INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

isep

# Implementation of an IoT-based Sensor Network Integrating IoT-based Sensor Networks with Large-Scale Message Distribution

FERNANDO PEDRO TEIXEIRA BESSA

Outubro de 2015

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

# Implementation of an IoT-based Sensor Network Integrating IoT-based Sensor Networks with Large-Scale Message Distribution

**Fernando Pedro Teixeira Bessa**

**Dissertation to obtain Master degree in Computer Science, Specialization in Architectures, Systems and Networks.**

**Advisor: Nuno Alexandre Pereira**

Porto, Maio de 2015 Calibri, 12pt

# Abstract

As technology advances not only do new standards and programming styles appear but also some of the previously established ones gain relevance. In a new Internet paradigm where interconnection between small devices is key to the development of new businesses and scientific advancement there is the need to find simple solutions that anyone can implement in order to allow ideas to become more than that, ideas.

Open-source software is still alive and well, especially in the area of the Internet of Things. This opens windows for many low capital entrepreneurs to experiment with their ideas and actually develop prototypes, which can help identify problems with a project or shine light on possible new features and interactions.

As programming becomes more and more popular between people of fields not related to software there is the need for guidance in developing something other than basic algorithms, which is where this thesis comes in: A comprehensive document explaining the challenges and available choices of developing a sensor data and message delivery system, which scales well and implements the delivery of critical messages. Modularity and extensibility were also given much importance, making this an affordable tool for anyone that wants to build a sensor network of the kind.

**Keywords**: Sensor Network, Programming, Internet of Things, Message Trading

# Acknowledgments

I have ISEP to thank the most for making equipment available for me to implement a solution and for always keeping their door open to their students in times of need, such as making classrooms for the development of this project available.

I also want to thank my thesis coordinator Nuno Pereira for doing his best to guide me through the making of this document and for trying his hardest to understand and critique most aspects of the project. Even when at times, time itself was scarce, there was always an opportunity for one more meeting before class.

To everyone who closely followed my thesis project, such as many friends and fellow ISEP students, thank you for showing interest in other people's work, for such is a mark of a truly exceptional engineer.

And finally, I would like to express my extended thank you to every engineer, graduated or not, who sees software development and programming not as a way to make money but as a tool to develop a better World.

# Index

**1 Introduction** ........ 1

1.1 Motivation ........ 2

1.2 Contribution ........ 2

1.3 Method ........ 3

1.4 Thesis structure ........ 3

**2 WSN's State of the art** ........ 5

2.1 Operating Systems ........ 5

2.1.1 Operating Systems Architecture ........ 5

Monolithic Architecture ........ 5

Microkernel Architecture ........ 6

Layered architecture ........ 7

2.2 Programming models in small Operating Systems ........ 8

2.2.1 Event-driven programming ........ 9

2.2.2 Thread-driven programming ........ 9

2.3 Available Operating Systems alternatives ........ 9

2.3.1 Contiki-OS ........ 10

2.3.2 TinyOs ........ 11

2.3.3 Comparison and decision ........ 11

2.4 IoT communication stack and related standards ........ 12

2.4.1 Link Layer ........ 13

LTE 4G ........ 13

Bluetooth Smart ( 802.15.1 ) ........ 15

Wi-Fi ( 802.11g ) ........ 17

IEEE 802.15.4 ........ 17

2.4.2 Network Layer ........ 18

6LowPAN ........ 19

ZigBee ........ 20

2.4.3 Transport Layer ........ 20

TCP ........ 21

UDP ........ 23

2.4.4 Application Layer ........ 24

CoAP ........ 25

HTTP ........ 26

MQTT ........ 27

3 Message delivery state of the art ........ 29
3.1 XMPP ........ 29
3.1.1 Openfire ........ 30
3.1.2 Smack API ........ 31
3.2 AMQP ........ 32
3.2.1 RabbitMQ ........ 33
3.3 Other implementations and architectures ........ 34
3.3.1 Xively ........ 34
3.3.2 ThingWorx ........ 35
4 Project Architecture and technologies ........ 37
4.1 System architecture ........ 37
4.1.1 WSN – Sensor network ........ 38
4.1.2 Linux Gateway ........ 39
4.1.3 Linux Server ........ 42
4.1.4 Android App ........ 42
4.2 Rejected approaches ........ 42
5 System implementation ........ 45
5.1 Some considerations ........ 45
5.2 Sensor Nodes ........ 45
5.3 Border Router ........ 47
5.4 Linux Gateway ........ 49
5.5 Linux Server ........ 51
5.6 Mobile App ........ 53
6 Conclusion ........ 55
7 Works Cited ........ 57

# List of Figures

Figure 1 - Visual representation of a Monolithic kernel [3] ........ 6
Figure 2 - Comparison between the previous monolithic system and a microkernel system..... 7
Figure 3 - Contiki-OS, an example of a layered architecture that became extremely popular [6] ........ 8
Figure 4 - Cooja running a simulation of 40+ nodes loaded with Contiki ........ 10
Figure 5 - TOSSIM Simulating nearly 30 motes and with its debug messages tab open [3]...... 11
Figure 6 - Communication stack - Link Layer ........ 13
Figure 7 - LTE vs other technologies, compared by speed........ 14
Figure 8 - A comparison between classic Bluetooth and Bluetooth Smart ........ 16
Figure 9 - The difference between ZigBee and 802.15.4 ........ 17
Figure 10 - Comparison between different technologies in terms of power usage ........ 18
Figure 11 - Communication stack - Network Layer ........ 19
Figure 12 - Mesh under and Route-over forwarding ........ 20
Figure 13 - Communication stack - Transport Layer ........ 21
Figure 14 - Machines trading messages by TCP, illustrating TCP's re-ordering feature ........ 22
Figure 15 - TCP header ........ 23
Figure 16 - Representation of UDP packet........ 24
Figure 17 - Communication stack - Application Layer........ 25
Figure 18 - Visual Representation of a CoAP message header [25] ........ 26
Figure 19 - Setup pages on Openfire, illustrating the use of external databases ........ 31
Figure 20 - Xively architecture ........ 35
Figure 21 - ThingsWorx architecture........ 36
Figure 22 - General Architecture........ 38
Figure 23 - WSN detailed view ........ 39
Figure 24 - Size comparison between a modern smartphone (left, iPhone 4s) and a Raspberry Pi (right, B+ model) ........ 40
Figure 25 - Energy consumption of various embedded boards. ........ 40
Figure 26 - Example of communication between the Linux Gateway and a WSN node ........ 41
Figure 27 - General Architecture, as proposed initially. ........ 43
Figure 28 - DAG netowrk captured in COOJA with radio environment ........ 48
Figure 29 - Openfire admin console, displaying information about the installation ........ 52
Figure 30 - The crud REST services running in the Linux Server ........ 53
Figure 31 - First idea of project to develop........ 61
Figure 32 - Early idea of sensor's data members and methods........ 62

# List of Tables

Table 1 - A comparison between the operating systems........................................................ 12

# Acrónimos e Símbolos

## Lista de Acrónimos

| | |
|---|---|
| **DAG** | Directed Acyclic Graph: Collection of vertices and directed edges with no loops. |
| **CoAP** | Constrained Application Protocol: Communication protocol. |
| **6LoWPAN** | IPv6 over Low power Wireless Personal Area Networks. |
| **IoT** | Internet of Things |
| **XMPP** | Extensible Messaging and Presence Protocol |
| **REST** | RESTful architecture |
| **QoS** | Quality of Service |
| **RPL** | Routing Protocol for Low power and Lossy Networks |
| **XML** | eXtensible Markup Language |
| **XEP** | XMPP Extension Protocols |
| **HTTP** | HyperText Transfer Protocol |

# 1 Introduction

Wireless Sensor Networks have an array of uses and have been applied to the most varied environments. There are home monitoring systems that often emphasize security and offer some kind of cost-reduction from existing services, there are also professional application WSNs at high-end establishments that count the number of visitors. Other applications such as agriculture-aimed or fire protection WSNs have been deployed and studied.

In the recent years development for this kind of distributed systems became increasingly popular given the all-time low prices of small sensors and processors. The hardware side of these systems scales well in terms of cost and make it easy to sell components for a good margin, therefore attracting both entrepreneurs and venture capitalists alike. This, aligned with the strong craze and popularity of tech start-ups gave way to many uncommon and one-of-a-kind use cases, i.e. full-fledged entertainment systems, upgrades for high-end homes or even an affordable monitoring system to aid the elderly and allow for quick action in case of a fall or other disaster.

The majority of applications are directed towards the common tech addict, someone who wants to be part of the technological advancement and is looking for entertainment. This is due to the fact that it is proven that selling technology nowadays is a particularly good way of making money, innovation for the sake of advancing the World has been neglected. Whether that's good, bad or irrelevant is debatable and not the purpose of this thesis. It does bring, however, the opportunity to develop a new tool based on different principles: Open source software, availability of information and modularity.

It was decided this software should also implement a new or uncommon functionality, as that is part of this thesis' requirements. Taking that into account it was ultimately decided to build a program that would put together the user-friendliness of mobile apps, the capabilities of a decentralized system, and the importance of critical messaging. The ultimate goal is for it to become a relevant tool that aids the monitoring of physicals spaces at a low cost using, whenever possible, readily-available components. One important contribution of this thesis is

the structured analysis of the several technology components and the proposal of an architecture that combines them.

## 1.1 Motivation

Partly because they are new and exciting and partly because research is expensive, there aren't many Wireless Network implementations that are open-source, or at least free. Even if someone wanted to install their own home-automation system and support the hardware costs they would have to have at least a deep understanding of low-level programming to achieve something functional. Frameworks aren't particularly popular yet and guides and architecture is still an open subject often discussed in technical papers.

Though the majority agrees that software engineering will play a big part in the development of these systems there are few specific tools for them. There are, however, programming tools that aid a lot of the development processes.

Examples of use cases of this tool could be:

- A biology research laboratory with plenty of cold chambers and delicate equipment that wants to make sure their results haven't been spoiled by a temperature variation or on the other hand, that want to seek environmental causes for the failure of an expected result.

- Monitoring of an aquarium containing algae that are very sensitive to temperature, to something as huge as monitoring the state of important equipment in an expedition.

The fact is that extensibility and modularity are key to anyone wanting to use these tools for various projects, and most wireless network software doesn't allow for those features, while this tool does.

## 1.2 Contribution

The following list give provides the reader with a summarized view of what the author has contributed to the subject, which and can be used by others in the following ways:

- **The gathering of an extensive state of the art, properly organized and documented in this report, described in chapters 2 and 3.**

- **The Implementation of a Wireless Sensor Network and Message Delivery System according to the technologies studied, with an analysis of a proper architecture for these projects.**

- **A demonstration on how a project of this kind would work.**

## 1.3 Method

The making of this thesis was composed of three main parts:

- A profound study of existing technologies, how they interact and which were more suitable for the use cases of the project. This included searching online for papers and articles on the subject of WSN's and instant messaging.
- A development phase, which went on for several months and where various different sets of technologies were considered and eventually discarded or accepted into the project. Some of the important extracts of code will be presented in chapter 5.
- A writing phase, where all the knowledge gained from the previous phases was condensed into a full thesis report. Here, not only the accepted technologies are studied, there is also discussion on other hypothesis if they are viable options when building a distributed system such as this. Some previous reports on the subject were studied to figure out which technologies and parts of the development process were usually studied and presented in these reports in order to make it more interesting.

All these phases spanned across the whole development of the project even though some were given more time and attention at the beginning, and others at the end. Still, even nearing the end of the project some small architectural decisions were made and development occurred. This stemmed from the fact that some new, better tools or versions came out after the part of the development that concerns them was made; the fact that as more and more knowledge of the subject was gained, some windows of opportunity of new ways of implementing the system appeared; and the fact that the writing had to take into account all of these changes.

The development phase took up much more time than planned. This ended up being a big system with several independent components running in several machines, so the amount of environments used to program the software escalated beyond expectation.

## 1.4 Thesis structure

This document is divided in six main chapters. Firstly it gives the reader an introduction on the subject and it follows with why it was decided to do such a project. It then presents some insight on what it does that's new and how it can be used and which use cases it covers.

