# Master's thesis

*Two years*

Datateknik
*Computer Science*

**Enabling communication between Wireless Sensor Networks and The Internet-of-Things**
A CoAP communication stack

**Alessandro Aloisi**

Mittuniversitetet
MID SWEDEN UNIVERSITY

**Campus Härnösand** Universitetsbacken 1, SE-871 88. **Campus Sundsvall** Holmgatan 10, SE-851 70 Sundsvall.
**Campus Östersund** Kunskapens väg 8, SE-831 25 Östersund.
Phone: +46 (0)771 97 50 00, Fax: +46 (0)771 97 50 01.

# Abstract

The growing presence of sensors around us is pushing the development of pervasive applications which will enable access sensor data from remote locations in an Internet-of-Things scenario. Many smart sensing nodes that cooperate to sense the environment may constitute a Wireless Sensor Network, providing sensing services to an ever growing application space. Based on this, the thesis focuses on enabling the communication between Wireless Sensor Networks and Internet-of-Things applications. In order to achieve this goal, the first step has been to investigate the concept of the Internet-of-Things and then to understand how this scenario could be used to interconnect multiple Wireless Sensor Networks in order to develop context-aware applications which could handle sensor data coming from this type of network. The architecture of Wireless Sensor Networks was then analyzed followed by a survey about the operating systems and communication standards supported by these network. Finally, some Internet-of-Things software platforms have been studied. The second step was to design and implement a communication stack which enabled Wireless Sensor Networks to communicate with an Internet-of-Things platform. The CoAP protocol has been used as application protocol for the communication with the Wireless Sensor Networks. The solution has been developed in Java programming language and extended the sensor and actuator layer of the Sensible Things platform. The third step of this thesis has been to investigate in which real world applications the developed solution could have been used. Next a Proof of Concept application has been implemented in order to simulate a simple fire detection system, where multiple Wireless Sensor Networks collaborate to send their temperature data to a control center. The last step was to evaluate the whole system, specifically the responsiveness and the overhead introduced by the developed communication stack. The results showed that the solution introduced just a little overhead to the platform and also that the value of the response time depends on the type of request sent to the Wireless Sensor Network. However, the performances of the system could be improved further and suggested future work involves some policies to manage multiple CoAP transactions at the same time. Also the challenge of implementing some security mechanisms for a safe communication between the platform and sensor nodes, requires further work.

# Acknowledgements

Firstly I would like to express my sincere gratitude to my supervisor Stefan Forsström for his patient guidance and his capacity to answer my numerous questions. Without his help and our numerous meetings, I could not have achieved the results that I had. Secondly, I would like to thank my examiner, Professor Ting Ting Zhang for her interest and helpful comments. Finally I would also like to thank my Italian exchange coordinator, Prof. Antonio Corradi and both University of Bologna and Mid Sweden University for giving me the chance to prepare this thesis as an exchange student. It was a valuable experience for me and it helped greatly in improving my professional and social skills. Lastly, I would like to thank my family and my friends for their love and all the support given me during this period as an exchange student.

# Table of Contents

# Terminology

6LoWPAN: IPv6 over Low power Wireless Personal Area Networks

AODV: Ad-hoc On Demand Distance Vector Routing

ASCII: American Standard Code for Information Interchange

CoAP: Constrained Application Protocol

FFD: Full Function Device

IoT: Internet of Things

ISM: Industrial, Scientific and Medical Radio Bands

LAN: Local Area Network

LLN: Low Power and Lossy Network

M2M: Machine to Machine

MAC: Media Access Control

NFC: Near Field Communications

PAN: Personal Area Network

PPP: Point to Point Protocol

REST: Representational State Transfer

RFD: Reduced Function Device

RFID: Radio Frequency Identification

TCP: Transmission Control Protocol

TLV: Type Length Format

UCI: Universal Context Identifiers

UDP: User Datagram Protocol

URI: Universal Resource Identifier

WSAN: Wireless Sensor and Actuator Network.

WSN: Wireless Sensor Network

# 1 Introduction

This report is a Master's thesis in Computer Science Engineering and it has been prepared in collaboration with Mid Sweden University in Sundsvall, Sweden. I am an exchange student from the University of Bologna (Italy) and I worked on this thesis within the Erasmus Exchange Program. This thesis deals with the challenging question of how to interconnect Wireless Sensor Networks over the Internet and describes a solution that has been developed within this thesis work.

## 1.1 Background and problem motivation

Historically, humankind has seen the emergence of different kinds of global data fields. The planet itself has always generated an enormous amount of data, as human systems and physical objects did too, but until recent years we were unable to capture it. We now can because we are able to embed sensors in all sort of things and to use them to retrieve data. A scenario in which objects, animals or people are provided with sensors and the ability to automatically transfer data over the Internet is called Internet-of-Things (IoT). This kind of network can then be used by applications that utilize information from sensors attached to different things in order to display context-aware behavior. However, since not all sensors may be directly connected to a device, they could be gathered in local networks such as the Wireless Sensor Networks, which nowadays are the most used technology in this field. Wireless Sensor Networks are composed of a large number of radio equipped sensor devices that autonomously form a network, through which sensors are capable of sensing, processing and communicating with each other. These networks can operate as standalone networks or be connected to other networks, but for many applications they do not work efficiently in full isolation. Therefore, one of the biggest challenges for the IoT developers is to find resources on how to interconnect several Wireless Sensor Networks over the Internet.

## 1.2 High-level problem statement

Wireless Sensor Networks rely on the collaborative efforts of many small wireless sensor nodes and on their ability to form networks which can be used to gather sensor information. Most sensor networks are usually deployed over a wide geographical area and their applications aim at monitoring or detecting phenomena. For such applications, Wireless Sensor Networks cannot operate efficiently in complete isolation because there should be a way for a remote user to gain access to the data produced by the network. By connecting these networks to an existing network infrastructure, remote access to the sensor data can be achieved. Since the Internet has the most widespread network infrastructure in the world, it is logical to look at some efficient methods for interconnecting Wireless Sensor Networks over the Internet; in order to make an Internet-of-Things. Many Internet-of-Things software platforms have already been devel-

oped in order to enable remote access to sensor data, but just a small quantity of these platforms deal with Wireless Sensor Networks. Thus, a communication stack is required for implementation in order to enable communication between Internet-of-Things applications and Wireless Sensor Networks. Another big challenge is the high heterogeneity between Wireless Sensor Networks, since these networks often are intended to run specialized communication protocols. As a consequence of this scenario, it is usually impossible to directly connect Wireless Sensor Networks to the Internet. Therefore, there is also the need to implement a second stack, which is able to export sensor data from these particular networks to other devices connected to the Internet. Therefore, this thesis will attempt to solve the following problem:

> Enabling communication between Internet-of-Things and Wireless Sensor Networks, irregardless of their network connection and then to utilize the sensor information available in Wireless Sensor Networks for context aware applications.

## 1.3 Concrete and verifiable goals

In order to solve the problem of this project, the following goals have to be accomplished:

1. Find three different solutions of connecting Wireless Sensor Networks to an Internet-of-Things.

2. Determine the most common operating systems used in Wireless Sensor Networks.

3. Investigate which communication protocols these operating systems support.

4. Implement a communication stack which enables communication between Wireless Sensor Networks and Internet-of-Things platforms.

5. Evaluate the performance and responsiveness of the implemented solution.

6. Find possible real world applications for the implemented solution in order to put together several Wireless Sensor Networks, defining policies for system federation and coordination.

## 1.4 Scope

This project is focused on creating a communication stack between IoT applications and wireless sensors and actuator networks and then to create a Proof of Concept application in order to evaluate it. However, since there are many different operating systems and communication protocols for Wireless Sensor Networks, in this thesis I will focus on how to enable communication only with networks which use the most common ones. The management of the physical layer below these systems and security issues are out of scope for this project.

## 1.5 Outline

The second chapter will present the general idea of Internet-of-Things and context awareness including the specific devices and protocols which have been developed in order to spread its diffusion. Next, some of the most popular IoT platforms are presented. The third chapter is about the methodology we used for the project. In this section all the goals that have been presented in this thesis are listed. The fourth chapter explains the approach that has been used in the project's implementation. In chapter five the tests made and their results are reported. Finally, the sixth chapter presents the conclusions and then discusses future work needed for this project.

## 1.6 Contributions

The SensibleThings platform and its source code was contributed by and is property of Mid Sweden University. My thesis work has contributed by adding functionalities to the existing framework in order to enable communication between IoT applications and Wireless Sensor Networks. The developed communication stack is independent of the platform itself, therefore it is possible to easily export it to any kind of implementation of the latter.

# 2 Theory

A first important step is to categorize the state of art based on current research literature. The following sections present the background theory and related work for this thesis. The first section provides a short introduction about the Internet-of-Things concept and also about context awareness and ubiquitous computing. The second section gives an overview of the Wireless Sensor Network technology and then a list of the most common motes. In the third section, two of the most used Wireless Sensor Network Operating System are presented and a comparison between them has been made. The fourth section provides an overview of the communication standards used in Wireless Sensor Networks. Finally, in the fifth section, three Internet-of-Things platforms are presented.

## 2.1 Internet-of-Things

The Internet-of-Things (IoT) is a novel paradigm that is rapidly spreading across the scenario of modern wireless telecommunications. This concept is based on is the pervasive presence around us of a variety of things or objects which, through unique addressing schemes, are able to interact and cooperate with each other in order to reach common goals. As the name suggests, the purpose of this architecture is to interconnect all kinds of objects over the Internet. It is considered a normal evolution of the Internet, which at the beginning was meant just to interconnect computers but now is developing into a world wide network which will be able to interconnect all kinds of devices; as represented in figure 2.1.

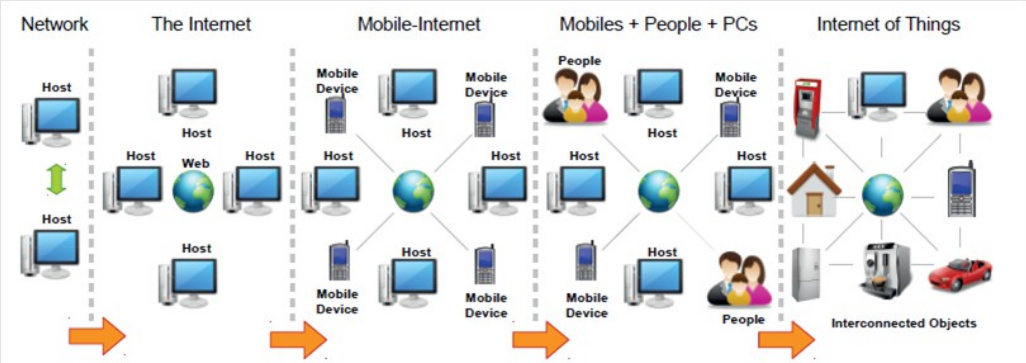



Figure 2.1: Evolution of the Internet [1]

However, IoT is a very broad vision, so the IoT research is still in progress. Therefore, there are many definitions of IoT within the research community but there are no standard definitions for IoT as of yet. The term 'Internet-of-Things' was originally introduced by Kevin Ashton [2] in a presentation in 1999. He noted that "The Internet-of-Things has the potential to change the world, just as the Internet did. Maybe even more so".

The very first vision of IoT was presented by The Auto-ID Labs [3], a world-wide network of academic research laboratories in the field of networked RFID and emerging sensing technologies. The group perceived things as very simple items: Radio-Frequency IDentification (RFID) tags having a unique identifier called Electronic Product Code. Their purpose was to realize a global system for object visibility (i.e. the traceability of an object and the awareness of its status).

However, according to the authors of [4], RFID still stands at the forefront of the technologies driving the vision just because of its maturity, low cost, and strong support from the business community. The group believes that a wide range of device, network, and service technologies will eventually build up the IoT. Near Field Communications (NFC) and Wireless Sensor and Actuator Networks (WSAN) together with RFID are recognized as ‘‘the atomic components that will link the real world with the digital world”. According to this heterogeneity, the following definitions are essential to understand the IoT:

> Definition by [5]: *“The Internet-of-Things allows people and things to be connected Anytime, Anyplace, with Anything and Anyone, ideally using Any path/network and Any service.”*


Figure 2.2: Representation of the first definition of IoT [5]

> Definition by [6]: *“The semantic origin of the expression is composed by two words and concepts: Internet and Thing, where Internet can be defined as the world-wide network of interconnected computer networks, based on a standard communication protocol, the Internet suite (TCP/IP), while Thing is an object not precisely identifiable. Therefore, semantically, Internet-of-Things means a world-wide network of interconnected objects uniquely addressable, based on standard communication protocols.”*

Many relevant institutions have stressed the concept that the road to full IoT deployment has to start from the augmentation in the Things’ intelligence. This is why a concept that emerged in parallel with IoT is the concept of Smart

Items, as a refinement of the general "Things" definition. Smart items are defined as:

> objects that can be tracked through space and time throughout their lifetime and that will be sustainable, enhanceable, and uniquely identifiable"[7]. "These are a sort of sensors not only equipped with usual wireless communication, memory, and elaboration capabilities, but also with new potentials. Autonomous and proactive behavior, context awareness, collaborative communications and elaboration are just some required capabilities [8].

The Internet-of-Things infrastructure allows combinations of different types of smart items, using different but interoperable communication protocols and realizes a dynamic heterogeneous network that can be deployed also in inaccessible, or remote spaces (oil platforms, mines, forests, tunnels, pipes, etc.) or in cases of emergencies or hazardous situations (earthquakes, fire, floods, radiation areas, etc.). Giving these objects the possibility to communicate with each other and to elaborate the information retrieved from the surroundings implies having different areas where a wide range of applications can be deployed. These can be grouped into the following domains: healthcare, personal and social, smart environment (such as at home or in the office), futuristic applications, transportation and logistics; as represented in figure 2.3.

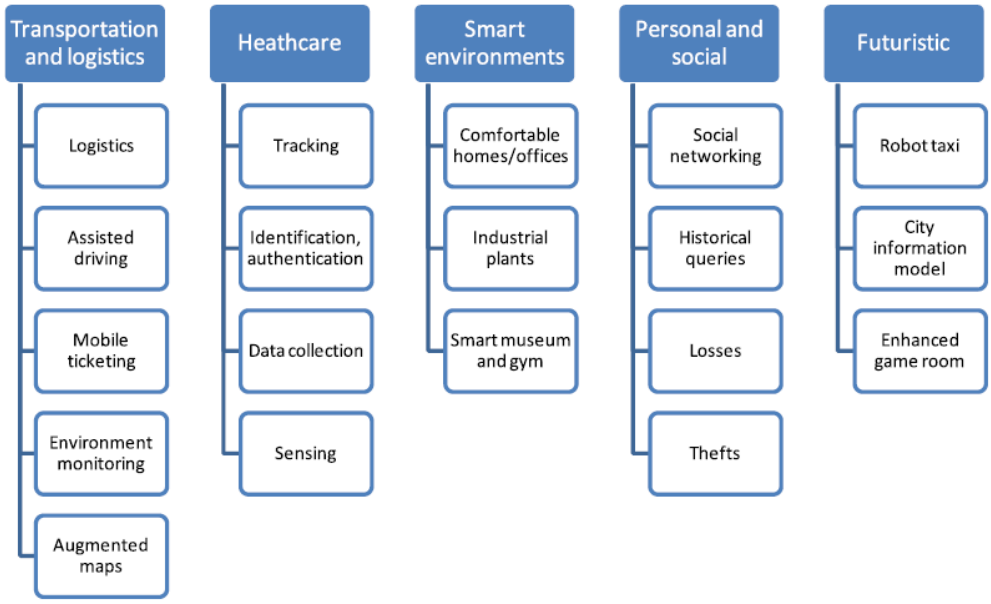



Figure 2.3: IoT application areas [8]

### 2.1.1 Context awareness

Context awareness plays an important role in the Internet-of-Things to enable services customization according to the immediate situation with minimal human intervention. Acquiring, analyzing, and interpreting relevant context information regarding the user will be a key ingredient to create a whole new range of smart applications. The concept of context is commonly understood as the

situation or surroundings of an entity. The main definition of context has been given by Dey and Abowd [9]:*"Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves."*. Therefore, context awareness is the result gained from utilizing context information, such as the ability to adapt behavior depending on the current situation of the users in context-aware applications. Dey and Abowd [9] gave this definition of context awareness:*"A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the users task."*

### 2.1.2 Ubiquitous computing

The focus on context-aware computing evolved from desktop applications, web applications, mobile computing, ubiquitous computing to the Internet-of-Things over the last decade. However, context-aware computing became more popular with the introduction of the term 'ubiquitous computing' by Mark Weiser [10], in his paper "The Computer for the 21st Century in 1991". He described a new era in which computer devices will be embedded in everyday objects, invisible at work in the environment around us; in which intelligent, intuitive interfaces will make computer devices simple to use and in which communication networks will connect these devices together to facilitate anywhere, anytime, always-on communication. Ubiquitous computing then, *"is the growing trend towards embedding microprocessors in everyday objects and refers to how they might communicate and process information, creating a world in which things can interact dynamically"*.

## 2.2 Wireless Sensor Networks overview

Wireless Sensor Networks (WSNs) became one of the most interesting and researched areas in the field of electronics in the last decade. WSNs are composed of a large number of radio equipped sensor devices that autonomously form a network, through which sensor nodes are capable of sensing, processing and communicating among each other. The sensor nodes are usually scattered in a sensor field as shown in Figure 2.4. Each of these sensor nodes has the capability to collect data and route data back to the sink and the end users. Data are routed back to the end user by a multi-hop infrastructureless architecture through the sink, which may communicate with the end user via the Internet or any type of wireless network (like WiFi, mesh networks, cellular systems, WiMAX, etc.), or without any of these networks where the sink can be directly connected to the end users [11]. There may be multiple sinks and multiple end users in the architecture shown in Figure 2.4.

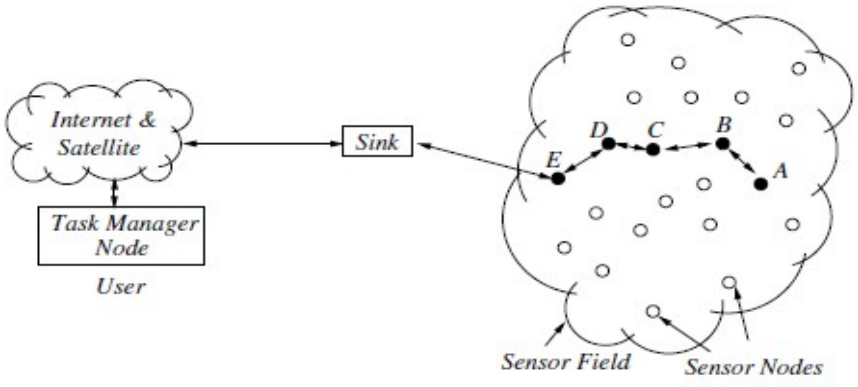



Figure 2.4: WSN architecture [11].

