstrap architecture

IoT enablement used in this thesis work is shown. The network technologies underlying the communication protocol are physical and datalink standards (e.g. 802.11 [48] or 802.15.4 [49]). Over the physical layers, there are the classic routing protocols, IPv4 and IPv6, along with the new 6LoWPAN, suitable for constrained devices.

On the application layer, the main application protocols are the RESTful HTTP and CoAP; while CoAP is suitable for constrained devices, i.e. sensors and controllers, HTTP is the main choice for the Web Server Interface, since a full CoAP-HTTP mapping has been designed in [2]. Above the RESTful communication, the service is provided by LWM2M protocol for device management and by the Web Server for the User Interface; in addition, it is possible to use other RESTful protocols, i.e. WebSocket that runs over HTTP, for constrained device management. Finally, IPSO Smart Objects are used as data model for both device management and user interface. The specific application relies on the full stack.

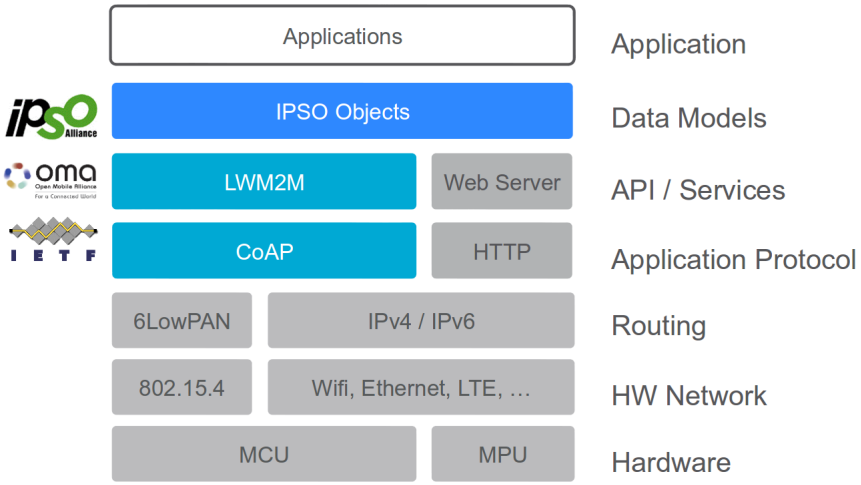

Figure 3.8: The full stack for IoT enablement

Chapter 4

Implementation

In order to evaluate the impact of the introduction of security and bootstrap capability in a constrained environment, a LWM2M prototype has been developed. The prototype has been tested on real constrained devices, represented by embedded computers used as constrained application development platform. This chapter is divided into two main sections: in the first section the hardware part of the prototype is explained, with a presentation of the embedded devices and some schematics of the sensor circuits; in the second section the software part of the application is presented, with an introduction of the open source libraries used to developed the prototype and the choices made to integrate LWM2M into a Web Server for device management.

4.1 Hardware

The hardware part of the prototype consists of a platform where to run the LWM2M application; in particular, the LWM2M Client is conceived to be executed on a device with sensors attached to it. The LWM2M Server can easily run on any server machine, while the LWM2M Bootstrap Server could be run either on a server machine or on intermediate devices, such as gateways.

4.1.1 Embedded Devices

In the field of research for IoT, new prototype applications are being developed every day; in order to test these applications, to evaluate the feasibility of their deployment

on constrained devices, it is possible to use small embedded devices that can run any kind of code to measure important parameters, such as power consumption, packet loss rate, network speed rate, and so on. New technologies are being developed nowadays, small constrained systems that allow the user to run applications on it in prevision of deploying them on a real IoT device. Many manufacturing companies are now developing a full set of IoT embedded systems, for testing monitoring applications, such as ARM (with the new Mbed Operative System), Intel, Samsung, and so on.

In this thesis work, the development of the device management application needed to be tested on a real embedded device, acting like a sensor. For this purpose, two embedded systems were used:

Raspberry Pi The Raspberry Pi [52] is a credit card-sized single-board computer, developed mainly for educational purposes and that can be used for IoT prototypes as well. The device has a Broadcom System-on-a-Chip, with CPU, GPU and RAM, and many other peripheral ports, such as USB port for keyboard and mouse, audio and video outputs, ethernet connectivity and the possibility to expand the system with many other functionalities: in particular, the Raspberry Pi has got 40 *General Purpose Input/Output* (GPIO) pins, where a wide selection of sensors can be connected.

Intel Edison The Intel Edison [53] is a tiny computer offered by Intel as a development system for wearable devices. The strength of the Edison is in its small dimensions: the device, in facts, is a small SD card shaped board, with WiFi connectivity, making it a perfect wearable developing platform. Many expansion boards exist, including a battery pack, making it possible to plug several sensors on it and providing full mobility characteristics.

In table 4.1 is showed a comparison between the two embedded devices previously described; both these devices run a linux operative system, hence it's very simple and straightforward to develop and run light applications on them. The two devices are compare in Figure 4.1.

4.1.2 Raspberry Pi Sensors - Hardware schematics

During the prototype development, the Raspberry Pi has been chosen as platform to connect sensors to and to run the LWM2M Client; the device should run the agent at

	Raspberry Pi	**Intel Edison**
Operative System	Raspbian	Yocto Linux
CPU	ARM1176 700 MHz	Intel Atom 500 MHz
GPU	Broadcom VideoCore IV	N/A
Memory	512 MB (shared with GPU)	1 GB
USB ports	2/4 USB 2.0	1 micro USB 2.0
Video output	HDMI	N/A
Audio output	3.5 mm jack	N/A
Storage	microSD	4 GB flash storage
Network	Ethernet	WiFi 802.11 a/b/g/n and Bluetooth 4.0
Low Level Peripherals	GPIO, SPI, I^2C and UART	N/A
Power Consumption	3.5 W	35 mW
Power Supply	5 V micro USB	5 V micro USB or battery
Dimensions	85 mm x 56 mm	35 mm x 25 mm

Table 4.1: Embedded Devices Characteristics

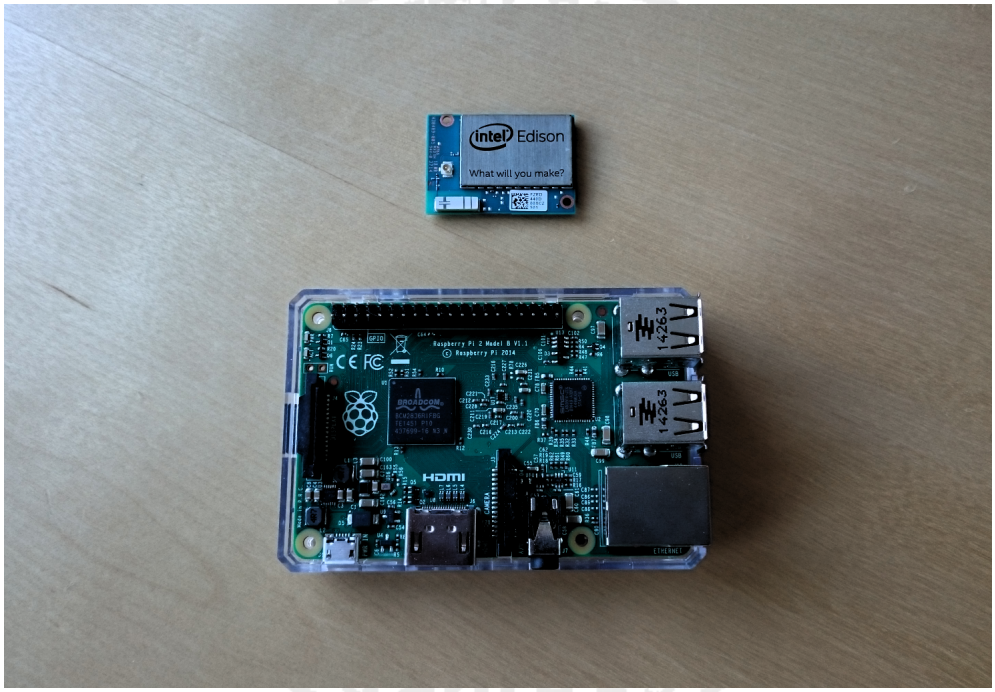

Figure 4.1: Comparison between the Raspberry Pi and the Intel Edison

startup, trying to bootstrap to a fixed LWM2M Bootstrap Server, connect to the bootstrapped LWM2M Server and get values periodically from the environment through some kind of sensors.

The sensors we used are small electronic components, usually used for test purpose. These little components are technically transducers: they converts measured values from the environment into voltage values. These analog values must be converted into digital burst of signal, since Raspberry Pi's GPIO only deals with digital values. For this purpose, an analog-to-digital converter microchip has been used. This microchip converts the analog values coming from the sensors into digital values, encoded on 1 byte, and sends them on the *Serial Peripheral Interface* (SPI) of the Raspberry Pi.

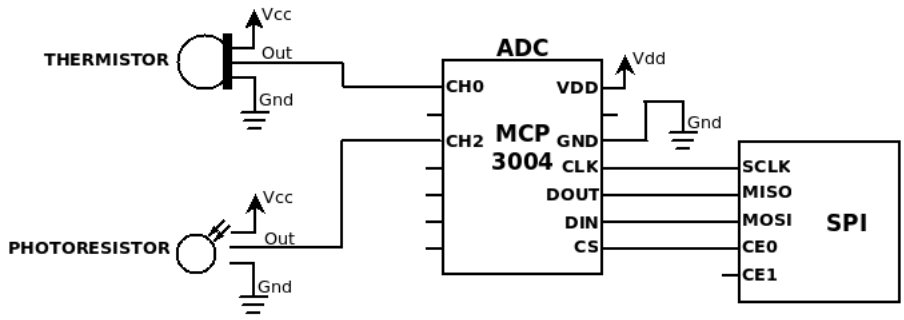

Figure 4.2: Raspberry Pi Sensors Schematics

In Figure 4.2 there are the schematics of the described system. The sensors used are a Thermistor, to get temperature values from the environment, and a Photoresistor, to get luminosity values. The sensors are attached to two ports of a MCP3004 ADC; this component acts like a slave for the master SPI of the Raspberry Pi, that drives the ADC clock and chip select. Input and Output of the ADC are connected, respectively, to the Master Output Slave Input (MOSI) port and to the Master Input Slave Output (MISO) port of the SPI.

The Serial Peripheral Interface is a standard communication bus between microcontrollers or integrated circuits. With the SPI it is possible to have a full-duplex communication, meaning that it is possible to transfer data in both ways simultaneously. It is possible to built the same configuration on the Intel Edison, connecting to it a GPIO extension board.

4.2 Software

The software part of the prototype consists of an application written in C language; the application is built using a client-server paradigm, though bidirectional, in the sense that both entities of the communication are listening and can start a communication with a request. In Figure 4.3 the implemented stack is showed. The LWM2M implementation used is the Wakaama Library, an eclipse foundation project; inside Wakaama, Erbium CoAP has been used as CoAP layer, for message building and parsing; lastly, an additional DTLS layer has been added using TinyDTLS library.

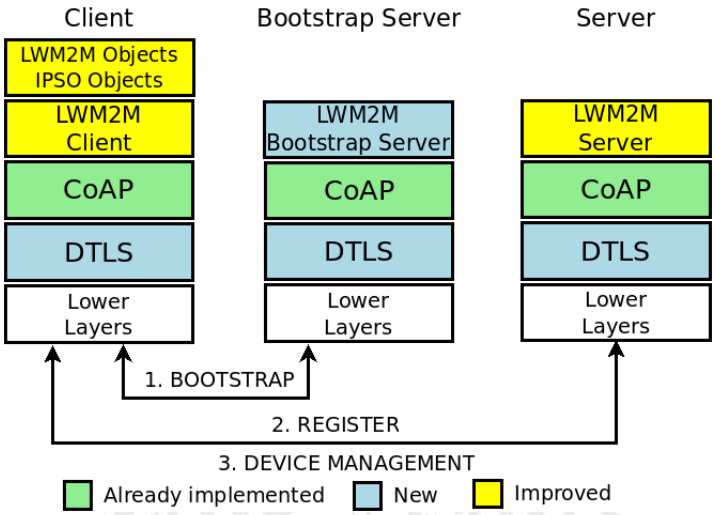

Figure 4.3: LWM2M Implemented Stack

4.2.1 Wakaama

Wakaama [54] is an open source C implementation of LWM2M under Eclipse Public License. Born as an Intel project known as LibLWM2M, it became an Eclipse Foundation project in June 2014 and it is still under development; no official releases has been produced yet, but the project has many contributors from several companies. Wakaama is not a true library, but it is a set of open source APIs to be built with an application, in order to provide support for the protocol.

At the beginning of this thesis work, Wakaama did not implement a bootstrapping interface; the Client Registration, Information Reporting and Device Management interfaces were only partially implemented. In the device management interface, some methods were missing, such as Resource Discovery or Write Attribute methods. In Figure 4.4 the architecture is described.