Following that are two chapters dedicated fully to the state of the art. Since this thesis features a distributed system and covers two distinct sets of technologies, these chapters correspond to two main subjects:

- The Wireless Sensor Network state of the art, which has information on available IoT operating systems, their features and types of Operating System architectures. This is useful information since the architecture of the Operating system has much influence in the performance of the nodes. Some types of WSN's might benefit from one type, while others can benefit from another. There is also space in that subchapter to present information on the most common programming models, which suit some Operating Systems more than others. And finally a comparison is done between available OS's, which is a very short, graphical view of all that was talked about. In this same subchapter the whole IoT communication stack is presented. In each layer of the stack the available technologies are compared and explained.

- The Instant messaging state of the art, which refers to the technologies used outside the WSN. They cover everything from the receiving of data from the WSN until it is delivered to the Android app. There's a final subchapter that studies other implementations with existing services

# 2 WSN’s State of the art

This is an analysis to the state of the art of WSN’s and technologies used in constrained environments. That means every layer of communications was studied and alternatives of methods of communication are presented. In order to try and make this analysis more comprehensive and make sure the reader doesn’t get lost in the information and the chapters, an explanatory image is displayed in each of the subchapters that displays where the layer is situated in the stack.

## 2.1 Operating Systems

This subsection an analysis of the different Operating Systems used when building WSN’s. They take into account a very different core of features from desktop computers OS’s and are mostly focused on architecture, programming models, memory management, communication protocols support and resource usage.

Operating system models seem to influence the programming model of Operating Systems for WSN’s that use them. Some might fit better in sensor-only networks and others for architectures where heavier data transmission is needed. The following study contains information on these properties and its purpose is to help decipher which type fits best for a certain architecture.

### 2.1.1 Operating Systems Architecture

The fact that architecture is considered stems from the importance of balancing the quality of the OS with its size. Because modularity and layering may require additional interfaces it also increases the size of the software and therefore the cost of the hardware.

**Monolithic Architecture**

A monolithic architecture is basic, and was the starting point when developing many systems. DOS for example had a monolithic architecture, so did BSD.

This kind of design allows for smaller OS’s but lacks in other fields. Monolithic OS’s are hard to understand and modify and that might slow down the development and release of new versions. [1]

All things considered the most obvious reason to consider this kind of architecture would be the small OS memory usage. To bet every bit of success of the OS in this feature could be a big

risk, so other types of OS for WSNs also have their place in the scene, even if they have to sacrifice a bit of memory size. [2]

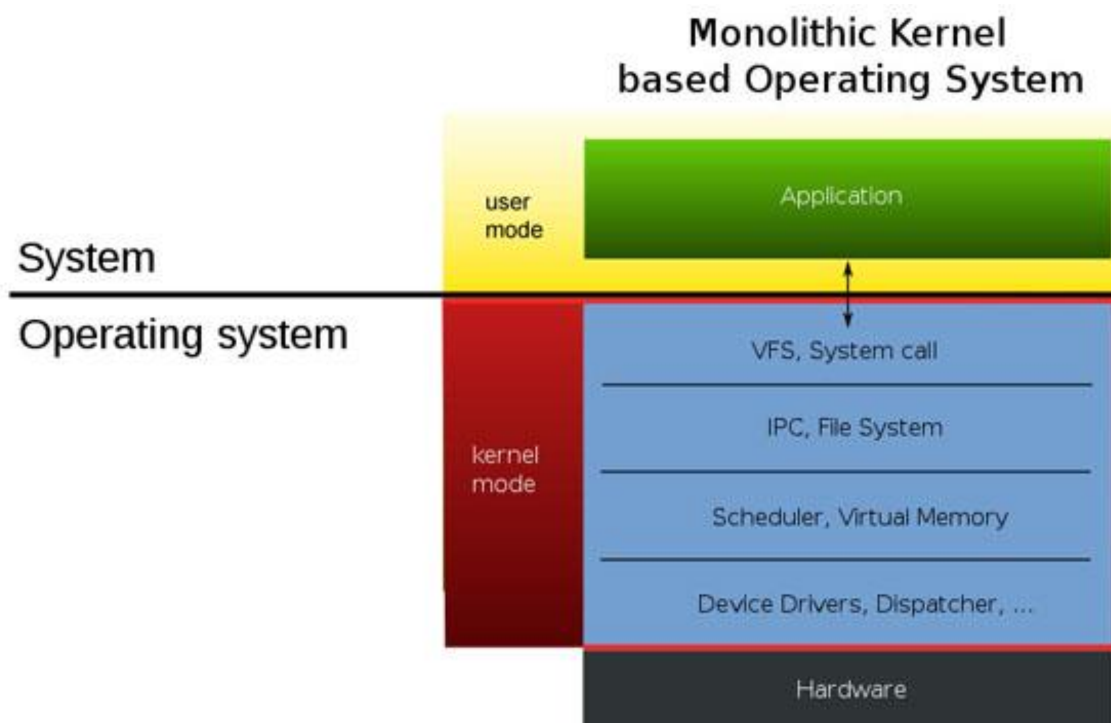


Figure 1 - Visual representation of a Monolithic kernel [3]

As represented above every service of the systems runs in the same kernel thread, meaning they all occupy the same memory area.

Even though is allows for powerful hardware access the fact that it's not modular represents a great danger: a small bug on a tiny part of the OS might very well crash the entire system. It's still a very popular architecture but it might not be good choice for rapidly evolving, complicated systems.

**Microkernel Architecture**

Microkernel architectures contradict mostly every principle of monolithic ones, they focus on implementing minimum functionality inside the kernel.

Since the rest of the functionality is usually provided in user-level servers, which are often independent, you can expand or contract a version of the OS to deploy it in various types of WSN nodes.

When (and this is might very well be the case in the future) someone figures out the very small WSN OS their company built needs another server for bigger and more powerful motes

they can be added with much less effort than in monolithic systems and with much less dependency.

Theoretically, and without considering anything else, microkernels might seem useless since a microkernel implementation provides poor performance when kernel calls are made. However, it's important to remember these calls are much reduced compared to a full desktop system. The fact that this allows for a very tiny kernel core is a great plus. [4]

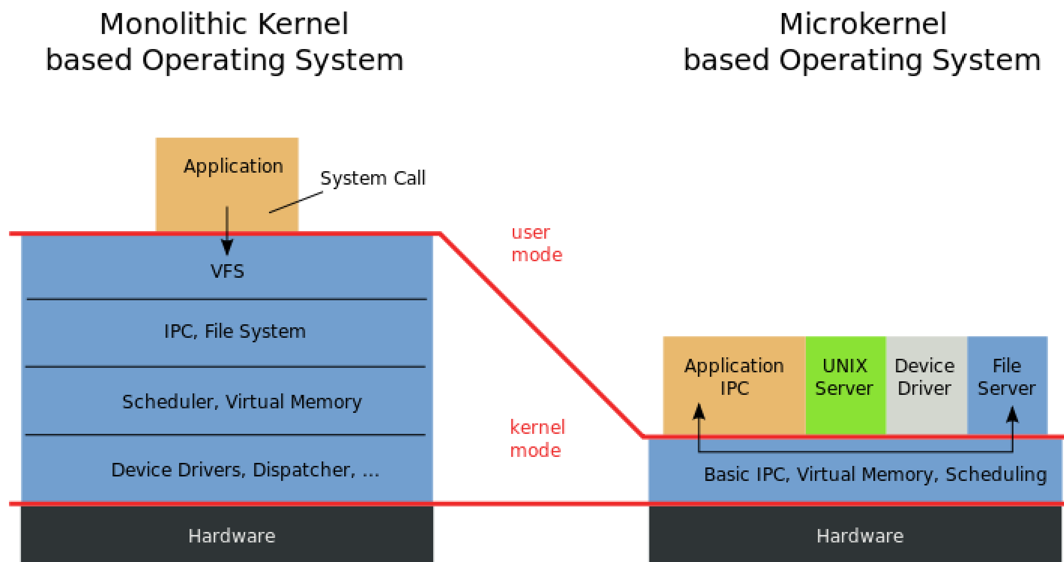


Figure 2 - Comparison between the previous monolithic system and a microkernel system

Though there aren't many WSN systems following this structure the fact that hardware is becoming cheaper and cheaper represents a great opportunity for these OS's. There is the opportunity to tailor the size, complexity and functionality of the OS and adjust to what current popular sensors are capable of doing.

While development of monolithic OS's will become harder and more expensive microkernel ones will keep their costs and complexity more or less the same.

**Layered architecture**

Layered architectures aren't about performance, they're more about reliability and easiness to understand.

OS's built in this architecture are usually not ideal if the goal is the fastest, smallest application possible but since many researchers and programmer hobbyists are new to the subject they present the only reliable alternative.

These WSNs use different technologies than what contemporary programmers are used to. Popular programming skills nowadays are typically high level languages, frameworks and programming standards, which have replaced low level programming.

From a purist view a layered architecture isn't at all ideal for WSNs, but when you consider the time and cost of learning a big array of new technologies for just a small project the idea makes much more sense.

Since our goal isn't speed or size and it's a more abstract concept of critical messaging, this architecture is very attractive and is the architecture of the OS that was ultimately chosen for development: Contiki OS [5].

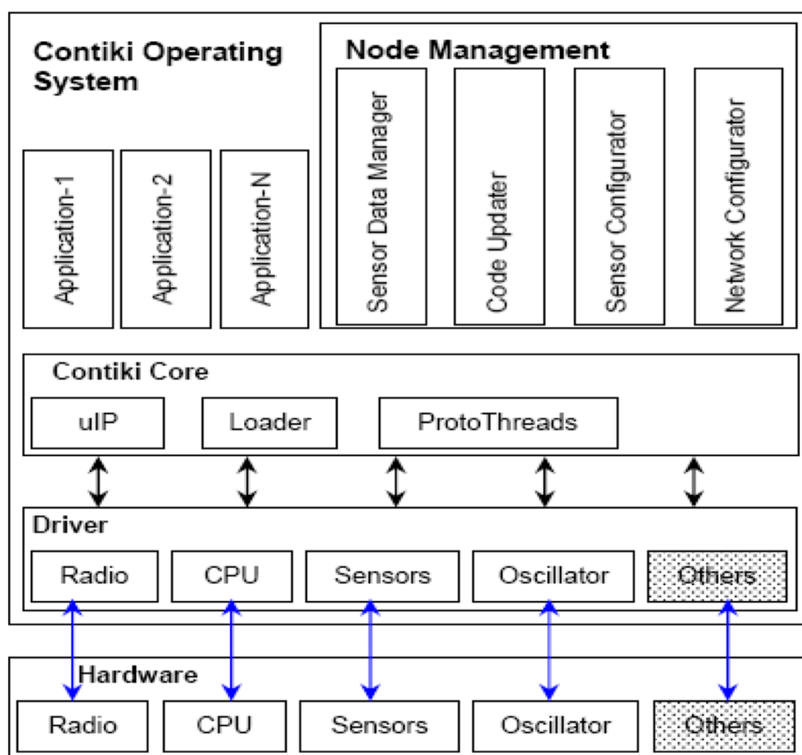


Figure 3 - Contiki-OS, an example of a layered architecture that became extremely popular [6]

## 2.2 Programming models in small Operating Systems

Two paradigms rule this aspect of the systems: Event-driven and multithreaded programming. The choice of using one, another or both will impact greatly the development of the application.

### 2.2.1 Event-driven programming

This type of programming isn't very convenient for who is programming but it's less resource-intensive. [7] The fact that it wasn't well accepted led to the development of alternate versions of threads, more pleasing to programmers. In here, the flow of the program is determined by events. In a sensor network, these can be button presses or even the arrival of a communication message.

This is, however, implemented in one of the most popular Operating Systems for the Internet of Things: TinyOs [8]. It's also the main barrier to overcome when deciding whether to develop for it.

### 2.2.2 Thread-driven programming

Multithreading is actually very widely spread and well accepted by programmers. The problem with using this in WSNs is that it's very resource intensive and therefore not at all suited for devices with very poor memory and processing resources.