Typical tasks for sensor nodes are: obtaining environmental data, storing, processing and transferring obtained data, receiving data from other nodes, using and forwarding received data. However, not every node in a sensor network has to perform all of these tasks. The sensor nodes, which are intended to be physically small and inexpensive, are equipped with one or more sensors for sensing operations, a short range radio transceiver in order to enable communication with other nodes, a small micro controller for computation, and a power supply in the form of a battery; as represented in figure 2.5.

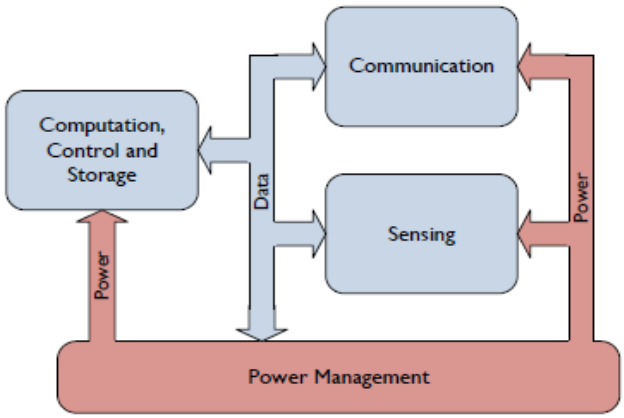



Figure 2.5: General sensor node structure

The main characteristics and challenges of WSNs are:

**Dynamic topology**: in many applications it is assumed that the topology of the network is stationary. However, in reality it is not, because WSN topology can change frequently. The topology of the WSNs can vary from a simple star network to a tree network or even to an advanced multi hop wireless mesh network.

**Limited data rate and short distance**: the sensor nodes electromagnetic range covers short distances (from one to several tens of meters). This determines the necessity of application multi-hop topology in WSN.

**Different traffic intensity**: the highest traffic density in WSN takes place around the central sensor nodes (that is the sink), because it collects all data coming from other nodes located in its vicinity. Quite the opposite, very little traffic takes place around sensor nodes which directly collect data and in the other direction, from sink to these nodes.

**Energy constraints**: the constraint most often associated with WSNs design is that sensor nodes operate with limited energy budgets. Typically, they are powered through batteries, which must be either replaced or recharged when depleted.

**Self management**: since many WSNs are required to operate in remote areas and harsh environments, without infrastructure and the possibility for maintenance or repair, sensor nodes must be able to self-configure and adapt to failures.

WSNs may consist of many different types of sensors including seismic, magnetic, thermal, visual, infrared, acoustic and radar, which are able to monitor a wide variety of ambient conditions that include: temperature, humidity, pressure, speed, direction, movement, light, soil makeup, noise levels, the presence or absence of certain kinds of objects, and mechanical stress levels on attached objects [11]. As a result, a wide range of applications are possible. However, in order to extend the applicability of these architectures and provide useful information anytime and anywhere, their integration with the Internet is very important. It is for this reason that during recent years the IoT research community has focused on WSNs as the upcoming technology for the IoT.

### 2.2.1 WSN motes

WSNs nodes are called “motes” and currently they range in size from disc shaped boards having diameters less than 1cm to enclosed systems with typical dimensions less than 5cm square. The term “mote” was coined by researchers in the Berkeley NEST to refer to these sensor nodes [13]. In figure 2.6 a list of the most common motes is reported. The values within this table show that all the motes have approximately the same size but the lightest one is SHIMMER, which is also one of the most expensive. Regarding memory and CPU power all the motes are almost identical except for the Sun SPOT which is currently the most powerful but the most costly.

| | Width x Length x Height(cm) | Weight (g) (with Battery) | Cost (per node) | Processor | Memory RAM/FLASH/EEPROM |
|---|---|---|---|---|---|
| TelosB | 3.2x6.6x0.7 | 63.05 | 139 $ | 4-8 MHz | 10 KB/48 KB/1 MB |
| Crossbow Mica2 | 3.2x5.7x0.6 | 63.82 | 99 $ | 8 MHz | 4 KB/128 KB/512 KB |
| SHIMMER | 2x4.4x1.3 | 10.36 | 276 $ | 4-8 MHz | 10 KB/48 KB/none |
| Crossbow IRIS | 3.2x5.7x0.6 | 69.40 | 115 $ | 8 MHz | 8 KB/640 KB/4 KB |
| Sun SPOT | 6.4x3.8x2.5 | 58.08 | 750 $ | 180 MHz | 512 KB/4 MB/none |

Figure 2.6: WSNs motes characteristics [13]

Each of these WSNs motes is equipped with a different set of sensors:

**TelosB**: it has a set of on-board sensors such as humidity, temperature and light intensity. In addition to the on-board sensors, the Tmote Sky provides access to 6 ADC inputs, a UART and I2C bus and several general purpose ports.

**Mica2**: it is not equipped with on-board sensors. However, Crossbow offers an extensive set of sensor boards that connect directly to the Mica mote, and are capable of measuring light, temperature, relative humidity, barometric pressure, acceleration/seismic activity, acoustics, magnetic fields and GPS position.

**Shimmer**: it has been designed for mobile health sensing applications. It incorporates a 3 axis accelerometer and allows connection of other sensors through its expansion board.

**Iris**: like the other mote from the Crossbow technology (Mica2 mote), it is not equipped with on-board sensors but it can be extended with the same sensor boards provided for the Mica2 mote.

**Sun SPOT**: it offers expansion boards with tri-axial accelerometer, temperature sensor and light sensors. Moreover, custom made sensors can be connected via five analogue inputs and five general purpose digital ports.

## 2.3 WSNs' Operating Systems

An operating system (OS) in a WSN is a thin software layer that logically resides between the node's hardware and the application and provides basic programming abstractions to application developers. Its main task is to enable applications to interact with hardware resources, to schedule and prioritize tasks, and to arbitrate between contending applications and services that try to seize resources. Other features of a WSNs OS are: memory and file management, power management, networking, providing programming environments. The choice of a particular operating system depends on several factors such as: data types, scheduling, stacks, system calls, handling interrupts, multithreading and memory allocation [12]. OS for WSNs nodes are typically less complex than

general purpose operating systems. They more strongly resemble embedded systems, for two reasons. First, Wireless Sensor Networks are typically deployed with a particular application in mind, rather than as a general platform. Second, a need for low costs and low power leads most wireless sensor nodes to have low power microcontrollers ensuring that mechanisms such as virtual memory are either unnecessary or too expensive to implement.

### 2.3.1 TinyOS

TinyOS is the most widely used runtime environment in WSNs and its compact architecture makes it suitable for supporting many applications. TinyOS has a component-based programming model, codified by the NesC language, a dialect of C and it is also based on an event driven programming model instead of multithreading. That means that when an external event occurs, such as an incoming data packet or a sensor reading, TinyOS signals the appropriate event handler to handle the event.

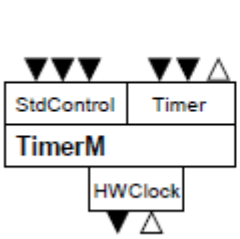
The architecture consists of a scheduler and a set of components each of which encapsulate a specific set of services, specified by interfaces. An application connects components using a wiring specification that is independent of component implementations. This wiring specification defines the complete set of components that the application uses. Components have three computational abstractions: *commands*, *events* and *tasks*. *Commands* and *events* are mechanisms for inter-component communication, while *tasks* are used to express intra- component concurrency. A *command* is typically a request to a component to perform some service, such as initiating a sensor reading, while an *event* signals the completion of that service. Rather than performing a computation immediately, commands and event handlers may post a *task*, a function executed by the TinyOS scheduler at a later time. The standard TinyOS task scheduler uses a non-preemptive FIFO scheduling policy [14].

TinyOS abstracts all hardware resources as components and it provides a large number of components to application developers, including abstractions for sensors, single-hop networking, ad hoc routing, power management, timers, and non volatile storage. A developer can then compose an application by writing components and wiring them to TinyOS components that provide implementations of the required services [14].

A component has two classes of *interfaces*: those it provides and those it uses. These interfaces define how the component directly interacts with other components. An interface generally models some service (e.g., sending a message) and is specified by an interface type. Interfaces contain both *commands* and *events* and they are bidirectional which means that the commands have to be implemented by the interface's provider whereas the events have to be implemented by the interface's user. The provided interfaces are intended to represent the functionality that the component provides to its user in its specification; the used interfaces represent the functionality the component needs to perform its job in its implementation.

NesC has two types of components: *modules* and *configurations*. *Modules* provide code for defining Tiny OS *components*. *Configurations* are used to wire other components together, connecting interfaces used by components to interfaces provided by others. They allow multiple components to be aggregated together into a single "supercomponent" that exposes a single set of interfaces.

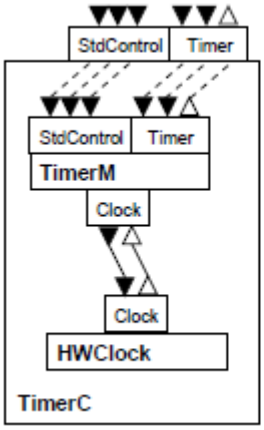
Figure 2.7 shows a simplified form of the TimerM component, a part of the TinyOS timer service, that provides the StdControl and Timer interfaces and uses a Clock interface.




```
module TimerM {
  provides {
    interface StdControl;
    interface Timer[uint8_t id];
  }
  uses interface Clock;
}
implementation {
  ... a dialect of C ...
}
```

Figure 2.7: Specification and graphical depiction of the TimerM component [14].

Figure 2.8 illustrates the TinyOS timer service, which is a configuration (TimerC) that wires the timer module (TimerM) to the hardware clock component (HWClock).




```
configuration TimerC {
  provides {
    interface StdControl;
    interface Timer[uint8_t id];
  }
}
implementation {
  components TimerM, HWClock;

  StdControl = TimerM.StdControl;
  Timer = TimerM.Timer;

  TimerM.Clk -> HWClock.Clock;
}
```

Figure 2.8: Example of TinyOS *configuration* [14].

### 2.3.2 Contiki

Contiki is a lightweight operating system with support for dynamic loading and replacement of individual programs and services. Contiki is built around an event driven kernel but provides optional preemptive multithreading that can be applied to individual processes. Contiki is implemented in the C language and has been ported to a number of microcontroller architectures.

A running Contiki system consists of the kernel, libraries, the program loader, and a set of processes. A process may be either an application program or a *service*. A *service* implements functionalities used by more than one application process. All processes, both application programs and services, can be dynamically replaced at run time.

Communication between processes always goes through the kernel. The kernel does not provide a hardware abstraction layer, but lets device drivers and applications communicate directly with the hardware. A process is defined by an event handler function and an optional poll handler function; interprocess communication is done by posting events [15].

A Contiki system is partitioned into two parts: the core and the loaded programs as shown in Figure 2.9. The core is made up of the Contiki kernel, the program loader, the most commonly used parts of the language run time and support libraries, and a communication stack with device drivers for the communication hardware. This part of the operating system cannot be modified dynamically.

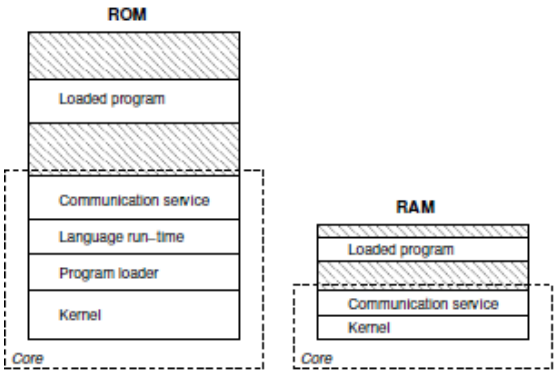



Figure 2.9: Contiki system partitioning[15].

The partitioning is made at compile time and is specific to the deployment in which Contiki is used.

The kernel is the central element of the OS. Its basic assignment is to dispatch events and to periodically call polling handlers. Subsequently, a program execution in Contiki is triggered by either events that are dispatched by the kernel or through the polling mechanism. Event handlers process an event to completion, unless they are preempted by interrupts or other mechanisms, such as thread preemption in a multithreading scenario. The kernel supports synchronous and asynchronous events. Synchronous events are dispatched to the target process as soon as possible and control is returned to the posting process once the event is processed to the end. Asynchronous events, on the other hand, are dispatched at a convenient time. In addition to these events, the kernel provides a polling mechanism, in which the status of hardware components is sampled periodically [12].

One of the interesting features of the Contiki OS is its support of dynamic loading and reconfiguration of services. This is achieved by defining *services*, *service interfaces*, *service stubs*, and a *service layer*. Services are to Contiki what modules are to TinyOS, that is a process that implements functionality that can be used by other processes. A Contiki *service* consists of a *service interface* and its implementation, which is also called a process. The *service interface* consists of a version number and the list of functions with pointers to the functions that implement the interface. A *service stub* enables an application program to dynamically communicate with a service through its *service interface*. A *service layer* is similar to a lookup service or a registry service. Active services register by providing the description of their *service interface* and ID and version number. This way, the *service layer* keeps track of all active services. Figure 2.10 illustrates how application programs interact with Contiki services [12].

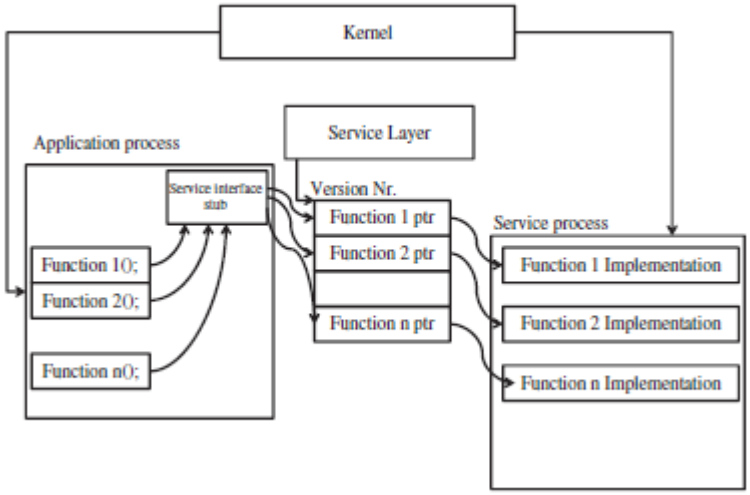



Figure 2.10: Contiki service interaction architecture [12].

When a service is called, the service interface stub queries the service layer and obtains a pointer to the service interface. Upon obtaining a service whose interface description as well as version number matches with the service stub, the interface stub calls the implementation of the requested function.

### 2.3.3 Tiny Os and Contiki evaluation

Ranking the strength of an operating system, like all ranking assignments, is a difficult assignment. However, in WSNs, there are several contexts pertaining to development, deployment, runtime performance, and code evolution. In view of these aspects, TinyOS is compact in size and efficient in its use of resources, since the cost of managing separate entities (operation system and application) is related to a single assignment of managing a single file. But replacement or reprogramming cost is high.

Contiki provides a flexible support for dynamic reprogramming and hence is well suited to applications which require intensive updating and upgrading processes; but this does not come without any costs.

Figure 2.11 and 2.12 provide summaries of the functional and nonfunctional aspects of both the OSs.

| OS | Programming paradigm | Building blocks | Scheduling | Memory allocation | System calls |
|---|---|---|---|---|---|
| TinyOS | Event-based (split-phase operation, active messages) | Components, interfaces, and tasks | FIFO | Static | Not available |
| Contiki | Predominantly event-based, but it provides optional multithreading support | Services, service interface stubs, and service layer | FIFO, poll handlers with priority scheduling | Dynamic | Runtime libraries |

Figure 2.11: Comparision of functional aspects of the OS [12].

| OS | Minimum system overhead | Separation of concern | Dynamic reprogramming | Portability |
|---|---|---|---|---|
| TinyOS | 332 bytes | There is no clean distinction between the OS and the application. At compilation time a particular configuration produces a monolithic, executable code. | Requires external software support | High |
| Contiki | ca. 810 bytes | Modules are compiled to produce a reprogrammable and executable code, but there is no separation of concern between the application and the OS. | Supported | Medium |

Figure 2.12: Comparision of non-functional aspects of the OS [12].

## 2.4 WSNs communication standards

In order to achieve interoperability between manufacturer components, a number of standards have been established in the WSN field. These standards can be mapped to the ISO-OSI layers. However, some standards cover only the bottom layers, others cover the full stack. No single standard has been established as the market winner. The most common standards used in WSN are: WiFi, Bluetooth, IEEE 802.15.4, ZigBee, 6LoWPAN. However, WiFi and Bluetooth are losing ground within the WSNs research community since they were not developed for low power devices such as WSNs nodes. On the other hand, IEEE 802.15.4 was created just for these kinds of devices and is thus becoming the most important communication standard for WSNs. Moreover, the ZigBee and 6LoWPAN standards have been developed in order to extend the features of IEEE 802.15.4.

### 2.4.1 IEEE 802.15.4

The key requirements for Low Rata Personal Area Networks (as the WSNs) are low complexity, very low power consumption and low cost. The IEEE 802.15.4 standard considers these requirements and provides a framework for the lowest two layers of the OSI mode. The standard defines two types of devices: a Full Function Device (FFD) and a Reduced Function Device (RFD). The FFD is capable of all network functionalities and can operate in three different modes: it can operate as a PAN coordinator, a coordinator or it can serve simply as a device. An RFD device is low on resources and memory capacity and is capable only of very simple applications such as sensing light or temperature [16]. There are two different topologies in which the PAN can operate: star or peer to peer, as represented in figure 2.13. In the star topology communication can only take place between the devices and the PAN coordinator, which has to be a FFD. The PAN coordinator is responsible for inaugurating or terminating communications in the network and is often mains powered. In the peer to peer topology all FFD devices in the network can communicate with each other while the RFD devices can only communicate with the PAN coordinator [16].