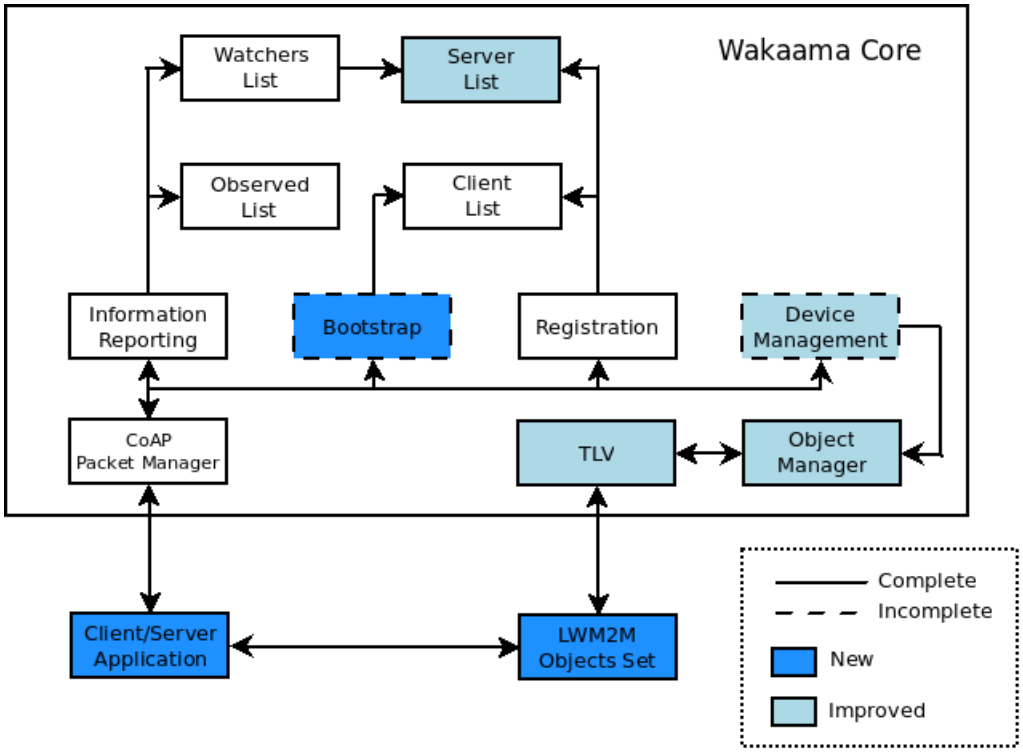

Figure 4.4: Wakaama Library Components

Wakaama uses multiple compilation switches in order to build a Client, with LWM2M_CLIENT_MODE, or a Server application, with LWM2M_SERVER_MODE; it is also necessary to define if the architecture where the application is built is big endian, with LWM2M_BIG_ENDIAN, or little endian, with LWM2M_LITTLE_ENDIAN, for a correct float encoding in the TLV module.

When the application starts, it is necessary to call some initialization APIs from Wakaama Core, such as *lwm2m_init()*, *lwm2m_configure()*, *lwm2m_start()* or *lwm2m_set_monitoring_callback()* to initialize *lwm2m_context* and set the proper callback functions. The main application should periodically call the *lwm2m_step()* API in order to perform the usual LWM2M operations, such as message retransmission, registration update or client bootstrapping.

Both Client and Server application have a command line user interface, where the user can send LWM2M messages by typing "Command ClientID URI Data"; in this way, the corresponding API is called in the Wakaama Core, the CoAP packet is built and sent over the network and the response is printed on the LWM2M Server screen. When a packet is received, the application calls the *lwm2m_handle_packet()* API, where the packet is parsed using CoAP library, then the control is sent to one of the four interfaces. The result of the elaboration is sent back through a callback

function, written in the application.

Information about the Clients and the Servers are stored in data structures. In particular, a Server application maintains a list of registered clients, assigning to each of them an InteralID. A Client application also maintains a similar server list, to which it tries to register each time the *lwm2m_step()* function is called. The Information Reporting interfaces maintains two similar data structures called Watchers List and Observed List.

After processing an incoming packet, if the request is done at the Device Management interface, the corresponding operation is called on the specific LWM2M Object, if it exists and has the requested resource. The communication between Wakaama Core and the LWM2M Objects Set is done using *Type-Length-Value* (TLV) format, a very compact data encoding in which it is possible to encode information about Type and Length of the data using only 3 bytes. TLV format is also used as payload format of the LWM2M packets if the request is done on the entire object (and not on a single resource).

Improvements

The main work of the thesis was done starting from a client-server test application, modifying Wakaama Core to improve it, adding missing functionalities. In particular, the main additions to the Wakaama library were:

Bootstrap Interface The bootstrap interface was missing from the Wakaama Core; in the packet parsing, there was a placeholder for the received "/bs" path case, which was completed. Also the Client Initiated Bootstrap was implemented.

Bootstrap Server An instance of the LWM2M Bootstrap Server was implemented, with an additional command: with "bs 0" the user can bootstrap LWM2M Client number 0 after receiving a bootstrap request from him; however, the bootstrap process is automatic on receipt of a bootstrap request.

Bootstrap Server Support New data structures, switches and conditions were added to handle bootstrap server cases as well.

DTLS Support TinyDTLS library has been integrated into Wakaama Core, making it possible to built a client-server application with DTLS capabilities.

Object Definition

With Wakaama, it is possible to define a great number of custom objects with any kind of resources. To define a new LWM2M Object, it is necessary to create a new variable of type *lwm2m_object_t* with a new unique ID, which will be communicate to the LWM2M Server and will be part of the address of a resource. The LWM2M Object data structure has the following declaration:

```
_lwm2m_object_t {
    uint16_t                objID;
    lwm2m_list_t *          instanceList;
    lwm2m_read_callback_t   readFunc;
    lwm2m_write_callback_t  writeFunc;
    lwm2m_execute_callback_t executeFunc;
    lwm2m_create_callback_t  createFunc;
    lwm2m_delete_callback_t  deleteFunc;
    lwm2m_close_callback_t   closeFunc;
    void *                  userData;
};

_lwm2m_list_t {
    struct _lwm2m_list_t * next;
    uint16_t               id;
};
```

The instances of the object are stored as a linked list, and must match the *lwm2m_list_t* data type, in order to allow explicit casting from a type to another; the rest of the instance structure can be freely defined by the developer, with an arbitrary number of fields. An important thing to do is to write static functions to be used as callbacks to access instance fields: read, write, execute, create, delete and close (to delete all the object instances). The arguments of these functions are the same across diverse ObjID: the URI path, the input/output TLV with its size and the object variable (to get its instances). The results of the callback function are CoAP response code, for success or errors in the execution of the operations. In Figure 4.5 a generic LWM2M Object instance implementation is described with a simple class diagram.

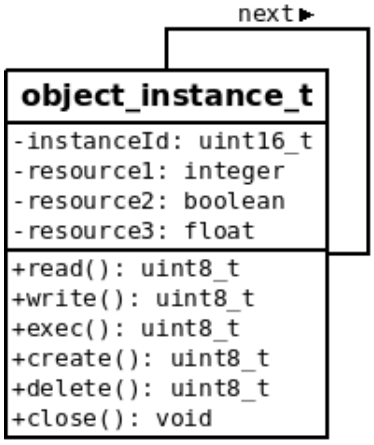

Figure 4.5: LWM2M Generic Object Instance Class Diagram

4.2.2 TinyDTLS

TinyDTLS [55] is an open source C implementation of DTLS protocol under MIT License. The implementation aims at providing a very lightweight version of DTLS, suitable for embedded devices with Linux OS or Contiki OS [56]. TinyDTLS is not complete: for this thesis work the TinyDTLS 0.8.2 version has been used; this version has basic functionalities to grant DTLS handshake and communication encryption.

To compile TinyDTLS code, two macro definitions have been used: WITH_ECC to enable Elliptic Curve Asymmetric Cryptography and WITH_SHA to add SHA-256 compliance. A new Wakaama compilation switch has been added: if WITH_DTLS is defined, the LWM2M application will use DTLS; otherwise, all the communication will be plaintext. Hence, it is possible to simply switch between secured and unsecured communication protocol by rebuilding and recompiling the application code.

TinyDTLS has been integrated into the application, between the application itself and the Wakaama Core: when a packet is received, it is firstly processed using TinyDTLS APIs; if the packet is of type Application Data, the payload is extracted, decrypted and sent to the Wakaama Core APIs to be handled normally. When the result of the request is ready, and a response must be sent, the packet will be again processed by TinyDTLS, encrypting data and putting it into another Application Data DTLS packet, and then sent normally over the socket.

TinyDTLS uses a similar approach of Wakaama: packets are processed by TinyDTLS APIs (specifically the *dtls_handle_message()* function contained in *dtls.c* source code file) and when the result of the processing is ready, the respective callback is called.

To start building a response packet, the *dtls_write()* function is used. There are four callbacks used in TinyDTLS:

1. *dtls_send_to_peer()* : used to send a newly built DTLS packet to a specific peer over the network.
2. *dtls_read_from_peer()* : when a packet is recognized as Data Application and decrypted, this callback function is called; inside this function there should be a *lwm2m_handle_packet()* call to return the control to Wakaama Core.
3. *dtls_handle_event()* : when a new event occurs in DTLS, this callback function is called; it is used by the application to recognize when the DTLS handshake is ended, the peers are connected and the LWM2M communication can start (with a bootstrap or registration request).
4. *dtls_get_ecdsa_key()* : this callback function is used by DTLS to obtain the elliptic curve key pair; this way, it is possible to customize the way the keys are obtained (they can be stored in an external file, randomly generated at each run or simply hardcoded).

4.2.3 Application Testbed

To test the Secure Bootstrapping Architecture, it was necessary to implement a distributed client-server application. In this application, three entities were compulsory: a LWM2M Client, acting as the device management agent, a LWM2M Server, acting as the device manager, and a LWM2M Bootstrap Server, to enable the service. Wakaama provides a Client and a Server test application, with a command user interface. These two entities were improved with new arguments, functionalities and capabilities, and a new one was created following the skeleton of the server application: the LWM2M Bootstrap Server. The Client was executed on two constrained devices: an Intel Edison and a Raspberry Pi; the server applications were executed both on a normal workstation or, in case of the Bootstrap Server, on a Raspberry Pi. Following, there's a short description of the application's entities.

LWM2M Client The LWM2M Client acts as a device management agent, running on a device; it implements LWM2M interfaces and data model. At startup, the application instantiates the context, allocate the LWM2M Objects needed by the

application and opens a socket on a random port; the objects allocated depends on the program arguments given at execution time, that are:

- **-b - Bootstrap**: with this option, the LWM2M Client will send a bootstrap request to the specified URI; if not used, the Client will send a normal registration request.
- **-s - Raspberry Pi Sensors**: this option is to be used only on a Raspberry Pi with the previously described configuration (sensors and ADC); the LWM2M Client won't start, otherwise.
- **-r - Resource Simulation**: with this option, the LWM2M Client will run in a sensor simulation mode, periodically changing a fake sensor value with a random integer between 0 and 100.
- **-o - IPSO Smart Objects**: with this option, the LWM2M Client will start with a full set of IPSO Objects, that are 22 in total.

It is possible to only define one parameter at a time between **-s**, **-r** or **-o**; if no arguments are defined, the Client starts with only a test object with simple resources. Once the LWM2M context is initialized, the application tries to bootstrap, or register, to a server specified in the command line (e.g. 10.0.0.98:5685). Both IPv4 or IPv6 are supported by the application, so it is possible to specify also an IPv6 address for the LWM2M Bootstrap Server. Once connected to a LWM2M Server, the Client receives the commands from the Server on the open socket and sends periodically a registration update to keep the registration alive. The encryption key pair used for DTLS communication are randomly generated on each program execution.

LWM2M Server The device manager is a command line program called LWM2M Server; the Server starts listening on port 5684, both on IPv4 and IPv6 sockets, after setting up the context. The Wakaama test server has LWM2M interfaces already implemented, and only little modifications have been made to it. The LWM2M Server listens on the socket for incoming registration requests, sends device management commands and just shows the result on the screen (through a result callback); it is possible to monitor Clients' registration, update and deregistration with a monitoring callback. The encryption key pair are stored in a file called "keys.config".

LWM2M Bootstrap Server The LWM2M Bootstrap Server is a new entity, built on the lines of a LWM2M Server; this server has a very limited command line

interface, making it possible only to bootstrap a specific client that sent a bootstrap request. To built the LWM2M Bootstrap Server application it is necessary to define two compilation switches: LWM2M_SERVER_MODE, to enable server capabilities, and LWM2M_BOOTSTRAP to enable the bootstrap interface (and disable the registration interface). The Bootstrap Server starts opening a socket on port 5685, both on IPv4 and IPv6 sockets, listening for bootstrap requests; the address of the LWM2M Server to be provided to Clients that wants to bootstrap is stored in a configuration file, called "server.config". The encryption key pair are stored in another file, called "keys.config".

4.2.4 Secure Bootstrap

In order to implement the secure bootstrap feature, Wakaama library has been integrated into TinyDTLS library, so that the application was easily built using few APIs; for a developer, it is only necessary to initiate a *lwm2m_context* and a *dtls_context* and call TinyDTLS API for the connection: *dtls_connect()*.

When the LWM2M Client starts, it checks its Security Object instances: for each instance, it generates a security key pair using a Pseudo-Random Number Generation function and stores it in memory. Then, in a Client initiated bootstrap, the Client tries to register or bootstrap to the relative LWM2M Server or LWM2M Bootstrap Server; the first steps consist in the establishment of the DTLS session through the handshake protocol using the key material provided by LWM2M Objects.

Once the DTLS session is established, the communication can continue through DTLS Application Data messages. In particular, the Client Initiated Bootstrap was implemented following the scheme of the Registration Interface. When the Client checks LWM2M Security Object and finds a LWM2M Bootstrap Server, it sends a CoAP POST message on the "/bs" resource, adding additional information in the query string, i.e. its endpoint name.