Contiki-OS has adopted this paradigm with a catch: an extremely lightweight version of a thread, called a protothread. In reality, a protothread is event-driven but has the convenient threaded programming style. Their main features are their single stack and a very low overhead. Once again we can see that convenience and easiness are very well favoured in these systems nowadays. [9]

The difference between these two paradigms is easily explained with an example, whenever a process is requested to do something:

- An event-driven runtime will dispatch the event, which will be eventually picked up by a handler.
- A thread-driven runtime will create a new thread to deal with the event.

As a result threads are very independent from one another, if a thread it stuck with an intensive task it won't affect the others a lot. If an event is stuck with a handler it might not be able to do anything else in the meantime.

## 2.3 Available Operating Systems alternatives

There are a couple of very popular alternatives as far as Operating Systems go. Considering the fact that this kind of development is hard, only popular active and documented systems were considered. The two main candidates were Contiki-OS and TinyOs, two very different systems.

### 2.3.1 Contiki-OS

This 12 years old system was made specifically for the Internet of Things. It's an open source, lightweight and modular system. It runs under a BSD license and is currently in version 2.7, undergoing development for versions 3.x. It has consistently been used in many IoT applications, particularly monitoring projects. It's designed specifically to deal with memory constrained and power-starved devices. [5]

Contiki only needs about 10KB of RAM and 30KB of ROM, so even though it's a system with a layered architecture it will fit it mostly any device. If a GUI is needed the system can be expanded and will still only take 30KB of RAM. With these values it's easy to see why it has been so well accepted. It's slower bigger than some competitors but not so much that it becomes unusable.

Contiki is distributed in Ubuntu with some examples, documentation, code, compilers and a simulator called COOJA where a network of Contiki systems can be built and run, aiding the development of applications and allowing developers to test the limits of their systems without investing a lot of money in hardware. [10]

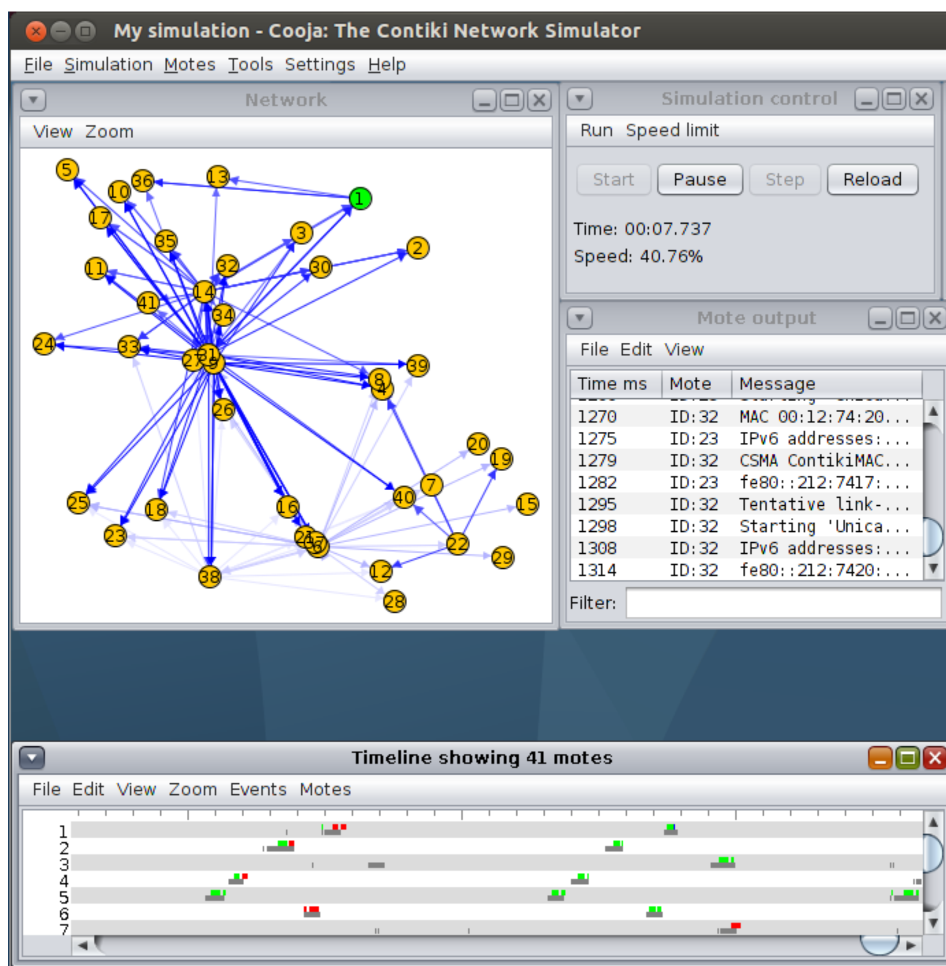


Figure 4 - Cooja running a simulation of 40+ nodes loaded with Contiki

Cooja seems to perform its duties well and is a great ally to any developer working with Contiki.

### 2.3.2 TinyOs

TinyOS has been released three years prior to Contiki and presents a monolithic architecture. It's also under a BSD License. Programming is done in the nesC language. One of the most well received features about TinyOs is that it's completely non-blocking. I/O either last microseconds or are done asynchronously and have a callback. [11]

It boasts a complete 6Lowpan/RPL IPv6 stack that is on par with Contiki's and there is documentation widely available. It can be installed in either Linux, Mac OS X or Windows machines and some virtual machines with it pre-installed are also distributed. [8]

Simulations is done primarily in TOSSIM. TOSSIM is a library that is analog to Contiki's Cooja and can also simulate entire TinyOS applications. Debugging is TOSSIM is much more powerful than in Contiki's Cooja, especially with its Python interface

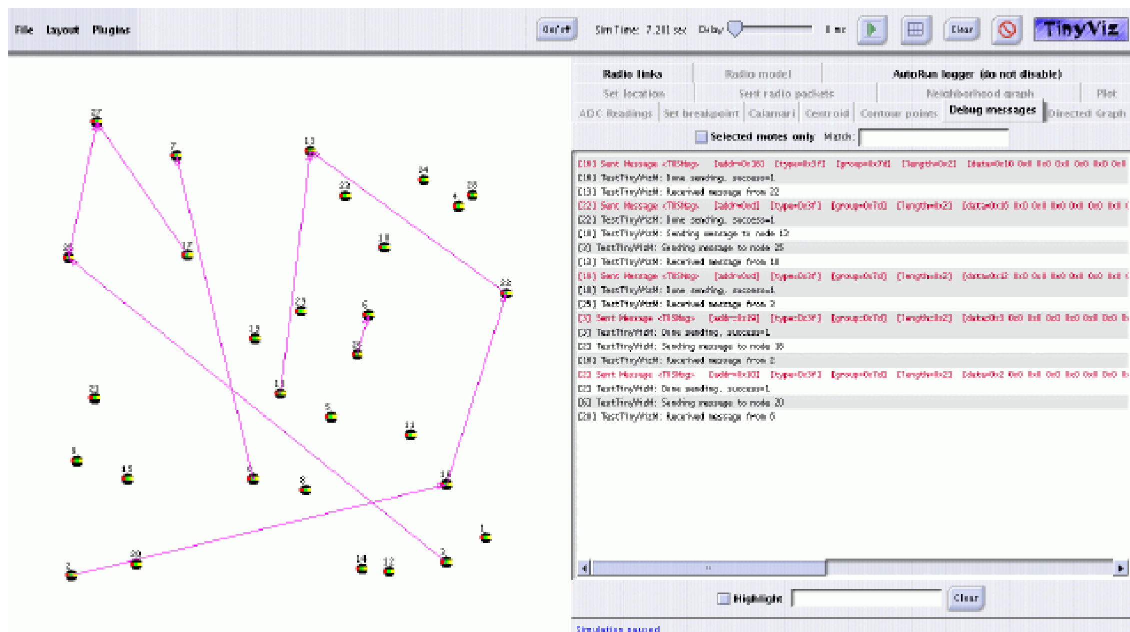


Figure 5 - TOSSIM Simulating nearly 30 motes and with its debug messages tab open [12]

### 2.3.3 Comparison and decision

Both systems have weaknesses and strengths. Although TinyOS includes powerful development tools, that doesn't translate into more quality in the end application. [13]

There's a lot of documentation about the OS and its tools but the technologies used are much less known and programming bigger systems in TinyOS has its obstacles. This was the main reason why Contiki was chosen.

Since the developed WSN was small project Contiki's familiar style and language and the fact that it follows the latest programming standards were deemed more important than a powerful debugging tool and small size. Furthermore, Contiki's weaknesses weren't a problem in this specific project. Although bigger than TinyOs, the compiled code will fit in the same nodes and even though the project is more mature, help and examples for Contiki are widely available.

Table 1 - A comparison between the operating systems.

| | **TinyOs** | **ContikiOs** |
|---|---|---|
| **Linking** | Static | Dynamic |
| Base OS memory usage | 400 bytes | 40kb |
| Programming language | NesC | c |
| Programming model | Event-driven | Hybrid (Based on protothreads) |
| License | BSD | BSD |
| Thread support | Partial(TinyThreads) | Full thread support |
| Simulation | Supported | Supported |
| Release Date | 2000 | 2003 |

## 2.4 IoT communication stack and related standards

Low-power networks have plenty limitations and require a different set of standards from normal applications. IEFT and OASIS standardized some protocols which have been adopted by nearly every OS designed for the IoT. [14]

In the following subchapters available technologies will be presented along with an analysis on their relevance for this project. Technologies will only be compared to others in their communication stack level and levels will be presented from the physical layer to application layer. In the beginning of each subchapter an image will guide readers, with layers being discussed signaled in blue, while the rest of the stack is painted in orange.

### 2.4.1 Link Layer

The link layer is comprised of the Physical and MAC layers. Only the relevant technologies will be discussed here. There are dozens of available protocols that were not adopted by some reason or another, and this report focus solely in viable alternatives to implement a sensor network.

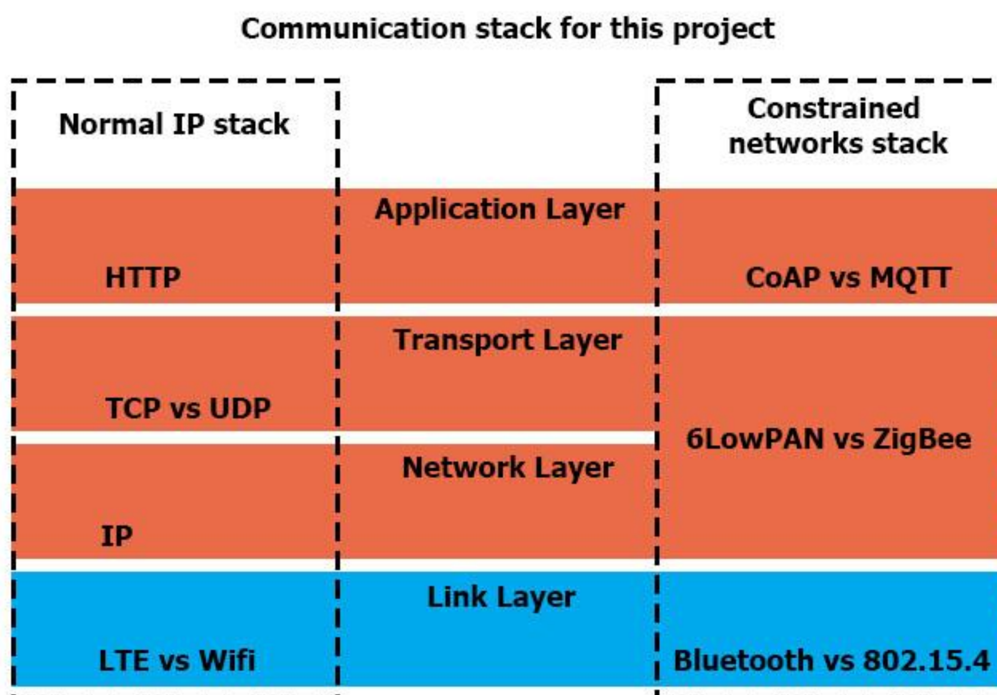


Figure 6 - Communication stack - Link Layer

**LTE 4G**

LTE, commonly marketed as 4G LTE, is a standard for wireless communication. It is currently marketed towards mobile phones and laptops but it could potentially be used in smaller, less

powerful devices. LTE 4G main attraction is its coverage, so it could it theory be used in a mobile sensor network, such as in a car or a plane. It could also have a place in a sensor system to aid the elderly.