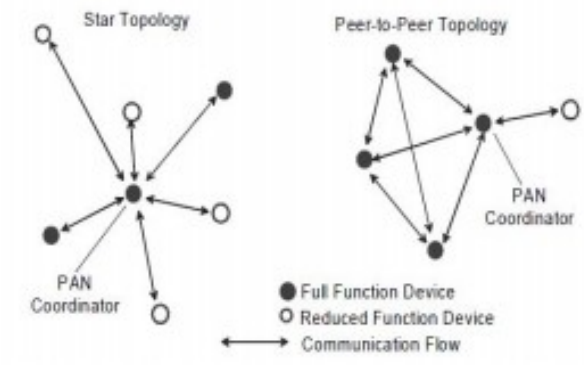



Figure 2.13 IEEE 802.15.4 network topology [16].

The physical layer is responsible for the transmission and reception of data. It defines the radio bands to be used and type of spreading and modulation techniques. The standard provide three different operational frequencies: 16 channels in the 2.4 GHz band, 10 channels in the 915 MHz band and 1 channel in the 868 MHz band. The MAC layer which appears just above the physical layer in the OSI model, is responsible for managing beacon transmission, access to channel and association/disassociation to the network.

The IEEE 802.15.4 standard defines four basic frame types which are **beacon**, used by a coordinator to transmit beacons, a **data frame**, used for all transfers of data, an **acknowledgment frame**, used for confirming successful frame reception and a **MAC command frame**, used for handling all MAC peer entity control transfers.

### 2.4.2 ZigBee

ZigBee is a specification for a suite of high level communication protocols used to create personal area networks; built for small, low power digital radios based on the IEEE 802.15.4 standard. ZigBee is used in applications that require a low data rate, long battery life, and secure networking. This standard has a defined rate of 250 kbit/s, best suited for periodic or intermittent data or a single signal transmission from a sensor or input device. The transmission distances range from 10 to 100 meters line of sight, depending on power output and environmental characteristics. The technology defined by the ZigBee specification is intended to be simpler and less expensive than other WPANs, such as Bluetooth or Wi-Fi.
The ZigBee standard defines a stack shown in figure 2.14 which has a layered structure with four distinct layers, the physical layer, the MAC layer, the network layer and the application layer. The two bottom layers are defined by the IEEE 802.15.4 standard. The network layer is the bottom layer defined by the ZigBee standard which provides network configuration, manipulation, and message routing. The routing protocol used by the network layer is the Ad hoc On-Demand Distance Vector Routing Protocol (AODV). In order to find the destination device, it broadcasts out a route request to all of its neighbors. The neighbors then broadcast the request to their neighbors, until the destination is reached. Once the destination is reached, it sends its route reply via unicast transmission following the lowest cost path back to the source. Once the source receives the reply, it will update its routing table for the destination address with the next hop in the path and the path cost. An application layer then provides the intended function of the device [17].

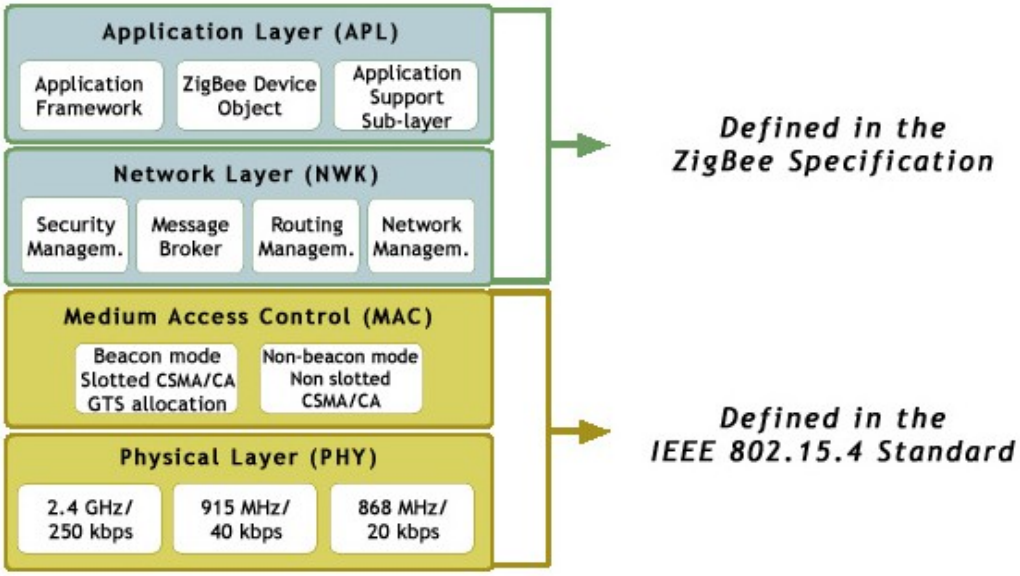



Figure 2.14: ZigBee stack architecture.

ZigBee operates in the industrial, scientific and medical (ISM) radio bands: 868 MHz in Europe, 915 MHz in the USA and Australia and 2.4 GHz in most jurisdictions worldwide. Data transmission rates vary from 20 kilobits/second in the 868 MHz frequency band to 250 kilobits/second in the 2.4 GHz frequency

band. The ZigBee network layer natively supports both star and tree typical networks, and generic mesh networks; as reported in figure 2.15. Every network must have one coordinator device, tasked with its creation, the control of its parameters and basic maintenance. Within star networks, the coordinator must be the central node. Both trees and meshes allow the use of ZigBee routers to extend communication at the network level [18].

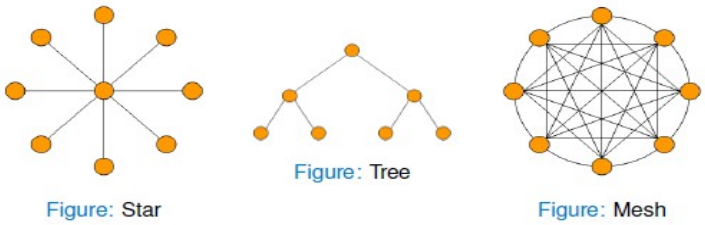



Figure 2.15 ZigBee network topologies

### 2.4.3 6LoWPAN

6LoWPAN is an acronym of IPv6 over Low power Wireless Personal Area Networks (WPAN). 6LoWPAN is the name of a working group in the Internet area of the Internet Engineering Task Force (IETF). The 6LoWPAN concept originated from the idea that “the Internet Protocol (IP) could and should be applied even to the smallest devices" and that low power devices with limited processing capabilities should be able to participate in the Internet-of-Things [19]. 6LoWPAN enables the use of IPv6 in Low Power and Lossy Networks (LLNs), such as those based on the IEEE 802.15.4 standard. Given the limited packet size and other constraints of this kind of devices, they cannot use the standard IPv6 directly. Therefore, an adaptation layer to perform header compression, fragmentation and address auto configuration is needed to use IPv6. The 6LoWPAN group thereby has encapsulation and header compression mechanisms that allow IPv6 packets to be sent to and received from over IEEE 802.15.4 based networks.

The 6LoWPAN architecture is made up of low-power wireless area networks (LoWPANs), which are connected to other IP networks through edge routers, as is shown in figure 2.16. The edge router plays an important role as it routes traffic in and out of the LoWPAN, while handling 6LoWPAN compression and NeighborDiscovery for the LoWPAN [19].

Each LoWPAN node is identified by a unique IPv6 address, and is capable of sending and receiving IPv6 packets. Typically LoWPAN nodes support ICMPv6 traffic and use the User Datagram Protocol (UDP) as a transport protocol. The whole 6LoWPAN protocol stack is shown in figure 2.17.

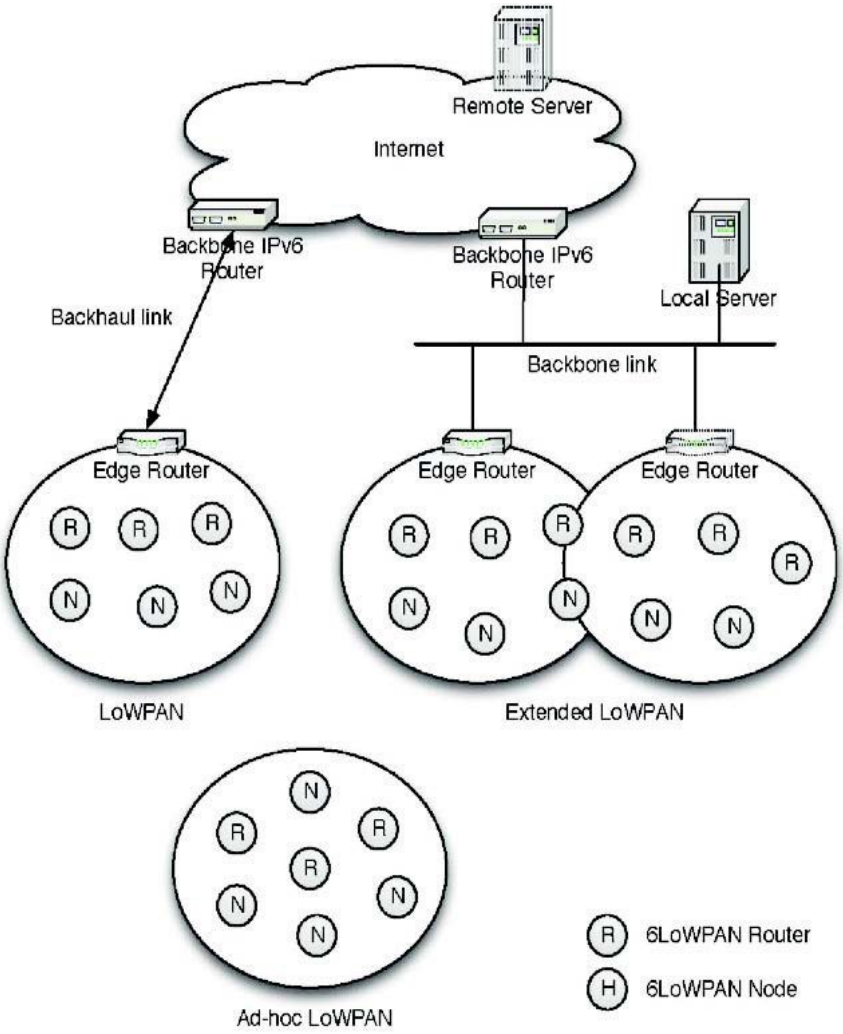



Figure 2.16 6LoWPAN architecture

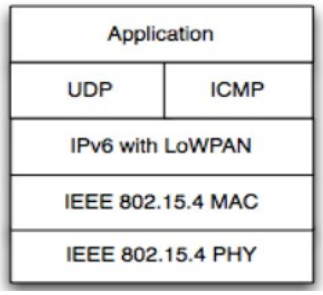



Figure 2.17 6LoWPAN protocol stack.

### 2.4.4 REST and CoAP

One of the major benefits of IP based networking in LLNs is to enable the use of standard web service architectures without using application gateways. As a consequence, smart objects will not only be integrated with the Internet but also with the web. This integration allows smart object applications to be built on

top of Representational State Transfer (REST) architectures and it is defined as the Web of Things (WoT) [20].

In a REST architecture a resource is an abstraction controlled by the server and identified by a Universal Resource Identifier (URI). The resources are accessed and manipulated by an application protocol based on client/server request/responses. REST is not tied to a particular application protocol, however, the vast majority of REST architectures currently use Hypertext Transfer Protocol (HTTP). HTTP manipulates resources by means of its methods GET, POST, PUT, DELETE [20].

REST architectures allow IoT applications to be developed on top of web services. However, the standard HTTP protocol cannot be used in LLNs since this protocol is relatively expensive for them, both in implementation code space and network resource usage. Therefore, the Constrained RESTful environments (CoRE) working group has defined a REST-based web transfer protocol called Constrained Application Protocol (CoAP). CoAP includes the HTTP functionalities which have been redesigned considering the low processing power and energy consumption constraints of small embedded devices [20]. CoAP is based on a REST architecture in which resources are server controlled abstractions made available by an application process and identified by Universal Resource Identifiers (URIs) and they can be manipulated by means of the same methods as the ones used by HTTP.

The first significant difference between HTTP and CoAP is the transport layer. HTTP relies on the Transmission Control Protocol (TCP). TCP' flow control mechanism is not appropriate for LLNs and its overhead is considered too high. Therefore CoAP has been built on top of the User Datagram Protocol (UDP), which has significantly lower overhead. As represented in figure 2.18, CoAP is organized in two layers. The transaction layer handles the single message exchange between end points, which can be of four types: Confirmable (it requires an acknowledgment), Non-confirmable (it does not need to be acknowledged), Acknowledgment (it acknowledges a Confirmable message) and Reset (it indicates that a Confirmable message has been received but context is missing to be processed). It also provides support for multicast and congestion control.

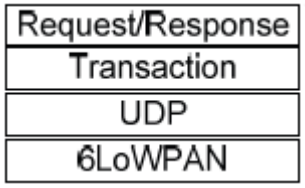



Figure 2.18 CoAP protocol stack [20]

The Request/Response layer is responsible for the transmission of requests and responses for the resource manipulation and transmission. A REST request is

piggybacked on a Confirmable or Non-confirmable message, while a REST response is piggybacked on the related Acknowledgment message. Figure 2.19 shows an example of a typical REST request-response transaction.

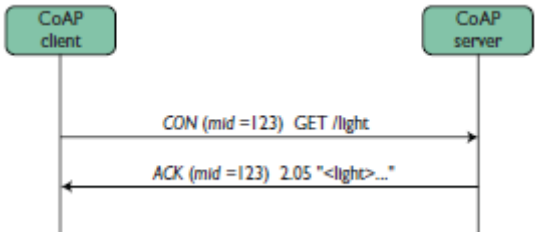



Figure 2.19 CoAP request-response example, using a confirmable message.

The dual layer approach allows CoAP to provide reliability mechanisms even without the use of TCP as transport protocol. In fact, a Confirmable message is retransmitted using a default timeout and exponential back off between retransmissions, until the recipient sends the Acknowledgement message. In addition, it enables asynchronous communication, because when a CoAP server receives a request which is not able to handle immediately, it first acknowledges the reception of the message and sends back the response in an off-line fashion [20].

One of the major design goals of CoAP has been to keep the message overhead as small as possible and limit the use of fragmentation. CoAP uses a short fixed length compact binary header of 4 bytes followed by compact binary options. A typical request has a total header of about 10-20 bytes.

Since a resource on a CoAP server likely changes over time, the protocol allows a client to constantly observe the resources. In a GET request, a client can indicate its interest in further updates from a resource by specifying the “Observe” option. If the server accepts this option, whenever the state of the resource changes it notifies each client having an observation relationship with the resource. The duration of the observation relationship is negotiated during the registration procedure.

Although CoAP is a work in progress, various open source implementations are already available. The two most known operating systems for WSNs, Contiki and Tiny OS, have already released CoAP implementation libraries, named Erbium and CoapBlib respectively.

## 2.5 Related work

Applications that utilize information from sensors attached to different things in order to provide more personalized, automatized, or even intelligent behavior are commonly referred to as Internet-of-Things applications.[8] The prediction is that these kinds of applications will be able to interact with an IoT, a worldwide network of interconnected everyday objects, and thereby be able to display context-aware behavior. [21] There is also an interesting relationship be-

tween the IoT and big data, since all of the connected things will produce and consume large amounts of data. In order to enable a widespread proliferation of IoT services there must be a common platform for dissemination of sensor and actuator information on a global scale. However, there is a large number of practical difficulties that must be solved to achieve this goal. The main requirements that an IoT platform should satisfy are the following:

**Scalable**: logarithmic or better scaling of communication load in end points;

**No central point of failure**: fully distributed platform;

**Bidirectional**: enabling communication between sensors/actuators and the IoT applications in both ways;

**Fast**: capable of signaling in real time between end points;

**Lightweight**: able to run on devices with limited resources;

**Seamless**: capable of handling heterogeneous infrastructures and different end user devices;

**Stable**: all queries into the platform should return an answer;

**Extensible**: capable of adding new features and modules without complete redistribution.

### 2.5.1 SensibleThings

The SensibleThings platform is an open source architecture for enabling IoT based applications, developed by Mid Sweden University. An overview of the platform and its components is presented in figure 2.20. It shows how the platform is distributed over a number of entities connected to the Internet. The figure shows how an application which is running a client of the SensibleThings platform (SensibleThinghs instance) communicates with other entities running the platform. A client can acquire sensor and actuator information of the other participants. Furthermore, the platform can act as both a producer and consumer of sensor and actuator information at the same time, enabling bidirectional exchange of context information [22].

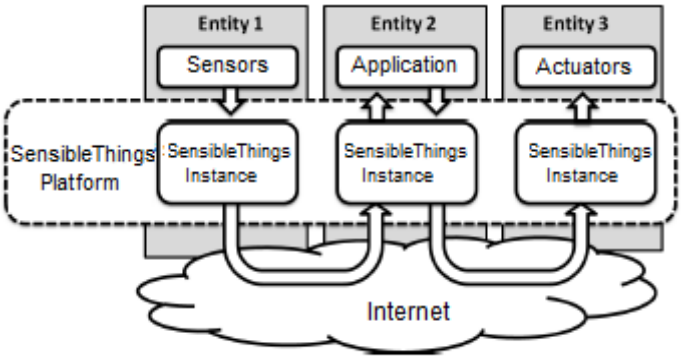



Figure 2.20: Overview on the function of the SensibleThings platform.

The SensibleThing platform is a realization and implementation of the MediaSense architecture explained in [22]. The code is based on a fork of the MediaSense platform, but with significant improvement. The focus has been on the open source aspect and maintaining the commercialization possibilities of applications that are utilizing the platform. The platform is organized in several levels, as represented in figure 2.21.

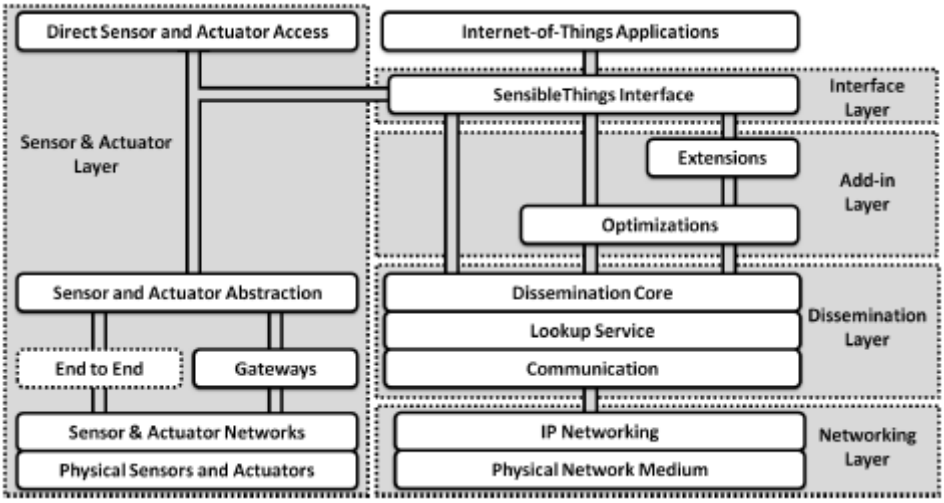



Figure 2.21: SensibleThings platform architecture.