The LWM2M Bootstrap Server is listening on port 5685; for simplicity, in this implementation we assumed that every Client requesting a bootstrap is authorized to be bootstrapped. Hence, the LWM2M Bootstrap Server will accept any bootstrap request, responding with the same LWM2M Security and Server Objects for each Client. The information of the objects are hardcoded in the application, except the LWM2M Server IP Address, stored in a configuration file.

In practice, the LWM2M Bootstrap Server maintains a Client list, in the form of a record, to keep track of the bootstrap requests; in the main application function, a *bootstrap_routine()* procedure is periodically called: this routine takes the Clients from the list, bootstraps them, and deletes them from the list. The bootstrap session consists in mainly two phases:

1. **Delete**: once the request is received and the 2.03 VALID ack message is sent back to the Client, the LWM2M Bootstrap Server proceeds to purge the LWM2M Security and Server Objects instances.
2. **Write**: to bootstrap the Client, new LWM2M Security and Server Objects instances are written by the LWM2M Bootstrap Server. Once this step is done, any connection pending can be optionally closed (e.g. TCP or DTLS).

Data sent during the Write phase is in TLV format, to sent the maximum amount of information in the minimum required space. Moreover, a new security key pair is generated at the LWM2M Bootstrap Server during the Write phase, in order to provide the Client with new keys material at each run.

After the procedure is ended, LWM2M Client should be able to register itself to the LWM2M Server, starting a new DTLS Handshake and establishing a secure connection for the device management session. The entire bootstrap procedure is automatic, and needs no human intervention at all; the user can easily change the LWM2M Server IP Address, editing the configuration file, or the security key material in the same way.

4.2.5 IPSO Smart Objects

IPSO Smart Objects are implemented as separate source file, with an initialization function to be called in the main initialization of the LWM2M Client: when the Client is started, a set of LWM2M Objects are created and stored in specific data structures (i.e. linked lists). Along with these LWM2M Objects, IPSO Smart Objects are created as well and stored in similar data structures; the number and the type of IPSO Objects deployed depends on the application.

In an IPSO Smart Object source code file there is a data structure, a record, that represent a generic instance of the object; inside this record, the variable representing the resources of the object are defined. Several macros are defined as well, e.g. the resource number, to simplify the legibility of the code. From a re-usability point of

view, the macros could be defined inside a header file (e.g. *IPSO_Smart_Objects.h*), being the resources reusable per definition: this means it is possible to call a macro inside each IPSO object without re-defining it in every source code file.

Inside an IPSO Smart Object source code file there are several functions that enables IPSO functionalities; an initialization function, generally called *get_IPSO_Object_N()*, instantiates a new *lwm2m_object_t* instance and creates a first (and only) instance of the object, populating the variables with initial values. On the other side, there is the *close()* function that frees memory space for every object instance and for the object data structure itself.

Since it is necessary to implement all the LWM2M resource access methods, in the source code file of a generic IPSO Object there are several additional functions, such as *read()*, *write()* or *exec()*; these functions accept a TLV object as an input/output parameter, in the sense that in this data structure there are information to be written, in case of a *write()* (or, optionally, a *create()*) function, or it must be filled with information read from the instance, in the case of a *read()* function. In the case of an *exec()* function, the presence of the TLV structure is optional. The *create()* function instantiate a new IPSO Object instance, filling the new instance's resources with the information provided optionally by the caller. The *delete()* function is a simple function that deletes a specific object instance, freeing the portion of memory assigned to that data structure.

Special functions can be written *ad hoc*, called by the application main function; for example, in the IPSO Smart Object 3300, the Generic Sensor Object, a function called *change_sensor_value()* is called periodically to change a specific resource value and simulate sensor's behaviour. In other objects, there has been defined functions that get the actual sensor values from a real device (i.e. in the Raspberry Pi Sensors mode). In Figure 4.6 is showed the class diagram for the IPSO Smart Object 3300 Generic Sensor, used by the application in Resource Simulation mode.

All these functions written for each IPSO Object are stored as callbacks in a *lwm2m_object_t* instance. When the Wakaama Core decodes a new incoming packet for the device management interface, the payload is encoded in the TLV format and the corresponding callback is called; the result of the callback will be sent again to the Wakaama Core, that elaborates somehow the response.

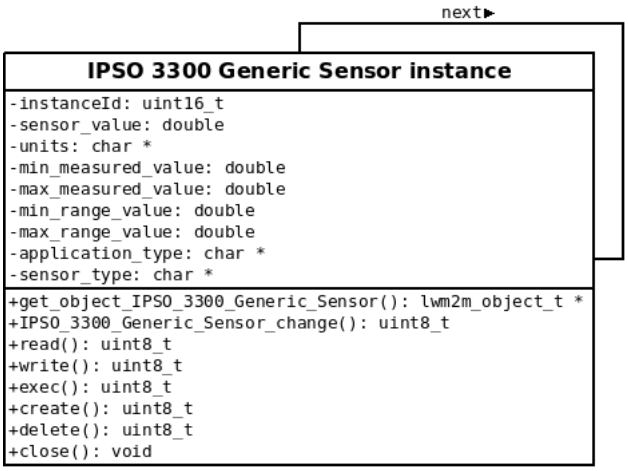

Figure 4.6: IPSO Generic Sensor class diagram

4.2.6 MINT Web Server Integration

To enable the entire bootstrap architecture, it has been found necessary to integrate the LWM2M Server and LWM2M Bootstrap Server instances into a Web Server; the Web User Interface was developed in another thesis work [47], using an Apache Web Server on a MINT Linux machine. The Web Server accepts any incoming HTTP traffic, using PHP technology and Node.js environment to provide a simple user interface for device management; a map, powered by Google Maps, shows the registered devices (according to their LWM2M Location Object).

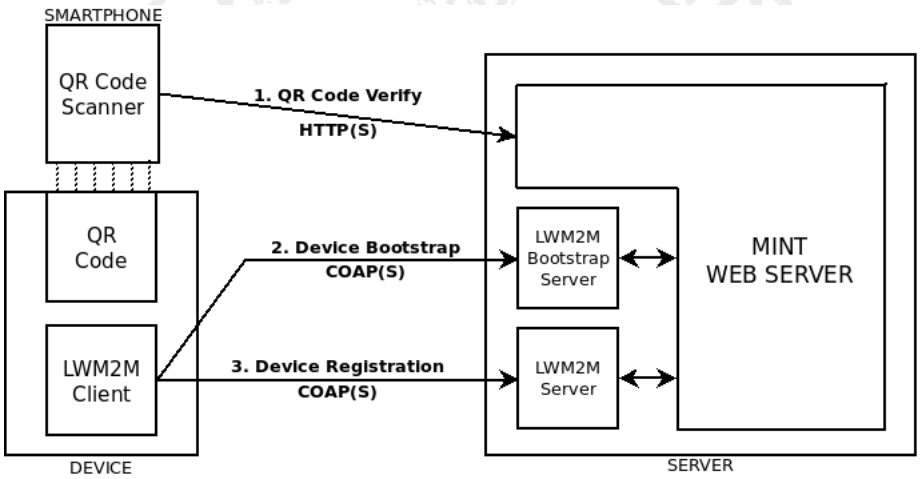

Figure 4.7: The implemented bootstrap architecture

The integration of the LWM2M servers has been done using UDP sockets. The two LWM2M instances are executed on the MINT machine, along with the Web Server; using a special argument, **-w** Web Server mode, the servers start in a particular mode:

they don't accept any more commands coming from the STDIN stream, but only from the open socket. When there's a new connection incoming, the content of the message is parsed in the main function: if the message is one of the commands (i.e. list, read, write, exec...), it is processed like a command coming from the STDIN stream; otherwise, the packet is sent to the Wakaama Core, through *lwm2m_handle_packet()* function.

This way, the communication between the Web Server and the LWM2M servers is made using UDP sockets. In Figure 4.7, the architecture is showed. Along with the LWM2M communication, the QR Code verification is implemented in the Web Server; this mechanism is used by the user to validate a raw public key assigned to a specific device, that is to be deployed.

Information exchanged between LWM2M servers and MINT Web Server is in JSON format [57]; this choice derives from the necessity to have a simple communication between the LWM2M Server and the Web Server in a human-understandable format, since the instances are both running locally and there's no need to encode the information. There are several type of communication between LWM2M Server and MINT Web Server; the most notable are described following. The first communication happens when a new client tries to bootstrap to a LWM2M Bootstrap Server (or tries to register to a LWM2M Server); the server sends a JSON object of "monitor" type, specifying in the "status" field the meaning of the message. Four "status" are possible, and they are: 2.01 (CREATED: new client registration), 2.02 (DELETED: client de-registration), 2.03 (VALID: client bootstrap request) and 2.04 (CHANGED: client registration update). Along with this information, there is the clientId number. The format of the Monitor JSON object is the following:

```
{
    "type" : "monitor",
    "status" : "2.01",
    "clientId" : "5a3f"
}
```

When a 'list' command is sent by the Web Server, a list of registered clients is sent back in response by the LWM2M Server. The format of the JSON object is the following:

```
{
```

```
    "clients" : [
        {
            "id" : "5a3f",
            "name" : "urn:dev:lmf:5555",
            "binding" : "U",
            "lifetime" : "300",
            "objects" : "/0/0,/1/0,/2/0,..."
        },
        {
            "id" : "8ec2",
            "name" : "urn:dev:lmf:5556",
            "binding" : "U",
            "lifetime" : "300",
            "objects" : "/0/0,/1/0,/2/0,..."
        },
        ...
    ]
}
```

A generic command can be executed from the Web Server interface, clicking on a specific button; the request will be translated into a command for the LWM2M Server. Possible commands are read, write, exec, create, delete. Once the request is elaborated by the LWM2M Server, a JSON object is sent back as a response, following the SenML [24] data format. Note that also error code are sent back to the Web Server, specifying the correct error code in the "status" field; the JSON object format for a generic command is the following:

```
{
    "status" : "2.05",
    "uri" : "/3303/0",
    "content" : {
        "e" : [
            { "n" : "/5700", "sv" : "22.37" },
            { "n" : "/5701", "sv" : "19.89" },
            { "n" : "/5702", "sv" : "24.65" },
            ...
        ]
```

```
    }
}
```

Finally, another important message type is the one used by the LWM2M Server to forward a notify message coming from a specific LWM2M Client. The JSON object contains all the information conveyed in a Notify message, such as the URI of the resource that changed its value, the new value, the number of the notify message, the internal clientId... Notify JSON objects has the following format:

```
{
    "type" : "notify",
    "status" : "2.04",
    "clientId" : "rd4a",
    "uri" : "/3303/0/5700",
    "count" : "144",
    "content" : "22.37"
}
```

In Figure 4.8 the message flow between LWM2M Client, servers and MINT Web Server is showed; in this scenario, the out-of-band mechanism is supposed to be implemented. When a LWM2M Client running on a device sends a bootstrap request, the LWM2M Bootstrap Server notifies the Web Server with a monitor JSON message; the clientId field should be something that authenticate the device, such as the hash of his raw public key. The notification coming from the LWM2M Bootstrap Server is showed in the Web User Interface, where the device manager can manually accept (or reject) the bootstrap request.

Once the request is accepted, the message flow will continue normally: the LWM2M Client is provisioned with information on the LWM2M Server, to whom it can register himself and start an encrypted device management session. The communication between LWM2M Server and MINT Web Server continues normally, with command messages coming from the Web Server and notification coming from the LWM2M Server.

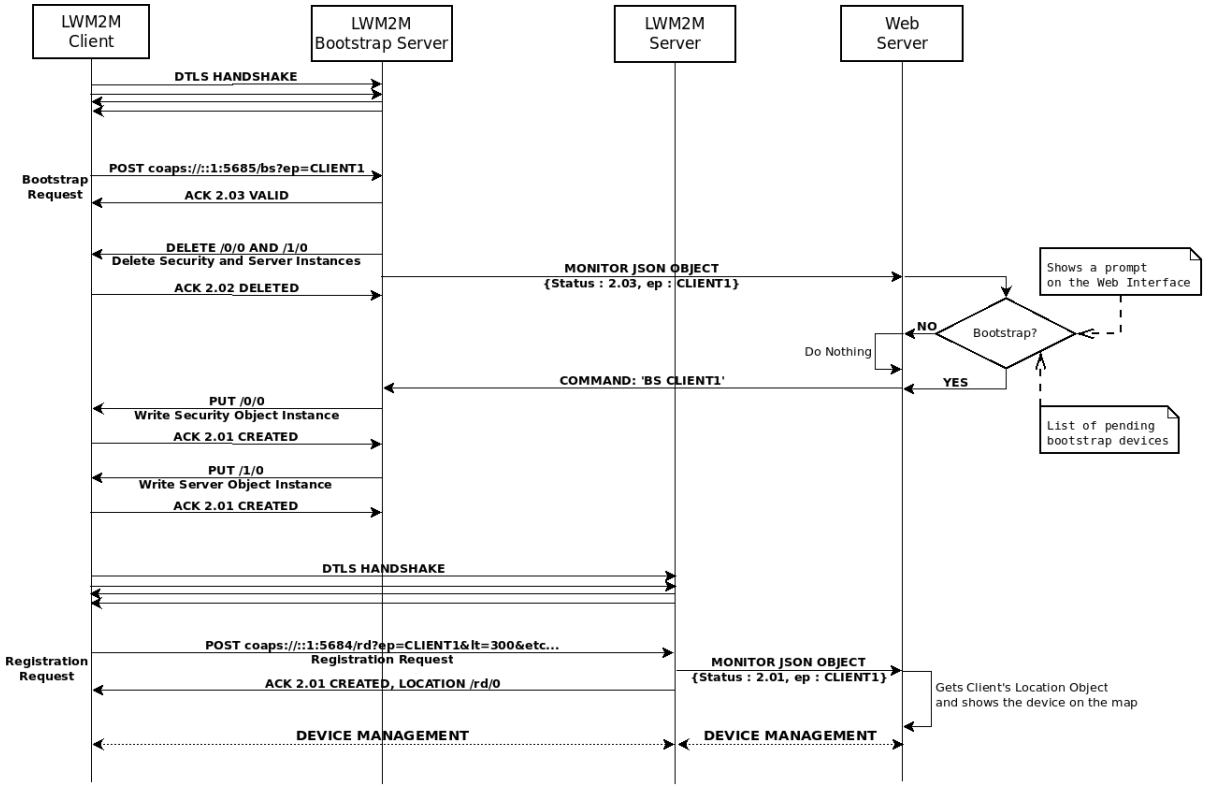

Figure 4.8: Secure Bootstrap message flow with MINT Web Server integration

4.2.7 Raspberry Pi Sensors - Software library

In order to read sensor's values coming from the hardware connected to the Raspberry Pi, it has been found necessary to use a specific library called *spidev*. Spidev is a linux library used to communicate with a SPI device; with spidev it is possible to send and receive byte streams using simple and standard I/O APIs (i.e. *ioctl()* function). Two IPSO Smart Objects has been provided with additional data structures and functions in order to make possible the communication with the sensor device; in Figure 4.9 the data structures of the IPSO Object 3303 for a Temperature Sensor is showed.