A use case could be when an elderly person goes out for a walk, he or she could take a myriad of sensors attached to them, such as a smartwatch, a cell phone, or small sensor modules to monitor vital signs. In this case, LTE might be the best available option for the sensor nodes to communicate an alert message since other types of networks are not available everywhere, whereas LTE normally has a nation-wide coverage in developed countries.

Another example could be a can or commercial plane being equipped with all kinds of sensors for increased security. Imagine a car is parked and the owner is away, if the car is broken into what usually happens is that the alarm goes off for a while, which does little to increase the chances of recovering the vehicle should it be also stolen. Besides, alarm systems can be overridden and silenced quickly, or may not be functioning well at all.

A car equipped with a movement sensor and a LTE 4G antenna could gather information from its GPS to alert the owner that something is wrong since Wi-Fi or other technologies might not be available. The main problem with 4G LTE is its cost. The service to connect to other devices with this technology is usually sold by telecommunications companies.

In a 2012 research study, the World's average price per 4G/LTE GB transfer is around 4.86 dollars. In Europe this number is lower, but that is not true for every developed country. In the United States of America for example, Verizon charged 7.50 dollars per GB. The price is not exorbitant for small sensor networks that trade very little data. But it's an extra cost that can be avoided by using other technologies. [15]

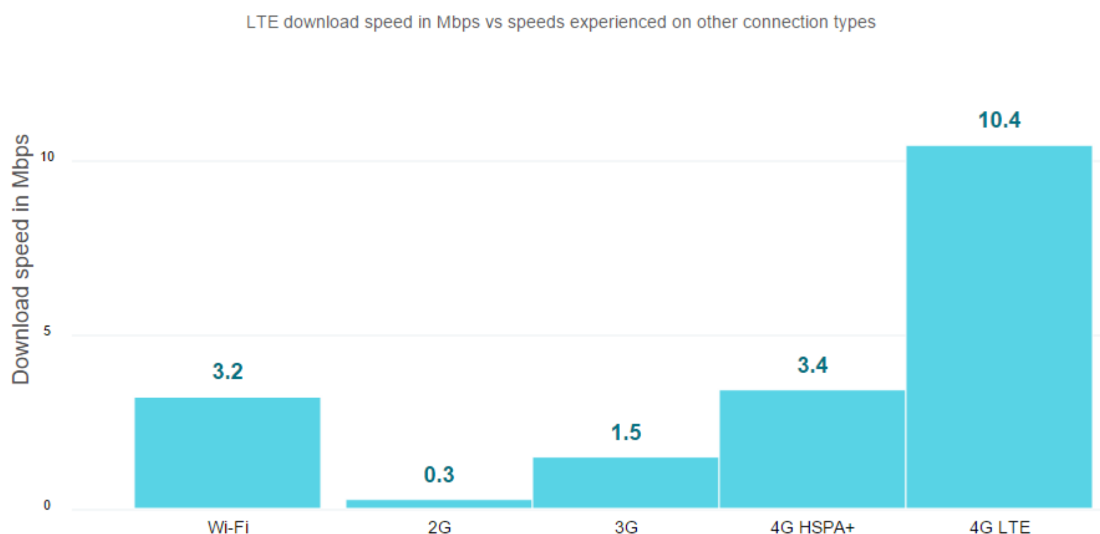


Figure 7 - LTE vs other technologies, compared by speed

4G LTE boasts higher speeds than home broadband services [16], but while such data rate might be suitable for entertainment systems, it would hardly have any use in a sensor network that trades small amounts of data. Besides, images like the one below provide distorted results that aren't always achieved in the real world. LTE coverage is stronger in some places and weaker in others, leading to varying speed in different places.

**Bluetooth Smart ( 802.15.1 )**

Bluetooth Low Energy, or Bluetooth Smart, as it is commonly marketed, is a wireless personal area network technology. Its main attraction is the amount of energy it uses. Compared to classic Bluetooth, one gets nearly the same communications area with considerably less energy consumption. This technology is marketed at healthcare, fitness, security and entertainment devices. The fact that nearly every popular computer and OS natively supports Bluetooth 4.0 makes it a very attractive choice for a communication protocol.

There are some drawbacks to this technology. Older devices won't be compatible, even if they already supported the classic Bluetooth. Since Bluetooth Smart is not backwards compatible every device in the network has to support version 4.0 of Bluetooth specifically in order for communication to work. [17]

Below is a comparison of classic and Smart Bluetooth. Notice how Bluetooth Smart sacrifices some of its functionality in order to leech less energy, while still improving on some areas.

| Technical Specification | Classic Bluetooth technology | Bluetooth low energy technology |
|---|---|---|
| Radio Frequency | 2.4 Ghz | 2.4 Ghz |
| Distance/Range | 30 meters | 50 meters |
| Over the air data rate | 1–3 Mbit/s | 1 Mbit/s |
| Application throughput | 0.7–2.1 Mbit/s | 0.2 Mbit/s |
| Active slaves | 7-16,777,184 | Unlimited |
| Security | 64/128-bit and application layer user defined | 128-bit AES and application layer user defined |
| Robustness | Adaptive fast frequency hopping, FEC, fast ACK | |
| Adaptive fast frequency hopping | | |
| Latency (from a non-connected state) | Typically 100 ms | 6 ms |
| Total time to send data | 100 ms | <6 ms |
| Government Regulation | Worldwide | Worldwide |
| Certification Body | Bluetooth SIG | Bluetooth SIG |
| Voice capable | Yes | Yes, with some limitation |
| Network topology | Scatternet | |
| Star-bus | | |
| Power consumption | 1 as the reference | 0.01 to 0.5 (depending on use case) |
| Peak current consumption | <30 mA | <15 mA |
| Service discovery | Yes | Yes |
| Profile concept | Yes | Yes |
| Primary use cases | Mobile phones, gaming, headsets, stereo audio streaming, automotive, PCs etc. | Mobile phones, gaming, PCs, watches, sports and fitness, healthcare, security & proximity, automotive, home electronics, automation, Industrial, etc. |

Figure 8 - A comparison between classic Bluetooth and Bluetooth Smart

Its main features are:

- A very easy way of implementation
- A very low power consumption in both peak, average and idle modes
- The components (i.e. antennas, chips, etc.) are of very small size and cost
- Range up to 15 meters
- Bitrate of 1Mbps

This last feature makes Bluetooth Smart a weak choice for heavier content such as high quality video streaming, but is not a real obstacle for the use cases of this project.

In conclusion, Bluetooth is a viable alternative for communicating across IoT devices, including the ones on this project.

**Wi-Fi ( 802.11g )**

Despite being the most popular and widespread of all the technologies, Wi-Fi has not been thriving in the low power devices of the IoT because this technology requires very high amounts of power to operate, and therefore drains battery life quickly. The main feature Wi-Fi offers that other technologies don't is a very high data rate and communication distance, which can go up to 54 mbps and 300 meters, respectively.

Wi-Fi is still viable for some devices, namely any device that streams audio, video or caps the use of other technologies data rate and distance. As long as an IoT device is not battery powered, i.e. has some kind of external source of energy like a power outlet, power consumption is not a problem, and the network is not going to scale in terms of number of these devices Wi-Fi is still a viable alternative, hence why it was analysed in this subchapter.

**IEEE 802.15.4**

As a brief note, the reader should understand that this IEEE standard is sometimes wrongfully named ZigBee. While ZigBee and 802.15.4 are often used together [18], 802.15.4 is actually just a standard for the link layer, while ZigBee operates on top of 802.15.4, on the network, transport and application layers as the below image suggests. ZigBee will be studied further down this document.

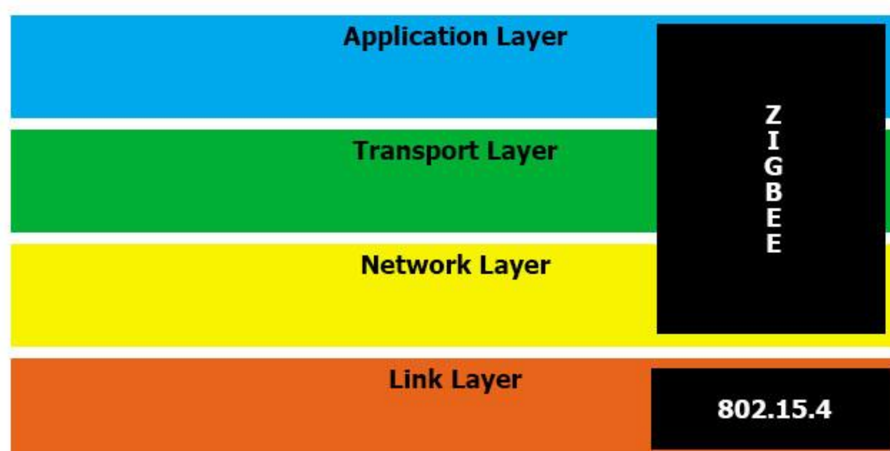


Figure 9 - The difference between ZigBee and 802.15.4

This standard is was built with low data rate networks in mind, in order to provide a low power alternative to other established standards. The emphasis really is on very low cost communication, making this the perfect standard for a sensor network. [19]

The 802.15.4 protocol initially supported transfer rates of only 20-40 kbit/s, but a new 250 kbit/s was added in the current version. When put together with the technologies in the upper levels of the stack, 802.15.4 consumes the following amount power compared to Bluetooth Low Energy and Wi-Fi:

Low Power Technologies

| Technologies | ZigBee/ 6LoWPAN (over 802.15.4) | Bluetooth Low Energy | WiFi |
|---|---|---|---|
| IEEE spec | 802.15.4 | 802.15.1 | 802.11 a/b/g |
| Frequency Band | 868/915 MHz; 2.4 GHz | 2.4 GHz | 2.4 GHz; 5 GHz |
| Nominal Range | 10-100 m | 10 m | 100 m |
| Chipset | cc2531 | cc2540 | cx53111 |
| RX | 25 mA | 19.6 mA | 219 mA |
| TX | 34 mA | 31.6 mA | 215 mA |

Figure 10 - Comparison between different technologies in terms of power usage

### 2.4.2 Network Layer

The network layer is responsible for packet forwarding, including routing. Protocols commonly used outside IoT networks include IPv4, IPv6, ICMP, IGMP, IPsec and others.

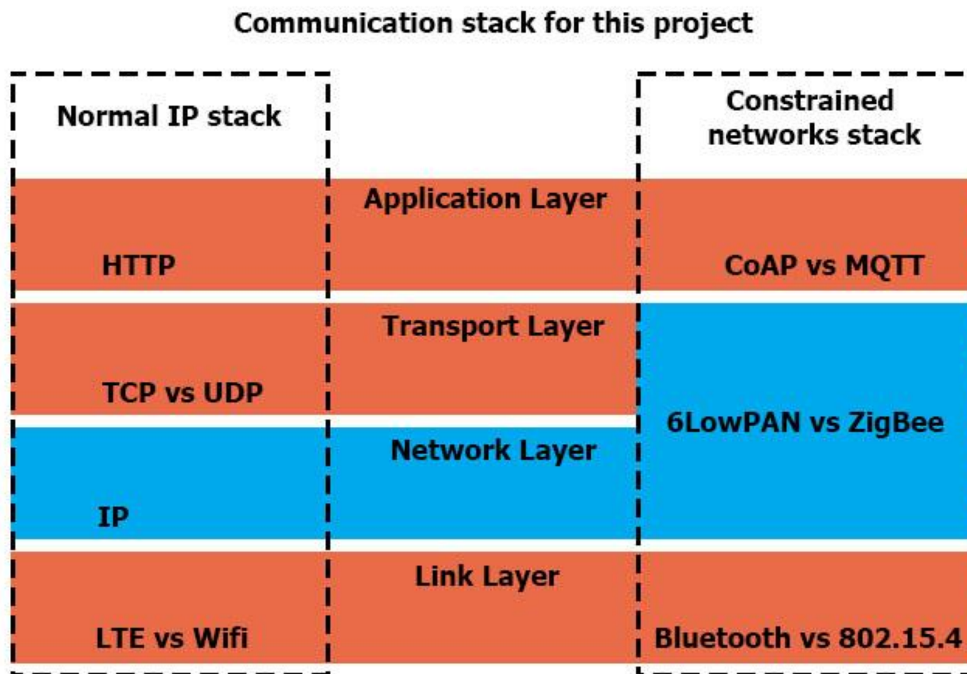


Figure 11 - Communication stack - Network Layer

**6LowPAN**

6LowPAN is an acronym for IPv6 over Low power Personal Area Networks. Its main attraction is the use of IP, making communication between IoT nodes simple and familiar.