**Interface Layer**: the public interface through which applications interact with the SensibleThings platform, using its API's.

**Add in Layer**: enables developers to add optional functionality and optimization algorithms to the platform, which can be loaded and unloaded in runtime when needed.

**Dissemination Layer**: it enables dissemination of information between all entities that participate in the system and are connected to the platform. Therefore, it enables registration of sensors in the platform, resolving the location of a sensor in order to find it, and the communication to retrieve the actual sensor values.

**Networking Layer**: it manages connection of different entities over current Internet Protocol (IP) based infrastructure.

**Sensor and Actuator Layer**: it enables different sensors and actuators to connect into the platform into two different ways. If they are accessible from the application code, they can be connected directly. Otherwise, the sensors and actuators can connect through the sensor and actuator abstraction, which enables connectivity either directly to Wireless Sensor Networks or via more powerful gateways.

### 2.5.2 ETSI M2M

The ETSI Machine to Machine (M2M) technical committee was created in January 2009 at the request of many telecom operators to create a standard system-level architecture for mass scale M2M applications. The ETSI M2M architecture is resource centric and adopts the RESTful style. It aims at integrating all of the existing standard or proprietary automation protocols into a common architecture. The ETSI M2M system architecture, represented in Figure 2.22, separates the M2M device domain and the network and applications domain.

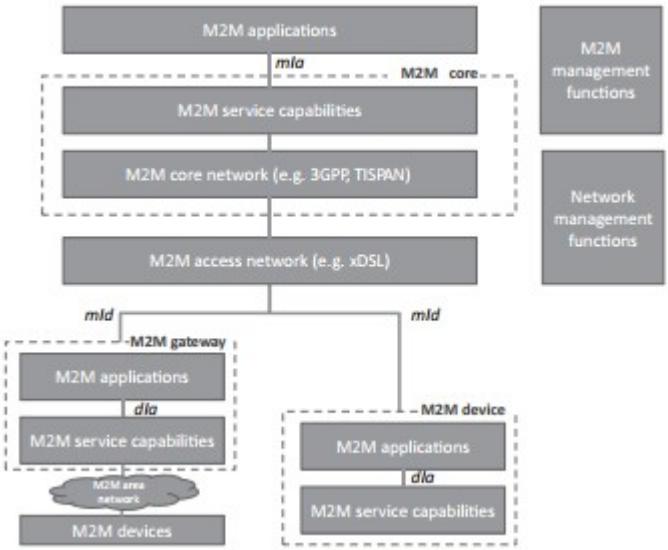



Figure 2.22 ETSI M2M architecture [23].

**M2M Device:** this kind of device can connect to the M2M network domain directly or via M2M gateways acting as a network proxy. A M2M Device is a device capable of replying to request for data contained within those devices or capable of transmitting data autonomously.

**M2M Gateway:** a gateway module runs a M2M application which offers M2M capabilities and act as a bridge between M2M devices and the M2M Access Network. Devices without M2M capabilities built-in can go through M2M gateway to interconnect and interwork with the M2M access network. M2M gateways can be cascaded or operate in parallel mode.

**M2M Area Network:** a wired or wireless access network provides connectivity and transport of M2M data/messages between M2M devices, M2M gateways and M2M servers. Some M2M area network technologies include: PWLAN, ZWave, Zigbee, Bluetooth.

**M2M Access Network:** it manages the communication between the M2M Gateways and M2M Applications. This layer is also responsible for defining the transport protocol used for network communication, such as IP transport networks.

**Core network layer:** it provides service and network control functions, network to network interconnect and roaming support. This is the central part of the M2M communication network that provides various services to service providers connected via the access network such as WiMAX, DSL, WLAN.

**M2M service capabilities layer:** this is an abstraction layer of the M2M software where common functionalities are implemented to serve the M2M application. It provides a set of APIs to expose the M2M service capabilities closest to the application using them.

**M2M Application:** this is a software running in the middleware layer designed to perform specific business processes over the M2M Core network [23].

### 2.5.3 SENSEWEB

SenseWeb is a IoT platform developed by Microsoft, through which IoT applications can initiate and access sensor data streams from shared sensors across the entire Internet. The SenseWeb infrastructure helps ensure optimal sensor selection for each application and efficient sharing of sensor streams among multiple applications. The SenseWeb layer architecture is shown in figure 2.23.

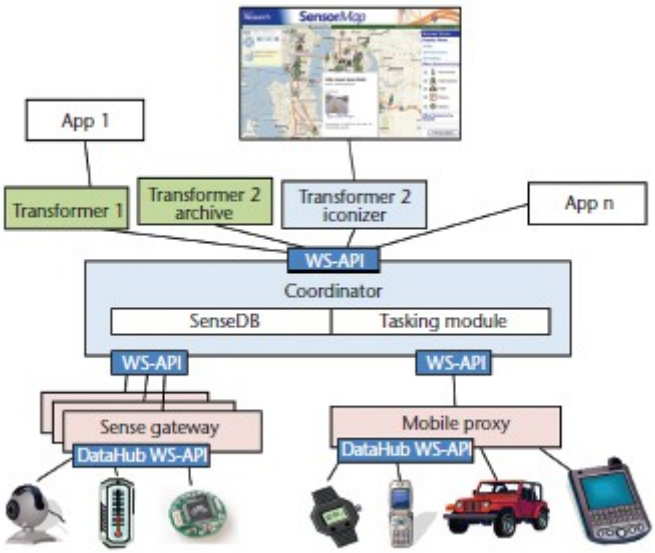



Figure 2.23: SenseWeb architecture [24].

**Coordinator layer**: is the central point of access into the system for all applications and sensor contributors. The functions of the coordinator are internally divided between two components: the *tasking module* and *senseDB*. The tasking module accepts the application's sensing queries and tries to satisfy these from available sensing resources considering their capabilities. The *senseDB* manages the overlap among multiple application needs. Specifically, when multiple applications need data from overlapping space time windows, senseDB attempts to minimize the load on the sensors or the respective sensor gateways by combining the requests for common data and using a cache for recently accessed data. SenseDB is also responsible for indexing the sensor characteristics

and other shared resources in the system to enable applications to discover what is available for their use.

**Sensor gateways***:* its main task is to hide the complexity regarding the heterogeneity of communications interfaces used by sensor nodes. The gateway might also implement sharing policies defined by the contributor of the sensors which are using it. For instance, the gateway might maintain all raw data in its local database, possibly for local applications the sensor owner runs, but only make certain nonprivate sensitive parts of the data or data at lower sampling rates available to the rest of SenseWeb.

**Mobile proxy**: is a special gateway built for mobile sensors, which makes the mobility of sensing devices transparent to the applications providing location-based access to sensor readings. Applications simply express their sensing needs and the mobile proxy returns data from any devices that can satisfy those needs.

**Data transformer**: a transformer converts data semantics through processing. Data transformers can also convert units, fuse data, and provide data visualization services. Transformers are indexed at the coordinator and applications might discover and use them as needed [24].

# 3 Methodology

In order to reach the goals described in chapter 1.3, this project will be divided into three different phases: a study phase, an implementation phase and an evaluation phase. During the first phase a survey about different possibilities of connecting WSN to an IoT will be made; then the most common operating systems and communication protocols used in WSN will be analyzed. After these surveys, a solution for the problem statement explained in chapter 1.2 will be designed and then implemented. In the last phase the performance of the developed solution will be evaluated and finally a Proof of Concept application will be created. During the whole work process, I will have weekly meetings with the Professor in order to show my own progress through PowerPoint presentations. To achieve all the goals the following methods are to be used:

To achieve goal 1 on finding three different solutions of connecting WSNs to an IoT scenario, documents will be collected regarding existing software platforms which enable the communication between WSN and IoT applications. This will will be done by searching articles and papers on research databases.

To achieve goal 2 on understanding the most common OS used in WSN, the most common operating systems for Wireless Sensor Networks will be assessed, by searching the Internet and find out what other people have used.

To achieve goal 3 on investigating which communication protocols these OS support, documentation about these OS will be scrutinized and some simulations will be executed using the supported communication protocol, in order to learn how to use it.

To achieve goal 4 on implementing a communication stack which enables communication between Wireless Sensor Networks and IoT applications, the documentation of the platform will be analyzed and some simulations will be run in order to discover its features. This platform will be extended by implementing a communication stack which connects Wireless Sensor Networks with IoT applications.

To achieve goal 5 on evaluating the performances and responsiveness of my implemented solution, tests will be executed to measure the response time, the scalability and the overhead introduced by this communication stack.

To achieve goal 6 on finding possible real-world applications for the implemented solution, various scenarios will be investigated in order to understand which would be the best application for the communication stack that has been developed. Finally, a Proof of Concept application will be developed in order to simulate the chosen application, implementing some policies to enable the collaboration between multiple Wireless Sensor Networks.

After having achieved all the goals, the entire thesis work process will be evaluated by investigating other possible approaches. A survey will be then performed in order to understand if I would have had different results using different systems, such as a different OS and communication protocol for the WSN. Finally, possible future work related to my thesis will be proposed.

# 4 Implementation

In this chapter the implementation of the CoAP communication stack is described. As presented in figure 4.1 the CoAP stack extends the SensibleThings platform and it is formed by two main classes: CoapSensorActuator and CoapSensorGateway. The first one allows the communication between the platform and a wireless sensor network which supports the CoAP protocol. The second one realizes a gateway between the CoapSensorActuator class and sensors which do not support the CoAP protocol. In this chapter the architecture of the wireless sensor network which has been utilized in this thesis is explained. Next, the structure of CoAP packets and the extended layers of the SensibleThings platform are described.

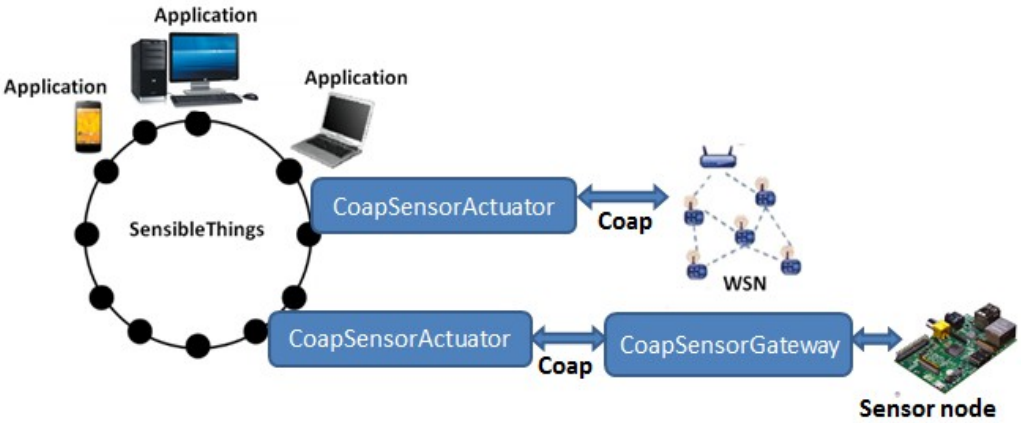



Figure 4.1: CoAP communication stack architecture

The architecture of the Wireless Sensor Network used in this work consists of one mote connected to a computer via a USB cable, which acts as a sink, and one or more motes that communicate with the sink through the IEEE 802.15.4 medium, which are the actual sensor nodes. The motes that have been used in this thesis are TelosB motes running Tiny OS as operating system. Figure 4.2 shows an example of a Wireless Sensor Network.


Figure 4.2: TelosB motes wireless sensor network

In order to use the CoAP protocol on the motes, the CoAPBlib library has been installed on the sensor nodes. Moreover, to enable the communication between the motes and the Linux machine the PPPRouter application needs to be installed on the sink mote. This application is IPv6 based and basically receives/forwards packets on a specified IEEE 802.15.4 channel and forwards/receive the packets to the computer using the Point to Point Protocol.

In appendix A some guidelines on how to install the CoapBlip library and the PPPRouter application are reported.

Each TelosB mote is equipped with multiple sensors which are identified by specific URI's, as represented in the following table:

| Sensor | URI |
|---|---|
| Led | \l |
| Temperature | \st |
| Humidity | \sh |
| Voltage | \sv |
| Temperature + Humidity + Voltage | \r |

In order to test the system, within the CoapBlip library an example client application is provided (at /support/sdk/c/coap/examples). With this application, it is possible to send CoAP requests to the motes from the Linux Terminal. For example, the request for getting the leds' status would be:"./coap-client coap://[fec0::3]/l" [25]. In figure 4.3 an output for this request is represented.

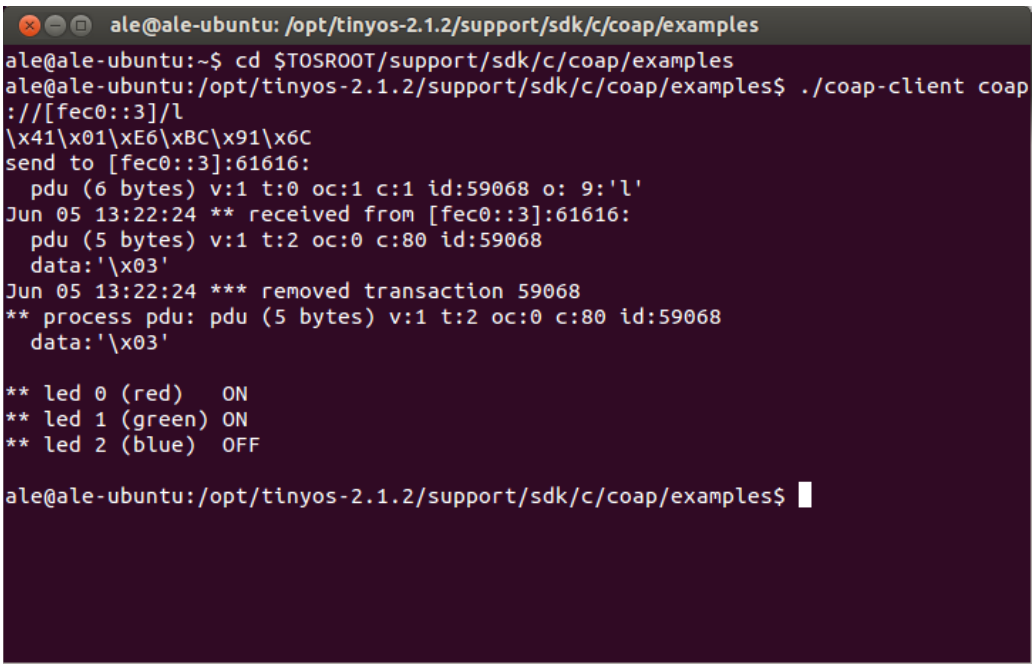


Figure 4.3: CoAP GET request example

## 4.1 SensibleThings platform

The CoAP communication stack extends the Sensor and Actuator layer of the SensibleThings platform, which has already been described in paragraph 2.5.1. This platform enables multiple nodes to communicate and to exchange data over the Internet. This feature then has been used to connect multiple remote Wireless Sensor Networks together and to build applications for managing the retrieved data from various nodes.

A component called SensorActuatorManager has been used in order to bind the CoAP stack with the SensibleThings platform. This component is included in the Sensor and Actuator layer and his main task is to manage the requests between the platform and this layer. It implements six methods:

- *connectSensorActuator()*: it is for connecting any sensor/actuator in the network. Basically, after this call, the sensor/actuator will be registered and available inside the platform.
- *disconnectSensorActuator()* and *disconnectAllSensorActuators():* are called for disconnecting a specific sensor and all the sensors from the platform, respectively.
- *HandleGetEvent()* and *HandleSetEvent()*: this method is called from the platform, to forward a getEvent/setEvent to the sensors.

In figure 4.4 the sequence of methods called within a GET request between two remote nodes is shown.

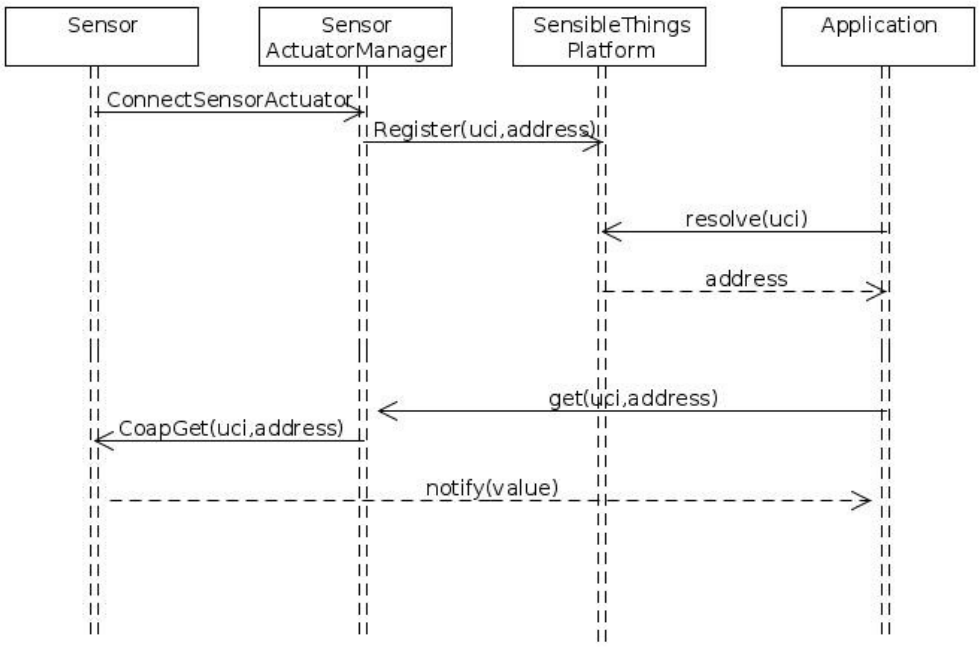



Figure 4.4: GET request

## 4.2 CoAP packet structure

A CoAP packet is formed by a 4 bytes binary header followed by an option field and a payload. The length of the message payload is implied by the datagram packet length. The structure of a CoAP packet is shown in figure 4.5.