To enable the SPI communication, it is necessary to initialize the SPI device when deploying the IPSO Object; first of all, a specific file is opened, "/dev/spidev0.0", that represents the SPI device of the Raspberry Pi; the communication between the application and the device will be done using the file descriptor and I/O functions. Then, some compulsory parameters need to be set, such as SPI mode, maximum clock speed rate, less significant bit settings or bits per word.

Once initialized the SPI device, it is possible to send and receive bytes from it in a simple way: first, it is necessary to instantiate a new *spi_ioc_transfer* object, filled with

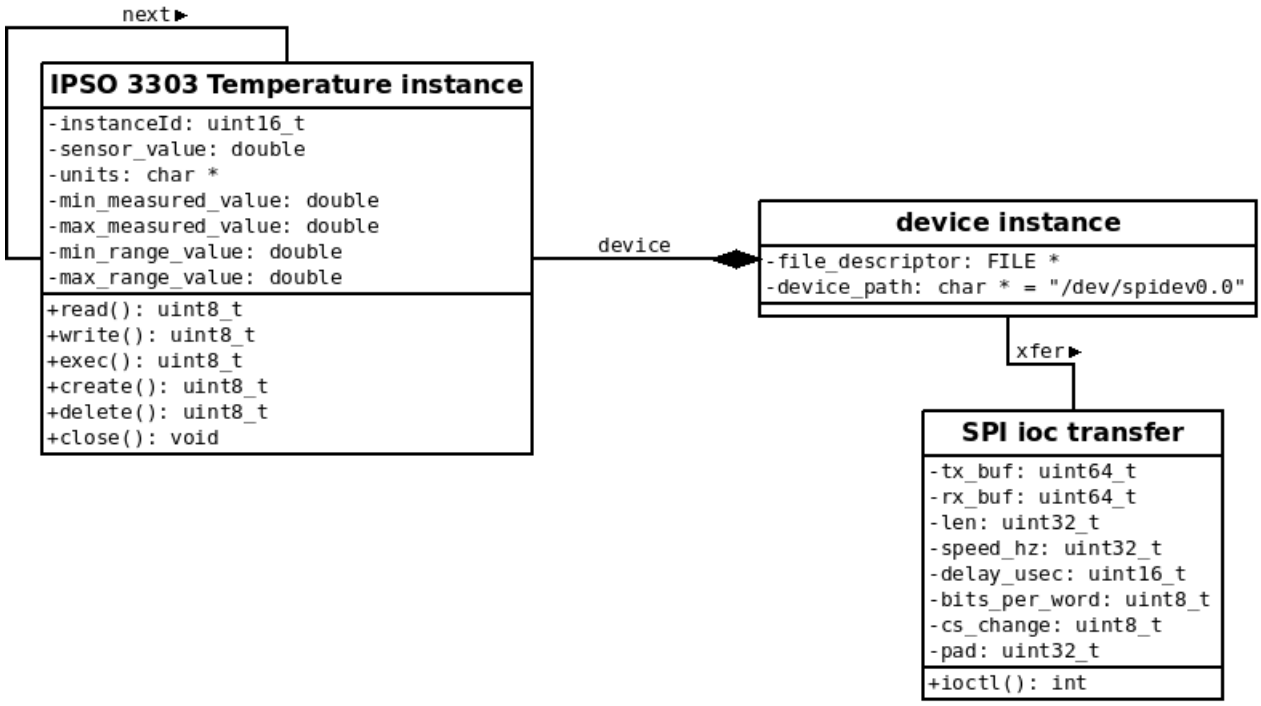

Figure 4.9: IPSO Temperature class diagram with device

the information to be sent. In particular, we used three byte: the first byte ($0x01$) is the read command, the second byte ($0x80$) is the address of the slave to be read (in this case, A0 port of the ADC) and finally the third byte is just a dummy byte where the result will be written.

The received value is a raw value that needs to be converted into voltage by multiplying it by 3.3 and dividing it by 1024; starting from the voltage value, between 0 and 3.3 volts, it is possible to derive the actual sensor value: a temperature or a luminosity value.

Chapter 5

Evaluation

In Chapter 3 the use cases and the requirements for a secure bootstrap architecture using LWM2M were showed; in this evaluation chapter we will show an execution of the use case, showing which ones of the requirements are fulfilled by the prototype. In the second section of the chapter, the results of performance measurements will be shown, evaluating the impact of adding a security layer in the communication stack; the metrics used are bytes count, number of packets sent over the network, memory occupation of the prototype and stress testing over long period of time.

5.1 Prototype Testbed

To evaluate the prototype, several tests with different configurations have been made. For instance, for the memory analysis test the prototype has been executed on a single workstation, where all the communications were made locally; the measurement has been made using a Linux tool called Valgrind [58]; this configuration has been also used for a simple use case execution and requirements fulfilment verification.

Another configuration used for prototype evaluation purposes includes the use of the Raspberry Pi and two sensors; to easily connect the sensors to the embedded device, a shield has been used. A shield is an expansion board used to interface Raspberry's GPIO pins to many small components; for our testbed, a Linker Kit Base Shield [59] has been used, with a temperature sensor (a thermistor) and a luminosity sensor (a photoresistor). Two jumpers of the board are connected to an integrated circuit, a MCP3004 analog-to-digital converter, soldered on the shield; the output of the ADC

module is connected to the MISO/MOSI ports of the Raspberry's SPI interface. In Figure 5.1 a picture of the testbed is showed; the two sensor are showed as well, connected to JP1 and JP2 of the shield.

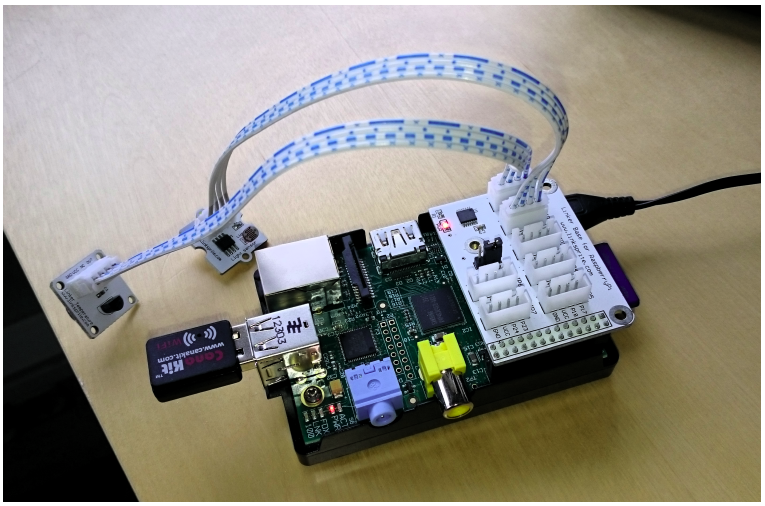

Figure 5.1: Picture of the testbed using a Raspberry Pi and a Linker Kit Base Shield

5.2 Functionality Evaluation

In this section, a normal execution of the prototype is commented; the focus is on the execution of a generic use case, using real sensors, and the evaluation of the requirement fulfilment with a simple test case execution.

5.2.1 Use Case Execution

A simple use case execution has been preliminary made; in this scenario, the LWM2M instances have been tested locally. The security keys for LWM2M Server and LWM2M Bootstrap Server have been written into the "keys.config" files; the keys are obtained from a TinyDTLS client-server test application. The first use case execution is a simple LWM2M Client Initiated Bootstrap; LWM2M servers are up and running, LWM2M Client is started with the **-b** argument. The Client sends the bootstrap request to the local LWM2M Bootstrap Server; the Bootstrap Server replies with an ACK message, followed by two delete messages (to purge Bootstrap Server information at the Client) and two create messages, with the new Security and Server instances.

Once the information is sent, the Client stores it locally and register itself to the

LWM2M Server. Every new connection is preceded by a DTLS handshake, in each of which a total amount of 15 DTLS packets are exchanged between client and server. When the Client finally registers to the Server, the device management session can start. Simple command are performed, such as read, write and exec, on the LWM2M 1024 Test Object; a simple observe/notify test has been made too, on a single resource (i.e. /1024/10/1): the Client responds to the request properly, and every change in the resource (using a shell command at the client application) is notified back to the LWM2M Server.

Another use case execution was made using the second testbed previously described, the one with the Raspberry Pi; the LWM2M Client was run on the Raspberry Pi in super-user mode (since it reads a '/dev/' file) with **-s** and **-b** arguments; the communication happens through the wireless channel, using a local WiFi network. The Client succeeds to bootstrap and register to the LWM2M Server (running on a laptop in the same network) and periodically updates the value of IPSO objects through sensors. The LWM2M Server can send an observe request for a resource; on every resource change, a notify message is sent from the LWM2M Client to the LWM2M Server correctly. The two executions described above were also made in NoSecurity mode, without using the DTLS layer.

The final use case execution is made using the entire bootstrap architecture, with the LWM2M server instances running on the MINT machine (with **-w** argument). The out-of-band mechanism is being implemented, but the LWM2M Client can properly bootstrap to the Web Server; when a bootstrap request is received by the LWM2M Bootstrap Server, a notification is sent to the Web Server and a prompt is showed in the user interface. If the bootstrap request is approved, the bootstrap process continues and the LWM2M Client can register to the LWM2M Server. When this happens, the Web Server automatically asks for its LWM2M Location Object instance in order to properly locate it on the map. At the end of the use case execution, the device is properly showed on the map. The read, write and exec commands work properly, returning the correct values to the Web Server; multiple instance resources are not supported yet.

5.2.2 Functional Requirements Fulfilment

The use case executions showed that most of the requirements listed in section 3.2 were satisfied; a simple overview is showed in Table 5.1.

#	**Functional Requirements**	
1	Automatic Bootstrap	Yes
2	Selective Bootstrap	Yes
3	Semantic Interoperability	Yes
4	Randomly Generated Key Pairs	Partly
5	Trust anchor for Raw Public Key	Yes
6	Protection against network attacks	Partly
7	Robust Bootstrapping	Partly
#	**Security Requirements**	
1	Confidentiality	Yes
2	Integrity	Yes
3	Non-repudiation	No
4	Authenticity	Yes
5	Availability	Partly

Table 5.1: Requirements fulfilment overview

1) Automatic Bootstrap When LWM2M Bootstrap Server is running in normal mode, every bootstrap request is served automatically.

2) Selective Bootstrap With the introduction of a new shell command, 'bs' command, it is possible to bootstrap only a specific set of LWM2M Clients when LWM2M Bootstrap Server is running in Web Server mode.

3) Semantic Interoperability The semantic interoperability is granted by the use of IPSO Smart Objects has data model to store information coming from sensors; the information is also made available using the resource model of the IPSO Objects, so that any LWM2M Server can fetch it from the LWM2M Client. A partial semantic interoperability test has been made at the IPSO InterOp event in Kista on May 20th.

4) Randomly Generated Key Pairs This requirement is partly fulfilled; in TinyDTLS library there is an API that allows a caller to randomly generate elliptic curve public and private keys using *Pseudo-Random Number Generator* (PRNG) functions. However, this function is not a perfect random generator since PRNG depends on actual time value; for better performance, a *Cryptographically Secure PRNG* (CSPRNG) algorithm should be used.

5) Trust anchor for Raw Public Key With the introduction of the out-of-band public key validation mechanism, using a QR Code sent via HTTP(S) to the Web Server, the Raw Public Key is verified and bound to the LWM2M Client. This requirement has been partially implemented.

6) Protection against network attacks DTLS provides protection against many network attacks. Full protection is granted against information eavesdropping; this is true as long as the shared key is not compromised and as long as the Diffie-Hellman assumption holds. Whenever the LWM2M Client is successfully authenticated, the channel is secure against Man-in-the-Middle attacks. Only completely anonymous sessions are inherently vulnerable to such attacks, but the implementation does not negotiate ciphersuites that support anonymous sessions. DTLS is, however, susceptible to a number of Denial-of-Service attacks: while the introduction of stateless cookie exchange in the Server's HelloVerifyRequest message prevents DoS amplification attacks, and therefore protects constrained nodes, the Server still remains a vulnerable target; thanks to the cookie mechanism, the Server is protected against CPU-intensive computation, but a large number of simultaneously initiated session can still undermine the availability of LWM2M Bootstrap Server or LWM2M Server.