It allows, through header compression and encapsulation, that IPv6 packets are sent over IEEE 802.15.4 networks. It's a competitor of ZigBee, which it is often compared to. It is widely accepted and every relevant player's chips can be used for 6LowPAN. [20]

Security has been given much attention in this standard since applications for the IoT often include valuable data like a way to open a door and to trigger or turn off alarms. It takes advantage of AES-128 link layer security, which is defined in IEEE 802.15.4. It provides link authentication, transport layer security (which runs over TCP, and requires the hardware to possess an encryption engine) and encryption. Since some IoT systems are very resource starved and might need to use UDP in favor of TCP, 6LowPAN also supports the RFC 6347 (datagram transport layer security) can be used to provide security at the transport layer.

To transmit packages 6LowPAN makes use of two different types of routing which are dependent on where the routing mechanism is defined. When the source and destination nodes are more than one hop away Route-over forwarding makes packets travel over the network layer whereas Mesh-under forwarding makes packets travel in the data-link layer. [21]

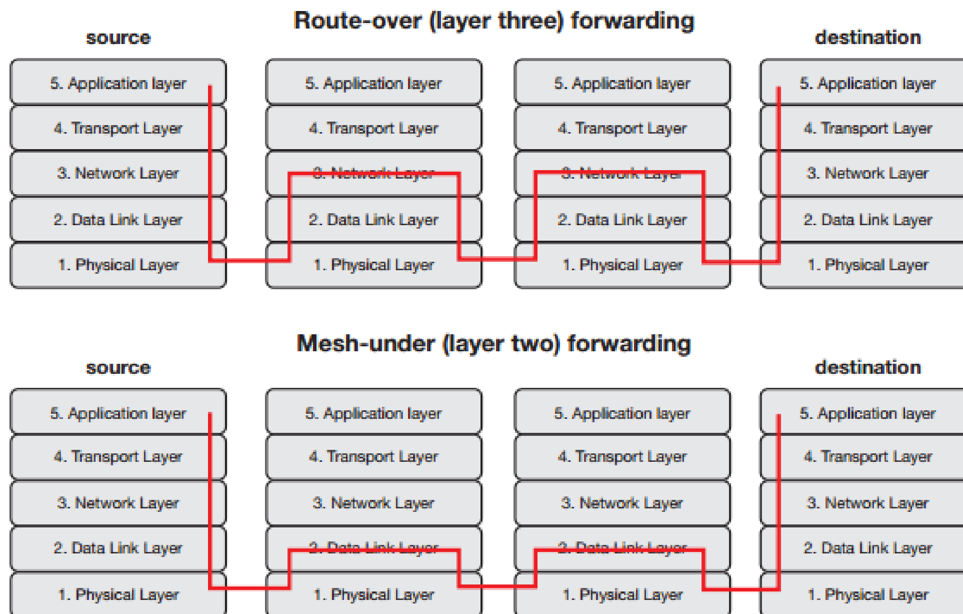


Figure 12 - Mesh under and Route-over forwarding

**ZigBee**

Unlike 6LoWPAN, ZigBee doesn't allow the option to apply ipv6 on 802.15.4 based nodes. It does however, allow for full interoperability between ZigBee devices from different manufacturers.

While 6LoWPAN requires extensive knowledge of IPv6 to be well implemented and therefore an extensive development time, ZigBee allows for significantly reduced development time by already implemented functions such as over-the-air firmware updates, channel agility, low-power mode, and the usage of safety certificates as well as different pre-assembled application profiles. [22]

### 2.4.3 Transport Layer

In this subchapter we will be focusing on the two most popular transport layer protocols, inside and outside the IoT: UDP and TCP.

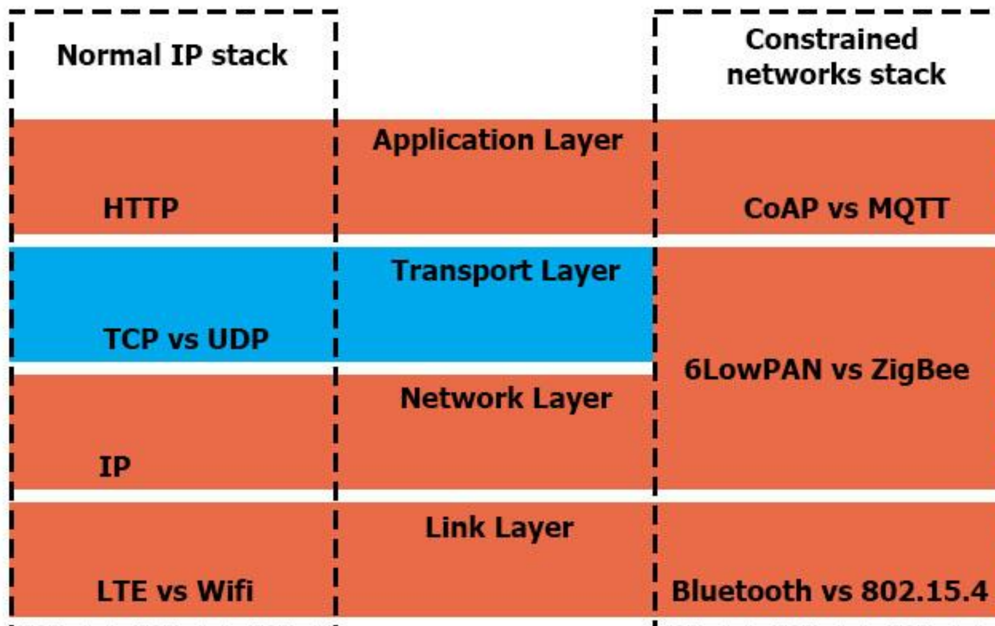


Figure 13 - Communication stack - Transport Layer

**TCP**

This protocol is currently the most used protocol in the Internet. TCP has many features including error correction, guaranteed delivery, recovery from congestion, etc.

TCP works by establishing a connection between the sender and the receiver in a 3-way handshake model. A 3-way handshake works as follows:

1. MachineA sends a SYN packet to MachineB.
2. MachineB receives the SYN packet, responds with a SYN-ACK packet
3. MachineA receives the SYN-ACK packet, responds with an ACK packet
4. MachineB receives the ACK packet. A connection is now established.

SYN and ACK messages are indicated by a either the SYN bit, or the ACK bit inside the TCP header, and the SYN-ACK message has both the SYN and the ACK bits turned on (set to 1) in the TCP header.

This model presents an opportunity for an attack. A malicious MachineA could not respond with the ACK packet in step 3, and instead send successive SYN packets as shown in step 1. This would allow for a type of attack called SYN Flood, which is a denial-of-service attack. The

main goal of this attack is to consume MachineB's resources to the point where it can no longer provide a connection to other legitimate machines.

After a connection has been successfully established, MachineA can now begin to send data packets to MachineB. MachineB will respond with an ACK packet after receiving them, but that doesn't mean the sequence of messages traded is always Data->ACK->DATA->ACK and so on. Since the machines have an established connection, and TCP does re-order packets if need be, MachineA can send multiple data packets even before receiving the ACK for the first. If a data packet is received by MachineB out of order it will respond with an ACK of the last received packet in order, so that MachineA will know what to re-transmit. The following image illustrates this point exactly.

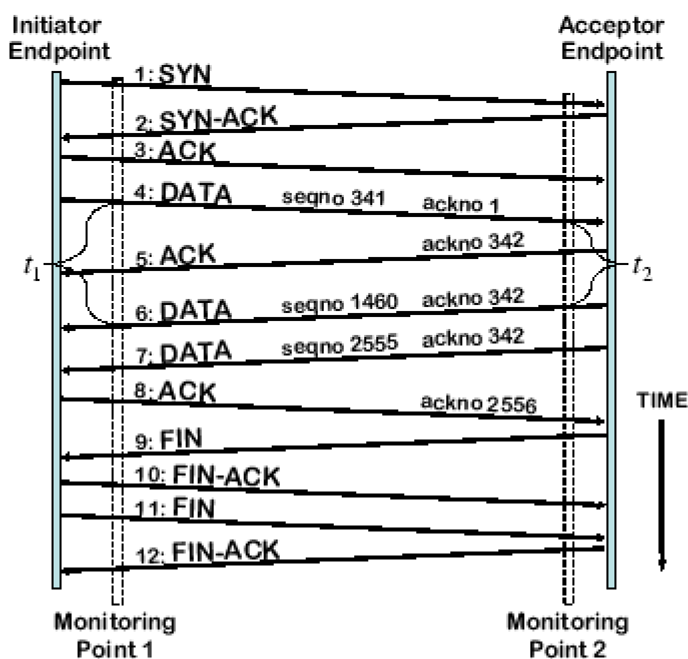


Figure 14 - Machines trading messages by TCP, illustrating TCP's re-ordering feature

As pointed out in the image above, the connection closing protocol also includes the trading of messages between the machines.

TCP has other features which might be interesting for this project, the most relevant of which might be its sliding window. This is a protocol that ensures TCP messages are not being sent too fast for the receiver to capture. This is another way to ensure the receiver can process incoming messages reliably.

The way sliding window works is by specifying a receive window size in the side of the receiver. This represents the number of bytes the receiver can buffer. When the sender fills the window size it stops sending messages and starts a persist timer. This is used to protect TCP from deadlocking, in case the update for the next window size never arrives for some reason, and the sender is locked in a waiting state, unable to send data.

When the receiver updates the sender with a message window that is not 0, the sender will start to send messages again. This can lead to some problems, however, especially in IoT devices with little memory. As an example imagine the receiver is only able to process data in very small increments.

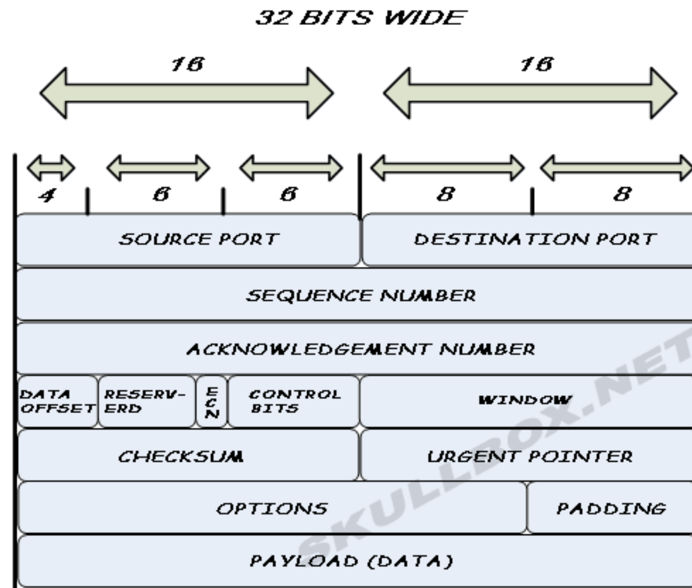


Figure 15 - TCP header

The advertised window will always be close to 0, so the sender is sending lots of packets with very little data, creating a huge overhead for a small message. This is commonly referred to as the silly window syndrome.

**UDP**

UDP has none of the features presented in the TCP chapter. It's unreliable, has no flow control, doesn't allow for delivery assurance and it's connectionless.

Still, many if not all IoT Operating Systems implement a stack that supports UDP. There are white papers defending UDP usage in the IoT and articles in reference web pages list UDP more often than TCP for IoT stacks and architectures. Why?

To answer that question lets first look at common cases where UDP is a viable solution. These include:

- Streaming audio
- Streaming video

- Continuous weather data
- Continuous stock market data feeds

These use cases have 3 things in common: Speed, the ability to handle unreliable data and the need for low overhead.