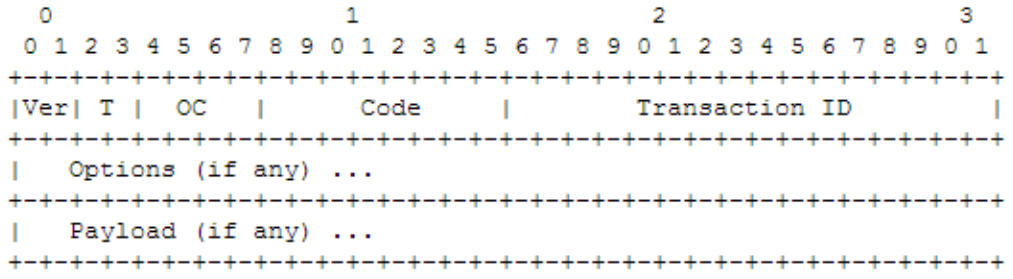



Figure 4.5: CoAP packet format

The fields within the packet header are:

**Ver**: Version, 2 bit unsigned integer. This value indicates the version of CoAP protocol. To set this field correctly for the CoAPBlib library, 1 has to be set as its value. Other values are reserved for future versions.

**T**: Transaction type field, 2 bit unsigned integer. This field indicates if this message is Confirmable (0), Non-confirmable (1), Acknowledgment (2) or Reset (3).

**OC**: Option count field, 4 bit unsigned integer. This field indicates how many option headers follow the base headers. If set to 0 the payload (if any) immediately follows the base header.

**Code**: 8 bit unsigned integer. It indicates the Method or the Response Code of a message. The method codes are reported in the following table:

| Method | Code |
|---|---|
| GET | 1 |
| POST | 2 |
| PUT | 3 |
| DELETE | 4 |

The CoAPBlip library only allows get and put methods, however. The values 40-225 are used for Response Codes. The CoAP stack developed in this thesis only uses the values 80 (HTTP code: 200 OK) and 160 (HTTP code: 400 Bad request).

**Transaction ID**: 16 bit unsigned integer. This value identifies each CoAP transaction since this is a unique ID assigned by the source. The response message for each request must contain the same transaction ID as the request message. This value must also be changed for each new request except when retransmitting a request.

CoAP messages may also include one or more header options in Type Length Format (TLV) and they have to appear in order of option type. The option types used in the CoAP stack were: URI path (for specifying the sensor URI within a sensor node, Type number: 9),Token (for sending the data payload in a PUT request, Type number: 11) and Content Type (which indicates the Internet media type of the token, Type number: 1). A delta encoding is used between each option header, with the Type identifier for each Option calculated as the sum of its Option Delta field and the Type identifier of the preceding Option in the message, if any, or zero otherwise. Each option header also includes a Length field, as represented in figure 4.6.

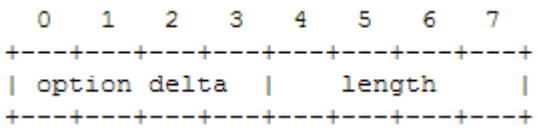


Figure 4.6: Option field format.

**Option delta**: 4 bit unsigned integer. This field defines the difference between the option Type of this option and the previous one (or zero for the first option). In other words, the Type identifier is calculated by simply summing the Option delta fields of this one and previous options.

**Length**: 4 bit unsigned integer. This field specifies the length of the option payload.

Figure 4.7 shows a basic request sequence. A client makes a Confirmable GET request for the resource/temperature to the server with a Transaction ID of 1234. The request includes one URI-Path Option (delta 0 + 9 = 9) "temperature" of Len = 11. The corresponding Acknowledgment is of Code 200 OK and includes a Payload of "22.3 C". The Transaction ID is 1234, thus the transaction is successfully completed. The response Content type of 0 (text/plain) is assumed as there is no Content type Option [26].

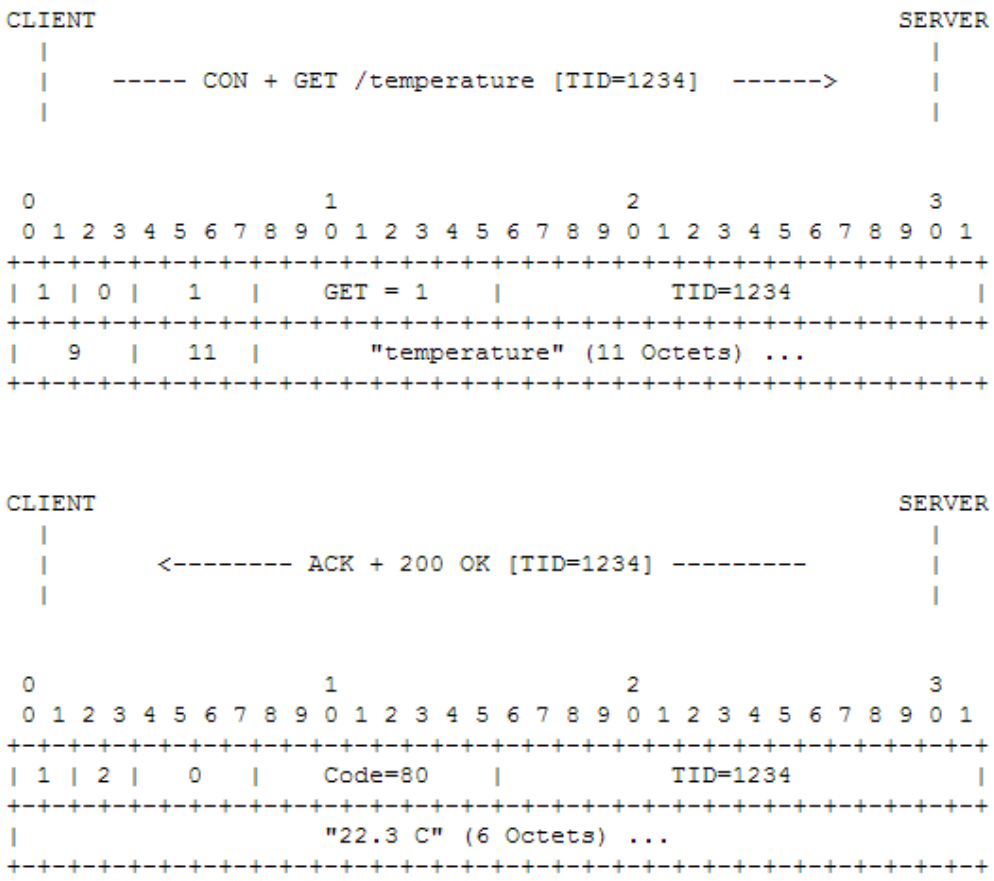



Figure 4.7: CoAP get transaction example.

## 4.3 CoapSensorActuator

CoapSensorActuator is responsible for the communication between the platform and the sink of a wireless sensor network, through the CoAP protocol. Its main task is to create CoAP packets, send them to a mote and parse the response message.

It extends the SensorActuator abstract class and implements its two methods *getValue()* and *setValue(),* as shown in figure 4.8. The constructor gets the IP address of the mote and the sensor UCI. At the end of the IP address the URI of the sensor also needs to be specified by the user.

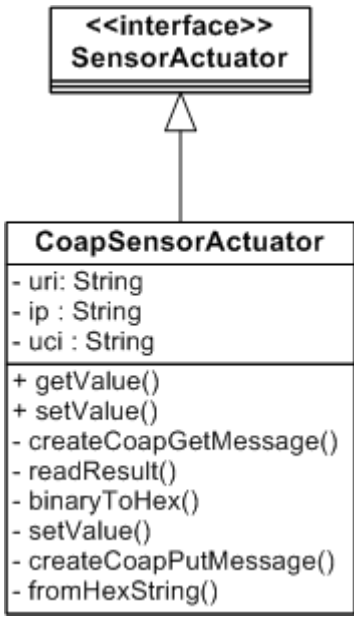



Figure 4.8: CoapSensorActuator UML scheme.

**getValue()**: this method is called by the SensibleThings platform every time a CoAP GET request has to be sent to a mote. It is a synchronized method because only one thread at a time can send a GET request to a mote. According to the CoAP protocol standard, getValue creates a CoAP packet using the *createCoapGetMessage()* method. Then it uses a DatagramSocket to send the packet to the mote at the specific IP address set by the user. However, the number of the port cannot be chosen by the user, since CoAPBlip on TelosB motes always uses the default port 61616 to receive the requests. If the request has been sent correctly, a response CoAP packet is received on the same socket. To parse the received packet, in order to extract the value of the sensor reading, the method *readResult()* is called.

Since a response message can never be received from the mote, a timer of 4 seconds is set during the creation of the DatagramSocket. If after that period of time the response has not been received, the lock on the SensorActuator object is released and a new GET request can be sent.

**createCoapGetMessage()**: this method builds a CoAP GET message, according to what has been explained in paragraph 4.3.

An example packet built by this method for a GET led request is shown in figure 4.9.

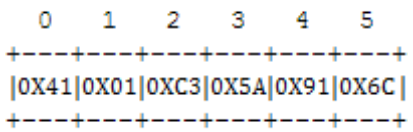



Figure 4.9: CoAP GET packet.

The first two bytes are constant for each packet created by the createCoapGetMessage method. Each byte has the following meaning:

0x41 (0100 0001): within this byte the first three fields of a CoAP packet are set. The two initial bits represent the protocol version number, which must be set to 1. Then the 2 bits Transaction Type is set to 0, which means that the current packet is a Confirmable message. The last 4 bits represent the number of the options that follows the packet header, which is set to 1. Since this request has to be sent to a specific sensor within the addressed mote, its URI has to be specified in the packet. Then the only option used for this request is the sensor's URI.

0x01: this byte represents the method code. For a GET request, the value of this field has to be 1.

0xC3\0x5A: this pair of bytes represent the Transaction ID. These values must be different every time a new packet is created. The Random Java Object was used to generate these values.

0x91 (1001 0001): this byte is the option header. The first four bits represent the option type, expressed in TLV format. Since this is the only option in this packet, the TLV value corresponds to the option type number (9 for the URI option). The length (in bytes) of the actual option value is set on the last 4 bits.

0x6C: this is the option payload, which contains the value of the option. Since the only option set by this method is the URI, this value represents that address in ASCII code. (6C is the hexadecimal ASCII code for the character 'l', which is the led URI.)

**readResult()**: this method parses a CoAP message and if in the response code field does not contain an error code, it extracts the payload data. Since the data sent by the motes are in binary format, in order to make them readable the *binaryToHex()* method is called. However, TelosB motes sensor data need another conversion to be correctly read. This conversion consists of swapping the order of bytes and then in dividing the data by 100. However, since *readResult()* is meant to read data from a general mote, this conversion needs to be implemented at application level.

Another issue that has been faced in this method was how to determine the end of the CoAP packet, since its size depends only on the datagram packet length and no termination characters were set by the CoAP protocol. Thus a packet was considered terminated if a sequence of five 0x00 bytes were found.

**binaryToHex()**: this method converts binary data to hexadecimal format. It uses the StringBuilder java object in order to format each byte to hexadecimal format.

**setValue()**: this method is called by the SensibleThings platform every time a CoAP set request is sent to a mote. As *getValue()*, this is a synchronized method

which uses a DatagramSocket to send a set request to a mote. To create a Coap-Put packet, the method *createCoapPutMessage()* is called, passing the value to set as an argument. After having sent the packet to a mote, a response message is received. It contains a response code which indicates the status of the PUT request.

**createCoapPutMessage()**: this method builds a CoAP packet for a PUT request, setting the specified value as payload of the packet. Since the value to set needs to be converted in binary format, the method *fromHexString()* is called.

An example packet created by this method is represented in figure 4.10.

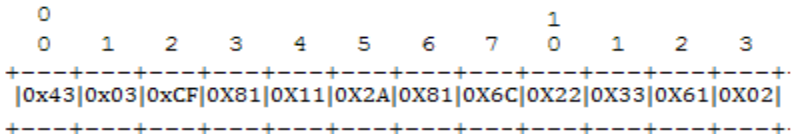


Figure 4.10: CoAP PUT packet.

The first two bytes and the first option field have the same value for every packet created by this method. These bytes have the following meanings:

0x43 (0100 0011): the only difference between this field and the first one of a GET request is that a PUT requires 3 options instead of 1.

0x03: this value represents the method code for a CoAP PUT request.

0xCF\0X81: Transaction ID. As for the createGetPacket method, the Random Java Object is used in order to have different values for each transaction.

0x11: this byte is the header of the first option in this packet. The option type number is 1 (Content type option) and the length of its payload is set to 1.

0x2A: it is the ASCII code for the Content type option, which corresponds to '*' (which means 'text/plain').

0x81: this byte is the header of the second option. Since there is another option before this field, the option type number does not correspond to the actual type number, but this value is calculated according to the TLV format. Therefore the actual type number of the current option is calculated by summing the option delta of the previous option with this one, that is 9 (URI). The length of the current URI is set to 1 byte.

0x6C: is the ASCII code of the 'l' character.

0x22: is the header of the third option. The option type number is calculated by summing this option delta with the option deltas of the two previous options, that is 11 (Token).

0x33\0x61: when a Token option is used, a constant value ('3a') must be set before the payload. Then these 2 bytes represent this value in ASCII code format.

0x02: is the packet payload. This value represents the led status that the user wants to set.

**fromHexString()**: this method converts a hexadecimal string in an array of bytes. It basically parses two hexadecimal values at a time from the string and it converts them to binary format using the *Integer.parseInt()* function.

## 4.4 CoapSensorGateway

CoapSensorGateway enables the communication between the CoapSensorActuator and sensor nodes which do not support the CoAP protocol. It is responsible for converting a CoAP request to the specific format used by the sensor node which is currently connected to the SensibleThings platform.

This class extends the SensorGateway abstract class and realizes a demon Java thread which is always listening to incoming messages from a Datagram Socket. Once a packet is received it checks if it is a CoAP packet and then parses all the single bytes in order to check if it is well formed. Then the type of the request and the sensor's URI are extracted from the packet.

If it is a GET request the actual request to the sensor node is sent with the call of the *getEvent()* method. This method has to be implemented by the developer within a new class which implements the SensorGatewayListener interface. This class has to be set as the argument of the CoapSensorGateway constructor. Then a response CoAP packet is built and, if the GET request was correctly sent to the sensor node, the sensor data are set as payload. Otherwise an error code is set in the response code field.

In case the packet received from the socket was for a PUT request, the *setEvent()* method is called for sending the actual request to the sensor node. As the *getEvent()* method, it also has to be defined by the developer within a new class which implements the SensorGatewayListener interface.

Eventually the response message is sent back to the address from which the request was received.

In figure 4.11 a flow chart representing the sequence of the main operations executed by the CoapSensorGateway is represented. Due to space limitations, a programming language syntax has been used: “packet[i]” represents the i-th byte of the received packet, while “||” and “&&” represents the conjunctions “or” and “and”, respectively.

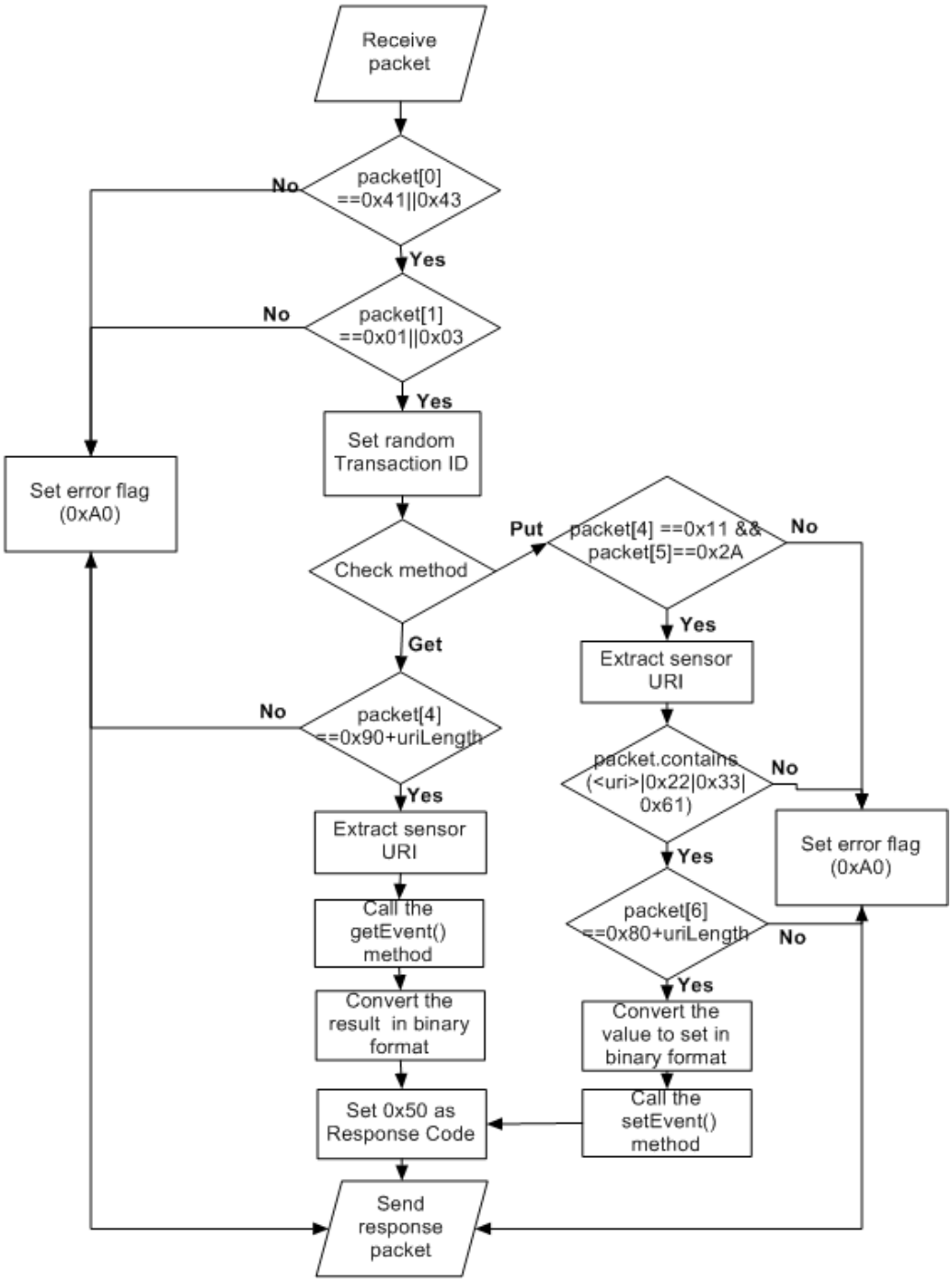



Figure 4.11: CoapSensorGateway's operations flowchart.