7) Robust Bootstrapping In case of sleeping nodes or failure of one of the endpoints, the implemented prototype doesn't manage the situation in a particular way; it only reports the most appropriate CoAP response code for problem like server errors or unresponsive/sleeping clients.

5.2.3 Security Requirements Fulfilment

The security requirements are instead granted by the addition of DTLS layer; they are:

1) Confidentiality The property of confidentiality is granted by DTLS by the use of an encryption algorithm to encipher LWM2M messages. The algorithm used is AES with a 128 bit long key; the shared key is agreed upon the two entities using ephemeral Diffie-Hellman algorithm, starting from raw public keys. Only the owner of the shared secret can decrypt the message and its contents.

2) Integrity The property of integrity is granted by the use of a MAC, a Message Authentication Code, in DTLS; during the handshake protocol phase, once the security parameters has been negotiated, a MAC is generated using all the messages exchanged since that moment. If the signature is verified by the two entities, the MAC key is validated and will be used to sign every Application Data message, providing the integrity property; the MAC signature provides an overhead of 16 byte for each CoAP message.

3) Non-repudiation Since a MAC is used instead of a digital signature, the property of non-repudiation could not be satisfied by the prototype; the MAC is generated using a shared key between two endpoints, so the property is not verified. The non-repudiation property is satisfied when the signature is generated starting from a private key and can be verified using a public key; the keys must be certified by a public *Certificate Authority* (CA).

4) Authenticity The MAC provides the property of authenticity, since only the owners of the shared secret can sign the message and verify the correctness of the MAC; the shared secret is agreed upon the entities with the Diffie-Hellman Key Exchange Algorithm.

5) Availability The property of availability was tested with a stress test in section 5.3.3; the test shows that the servers have an high availability over a period of one week, and the Client was not failing over this period of time (e.g. due to a segmentation fault). This was only a software availability test that shows that the prototype has a high durability over time; no hardware was tested over long period of time.

5.3 Performance Evaluation

The performance tests of the prototype has been made using LWM2M Client in Resource Simulation mode; several tools has been used to evaluate prototype's performances in terms of bytes transmitted and memory used. The tests were made to compare two security modes, with or without DTLS, in order to evaluate the impact of adding a DTLS layer in the device management stack. A stress test has also been made, running the application locally on a period of one week for stability.

5.3.1 Packet Size and Number

This experiment was designed to evaluate the overhead introduced by the DTLS layer in the LWM2M stack; the prototype was executed using a wireless interface for communication, meanwhile a packet sniffer (i.e. Wireshark) ran in background to capture packets sent and received through the interface. Wireshark is capable of parsing the captured data in order to distinguish the various protocols headers, providing access to all their fields as well as their sizes and, in particular, the effective DTLS header

DHCPv6 Message Type	Packet Size	Packets Number	Direction
Solicit	56 bytes	1	Uplink
Advertise	102 bytes	1	Downlink
Request	70 bytes	1	Uplink
Reply	102 bytes	1	Downlink
Total	330 bytes	4	2 Uplink 2 Downlink

Table 5.2: DHCPv6 packets overview

dimension. Moreover, analysing the timestamp of the packets it is also possible to estimate the time of the elaboration in each step of the protocols.

To estimate DTLS overhead, we assumed a typical scenario: a device is turned on, the LWM2M Client starts, gets an IPv6 address through DHCPv6 protocol and then tries to bootstrap and register to its LWM2M Bootstrap Server and LWM2M Server instances; a regular device management session is executed, with a certain number of notify messages. This experiment was made with and without DTLS-provided security mechanisms. In Table 5.2 there is an overview of the messages of the DHCPv6 protocol that a device needs to run when turned on to obtain an IPv6 address; the messages needed to negotiate an IPv6 address are four, one per kind, with a total amount of 330 bytes transmitted over the network between the DHCPv6 Client (on the device) and the DHCPv6 Server.

In Table 5.3 a full overview of DTLS Handshake Protocol is showed; a total amount of 1128 bytes over 15 packets are sent over the network just to establish a secure connection between two endpoints. In Table 5.4 there is an overview of the packets exchanged during LWM2M protocol communication, divided into a bootstrap phase, where the communication is between a LWM2M Client and a LWM2M Bootstrap Server, and a device management phase, where the communication is between a LWM2M Client and a LWM2M Server. Note that in the latter communication, the number of packets depends on a parameter k, that is the number of Notify messages that are sent by the Client to the Server; so the amount of data transmitted depends on the parameter k.

For LWM2M message exchange, we have analysed two cases: in NoSec mode there is no DTLS additional layer for end-to-end security, while in DTLS mode each LWM2M message is wrapped inside the DTLS Record Layer; this translates in an overhead of 29 bytes per packet, corresponding to 13 bytes for the Record Layer and 16 bytes for the Message Authentication Code. Moreover, in DTLS mode there is the overhead given by the DTLS Handshake Protocol for each phase. The final result shows that there

DTLS Message Type	Packet Size	Packets Number	Direction
ClientHello	95 bytes	1	Uplink
HelloVerifyRequest	44 bytes	1	Downlink
ClientHello (with cookie)	111 bytes	1	Uplink
ServerHello	81 bytes	1	Downlink
Certificate	122 bytes	1	Downlink
ServerKeyExchange	169 bytes	1	Downlink
CertificateRequest	33 bytes	1	Downlink
ServerHelloDone	25 bytes	1	Downlink
Certificate	122 bytes	1	Uplink
ClientKeyExchange	91 bytes	1	Uplink
CertificateVerify	101 bytes	1	Uplink
ChangeCipherSpec	14 bytes	1	Uplink
Finished	53 bytes	1	Uplink
ChangeCipherSpec	14 bytes	1	Downlink
Finished	53 bytes	1	Downlink
Total	1128 bytes	15	7 Uplink 8 Downlink

Table 5.3: DTLS Handshake packets overview

is a huge difference between the communication with and without DTLS, since DTLS was not developed for constrained environment.

In Table 5.5 there is the final result for a short device management session with only 10 Notify messages; a comparison between NoSec mode and DTLS mode is showed, both for total amount of bytes transmitted and number of packets exchanged. In DTLS mode, the number of bytes transmitted is increased by a factor of 5; the handshake protocol contributes the most to the bytes transmitted, representing the 58% of the total amount of data transmitted. The other 42% is Application Data. The number of packets is doubled in DTLS mode compared to NoSec mode.

In Table 5.6 a similar analysis is showed; in this case, the device management session is longer: 100 Notify messages has been transmitted from LWM2M Client to LWM2M Server. In this case, the amount of bytes transmitted in DTLS mode is increased by a factor of 4 compared to NoSec mode; moreover, in DTLS mode only the 29% of transmitted data is used for the handshake. Also the results on the number of packets are improving: the difference between the two modes is given only by the 30 packets necessary for two DTLS handshakes.

Finally, the impact of the DTLS layer is analysed in Figure 5.2; this analysis was made using four values of the k parameter, that are 10, 100, 1000 and 10000 Notify

LWM2M Message Type	**NoSec Packet Size**	**DTLS Packet Size**	**Packets Number**	**Direction**
POST Bootstrap Request	28 bytes	57 bytes	1	Uplink
ACK 2.03 Valid	4 bytes	33 bytes	1	Downlink
DELETE	8 bytes	37 bytes	2	Downlink
ACK 2.02 Deleted	4 bytes	33 bytes	2	Uplink
POST Security Obj	42 bytes	176 bytes	1	Downlink
POST Server Obj	19 bytes	48 bytes	1	Downlink
ACK 2.01 Created	4 bytes	33 bytes	2	Uplink
DTLS Handshake		1128 bytes	15	7 Uplink 8 Downlink
Bootstrap Phase	125 bytes	1648 bytes	10/25	5/12 Uplink 5/13 Downlink
POST Registration Request	115 bytes	144 bytes	1	Uplink
ACK 2.01 Created	9 bytes	38 bytes	1	Downlink
GET Observe Request	21 bytes	50 bytes	1	Downlink
Notify 2.05 Content	13 bytes	42 bytes	**k**	Uplink
PUT Registration Update	9 bytes	38 bytes	1	Uplink
ACK 2.04 Changed	4 bytes	33 bytes	1	Downlink
DELETE De-registration	9 bytes	38 bytes	1	Uplink
ACK 2.02 Deleted	4 bytes	33 bytes	1	Downlink
DTLS Handshake		1128 bytes	15	7 Uplink 8 Downlink
Device Management Phase	171+13k bytes	1502+42k bytes	7/22+k	3/10+k Uplink 4/12 Downlink
Total	296+13k bytes	3150+42k bytes	NoSec: 17+k DTLS: 47+k	8/22+k Uplink 9/25 Downlink

Table 5.4: LWM2M packets overview

for k=10	**NoSec Packet Size**	**DTLS Packet Size**	**Packets Number**
DHCPv6	330 bytes	330 bytes	4
DTLS Handshake		1128 bytes	15
LWM2M Bootstrap	125 bytes	1648 bytes	10/25
LWM2M Device Management	301 bytes	1922 bytes	17/32
Total	756 bytes	3900 bytes	NoSec: 31 DTLS: 61

Table 5.5: Final results of the packet size experiment (for k=10)

for k=100	**NoSec Packet Size**	**DTLS Packet Size**	**Packets Number**
DHCPv6	330 bytes	330 bytes	4
DTLS Handshake		1128 bytes	15
LWM2M Bootstrap	125 bytes	1648 bytes	10/25
LWM2M Device Management	1471 bytes	5702 bytes	107/122
Total	1926 bytes	7680 bytes	NoSec: 121 DTLS: 151

Table 5.6: Final results of the packet size experiment (for k=100)

messages. The graphs show that for longer device management session, the overhead introduced by the bootstrap phase decreases. The amount of additional data added by DTLS, instead, always represents the majority of the bytes exchanged through the network, not only because of the handshake phases, but substantially because of the Record Layer added in each transmitted packet.

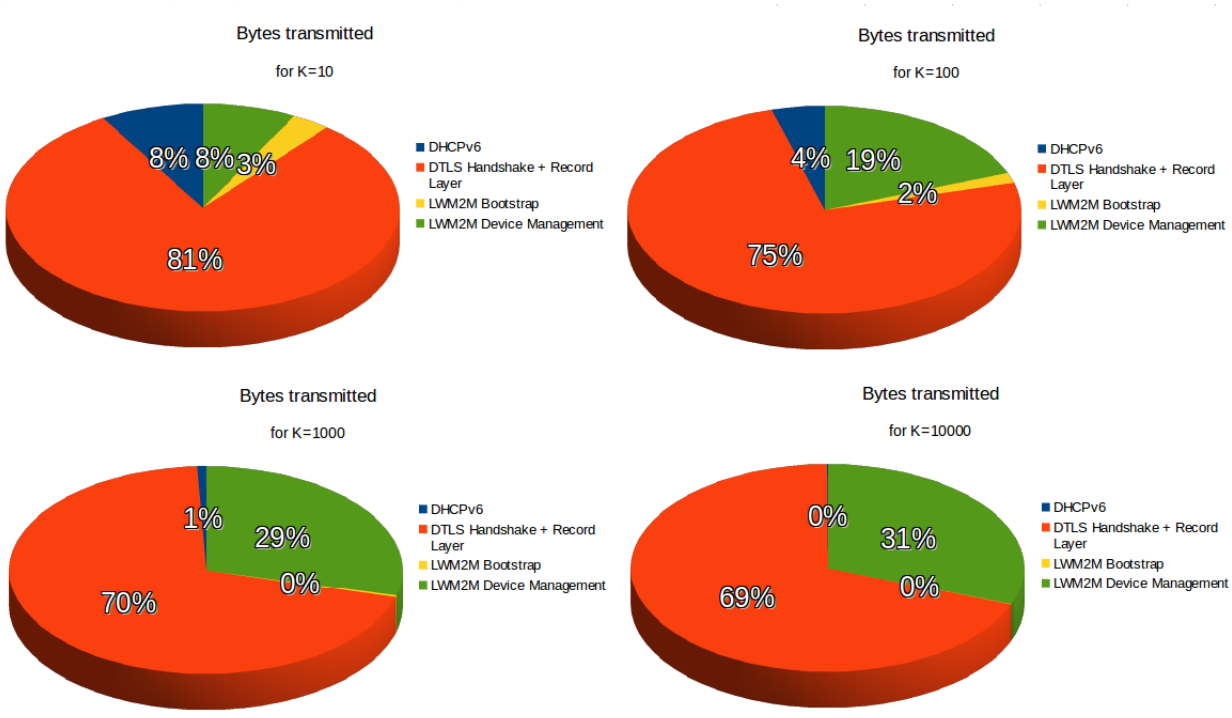

Figure 5.2: The amount of bytes transmitted by each protocol

The packet overhead experiment showed the huge amount of overhead introduced by the DTLS layer; this overhead drastically reduces the 102 bytes maximum frame size available at the media access control (without link-layer security) [16]. Analysing wireshark captures, we can notice that the overhead coming from the lower layers is given by: 14 bytes for data link layer (i.e. Ethernet II), 40 bytes for IPv6 and 8 bytes

for UDP; this leaves only 40 bytes for application data. If DTLS requires 29 bytes for each packet, only 11 bytes are left for CoAP and LWM2M. Moreover, many packets exchanged during the DTLS handshake phase exceed the 102 bytes MTU, e.g. the Key Exchange messages.