Below is a graphical representation of a UDP packet, compare it to the TCP packet above.

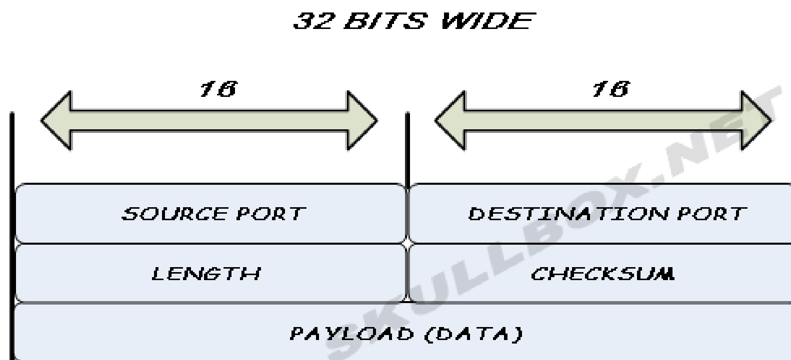


Figure 16 - Representation of UDP packet

Notice how TCP uses a minimum size of 224 bytes to send one message. Notice as well how UDP only uses a fragment of that, 96 bytes, to send the same message.

Although computers have evolved and became cheaper and more powerful, programming for the IoT can't take the current capabilities of computers and develop for them. In networks where the number of hardware devices can increase by a factor of 10 or 100, programming has to take into account the hardware restrictions of these devices.

UDP is faster because it has very little features, and very little overhead. UDP is not concerned whether the receiving end is able to save every bit of information effectively. UDP is the technology to go for if speed is more important than reliability.

In a sensor network, how serious is it that one sensor out of 100 fails a message that it is sending every 5 seconds anyway? If the new data is coming in and is going to replace the previous one, then missing one message once in a while has no real impact in the quality of information flowing through the system.

### 2.4.4 Application Layer

In this layer it might be necessary to import libraries that are foreign to the Operating System we are using. There are several different technologies to choose from, this analysis will focus

on only three of those, which are the most common in IoT systems. They are HTTP, MQTT and CoAP.

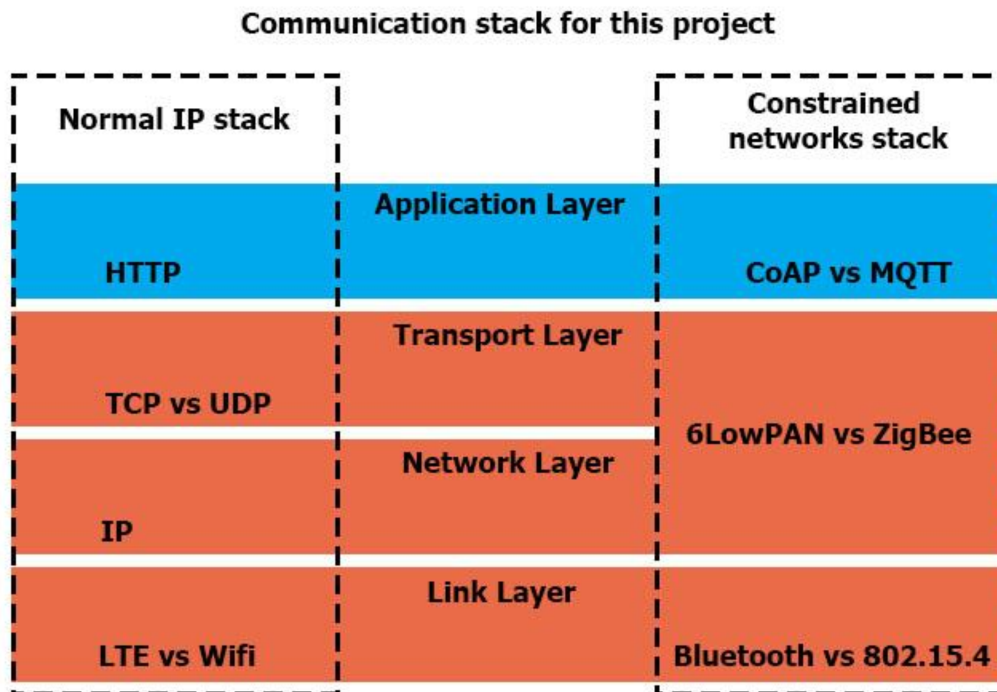


Figure 17 - Communication stack - Application Layer

**CoAP**

CoAP is short for Constrained Application Protocol and was designed for very simple hardware components to communicate over the Internet [23]. Its description is in line with what we want to achieve in our project and the technology targets specifically low power nodes. It's easily translatable to HTTP and supports multicast communication but still maintains a very low overhead, which is mighty important in a WSN since it decreases bandwidth usage. [24]

Its message formats are divided into requests and responses and it is bound by default to UDP. Below is a visual representation of a CoAP header. As you can see, it uses only 4 mandatory bytes of header information if no other options or tokens are chosen, making it extremely lightweight compared to HTTP, which uses no less than 16 bytes (or 18 bytes for HTTP 1.0 and 1.1 which expects headers and therefore an empty line to signify their end).

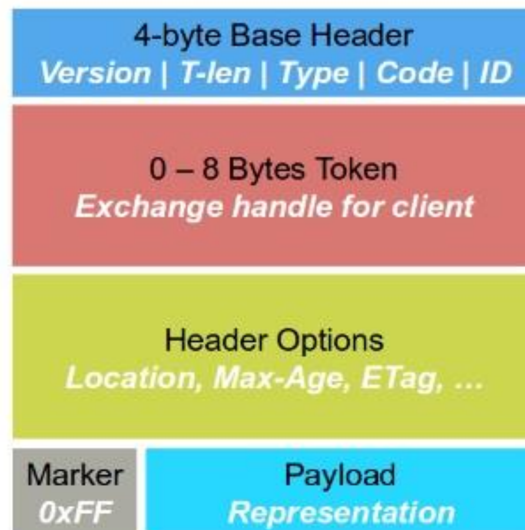


Figure 18 - Visual Representation of a CoAP message header [25]

Low overhead is not CoAP's only strength. This protocol allows for the discovery of resources provided by known CoAP services. This is incredibly useful since it allows us to connect to our network via REST services which potentially can become easily available to other devices in the future and not only to our Linux gateway.

CoAP features something called resources. These are available routines in a program than can be discoverable by querying a conventional resource path. These can even be subscribed to, which result in push notifications.

**HTTP**

HTTP is the most commonly used commonly used application protocol today. It's even considered to be the foundation for communication in the World Wide Web. It's a much more complete and powerful technology than CoAP but it takes up much more ROM and RAM space and requires powerful WSN nodes to operate in.

Its main features include, amongst others [26]:

- Authentication
- Statelessness

- Status codes
- Persistent connections
- Encrypted connections (HTTPS)

Regardless, its operation specifications are compatible with our system, there are compiled versions of http clients and servers for Contiki and ways to take advantage of its architecture. As electronics evolve the cost difference between a 512 KB ROM node and an 8 MB Rom one may become irrelevant but for now HTTP, when compiled, takes up so much space that other lighter, swifter communication protocols become relevant. ´

**MQTT**

MQTT is advertised as a lightweight messaging protocol for small sensors and mobile devices. According to mqtt.org it is optimized for high-latency and unreliable networks and is implemented in a publish/subscribe architecture. [27]

MQTT is not just a good fit for the IoT, it was built exactly for the IoT. It is not a standard yet, but as of March, 2013 it is undergoing standardisation at OASIS. It supports security in the form of username/password fields in a MQTT packet. Encryption across the network can be handled with SSL.

The notion of publisher and subscriber in MQTT can be explained as follows:

1. A MachineX publishes a message that contains a topic A.
2. A MachineY subscribes to messages with topic A.

A message server, also known as a broker will then match publications with subscriptions, meaning it will deliver the message published by MachineX to every machine that subscribes to topic A, MachineY included. This kind of architecture is very efficient in *One to Many* messaging, which does not match this project's decided architecture, presented in chapter 4 of this document. [28]

Nevertheless, MQTT is a very useful protocol in constrained networks. It has a smallest possible package size of two bytes and is compressed into bit-wise headers, having also variable length fields. MQTT also features some interesting connection properties such as supporting always-connected and sometimes-connected models, session awareness (which includes configurable keep-alives and last will and testament which enables applications to know when a client goes offline abnormally).

Its networks are usually TCP-based, implement web sockets and QoS is also present in MQTT, and allows the sender to know whether the message was delivered at most once (Code 0), delivered at least once (Code 1), or delivered exactly once (Code 2)

# 3 Message delivery state of the art

This chapter provides a description of technologies used to deliver the data of the WSN to different entities. There are several ways to implement this type of distribution of data but as everything else in this project.

The idea behind this part of the system is that there is a pair of machines trading real-time messages in both ways, which is comparable to a chat between humans except for machines deciding who to talk to and what to say. Therefore, a message trading (or chatting) architecture is suitable for this project.

There are several message trading protocols available. Their components usually include a server and several clients which trade messages. This analysis will focus some of the most used around the World and what software they're usually used with. It will also include an analysis on alternative message delivery services or architectures.

## 3.1 XMPP

XMPP is short for Extensible Messaging and Presence Protocol and it's a communications protocol. It's based on XML and enables real-time message trading between two agents. Being based on XML means that the traded information is extensible and structured. This brings some problems of course, an XML structure is very heavy to handle and retrieving a high number of offline messages in a slow network can take a while.

If this system tried to deliver every sensor message it receives to a client over the internet the initial fetch of offline messages would take an unreasonable amount of time, making XMPP unviable for this project. However, since the goal is to real-time talk between sensors and an Android app and to store only critical offline messages, XMPP will be a viable protocol.

One of the main selling points of this protocol is that, like everything else used in the project, it's an open standard [29]. Another selling point is that XMPP follows a decentralized structure, meaning anyone can implement and host any kind of XMPP service in their own machines. XMPP is also flexible, things can and have been built on top of XMPP, there are several well-known extensions that can be used like publish-subscribe ones (Note: To maintain interoperability, common extensions are managed by the XMPP Standards Foundation.).

The value of XMPP has been shown by many of the technology giants. Facebook's IM system is based on XMPP. Google Talk is based on XMPP and is not discontinued, despite reports of the contrary. The following subchapters specify software commonly used to implement this protocol.

### 3.1.1 Openfire

Openfire is the server which handles the messaging between two clients since they do not talk directly to each other. It is a real time collaboration (RTC) server licensed under the Open Source Apache License.

Being built to handle the XMPP protocol, Openfire can receive and send messages from/to any client that also implements XMPP. This means that receiving of the messages sent by our system is not locked to our application. Should you need to check them on an iPhone or a BlackBerry in an emergency you could very well use a XMPP client like Crosstalk or Vayusphere to read raw data.

This server supports multi-user chat by the XMPP extension XEP-0045 [30] and has a user-friendly web-based installation that makes it in line with all the other components in this project. It supports Offline messages, Server-to-Server connectivity, and database connectivity with MySql for example.