# 5 Results

This chapter explains all the tests that have been made in order to evaluate the CoAP communication stack. The first parameter that has been measured is the response time; which describes how fast a CoAP transaction is completed. The second paragraph describes measurements of the size of a CoAP packet followed by a comparision between CoAP packet and standard UDP packet sent over the SensibleThings platform. Finally the scalability of the CoAP stack has been evaluated.

## 5.1 Response time

The first measurement that has been made to evaluate the CoAP stack was to measure how long a CoAP transaction takes to be completed. The Java API *System.nanoTime()* was used to make these measurements. This API returns the current value of the most precise available system timer, in nanoseconds. The criteria used in determining the response time was to sample twenty different response times and then to calculate the average and the standard deviation of these values.

First, the response time between the CoapSensorActuator component and a TelosB sink was measured. The measurements are related to the GET and PUT led requests and also GET temperature requests. To measure the transaction time, two timers have been used: the first one samples the current time just before the CoapSensorActuator sends a CoAP request packet and the second one measures the time after the data payload has been extracted from the response packet. The difference between these two values represents the duration of a CoAP transaction.

Within a single transaction, the CoAP request packet building time and CoAP response parsing time were also measured. In figure 5.1 the scenario of this test is shown, while in figures 5.2, 5.3 and 5.4 these measurements are reported.

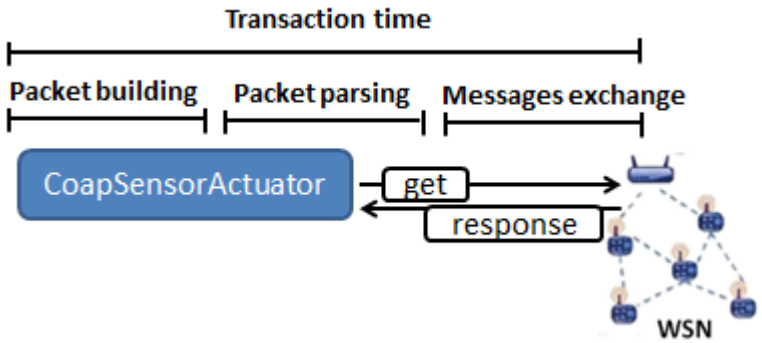



Figure 5.1: CoapSensorActuator response time test scenario.

The values reported in figure 5.2 show that building packets for PUT requests takes a little bit more time than building get packets. This is because CoAP packets for PUT requests have more fields than GET packets (as explained in paragraph 4.3) and then more calculations are made to build the packet. However, building CoAP packets takes very little time, so this difference is insignificant.

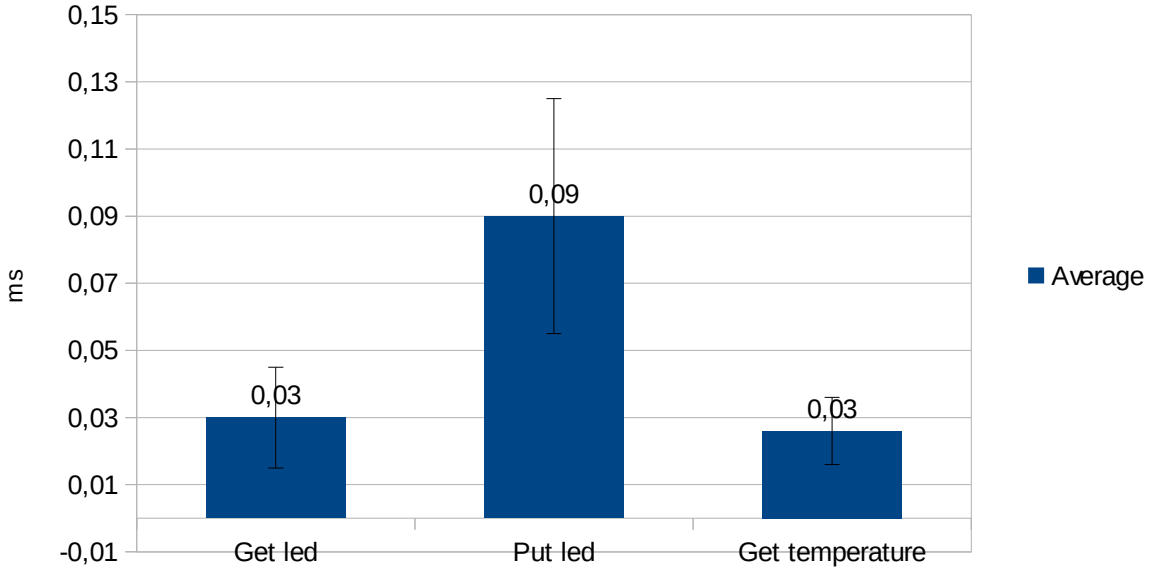



Figure 5.2: Packet building time.

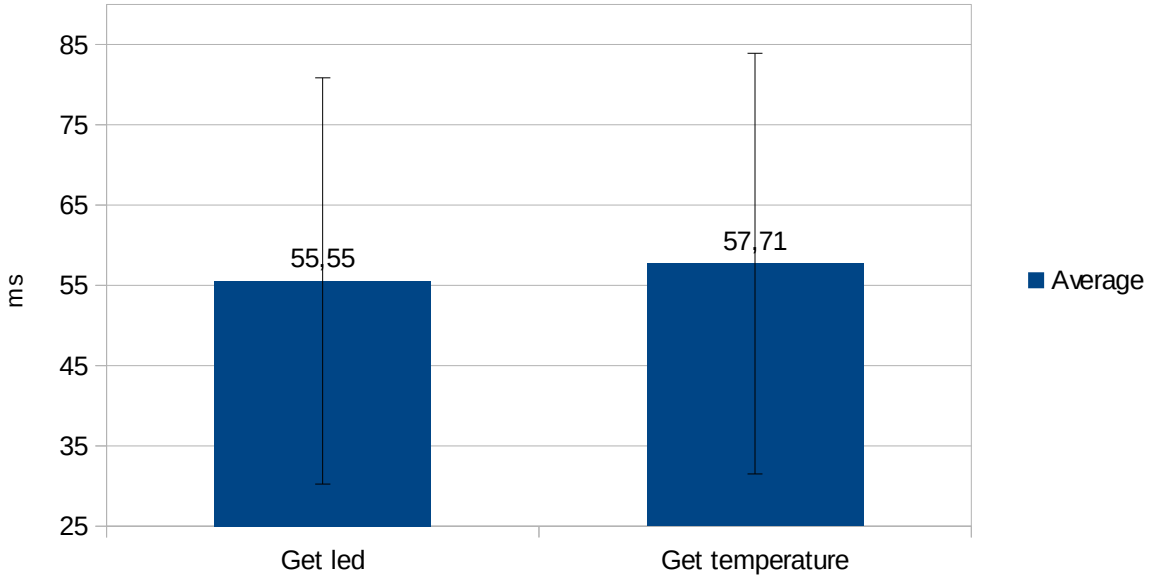



Figure 5.3: Packet parsing time measurements.

Figure 5.3 shows the time required for parsing a CoAP response packet sent back from the sink of a WSN. The duration for both the requests is almost equivalent, however, parsing a GET temperature response packet takes a little bit more time because the data payload of these packets is always bigger than the one in the response packet for a led request.

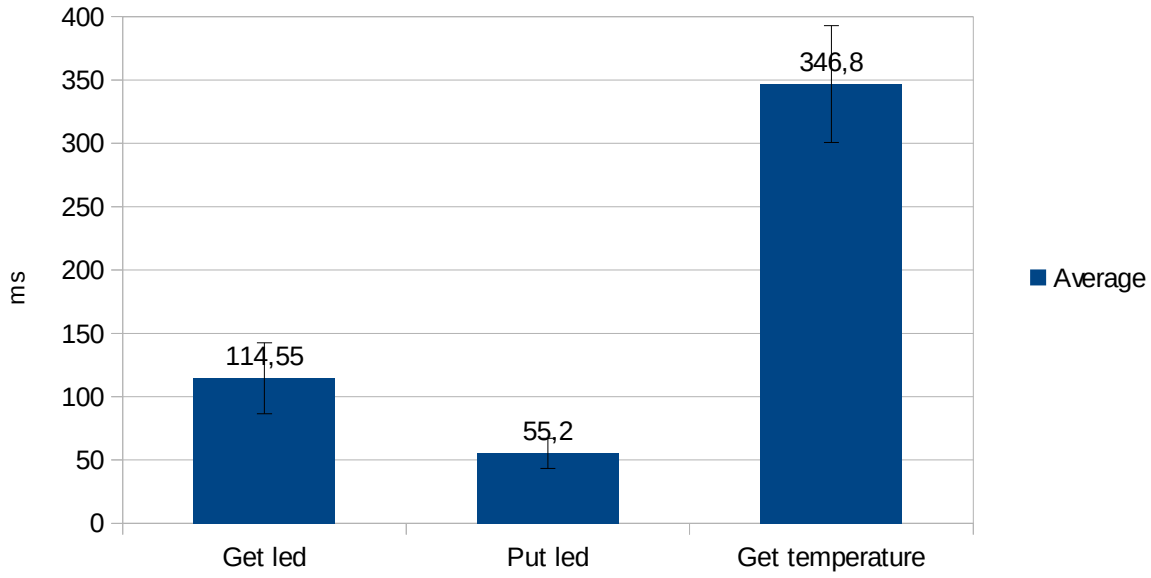



Figure 5.4: Transaction time measurements.

In figure 5.4 the whole duration of a CoAP request is represented. From this figure it is clear that GET temperature is the request that takes more time so far. Comparing this figure with figure 5.3 it can be assumed that for a get led request almost half of the transaction time is taken for parsing the response packet and the remaining time is related to the communication delay with the WSN and for the computation inside the sink. On the other hand, for a GET temperature request the packet parsing time is just a small part of the transaction time. This is because for this type of request the sink has to perform more computations than for a led request. Many of these operations are performed to format the temperature data, which have to be divided by 100 and sent in reverse order.

Another value that was measured was the overhead introduced by the SensibleThings platform. Time was measured between a request and a response packet within a single CoAP transaction to travel between two remote nodes connected by the platform. The first node was a computer which was running an application for retrieving sensor data from a remote node, which was a WSN connected to the SensibleThings platform; as represented in figure 5.5. GET led and GET temperature requests were measured, as shown in figure 5.6.

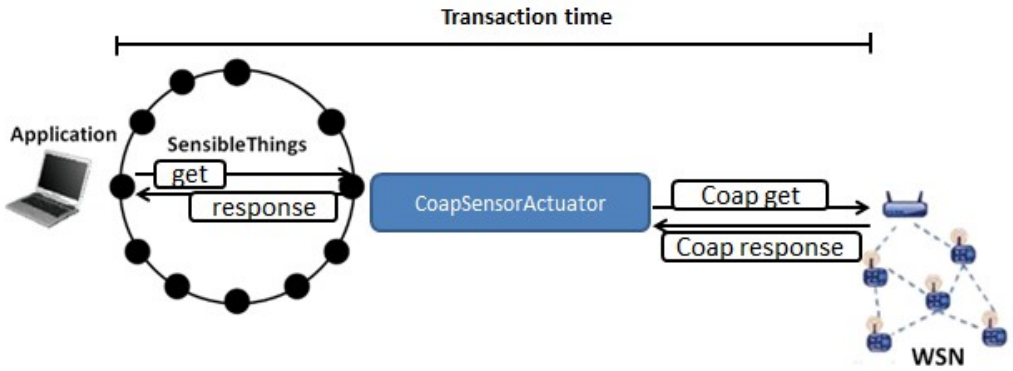



Figure 5.5: SensibleThings overhead test scenario.

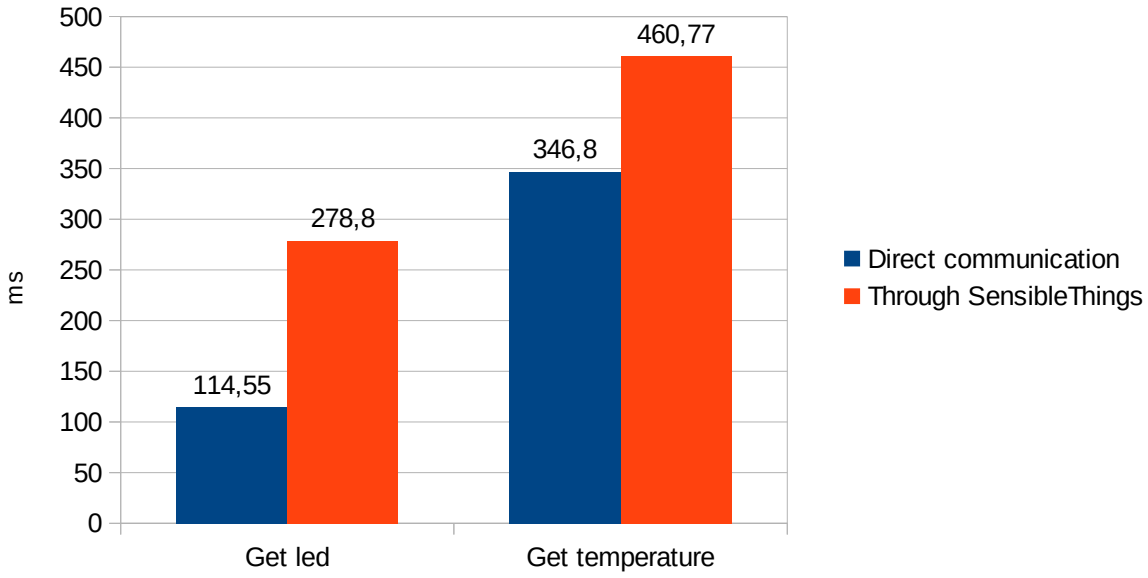



Figure 5.6: Average transaction time.

In order to evaluate the CoAP stack, the CoapSensorGateway was taken into consideration. To measure the overhead introduced by this component a Raspberry Pi device was used.

The first step was to measure the transaction time between the Raspberry Pi (which was directly connected to the SensibleThings platform) and a remote computer which was running a simple application for sending GET requests through the platform. To connect the Raspberry Pi to the SensibleThings platform, the software of the platform was installed on the Raspberry Pi and then a simple application was run on it to register its IP address and its sensor UCI inside the platform. In this way the Raspberry Pi could directly receive requests and send responses from and to the platform.

The second step was to use the CoapSensorGateway to connect the Raspberry Pi to the SensibleThings and then measure the transaction time between this de-

vice and a remote computer. In this scenario the requests sent by the remote application are converted in CoAP requests by the CoapSensorActuator and then handled by the CoapSensorGateway which is directly connected to the Raspberry Pi through a datagram socket. In figure 5.7 both the scenarios are represented, while in figure 5.8 the measurements are reported.

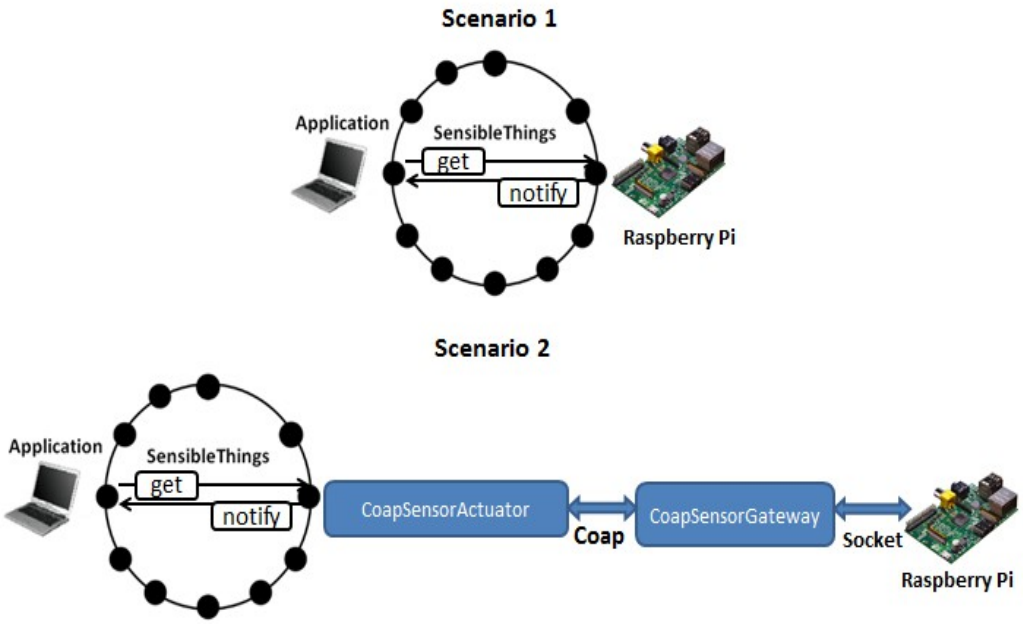



Figure 5.7: CoapSensorGateway test scenarios.

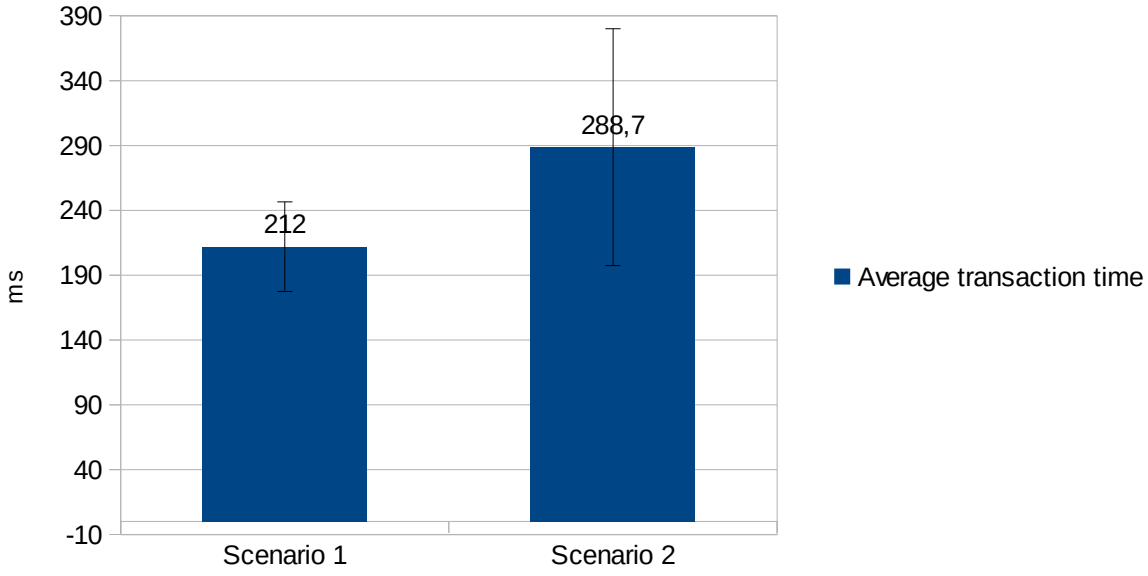



Figure 5.8: CoapSensorGateway test results.