5.3.2 Memory Analysis

The memory analysis of the application was executed in Unix environment using a framework for dynamic analysis called Valgrind [58]. Valgrind has many tools, useful for different purposes; in our analysis we have used two of them: *MemCheck* and *Massif*. MemCheck is a tool that monitors the execution of the application in order to keep track of memory allocation and discover any memory leaks or errors in the code, and it is also useful to derive the memory footprint of the application; Massif instead is a very simple heap profiler, that gives an overview of the memory usage overtime. Experiments have been executed both in NoSec and DTLS mode, in order to evaluate the impact of the introduction of DTLS in LWM2M.

Memory Footprint

To measure the memory footprint of the prototype, the MemCheck tool of Valgrind has been used; this tool monitors the memory usage of an application, by checking its calls of functions such as *malloc()* or *free()*. Hence, the result of the MemCheck tool is the total amount of memory allocated and freed during the execution of the program. Additional information are given about any memory leaks that happens during the execution, for example if an allocated portion of memory is not properly freed when it is not needed any more.

The MemCheck tool was run on the prototype in both NoSec and DTLS mode, at the same condition. The LWM2M Client, in Resource Simulation mode, sends a bootstrap request and the registers to the LWM2M Server; then the LWM2M Server sends an observe request to the Client and receives a Notify message every 10-15 seconds, randomly. When 100 Notify messages has been sent, the execution of the prototype is stopped and the results are collected.

The first result of the MemCheck experiment is that in both modes and in the three applications all the heap blocks are regularly freed, so no leaks are possible. This is important, especially for the LWM2M Client, since we are testing a functionality that

Endpoint Type	LWM2M Memory Consumption	DTLS Memory Consumption
LWM2M Client	155087 Bytes - 73%	56173 Bytes - 27%
LWM2M Bootstrap Server	2825 Bytes - 31%	6423 Bytes - 69%
LWM2M Server	1730 Bytes - 22%	6004 Bytes - 78%

Table 5.7: DTLS Memory Footprint

needs to work in a constrained environment, on devices with constrained resource (i.e. low RAM memory availability); if the LWM2M Client had memory leaks, on long executions that would be translated into memory saturation and possibly a segmentation fault into the software, making the device useless.

In Table 5.7, an overview of the experiment is showed. To calculate the impact of DTLS on the memory, a simple difference between the memory usage in the two modes has been made. This result is also graphically showed in Figure 5.3. We can deduce from graphics that TinyDTLS occupies a big part of memory in the server applications, probably due to the allocation of the DTLS context to keep track of all the security parameters and the sessions between the endpoints. Moreover, in the client application TinyDTLS represents only a small share of the memory usage, since the LWM2M Wakaama application periodically updates the simulated sensor value and sends back a Notify message to the Server.

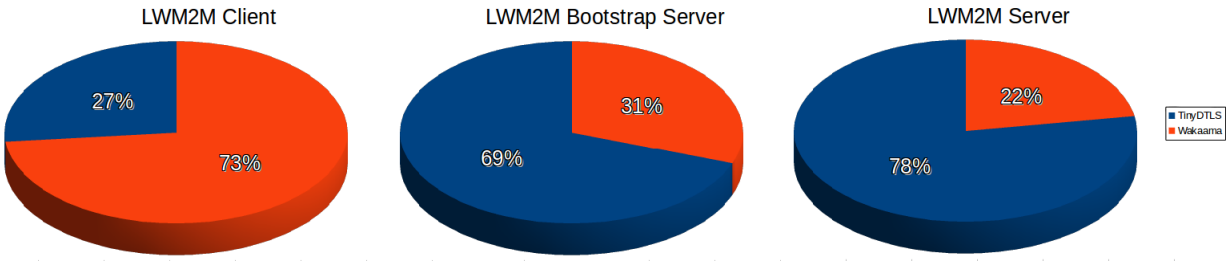

Figure 5.3: Memory Footprint of the prototype

Heap Profile

To have an idea of the heap usage of the prototype, a heap profiling tool has been used; Valgrind's Massif tool was executed when the prototype was running in the usual configuration. The Massif tool operates making some snapshots of the heap memory every time a particular condition is verified (e.g. a portion of memory is freed by the application). Taking into account each snapshot, a trend of the memory usage over time is plotted. The experiment was executed both in NoSec and DTLS mode,

at the same conditions: 100 Notify messages are sent by the Client after a regular bootstrapping phase is executed.

In Figure 5.4, 5.5 and 5.6 the memory usage plots in NoSec mode are showed. LWM2M Client presents mainly two peaks at the beginning of the experiment, representing the bootstrapping phase and the registration phase; after that, the device management session maintains a regular memory usage, despite the number of Notify messages sent. This could be because LWM2M Client doesn't keep track of the sensor values gathered during the device management session; it only sends the Notify packet and frees the memory relative to that reading. LWM2M Bootstrap Server is an almost idle node, serving just one bootstrapping request at the beginning of the execution and then just listening on the socket; there is another peak when the application is about to close. Finally, LWM2M Server presents a more irregular plot, due to the Notify messages coming from the LWM2M Client. Note that the memory usage peaks are relative to the stack area, and not to the heap area, meaning that the memory fills on specific function calls.