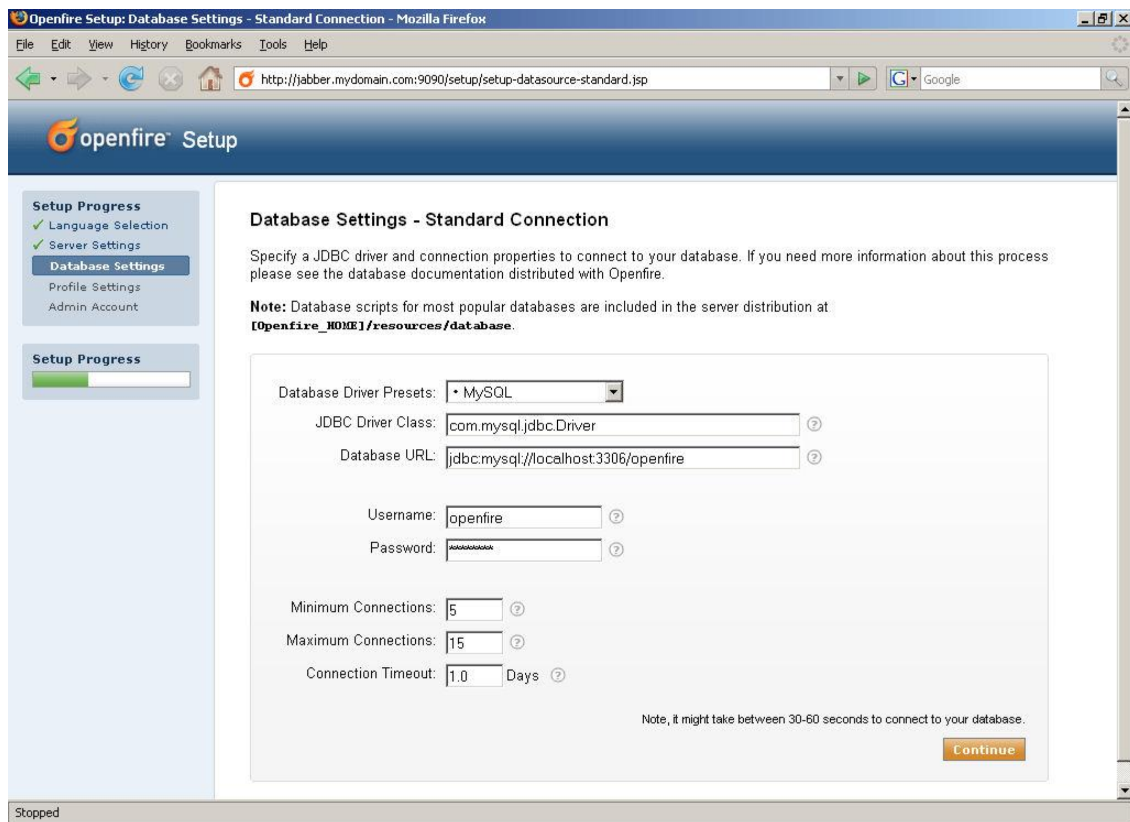


Figure 19 - Setup pages on Openfire, illustrating the use of external databases

Another interesting feature that could be used in a future version of this project is the ability to archive and log messages. It's just another example of how the decisions on the technologies to be used was made to make the system modular, decentralized and extensible, and not by which were theoretically the best or the most popular, or even the easiest to implement.

### 3.1.2 Smack API

Smack is the API on which the clients trading messages will be built with. Since the goal is to have machines trade messages and not humans, the installation of a regular XMPP client like Spark would not suffice. Hacking the clients to accept messages from a machine instead of keyboard input is not the way to go in this scenario. Why Smack? It's a pure Java library for JVMs/ Android and has integration with Openfire, so Smack can be implemented inside a Java program to run on any machine.

Recently versions 4.1.+ up have been released after a long time of little to no updates. [31] Before 4.1, smack could not run natively and unmodified on Android. This version was released after the development of this project started, and until then a wrapper for versions 3.x of Smack was used in Android. This library was called aSmack and was not official, poorly

documented and unreliable, but it was the only available alternative. Fortunately all these problems were fixed by migrating from aSmack to Smack 4.1.+.

## 3.2 AMQP

Advanced Message Queuing Protocol (AMQP) is an open standard application layer protocol for message-oriented middleware. Is has been approved as an international standard which is often used in enterprise-level applications and it's often referred to as the Internet Protocol for Business Messaging.

This protocol's features include [32]:

- All AMQP clients interoperate with all AMQP servers
- Diverse programming languages can communicate easily
- Legacy message brokers can be retrofitted to remove proprietary protocols from your networks
- Enables messaging as a cloud service
- Advanced publish-and-subscribe
- Transactional messaging functionality
- Delivery of offline messages
- Encrypted transactions
- Sending of one big message while still receiving status updates on the same network
- Implementation in most popular operating systems and languages

The broker concept in AMQP is very important, since AMQP works in a publish-subscribe architecture brokers receive messages from the publishers and route those messages to all the subscribers of that publisher. When the subscribers receive the messages an acknowledgment is sent, ensuring the delivery of the message which usually terminates the transaction. However, in AMQP an acknowledgement can have one of two types:

- Normal acknowledgement, using the *basic.ack* AMQP method which announces everything went well with the receiving.
- Negative acknowledgements, using the *basic.reject* AMQP method, which announces that either there was a problem with the transmission or the processing of the message after receiving.

If a negative acknowledgement is sent, the subscriber can also ask the message broker to either discard it or repeat the sending. Below is described a popular software implementation of this protocol, RabbitMQ. There are many others, notably Windows Azure Service Bus. However, since other implementations are often not open source or widely accepted they aren't of much interest to this project.

### 3.2.1 RabbitMQ

RabbitMQ is an open source message broker project which has its code released under the Mozilla Public License. The project consists of [33]:

- The RabbitMQ exchange server itself
- Gateways for HTTP, Streaming Text Oriented Messaging Protocol (STOMP), and MQTT protocols
- AMQP client libraries for Java, .NET Framework, and Erlang. (AMQP clients for other languages are available from other vendors.)
- A plug-in platform for custom additions, with a pre-defined collection of supported plug-ins, including:
    - A "Shovel" plug-in that takes care of moving or copying (replicating) messages from one broker to another.
    - A "Federation" plug-in that enables efficient sharing of messages between brokers (at the exchange level).
    - A "Management" plug-in that enables monitoring and control of brokers and clusters of brokers.

Setting up a connection between RabbitMQ clients is pretty simple. The programmer has to import the connection libraries of the client:

```
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
```

And subsequently create a connection and channel:

```
ConnectionFactory factory = new ConnectionFactory();
  factory.setHost("localhost");
  Connection connection = factory.newConnection();
  Channel channel = connection.createChannel();
```

After that the only thing to do is to declare a channel and publish the message to it:

```
  channel.queueDeclare(QUEUE_NAME, false, false, false, null);
  channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
```

The client only has to subscribe the messages to that queue to receive them:

```
Consumer consumer = new DefaultConsumer(channel) {
   @Override
```

```
    public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties
properties, byte[] body)
        throws IOException {
      String message = new String(body, "UTF-8");
      System.out.println(" [x] Received '" + message + "'");
    }
  };
  channel.basicConsume(QUEUE_NAME, true, consumer);
```

## 3.3 Other implementations and architectures

Implementing the whole message trading system is not the only option, nowadays there are several cloud based services that allow you to send them your IoT sensors information which is stored and possibly even processed online.

### 3.3.1 Xively

Xively [34] is a PaaS for the Internet of Things. It allows for cloud messaging, data archiving, directory services and provisioning to be accessible from Xively API. The difference from the previous architectures is that such services allow a developer to not implement or host a server with the message trading logic. This simplifies work and removes worrying risks such as problems in the server configurations and uptime.

Xively seems to have a strong security component, including:

- From and to connections can be done with HTTPS
- Access to the API is done using API access keys which allow for fine-grained access controls
- Support for OAuth to allow third party applications to access your resources securely without giving them access to your username and password
- Public and Private feeds, which specify whether or not search engines can index your data

The following image provides a good explanation of Xively API:

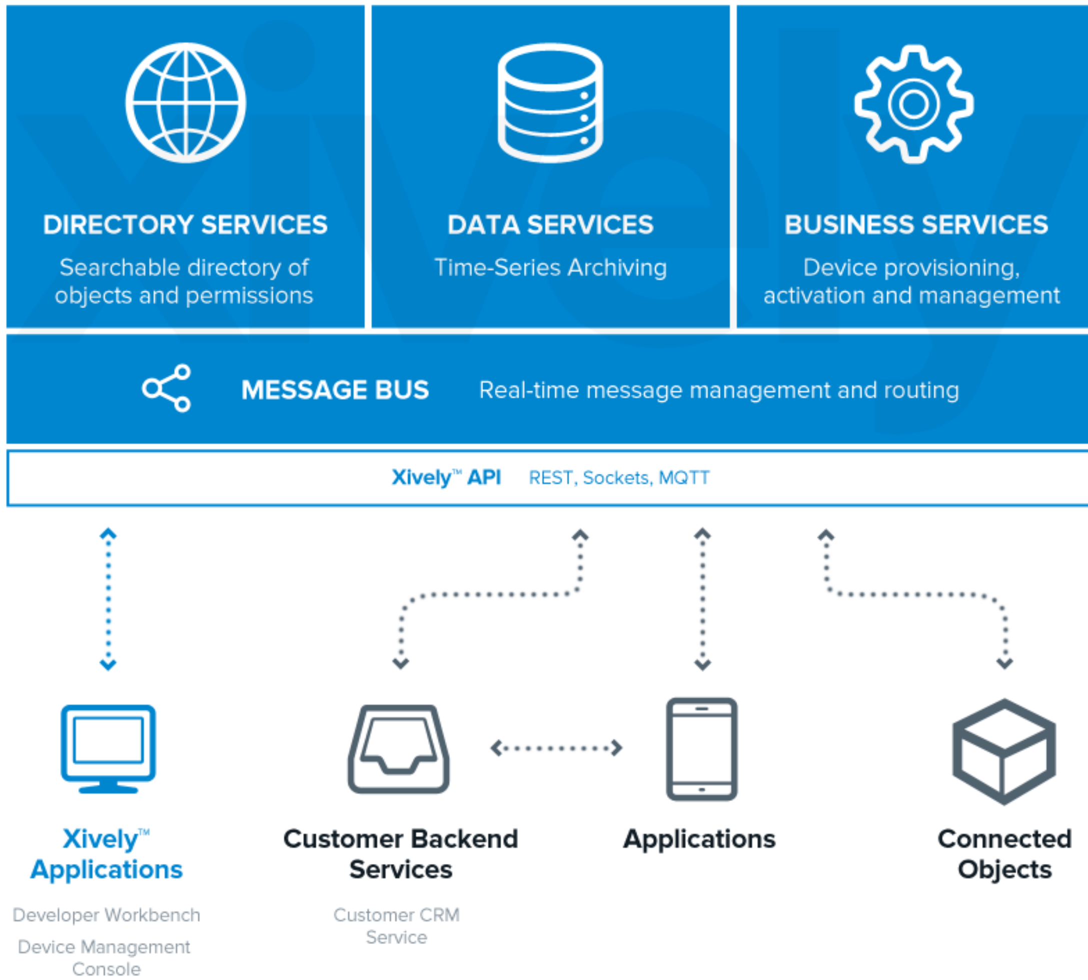


Figure 20 - Xively architecture

### 3.3.2 ThingWorx

ThingWorx [35] is similar to Xively in that it is also a PaaS for the Internet of Things. The promise is that you can save time and reduce risk and cost by implementing this platform over your own server. Features include:

- Speed - Deploy 10 times faster with model-based development
- Flexibility - Deploy how you like
- Scalability - Evolve and grow your application over time
- Modern and complete platform
- Mashup people, systems & machines

ThingWorx offers a wide range of tools to aid development of applications for the Internet of Things such as a codeless mashup builder for network and application planning. In fact, this is

probably the main selling point of this service and makes it more of a framework than just a service. There are tools for:

- Application modelling
- Search-based intelligence
- Dynamic collaboration
- Business Process Management
- "Thing" Management, where thing is how the company refers to what in our project are the sensors, the "things" in the Internet of Things.