From the test results it appears that the overhead introduced by the CoapSensor-Gateway is pretty low, however, the high value of the standard deviation says that the duration of each transaction is much more variable than the one in the first scenario.

## 5.2 Packet size

The second parameter used to evaluate the CoAP stack was the size of the packets. In order to extract packets from the network dataflow a software named Wireshark was used. Wireshark is a free and open source packet analyzer. It is used for network troubleshooting, analysis, software and communications protocol development. This software allows the user to put network interface controllers in order to see all traffic visible on that interface.

The goal was to compare the GET packets sent over the SensibleThings platform with the CoAP GET packets sent from the CoapSensorActuator to the WSN and then analyze their size. Two computers and a WSN have been used in this test. The first computer was running a simple application to send GET requests to a remote WSN through the SensibleThings platform, while the second computer was directly connected to the WSN and had the task of managing the CoAP communication with the WSN (as represented in figure 5.9).

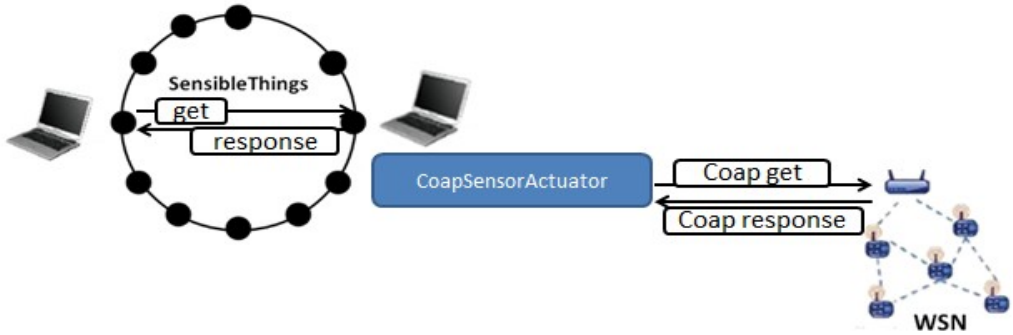



Figure 5.9: Packet size test scenario.

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 95 | 5.142400 | 192.168.1.3 | 193.10.119.42 | TCP | 863 | 46346 > 14523 [PSH, ACK] Seq=727 Ack |
| 96 | 5.156054 | 193.10.119.42 | 192.168.1.3 | TCP | 264 | 14523 > 46346 [PSH, ACK] Seq=940 Ack |

```
▸ Frame 95: 863 bytes on wire (6904 bits), 863 bytes captured (6904 bits)
▸ Ethernet II, Src: Intel_68:fa:3e (00:18:de:68:fa:3e), Dst: 08:bd:43:64:1a:f6 (08:bd:43:64:1a:f6)
▸ Internet Protocol Version 4, Src: 192.168.1.3 (192.168.1.3), Dst: 193.10.119.42 (193.10.119.42)
▸ Transmission Control Protocol, Src Port: 46346 (46346), Dst Port: 14523 (14523), Seq: 727, Ack: 940
▸ Data (797 bytes)
```

Figure 5.10: SensibleThings GET packet.

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 1 | 0.000000 | fec0::100 | fec0::3 | COAP | 71 | Confirmable, GET |
| 2 | 0.286539 | fec0::3 | fec0::100 | COAP | 70 | Acknowledgement, 200 OK |

```
▸ Frame 1: 71 bytes on wire (568 bits), 71 bytes captured (568 bits)
▸ Linux cooked capture
▸ Internet Protocol Version 6, Src: fec0::100 (fec0::100), Dst: fec0::3 (fec0::3)
▸ User Datagram Protocol, Src Port: 33522 (33522), Dst Port: 61616 (61616)
▸ Constrained Application Protocol, TID: 57300, Length: 7
```

Figure 5.11: CoAP GET packet.

Comparing the packets shown in figures 5.10 and 5.11 it appears clear that CoAP packets are smaller in size than the packets sent over the platform. The actual size of a CoAP packet for a GET temperature request (that is the packet from figure 5.9) is only 7 bytes but the whole size of the packet sent from the CoapSensorActuator to the WSN is 71 bytes. This is because the CoAP protocol relies on UDP as transport layer and IPv6 as network layer (as explained in paragraph 2.4.4), so a packet must include both IPv6 and UDP headers in order to be sent over the network. On the other hand, packets sent over the SensibleThings platform are much bigger in size than CoAP packets because they are serialized before being sent. An increase in the size of serialized data is one of the consequences in using the serialization. Another difference between these two types of packets is that SensibleThings packets rely on TCP and IPv4 as transport protocol and network protocol, respectively.

## 5.3 Scalability

The third test that has been conducted was regarding the scalability of the developed communication stack. The first idea was to test the scalability of the CoapSensorActuator component in order to measure how many motes could be connected to the same sink before the requests coming from this WSN had caused a drop in the performances of this component (that means a high increase of the response time for each request). However, since each CoapSensorActuator component can be bound to just one mote, this kind of test was then considered not of interest. Nevertheless, the scalability of the CoapSensorGateway was considered of interest. Since this component is implemented as a daemon thread which is continuously listening for incoming CoAP requests, a test was run to see how many requests it could have managed before having a significant drop in the response performance. The idea was to create a simple application which created and then run a set of threads, where each one created and used its own CoapSensorActuator object for sending multiple CoAP GET requests to the CoapSensorGateway; as shown in figure 5.12. It was then measured how many requests coming at the same time from different threads could have been managed by the CoapSensorGateway before having a significant increase in the response time. To not add any further overhead, no sensor nodes were connected to the CoapSensorGateway; therefore a static value was sent as a response value for a GET request.

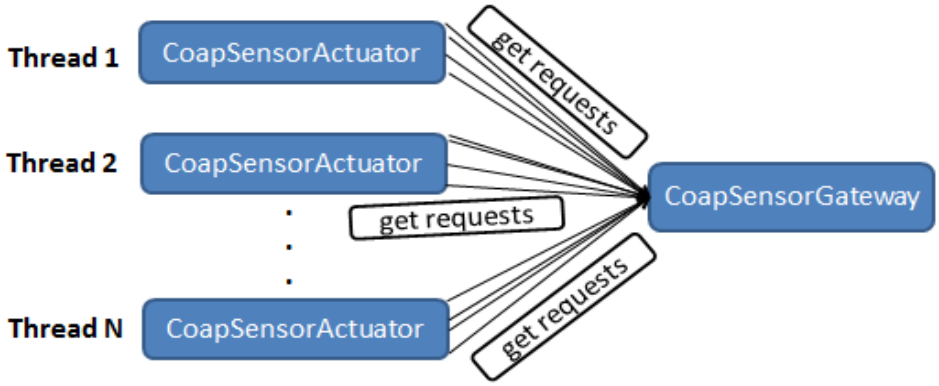



Figure 5.12: Scalability test scenario.

Three measurements with different number of threads have been conducted: 10 threads in the first one, 100 in the second one and 1,000 in the last one. The results of the first two measurements, after 100 iterations, are shown in figure 5.13.

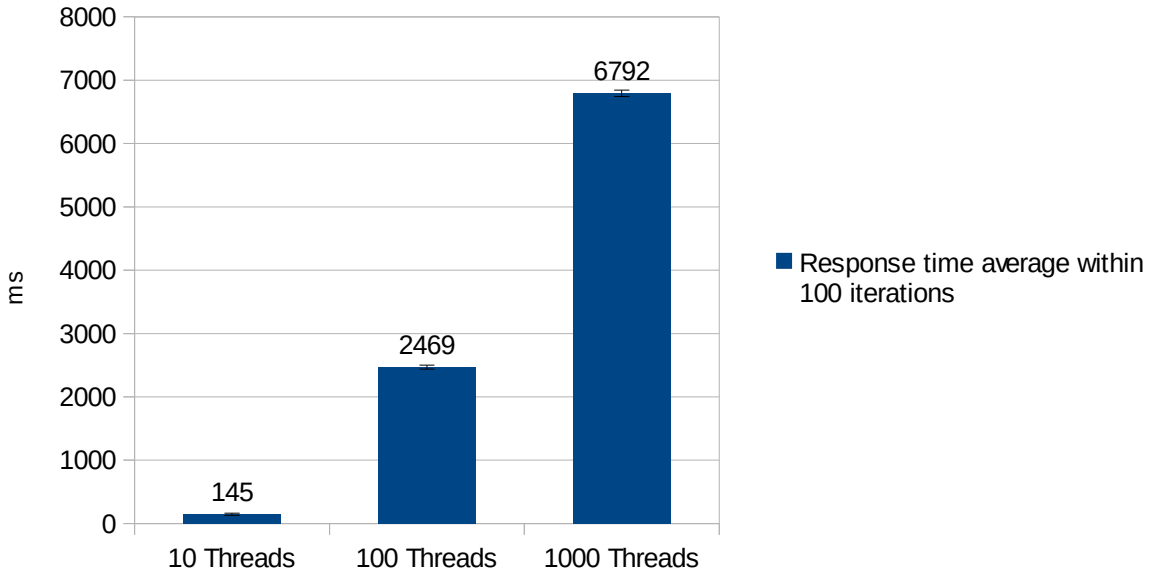



Figure 5.13: Scalability test results (a).

The results represented in figure 5.13 show an increase in the response time, related to the number of the running threads as expected. However, in both the scenarios with 10 and 100 running threads, after the first iteration where the whole set of threads were created, there was a huge increase in the response time; for the remaining 99 iterations the value of the response time remained almost steady signifying that the CoapSensorGateway was able to manage such a number of requests without having further drop in the performance. On the other hand, the results with 1,000 threads running at the same time were different, since the response time started to be significant after the first iteration and then kept increasing during the next 10 iterations as shown in figure 5.14; where the difference of the response time between the values within the second, fifth, tenth iterations with that one within the first iteration are reported. This is an unwanted result which suggests that this number of requests was too high to be managed by the CoapSensorGateway. One of the main reasons for this behavior is caused by the synchronization used in both get and set methods within the CoapSensorActuator. When the number of requests is high as in this last scenario, each request sent from the same CoapSensorActuator object has to wait for the previous one to be terminated, thus there is a huge increase in the response time for each new request. This result could be improved by modifying the code of both get and set methods in a way that they could manage multiple requests at the same time.

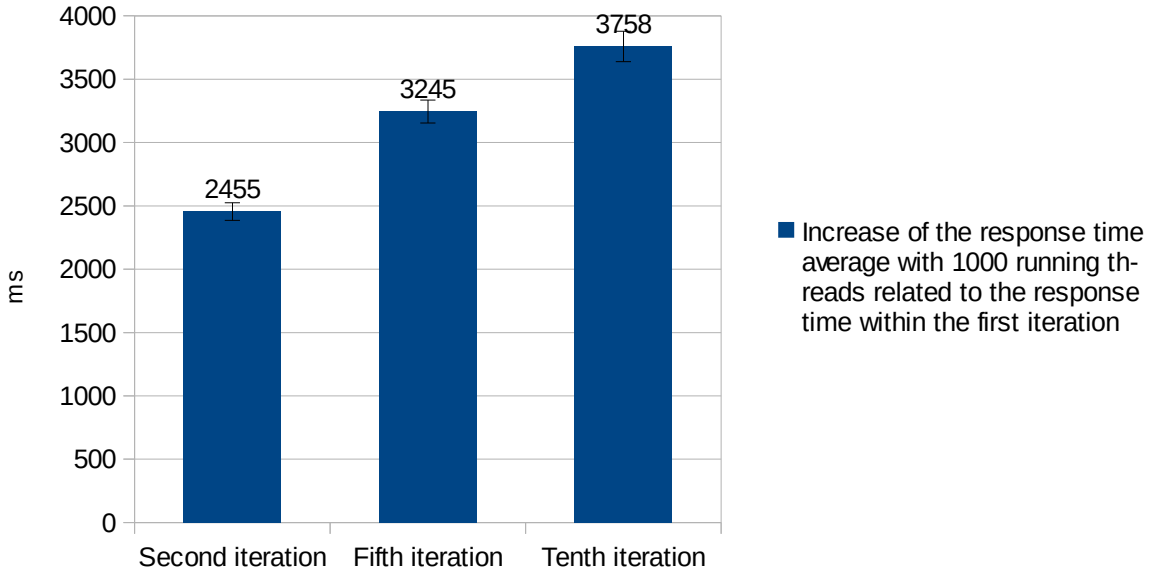



Figure 5.14: Scalability test results (b).

## 5.4 Proof of Concept application

In order to test both the CoapSensorActuator and the CoapSensorGateway, a Proof of Concept application has been developed. This application simulated a real-world application that is very common in Sweden: a fire detection system. This test was an important part of this thesis work since it was helpful to test all the developed components together and to discover problems which would not surface if only testing each component individually.

### 5.4.1 Potential real-world scenario

The forest is considered one of the most important and indispensable resources as well as the protector of the Earth's ecological balance. Forest fires are a constant threat to these ecological guardians. Recently, with the effect of factors such as climatic fluctuations, human activities, etc., a tendency of intense increase of forest fires was showed. At present, traditional forest fire prevention measures have been ground patrolling, watch towers, aerial prevention, long distance video detection and satellite monitoring and so on. In view of all the deficiencies of conventional forest fire detection, it is necessary to bring in a new method for a more efficient ground forest fire detection system.

Compared with the traditional techniques of forest fires detection, WSNs technology is a very promising green technology for the future in efficiently detecting forest fires; according to the features explained in paragraph 2.2. In this case, a Wireless Sensor Network could be deployed to detect a forest fire in its early stages. A number of sensor nodes would need to be pre-deployed in a forest. Each sensor node could then gather different types of row data from sensors, such as temperature, humidity, pressure and position. All sensing data would be sent wirelessly in ad-hoc fashion to a sink station, which in turn

would transmit data to the control center via a transport network such as GSM, UMTS, Satellite, TCP/IP networks. In figure 5.15 a possible scenario of a WSN deployment for fire detection is shown.

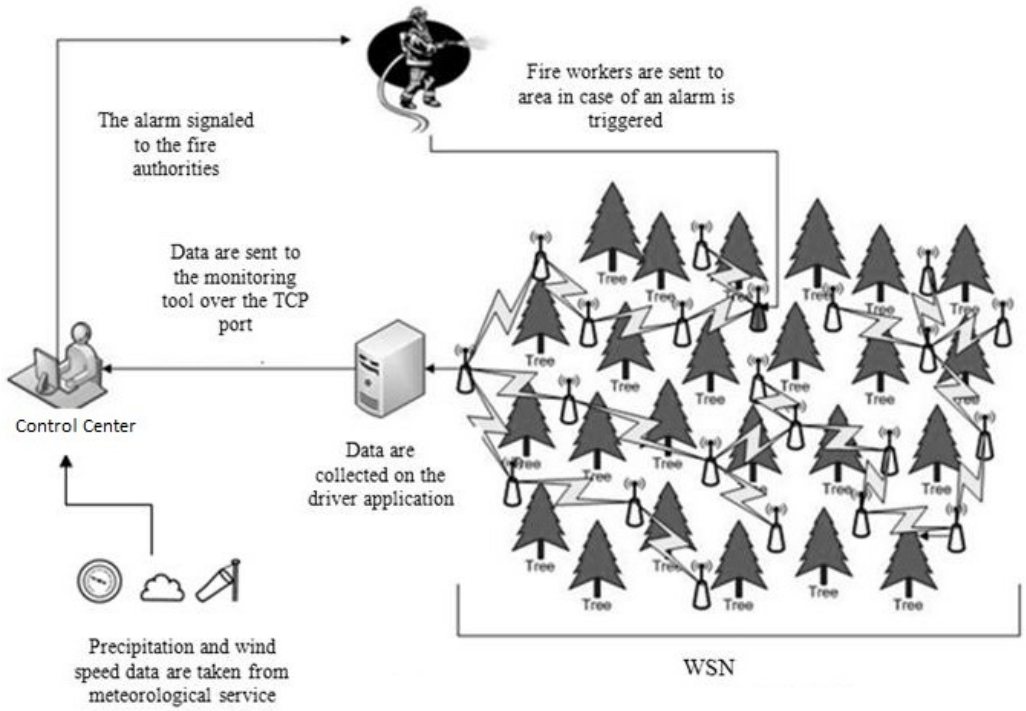



Figure 5.15: WSN fire detection scenario.

On the control center the sensor data could then be used to detect forest fires. To detect fires many different techniques could be implemented, however the two most common solutions are the Canadian system and the South Korean system, as explained in [27].

### 5.4.2 Implementation and results

The scenario explained in the previous paragraph was implemented in small scale in a computer lab of the university. In this test three machines have been used, which were interconnected by the SensibleThings platform. The first machine was connected to a Raspberry Pi (equipped with a temperature sensor) through a Local Area Network (LAN) and was running both the CoapSensorActuator and CoapSensorGateway classes in order to communicate with the Raspberry Pi. The second node was a notebook which was connected through a USB cable with two WSNs formed by 2 TelosB motes each (the first one used as a sink and the other one as a standard WSN mote). In this node two different instances of the CoapSensorActuator object were created in order to enable the communication between the SensibleThings node and both the WSNs. The third machine represented the control center of the system and was running the Proof of Concept application. In figure 5.16 the before mentioned scenario is represented.