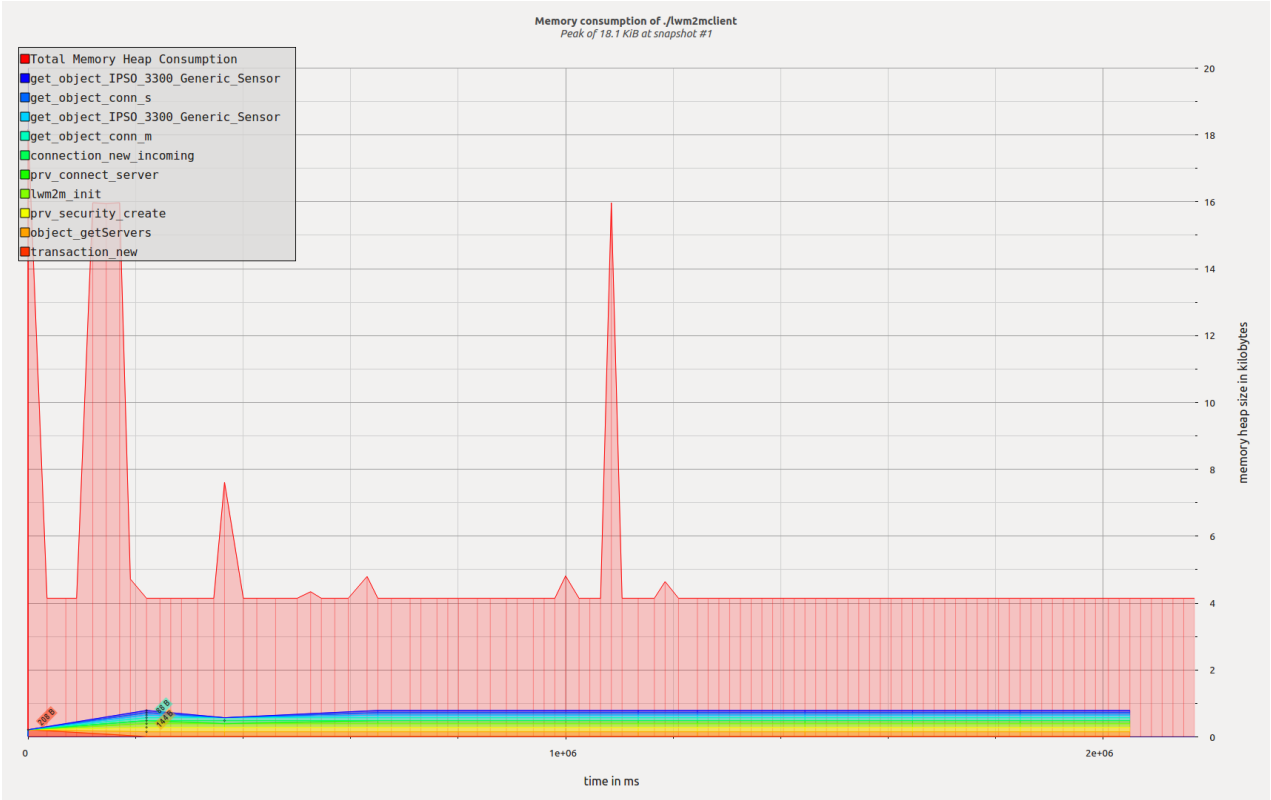

Figure 5.4: Memory consumption of LWM2M Client in NoSec mode

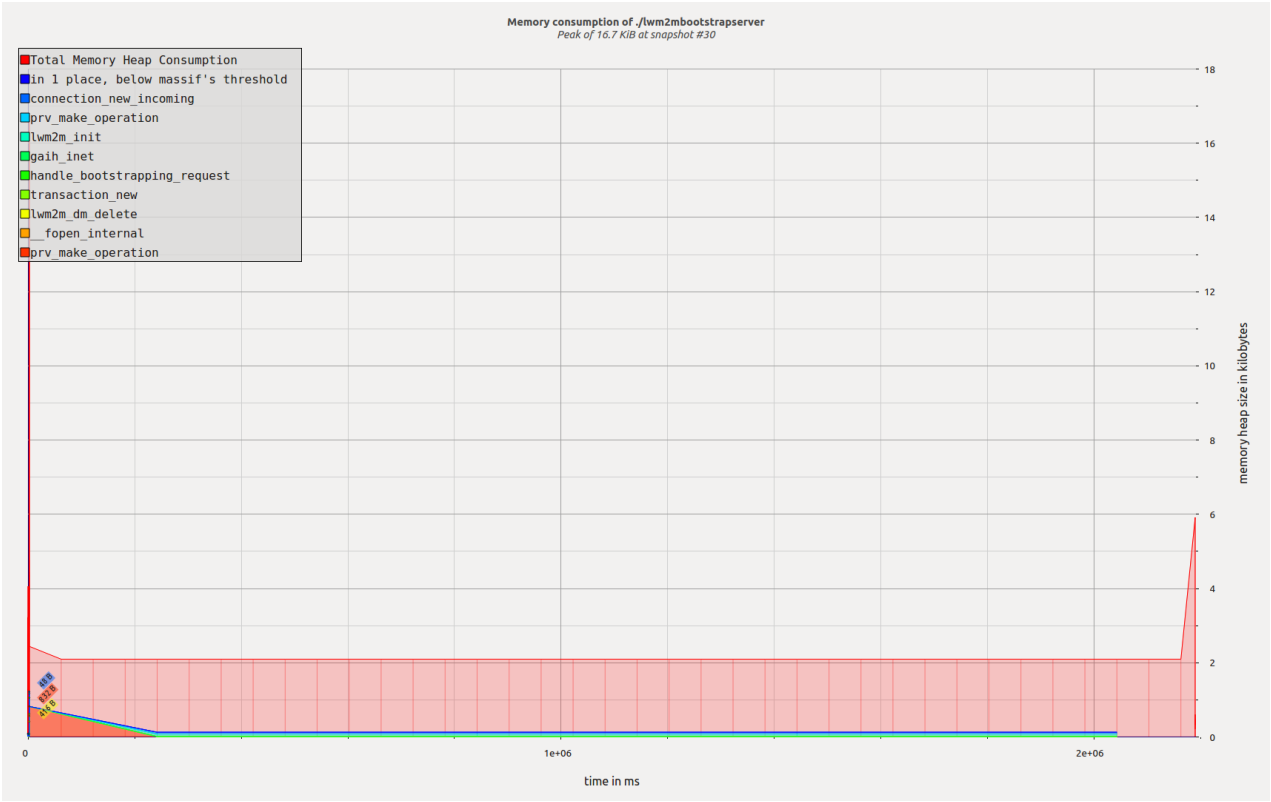

Figure 5.5: Memory consumption of LWM2M Bootstrap Server in NoSec mode

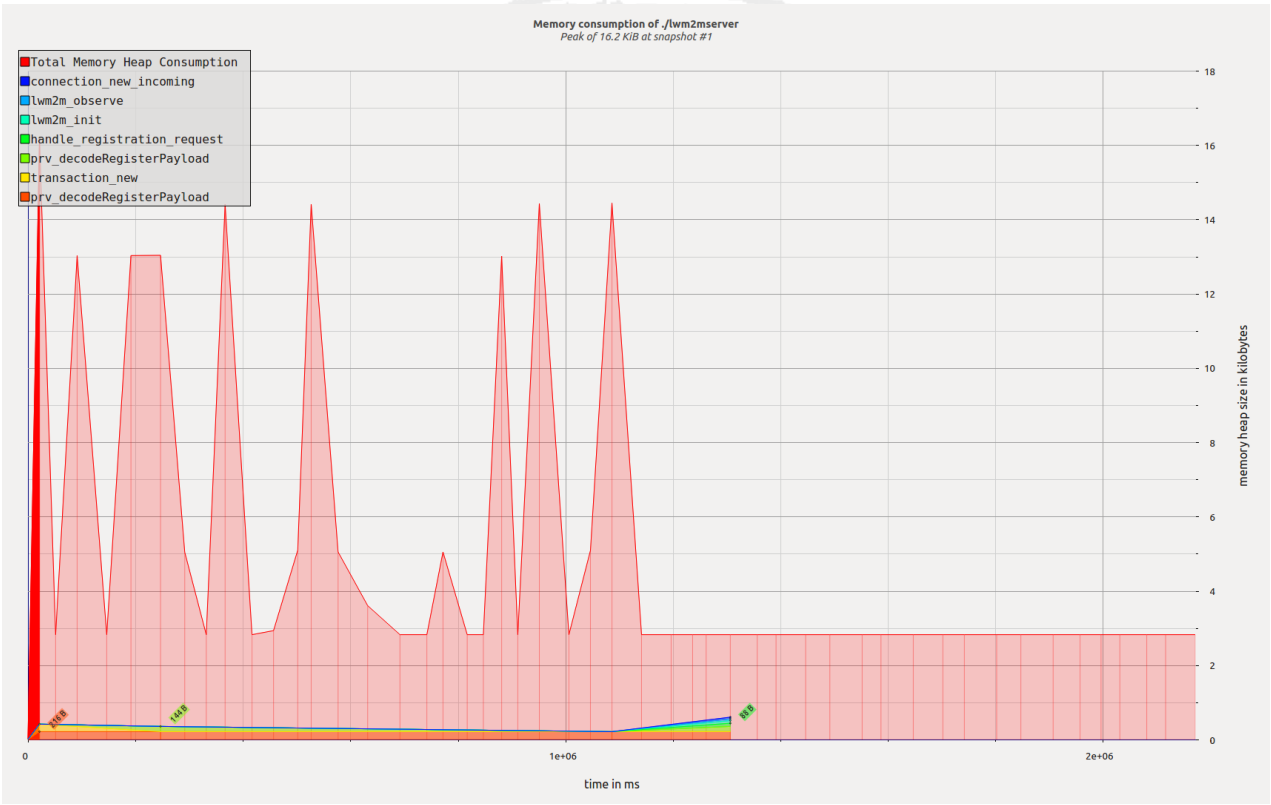

Figure 5.6: Memory consumption of LWM2M Server in NoSec mode

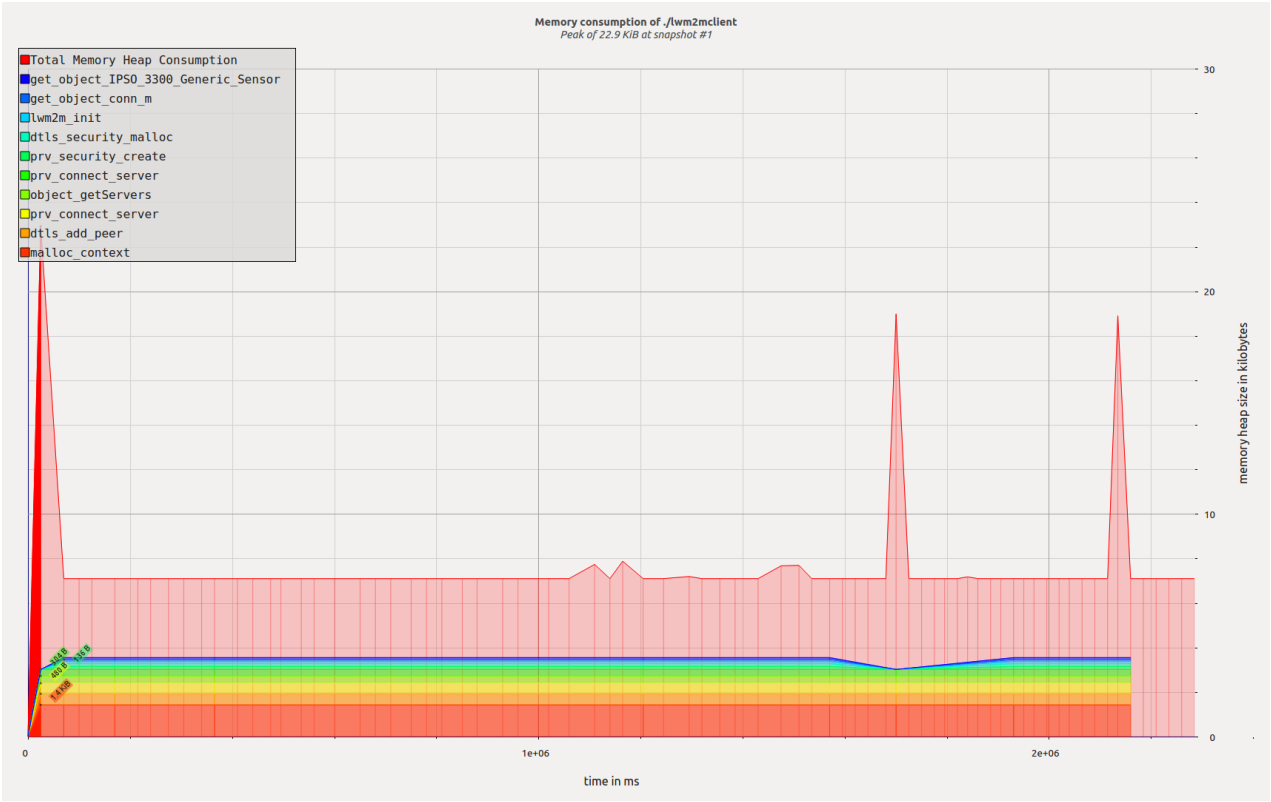

Figure 5.7: Memory consumption of LWM2M Client in DTLS mode

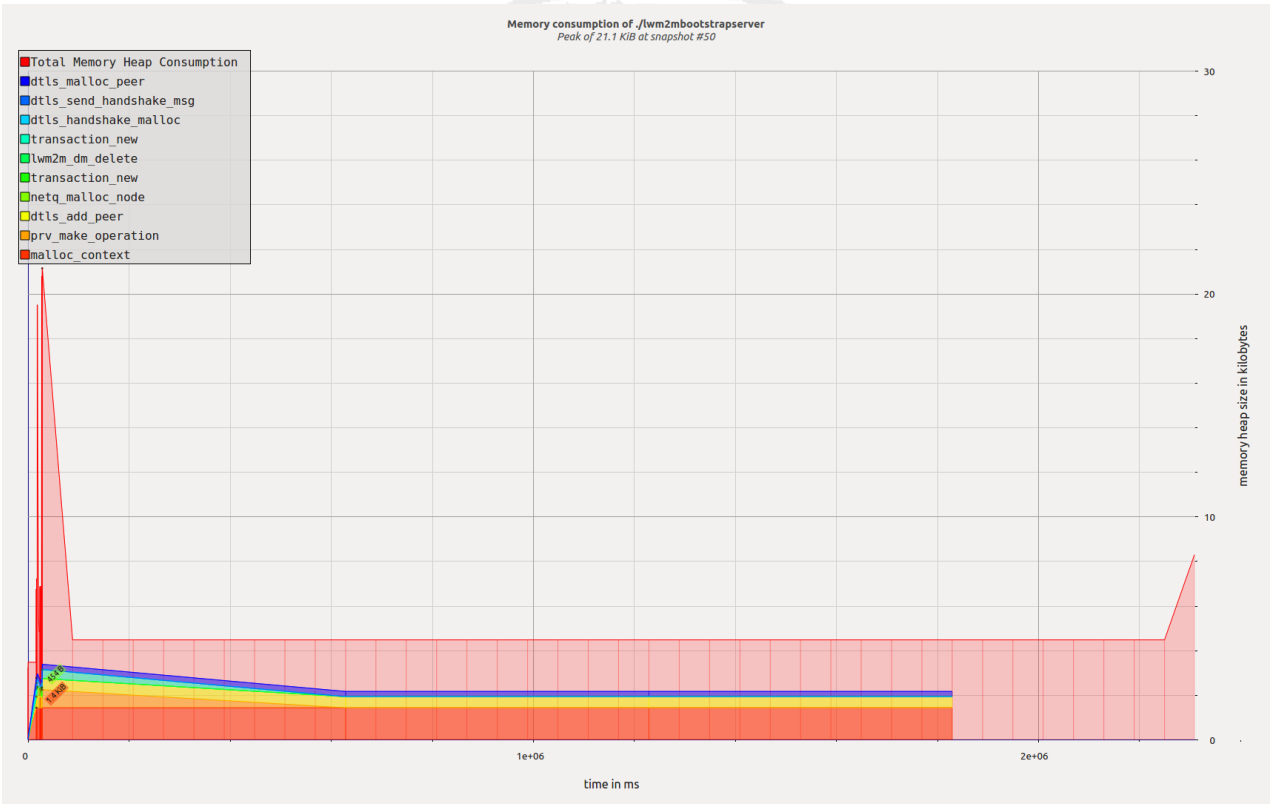

Figure 5.8: Memory consumption of LWM2M Bootstrap Server in DTLS mode

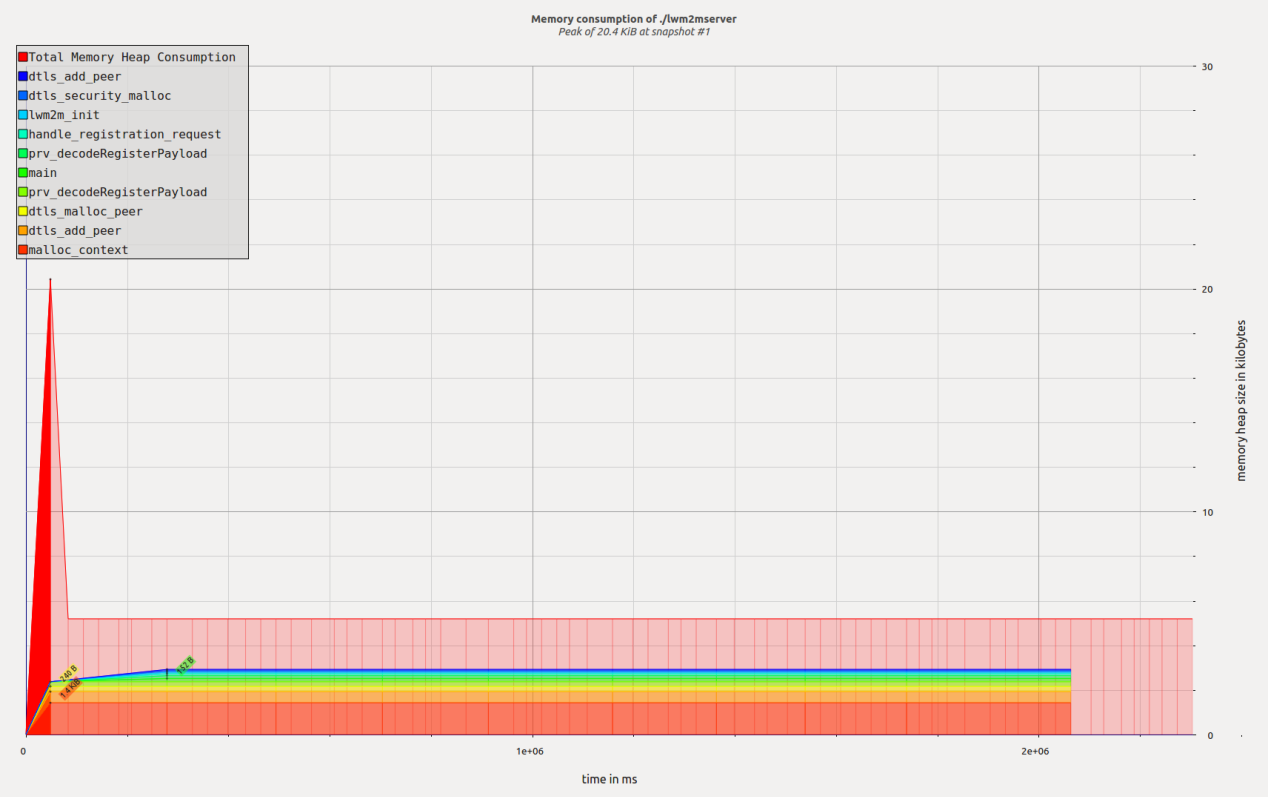

Figure 5.9: Memory consumption of LWM2M Server in DTLS mode

In Figure 5.7, 5.8 and 5.9 the same analysis is made for the prototype in DTLS mode. LWM2M Client shows similar results to the NoSec mode, except for the total heap usage (the coloured lines at the bottom of the plot) that is greater, due to the TinyDTLS's *malloc_context()* (to store context information), showed in red at the bottom. Similar considerations can be made for the LWM2M Bootstrap Server. Finally, LWM2M Server in DTLS mode presents a more regular trend of memory usage.

An interesting focus of the experiment is on the memory usage peaks of each endpoint, representing the worst cases of the execution. These values could be seen as a upper bound to the memory usage of the application, meaning that the prototype could be executed on a device with a certain RAM requirement. In Table 5.8 the memory usage peaks are showed, keeping into account the differences between NoSec and DTLS modes in the three application's endpoints. The DTLS layer introduces a 4.5 KB of memory usage improvement, by average; in the LWM2M Client, that is the critical endpoint of the application running on a constrained device, DTLS introduces an additional memory consumption of 4.8 KB, representing the 21% of the total memory usage of the application.

Endpoint Type	NoSec Mode Memory Peak	DTLS Mode Memory Peak
LWM2M Client	18.1 KB	22.9 KB
LWM2M Bootstrap Server	16.7 KB	21.1 KB
LWM2M Server	16.2 KB	20.4 KB

Table 5.8: Memory consumption peaks

5.3.3 Stress Test

A stress test has been executed on the prototype. The test consisted in a long period run of the prototype, in which normal sensor's update where sent from LWM2M Client to LWM2M Server. The purpose of the test is to check if the conditions for the correct execution of the prototype are held over long periods of time; in particular, we were interested in testing the durability of the DTLS session, but also the durability of the LWM2M Registration mechanism (related to the lifetime of the Client, fixed to 300 seconds).