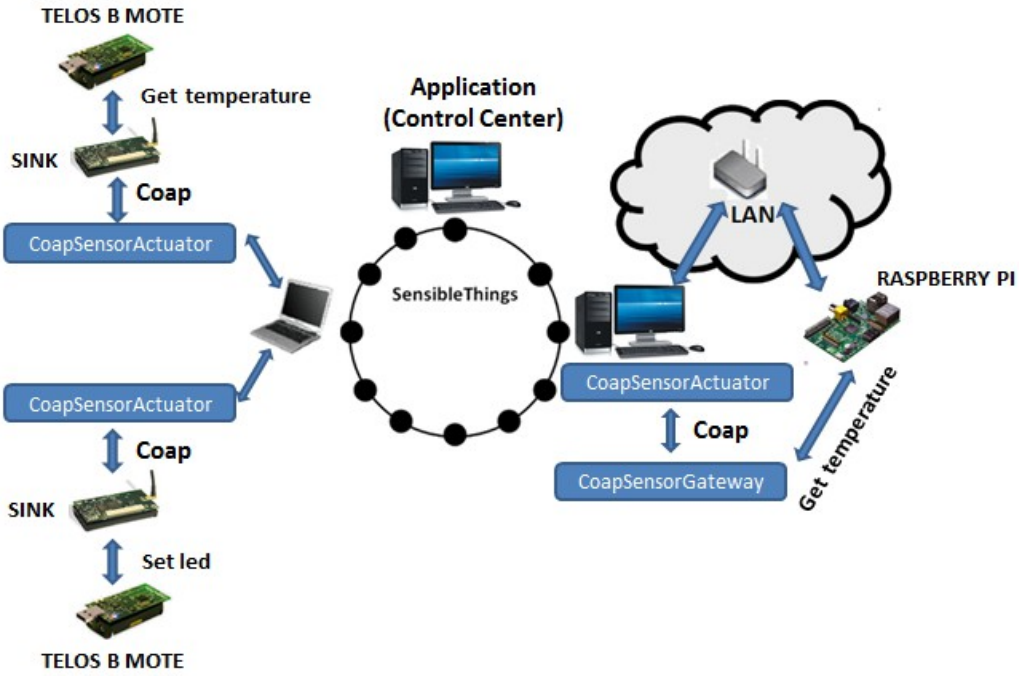



Figure 5.16: Proof of Concept application scenario.

This test tried to simulate a scenario where the sensors deployed in a forest were developed by different manufacturers and the CoAP protocol was not supported by all of them. For this reason, two different kinds of devices were used: TelosB motes and a Raspberry Pi, which is a credit card sized single board computer. Since the Raspberry Pi did not support the CoAP protocol, it was necessary to use the CoapSensorGateway in order to enable the communication between the platform and this device. Moreover, a server application for the Raspberry Pi was implemented. This application realizes a Java daemon thread which is always listening for incoming requests from the CoapSensorGateway on a datagram socket and then uses the same socket to send its sensor data (which are periodically saved in a file on the Raspberry Pi's memory).

On the third node the actual Proof of Concept application has been developed. The application uses the SensibleThings platform to obtain the actual address of both the temperature sensors within the Raspberry Pi and the first TelosB mote, specifying their UCIs ('*alessandro@miun.se/tinyos/temperature*' and '*alessandro@miun.se/raspberrypi/temperature*', respectively) and then to collect temperature data from them. Once it has received the temperature from both the sensors, the application checks if the average of both the values is above a certain threshold (19 °C). If it is, a Set led request is sent to the second TelosB mote (specifically to its UCI: '*alessandro.aloisi@miun.se/tinyos/led*') and then a led is switched on. This action represents an alarm sent to the nearest fire station in the real world scenario. In figure 5.17 the output of the Proof of Concept application is shown.

```
SensibleThings ProofOfConceptApp is running

Press any key to shutdown
ResolveResponse: alessandro@miun.se/raspberryPi/temperature 193.10.119.42:51318
ResolveResponse: alessandro@miun.se/tinyos/temperature 193.10.119.42:35565
ResolveResponse: alessandro@miun.se/tinyos/led 193.10.119.42:35565
GetResponse: alessandro@miun.se/tinyos/temperature 19.55 °C 193.10.119.42:35565
GetResponse: alessandro@miun.se/raspberryPi/temperature 20.312 °C 193.10.119.42:51318
Average temperature= 19.931 C ,the led have been set to status 1
```

Figure 5.17: Proof of Concept application output.

# 6 Conclusion

The problem treated in this thesis is perhaps one of the most challenging for IoT developers. The thesis investigated enabling communication between Internet-of-Things and Wireless Sensor Networks, irregardless of their network connection and then to utilize the sensors data for context aware applications. In order to solve the problem, the goals described in chapter 3 were achieved using the following methods:

Goal 1: evaluating three different solutions of connecting Wireless Sensor Networks to an Internet-of-Things scenario was achieved by searching for articles and papers in many research databases and then analyizing three different IoT platforms: SensibleThings, ETSI M2M and SENSEWEB.

Goal 2: understanding the most common operating systems used in Wireless Sensor Network, resulted in discovering TinyOS and Contiki. I studied the documentation about these OS in order to assess which was the best to be used in my thesis. Eventually I concluded that TinyOS was the best option, since Contiki does not work well with TelosB motes because the commands are set for a type of sky motes that is not valid for TelosB motes new versions.

Goal 3: investigating which communication protocols these operating systems support, was achieved analyizing the documentation about these OSs. This research led me to choose the CoAP protocol for its compatibility with TelosB motes and its lightweight protocol stack.

Goal 4: to implement a communication stack which enables communication between Wireless Sensor Networks and Internet-of-Things applications, led to the analysis of the SensibleThings platform and running some simulations to discover its features. The Sensor and Actuator Layer of the platform was then extended with the CoapSensorActuator and CoapSensorGateway classes.

Goal 5: to evaluate the performance and responsiveness of the implemented solution, some tests were run in order to measure the get/set transactions response time and then compared these values with the response time of the SensibleThings get/set transactions to measure the overhead introduced by the CoAP stack. Moreover, a packet sniffer software was used in order to extract the packets from the network data flow and then, some comparisons between the packets created by the SensibleThings platform with the ones built by the CoAP stack, were made. Eventually, the scalability of the CoapSensorGateway was tested, measuring how many requests could it manage before having a drop in the performance.

Goal 6: investigating possible real-world applications for the implemented solution, I concluded that this research could be applied to a fire detection system, especially introduced in the prevention of fires in forests, a prominent

geographical feature in Sweden. Eventually I developed a Proof of Concept application in order to simulate this system in a small scale and to also test the developed classes.

## 6.1 Discussion

Many choices regarding the tools and the methods that have been used in this thesis work were forced by the type of motes that I had to use, the TelosB motes. Since this type of mote supports only Tiny OS and Contiki I could not test other operating systems, like MantisOS or Nano-RK; which would have been interesting to install in order to compare the performances of all these IoT OSs. I did experience some problems using Contiki on the TelosB motes, probably due to some compatibility issues with the new version of the motes and thus eventually I decided to use TinyOS. Nevertheless, my choice was consistent with that of most of the IoT developers, since TinyOS is the most used OS in WSNs at the moment. The choice to use CoAP as application protocol (for retrieving sensor data) was forced by certain limitations. Indeed, the documentations of TinyOS is quite unspecific and I was not able to figure out if this OS supported other kinds of application protocols. However, CoAP is supported by the majority of WSN operating systems, therefore the choice of using this protocol allowed the use of the communication stack developed in my thesis not only to communicate with TelosB motes but also with any kind of mote which supports CoAP.

As a result of my thesis work, the Sensor and Actuator layer of the SensibleThings platform was extended and as a consequence, the communication between the platform and the WSNs has been enabled. My work was based on the SensibleThings platform because it was the IoT platform developed by Mid Sweden University and to connect the platform with WSNs was one of the main features which was not implemented yet. However, the CoAP stack that has been implemented could be easily exported to other Java based IoT platforms, since the SensibleThings APIs were not used within the code. In the following tables the values collected during the evaluation phase (explained in paragraphs 5.1 and 5.2) are summarized.

| | **Through SensibleThings** | **Direct communication to the WSN** |
|---|---|---|
| GET led transaction time | 278,8 ms | 114,55 ms |
| GET temperature transaction time | 460,77 ms | 346,8 ms |

Table 1: CoapSensorActuator response time measurements.

| | **Raspberry Pi connected to the CoapSensorGateway** | **Raspberry Pi directly connected to SensibleThings** |
|---|---|---|
| GET transaction time | 288,7 ms | 212 ms |

Table 2: CoapSensorGateway response time measurements.

| | **SensibleThings packet** | **CoAP packet** |
|---|---|---|
| Request packet size | 863 bytes | 71 bytes |
| Response packet size | 264 bytes | 70 bytes |

Table 3: Packet size test measurements.

Analyzing these values it appears clear that the CoAP stack added an overhead in terms of response time to the SensibleThings platform, since the requests coming from a remote node have to be translated into CoAP requests before being sent to a WSN. However, the CoAP protocol brings a decrease in the packet size and makes the developed communication stack 'open', since it can be used to communicate with several different types of WSN motes.

### 6.1.1 Ethical issues

There are many ethical issues that may arise from the IoT. The biggest one is related to individual privacy. Many people today wear sensors when they move through their daily lives to track their heart rate, miles traveled, or steps taken. These activity monitor sensors are connected wirelessly to smart phones and to the Internet to enable users to track metrics over time. By collecting information on people and their habits, companies will have the ability to infringe upon consumers. Therefore, when companies have this information readily available to them, and they have the possibility to increase their revenue tremendously, they are more likely to infringe upon our privacy. Another ethical issue of the IoT is that it can discriminate against certain groups of people that do not have access to the Internet. There are many countries where lower income families do not have access to the Internet, so they will not be able to reap the benefits offered by the Internet-of-Things. In other words, families that do not have the money to purchase some of these devices will be as well off as other more affluent families. In the end this could cost the lower socioeconomical families more, and decrease the inefficiencies in higher socioeconomical classes. The third ethical issue of the IoT is related to security. In this new media, which is no longer in its infancy, the vulnerabilities and attacks are various, caused by technological advances and proliferated through lack of user awareness. This problem is particularly related to the CoAP communication stack which has been implemented in this thesis work, since no security mechanisms were used. All the packets sent between a WSN sink and a SensibleThings node could be intercepted and modified. For example, one threat for a real world fire detection system could be a fake packet with a high temperature value sent to the control center in order to simulate a fake fire.

## 6.2 Future work

Some improvements can be applied to the current work. In relation to the written code, firstly the system needs to implement some security mechanism in order to be adapted to real world applications. For instance cryptographic protocols like SSL and RSA could be implemented in order to enable secure communication channels between WSNs and the SensibleThings platform; specifically between the WSN sink and the CoapSensorActuator and also between the CoapSensorGateway and the attached sensor node. Another issue which needs to be solved is how to handle multiple CoAP transactions. At the moment, both the GET and PUT methods defined in the CoapSensorActuator class are synchronized methods, which means that when a request comes to this component it needs to wait until the previous one has received a response from the WSN sink. In order to improve the performance of the system it would be useful to improve the before mentioned methods in a way where they could manage multiple CoAP transactions at the same time.

Related to the motes which have been used in this thesis work it would be interesting to test the implemented communication stack with other types of motes other than the TelosB mote. Only motes which support CoAP could communicate with the CoapSensorActuator and could then be used instead of TelosB motes. However, if these different types of motes support CoAP, there should not be any compatibility issues and they should be able to receive and send CoAP packets from and to the CoapSensorActuator, respectively.

Other future work that could be interesting would be to export the classes implemented in this work to other IoT platforms, like Senseweb and ETSI M2M, in order to figure out which platform has the best performance. Therefore, both the response time and the packet size could be measured using different types of motes and IoT platforms. A comparison between all these values could be made in order to investigate which would be the best solution to implement.

# References

[1] Charith Perera, Arkady Zaslavsky, Peter Christen, Dimitrios Georgakopoulos, Context Aware Computing for The Internet of Things: A Survey, IEEE Communications Surveys & Tutorials, Volume xx, Issue x, Third Quarter 2013.

[2] K. Ashton, “That ’internet of things’ thing in the real world, things matter more than ideas,” RFID Journal, June 2009, http://www.rfidjournal.com/article/print/4986.

[3] D. L. Brock, “The electronic product code (epc) a naming scheme for physical objects,” Auto-ID Center, White Paper, January 2001, http://www.autoidlabs.org/uploads/media/MIT-AUTOID-WH-002.pdf

[4] M. Presser, A. Gluhak, The Internet of Things: Connecting the Real World with the Digital World, EURESCOM mess@ge – The Magazine for Telecom Insiders, vol. 2, 2009.

[5] P. Guillemin and P. Friess, “Internet of things strategic research roadmap,” The Cluster of European Research Projects, Tech. Rep., September 2009.

[6] European Commission, “Internet of things in 2020 road map for the future,” Working Group RFID of the ETP EPOSS, Tech. Rep., May 2008.

[7] B. Sterling, Shaping Things – Mediawork Pamphlets, The MIT Press.

[8] L. Atzori, A. Iera, and G. Morabito.The Internet of Things: A Survey.-Computer Networks.Vol. 54, No. 15, pp. 2787-2805. October 2010.

[9] A. Dey and G. Abowd, “Towards a Better Understanding of Context and Context-Awareness,” in CHI 2000 Workshop on The What, Who, Where, When, and How of Context-Awareness, 2000, pp. 304–307.

[10] M. Weiser, “The computer for the 21st century,” Scientific American, vol. 265, no. 3, pp. 66–75, July 1991.

[11] Ian F.Akyildiz, Mehmet Can Vuran, Wireless Sensor Networks, ISBN 978-0-470-03601-3, WILEY, 2010.

[12] Fundamentals of Wireless Sensor Networks , Waltenegus Dargieand, Christian Poellabauer, Wiley, 2010.

[13] M. Johnson, P. Van De Ven, M. Healy, And M. J. Hayes, "A Comparative Review Of Wireless Sensor Network Mote Technologies", Proc. IEEE Sensors Conference, Pp 1695-1701, Auckland, New Zealand, November, 2009.

[14] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, D. Culler, *"TinyOS: An Operating System for Sensor Networks",* Ambient Intelligence 2005, pp 115-148.

[15] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, November 2004.

[16] Salman N; Rasool I; Kemp AH "*Overview of the IEEE 802.15.4 standards family for low rate wireless personal area networks"* in:"Proceedings of the 2010 7th International Symposium on Wireless Communication Systems, ISWCS'10," pp.701-705. 2010.

[17] Prativa P. Saraswala "*Survey on upcoming ZigBee technology in future communication system",* IJECSE: International Journal of Electronics and Computer Science engineering Volume 1, Number 3, 2012.

[18] Wikipedia about ZigBee standard 21-02-2014 (Retrieved March 2014) [www] Available: http://en.wikipedia.org/wiki/ZigBee.

[19] Zach Shelby and Carsten Bormann,*"6LoWPAN: The wireless embedded Internet-Part 1: Why 6LoWPAN?"*,EE Times, May 23, 2011.

[20] Walter Colitti, Kris Steenhaut, Niccolò De Caro, *"Integrating Wireless Sensor Networks with the Web",* IPSN 2011 – Extending the Internet to Low power and Lossy Networks (IP+SN 2011), April 12-14, 2011, Chicago, Illinois, USA.

[21] Hong, E. Suh, and S. Kim, "Context-aware systems: A literature review and classification,," Expert Systems with Applications, vol. 36,no. 4, 2009, pp. 8509–8522.

[22] T. Kanter, S. Forsström, V. Kardeby, J. Walters, U. Jennehag, and P. Österberg, "Mediasense–an internet of things platform for scalable and decentralized context sharing and control," in ICDT 2012, The Seventh International Conference on Digital Telecommunications, 2012, pp. 27–32.

[23] Hersent, Olivier; Boswarthick, David; Elloumi, Omar, "The Internet of Things: key applications and protocols", Chapter 14, p.237-267, 2011.

[24] Grosky, W.I. ; Kansal, A. ; Nath, S. ; Jie Liu, Jie Liu ; Feng Zhao, Feng Zhao, "SenseWeb: An Infrastructure for Shared Sensing", Journal IEEE MultiMedia, Volume 14 Issue 4, October 2007.

[25] CoapBlip guide 11-07-2013 (Retrieved April 2014) [www] Available: http://tinyos.stanford.edu/tinyos-wiki/index.php/CoAP_-13.

[26] Coap draft 26-10-2010 (Retrieved April 2014) [www] Available: http://tools.ietf.org/html/draft-ietf-core-coap-03.

[27] Kechar Bouabdellah, Houache Noureddine, Sekhri Larbi, "Using Wireless Sensor Networks for Reliable Forest Fires Detection", Procedia Computer Science, Volume 19, 2013, Pages 794-801, The 4th International Conference on Ambient Systems, Networks and Technologies (ANT 2013)

# Appendix A: CoapBlip installation guide

The first step to install the CoapBlib on the motes has been to install Tiny OS on Linux Ubuntu 12.04 LTS machine. The main guidelines to install TinyOS are the following:

1. Add the TinyOS respository link (deb http://tinyos.stanford.edu/tinyos/dists/ubuntu natty main) at the end of the file: /etc/apt/sources.list;

2. Run the command: sudo apt-get install tinyos-2.1.2;

3. Configure permission for user: sudo chown user:user -R /opt/tinyos-2.1.2/ ; sudo chown user -R /opt/tinyos-2.1.2;

4. Add environment variables to bashrc: at the end of that file add the following lines (export TOSROOT=/home/user/tinyos-2.1.2 ; export TOSDIR=$TOSROOT/tos; export CLASSPATH=$TOSROOT/support/sdk/java/tinyos.jar:.$CLASSPATH; export MAKERULES=$TOSROOT/support/make/Makerules).

Once Tiny OS has been installed on the machine, it is possible to compile the CoapBlib library. To compile the library, change directory to /support/sdk/c/coap within the home directory of Tiny OS and run the following commands: 1- autoconf, 2-./configure, 3-make.

At this point is possible to install the CoapBlib via a USB connection on each mote, running this command:”make telosb blip coap install,<addr> bsl,/dev/ttyUSB0” within the following directory: /apps/CoapBlip. It is possible to set the last field of the mote's IPv6 address, writing the selected value in the <addr> field.

Then, to enable the communication between the computer and the motes, the PPPRouter has to be installed on the sink node. To install this application, connect the sink to the computer with a USB cable and then execute the following command:”make telosb blip install bsl,/dev/ttyUSB0” within the following directory: /apps/PppRouter. Next, to enable the actual PPP connection the following command needs to be run:”sudo pppd debug passive noauth nodetach 115200 /dev/ttyUSB0 nocrtscts nocdtrcts lcp-echo-interval 0 noccp noip ipv6 ::23, ::24”. Eventually, to make the computer reachable from the sink a IPv6 address has to be provided to it. Then, in a new terminal run the following command: ”sudo ifconfig ppp0 add fec0::100/64”. Now it is possible to send CoAP requests to the motes.