The execution was the usual: the LWM2M Client sends a bootstrapping request to the LWM2M Bootstrap Server and it receives the information needed to securely register to the LWM2M Server. After that, the Server sends an Observe request to the Client, which starts to sends back Notify messages every 10-15 seconds. During the period of the test, Valgrind's MemCheck tool was also used to test if the prototype is leaking memory on long time period.

A preliminary stress test was made over 3 days, from 27 July 2015 until 30 July 2015; the test showed that after more than 11,000 Notify messages, the LWM2M Client was still running and correctly registered to the LWM2M Server, that was still receiving correctly the encrypted Notify messages; no DTLS errors or alerts were detected. From a memory point of view, no leaks are detected; the total allocated (and freed) bytes of the application were 20,672,314 for the LWM2M Client, 9,249 for the LWM2M Bootstrap Server and 45,720 for the LWM2M Server.

A longer stress test was made from 5 August 2015 until 12 August 2015; the application ran for 7 days, a total amount of 27,000 Notify messages were sent from the LWM2M Client to the LWM2M Server, after a correct bootstrapping procedure has been executed. The application showed no particular issues, running consecutively for the time of the test; no segmentation fault were noticed, nor DTLS errors or alerts. The total amount of memory allocated and freed by the endpoints was 52,085,484 bytes for the LWM2M Client, 9,249 bytes for the LWM2M Bootstrap Server and 126,792 bytes

for the LWM2M Server.

Chapter 6

Conclusions

The goal of this thesis work was to present a secure bootstrapping architecture for IoT devices, providing a design, an implementation and an evaluation of a working prototype. This work gives an overview of the underlying protocol implementations and new components merged to assemble the proposed application and outlines their relationships to ensure their correct collaboration.

After the description of the communication patterns and protocols used in the IoT field, a solution to provide end-to-end security to a distributed constrained application has been described. In the design of the prototype a use case and the software requirements have been introduced, along with a description of the architecture built. In the implementation chapter, a detailed description of both hardware and software parts of the prototype has been provided, focusing on the protocol implementation used to built the secure bootstrapping application for device management in IoT.

Experimental results showed that the memory footprint of the application is very low, despite the unoptimized state of protocols implementation; the memory consumption of the prototype fits well the requirements of the constrained environment of the IoT field. On the other hand, the security operations of DTLS implementation introduce a considerable delay for each handshake carried out every time two nodes begin a new communication; once the security session has been established, the security operations introduce another little delay for the decryption and signature validation of each Application Data packet, as well as a little delay for encrypt and sign any newly generated packet.

Moreover, the introduction of DTLS layer translates in a huge protocol overhead,

since each CoAP message used by LWM2M will carry additional 29 bytes of security information. The overhead of DTLS has been evaluated to be the 69% of the total information transmitted over long device management sessions. Furthermore, in the handshake phase the two endpoints involved in the negotiation often exchange packets that are too big for a constrained network with strict bounds on the Maximum Transmission Unit.

While the memory overhead of the introduction of DTLS is very low, settling to 4.5 KB of additional memory required for a secure communication, the message overhead introduced by the security layer is almost unbearable for IoT applications. Since the memory footprint of the prototype, specifically of the LWM2M Client, is very restrained, the prototype is suitable to run in many small constrained devices; in the other end, packets encapsulated into DTLS reach sizes that are not suitable for constrained environment, translating in resource and power consumption into the devices.

Summarizing, the prototype implements the secure bootstrapping architecture designed during this thesis work, it executes with no errors on a Raspberry Pi and the performance evaluation stated that DTLS protocol might introduce an heavy overhead in the number of transmitted bytes, translating into an undesirable fragmentation in a low power and lossy network.

6.1 Future Work

The device management application developed during this thesis work can be improved, implementing missing functionalities or optimizing existing ones, for a better overall performance in terms of memory occupation, packets size and power consumption.

Wakaama Improvement By the end of the thesis, Wakaama implementation was still an alpha version and no releases were made yet. Some important LWM2M functionalities were missing, hence one of the future work for the prototype could be improving Wakaama library, for example by adding support for the discover method, JSON support as data encoding or the Access Control List mechanism.

DTLS Compression As stated in [36], a stateless DTLS compression is possible in order to reduce the overhead generated by the introduction of the security layer in the application; a compressed version of TinyDTLS library could be developed,

optimizing packets size.

Fragmentation Support Both Wakaama and TinyDTLS library are not providing support for fragmentation in the application layer; hence, it is possible that the application developed during this thesis work sends packets that are exceeding the upper bound of 102 bytes for 802.15.4 [49] standard. The support for fragmentation could be added in Wakaama using CoAP's Block Option.

Multiple Instances Resource support for Web Interface The Web Server could ask for a multiple instances resource in the LWM2M Client; a support for this feature should be added in the LWM2M Server's Web Interface, in order to build the proper JSON response for the Web Server.

Testing on other devices The prototype has been tested on two embedded devices, the Raspberry Pi and the Intel Edison; as further analysis, a run test could be made on other embedded devices, for example on Arduino, Samsung Artik 1 or Waspmote.

Power Measurements A comparison between several different devices could be made in terms of power consumption, to test the impact of the prototype on embedded devices.

Bibliography

[1] A. Bruneau, "REST - REpresentational State Transfer - Les Concepts," March 2008.

[2] Z. Shelby, K. Hartke, and C. Borman, "The Constrained Application Protocol (CoAP)." http://tools.ietf.org/html/rfc7252, June 2014.

[3] Open Mobile Alliance, "Lightweight Machine to Machine: Technical Specification." Candidate version, 1(10), December 2013.

[4] A. Tanenbaum and D. Wetherall, *Computer Networks*. 5th ed. Prentice Hall, 2011.

[5] IPSO Alliance, "IPSO Smart Objects Guideline," September 2014.

[6] J. Höller, V. Tsiatsis, C. Mulligan, S. Avesand, S. Karnouskos, and D. Boyle, *From Machine-to-Machine to the Internet of Things: Introduction to a New Age of Intelligence*. 1st ed. Oxford, The UK: Academic Press, 2014.

[7] S. Gupta and A. Hirdesh, "Overview of M2M," 2007.

[8] C. Kumar and R. Paulus, "A prospective towards M2M Communication," *Journal of Convergence Information Technology (JCIT)*, vol. 9, March 2014.

[9] K. Moummadi, R. Abidar, and H. Medromi, "Distributed Resource Allocation: Generic Model and Solution Based on Constraint Programming and Multi-Agent System for Machine to Machine Services," *International Journal of Mobile Computing and Multimedia Communications*, vol. 4, pp. 49–62, April-June 2012.

[10] K. Ashton, "That 'Internet of Things' Thing," *RFID Journal*, 2009.

[11] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey." Computer Networks 54, 2787-2805, 2010.

[12] Ericsson, "Ericsson Mobility Report: on the Pulse of the Networked Society," June 2015.

[13] Ericsson, "More than 50 Billion Connected Devices," February 2011.

[14] INFSO D.4 Networked Enterprise & RFID - INFSO G.2 Micro & Nanosystems, "Internet of Things in 2020 - Roadmap for the Future," May 2008.

[15] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks." http://tools.ietf.org/html/rfc6550, March 2012.

[16] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks." http://tools.ietf.org/html/rfc4944, September 2007.

[17] R. Cohn and R. Coppen, "MQTT version 3.1.1." OASIS Standard - http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html, October 2014.

[18] R. Fielding, *Fielding PhD Dissertation. Chapter 5: Representational State Transfer (REST).* PhD thesis, University of California, Irvine, 2000.

[19] R. Fielding, J. Gettys, J. Mogul, H. F. Nielsen, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1." http://tools.ietf.org/html/rfc2616, June 1999.

[20] Z. Shelby, "Constrained RESTful Environments (CoRE) Link Format." http://tools.ietf.org/html/rfc6690, August 2012.

[21] Z. Shelby, S. Krco, and C. Bormann, "CoRE Resource Directory." http://tools.ietf.org/html/draft-shelby-core-resource-directory-05, February 2013.

[22] K. Hartke, "Observing Resources in CoAP." http://tools.ietf.org/html/draft-ietf-core-observe-16, December 2014.

[23] M. Koster, A. Keranen, and J. Jimenez, "Publish-Subscribe in the Constrained Application Protocol (CoAP)." http://tools.ietf.org/html/draft-koster-core-coap-pubsub-01, March 2015.

[24] C. Jennings, Z. Shelby, and J. Arkko, "Media Types for Sensor Markup Language (SENML)." http://tools.ietf.org/html/draft-jennings-senml-10, October 2012.

[25] G. Klas, F. Rodermund, Z. Shelby, S. Akhouri, and J. Höller, "Lightweight M2M: Enabling Device Management and Applications for the Internet of Things," February 2014.

[26] J. Case, M. Fedor, M. Schofstall, and J. Davin, "A Simple Network Management Protocol (SNMP)." http://tools.ietf.org/html/rfc1157, May 1990.

[27] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and T. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile." http://tools.ietf.org/html/rfc5280, May 2008.

[28] P. Salmela and M. Sethi, "Secure Bootstrapping."

[29] C. Neuman, T. Yu, S. Hartman, and K. Raeburn, "The Kerberos Network Authentication Service (V5)." http://tools.ietf.org/html/rfc4120, July 2005.

[30] R. Needham and M. Schroeder, "Using Encryption for Authentication in Large Networks of Computer," *Communications of the ACM*, vol. 21, pp. 993–999, December 1978.

[31] ETSI 3rd Generation Partnership Project (3GPP), "Generic Bootstrapping Architecture (GBA) Technical Specification," October 2014.

[32] M. Sher and T. Magedanz, "Secure Access to IP Multimedia Services Using Generic Bootstrapping Architecture (GBA) for 3G & Beyond Mobile Networks," *ACM International Workshop on QoS and Security for Wireless and Mobile Networks*, October 2006.

[33] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication." http://tools.ietf.org/html/rfc2617, June 1999.

[34] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2." http://tools.ietf.org/html/rfc6347, January 2012.

[35] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2." http://tools.ietf.org/html/rfc5246, August 2008.

[36] S. Raza, D. Trabalza, and T. Voigt, "6LoWPAN Compressed DTLS for CoAP," *8th IEEE International Conference on Distributed Computing in Sensor Systems*, pp. 287–289, 2012.

[37] P. Wouters, H. Tschofenig, J. Gilmore, S. Weiler, and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)." http://tools.ietf.org/html/rfc7250, June 2014.

[38] R. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," tech. rep., Massachusetts Institute of Technology, Laboratory of Computer Science, Cambridge, MA, 02139, 1978.

[39] R. Rivest, A. Shamir, and L. Adleman, "Cryptographic communications system and method," September 1983.

[40] V. Miller, "Elliptic Curves and their use in Cryptography," March 1997.

[41] W. Diffie and M. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. 22, pp. 644–654, November 1976.

[42] E. Rescorla, "Diffie-Hellman Key Agreement Method." http://tools.ietf.org/html/rfc2631, June 1999.

[43] D. McGrew and K. Igoe, "Fundamental Elliptic Curve Cryptography Algorithms." http://tools.ietf.org/html/rfc6090, February 2011.

[44] D. McGrew, D. Bailey, M. Campagna, and R. Dugal, "AES-CCM Elliptic Curve Cryptography (ECC) Cipher Suites for TLS." http://tools.ietf.org/html/rfc7251, June 2014.

[45] E. Rescorla, "TLS Elliptic Curve Cipher Suites with SHA-256/384 and AES Galois Counter Mode (GCM)." http://tools.ietf.org/html/rfc5289, August 2008.

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

[47] M. C. Ocak, "Implementation of an Internet of Things Device Management Interface," Master's thesis, School of Electrical Engineering, Aalto University, November 2014.

[48] IEEE 802.11, "Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications," March 2012.

[49] IEEE 802.15.4, "Low-Rate Wireless Personal Area Networks (LR-WPANs)," September 2011.

[50] P. Karn and W. A. Simpson, "Photuris: Session-Key Management Protocol." http://tools.ietf.org/html/rfc2522, March 1999.

[51] P. Hoffman and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA." http://tools.ietf.org/html/rfc6698, August 2012.

[52] "The Raspberry Pi Foundation." http://www.raspberrypi.org, last visited on 25/06/2015.

[53] "Intel Edison." http://www.intel.eu/content/www/eu/en/do-it-yourself/edison.html, last visited on 25/06/2015.

[54] The Eclipse Foundation, "Wakaama." http://projects.eclipse.org/projects/technology.wakaama, last visited on 04/06/2015.

[55] O. Bergmann, S. Bernard, and H. Mehrtens, "TinyDTLS." http://sourceforge.net/projects/tinydtls/, last visited on 04/06/2015.

[56] "Contiki Operative System." http://www.contiki-os.org, last visited on 14/07/2015.

[57] T. Brady, "The JavaScript Object Notation (JSON) Data Interchange Format." http://tools.ietf.org/html/rfc7159, March 2014.

[58] "Valgrind Website." http://valgrind.org/, last visited on 01/07/2015.

[59] "Linker Kit Base Shield for Raspberry Pi." http://linksprite.com/wiki/index.php5?title=Linker_kit_Base_Shield_for_Raspberry_Pi_with_ADC_Interface, last visited on 16/06/2015.