The following image mirrors this service's architecture:

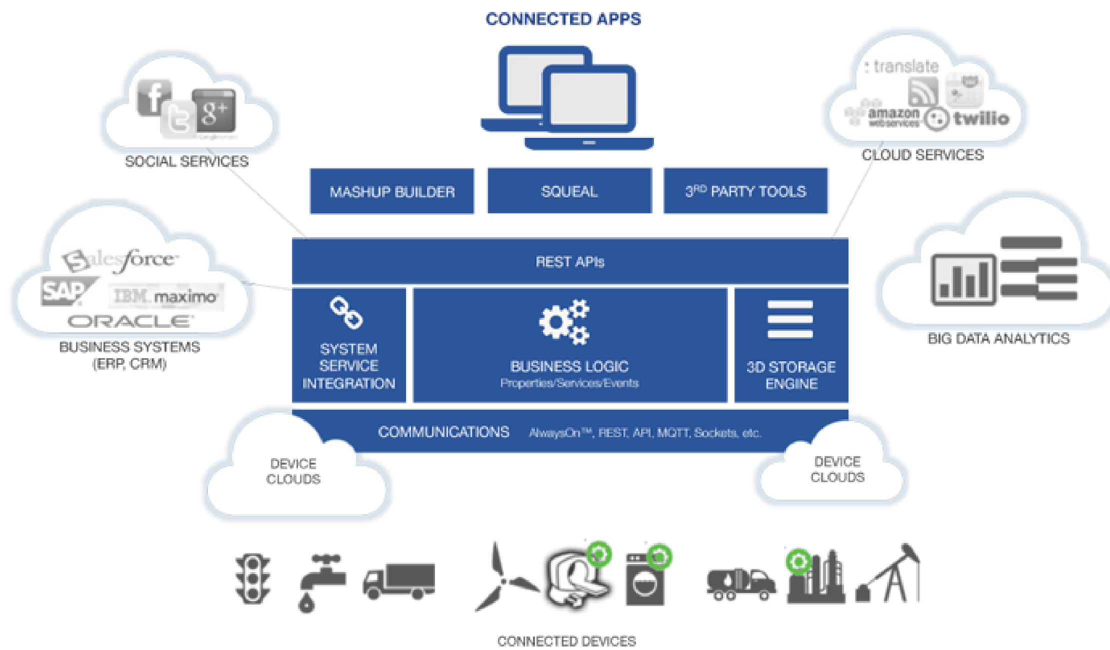


Figure 21 - ThingsWorx architecture

If you compare it to Xively's architecture you can tell how ThingsWorx relies heavily in their tools to provide a more complete service.

# 4 Project Architecture and technologies

There are several hardware and software components in this project. Some decisions had to be made regarding the types of communication protocols and whether or not to have cloud-based processing of data. This report describes the options considered and offers insight on the decisions that were made. Initial proposals and rejected approaches are also presented in Section 4.2.

## 4.1 System architecture

The following diagram represents the system physical architecture. It contains every piece of hardware this system acts on and is captioned with a small text about each type of component. An effort was made to make the system use as much open source software and well known formats as possible. Since the aim of the project is to deliver a free and Open Source option to data gathering any technology with a costly license was automatically discarded.

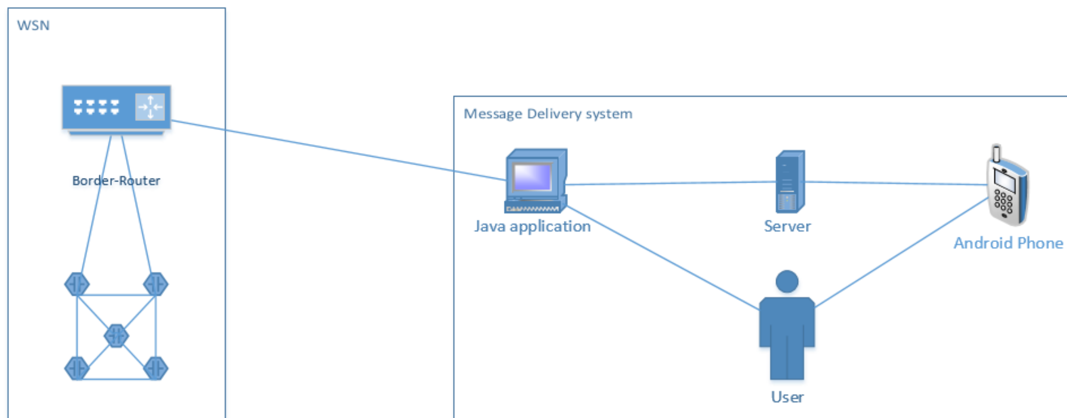


Figure 22 - General Architecture

The system architecture has four main components: The WSN (divided into sensor nodes and border router), Linux Gateway, Linux Server and Mobile App. The following subsections have detailed explanations on every component of the architecture. They include reasoning behind every design choice.

### 4.1.1 WSN – Sensor network

Every Sensor node in this network would be at the very least composed of a board with the actual sensor, a microprocessor and an antenna. Loaded into the node is Contiki-OS, an operating system for the Internet of Things. Each node is running a different program written for Contiki-OS. The routines of these programmes gather sensor data and send them to the Border Router.

Every Sensor has its own IPV6 address in the network and can communicate with every other node in the network. Furthermore, since this network uses a routing protocol called RPL, even sensors that are not connected directly to the border router can send him messages. They will simply be transported through a route in the network defined by the algorithm. [36]

The reason why the Linux Gateway is directly gathering sensor data from these sensors is because of hardware limitations. There would be a need for more advanced and therefore expensive nodes which could store information to act as sinks for the Linux Gateway to connect to, or at least to upgrade the sensor nodes themselves.

This option was actually largely considered as the best approach for some time while developing the project, simply because it was a simpler way of communicating with the outside of the network. However, after trying to run a simulation considering the storage of values in the nodes it was verified that the cost of the network would quickly increase as the

number of sensors increased, while the current approach allows for a much slower increase in costs, even though the initial cost is higher.

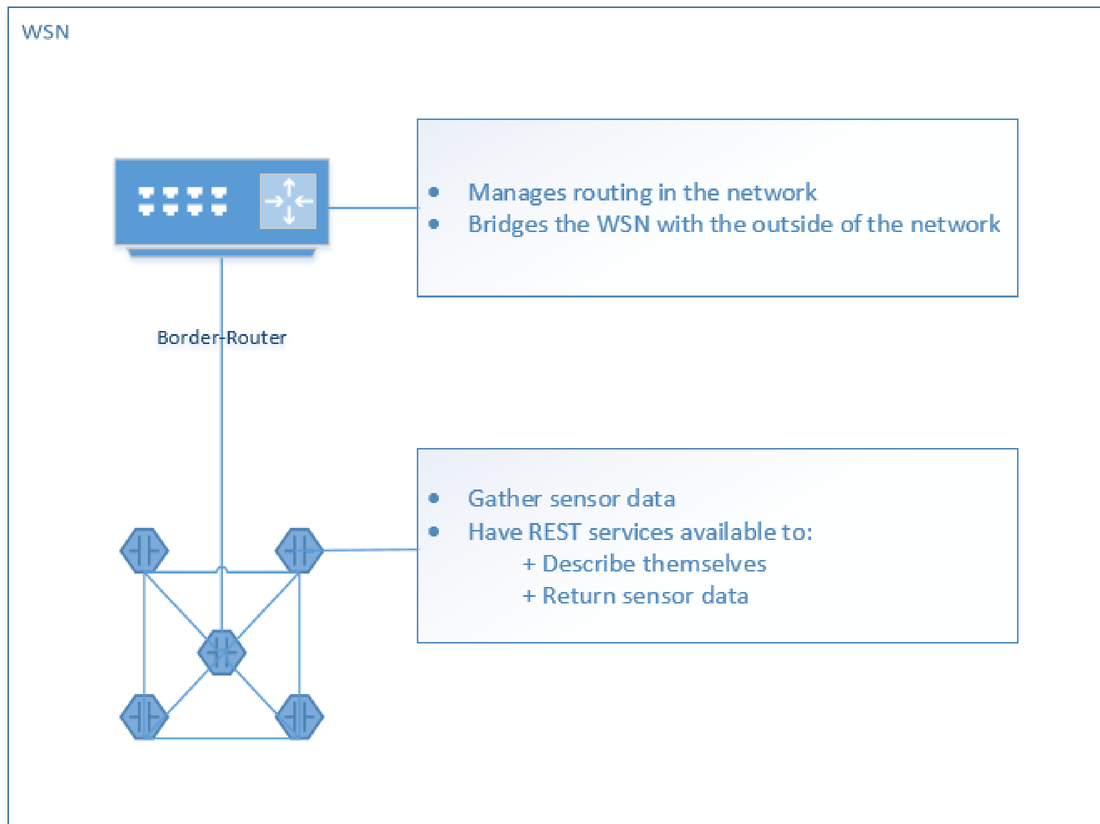


Figure 23 - WSN detailed view

The Border Router will not have any kind of sensor installed. It is simply router with the responsibility to coordinate the network and communicate with the outside.

### 4.1.2 Linux Gateway

Any small computer, the likes of the popular Raspberry Pi, can be used to run this module. It can also use the users home or laboratory computer.

Once again, this was thought so that the cost of building the network was as low as possible. Even in the event a Raspberry Pi has to be bought to continuously run this application, the cost of doing so would be negligible compared to existing solutions. The physical space one would need to run such a computer is also very small.

Figure 24 - Size comparison between a modern smartphone (left, iPhone 4s) and a Raspberry Pi (right, B+ model)

Another interesting feature of using these small computers is their low energy consumption. Once again this is in line with the mission of developing a cost effective solution. Below is a graph comparing the energy consumption of various mini PCs. The graph shows values for different models of the Rasp Pi and even other boards. The main difference between model A and model B are the hardware specs.

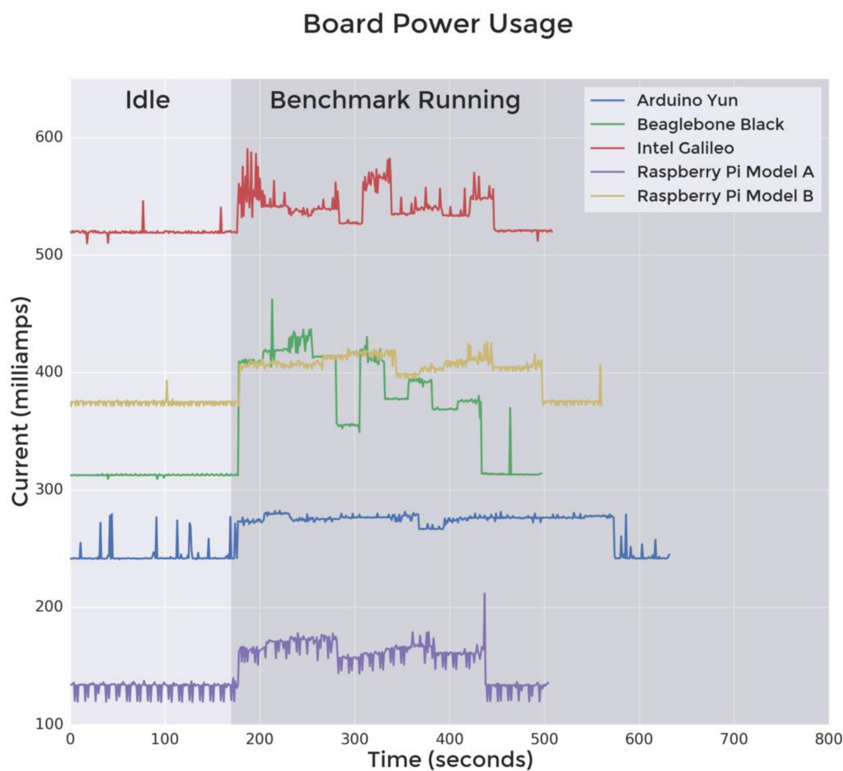


Figure 25 - Energy consumption of various embedded boards.

Notice how much more energy the model B consumes than the model A. The upgraded model B+ (presented above) already addresses this concern and is slightly more energy efficient. However, we are talking differences of milliamps, and although they add up the more you keep your Raspberry Pi on, the difference in the final energy bill will be negligible, especially when we have only one computer running. For a long running operation it might actually be cheaper to buy a model A, though the cost won't differ astronomically.

It's also important to notice how these values are the minimum required to run the boards. Add a monitor and the milliamp usage will spike, add a keyboard and you will also feel some difference. Even connecting the Ethernet cable and sending packets via 802.11 will consume noticeably more energy. However, there is little or nothing that can be done in this regard. Hardware components cost energy to use and it is not the goal of this thesis to manage energy consumption of hardware devices.

It runs a Java application, and therefore can be ran anywhere where a Java VM is installed. It sends (and can potentially receive, this is a two-way chat, although not implemented yet), XMPP messages to the Openfire server, which will be received by the Android App when it is online.

A typical message will be composed of the sensor node identification string, the value and the time at which that data was recorded, along with a flag that determines whether or not a message is critical. As far as communication with the WSN goes, the Gateways uses a CoAP library to connect to the nodes and reach out to their REST services.

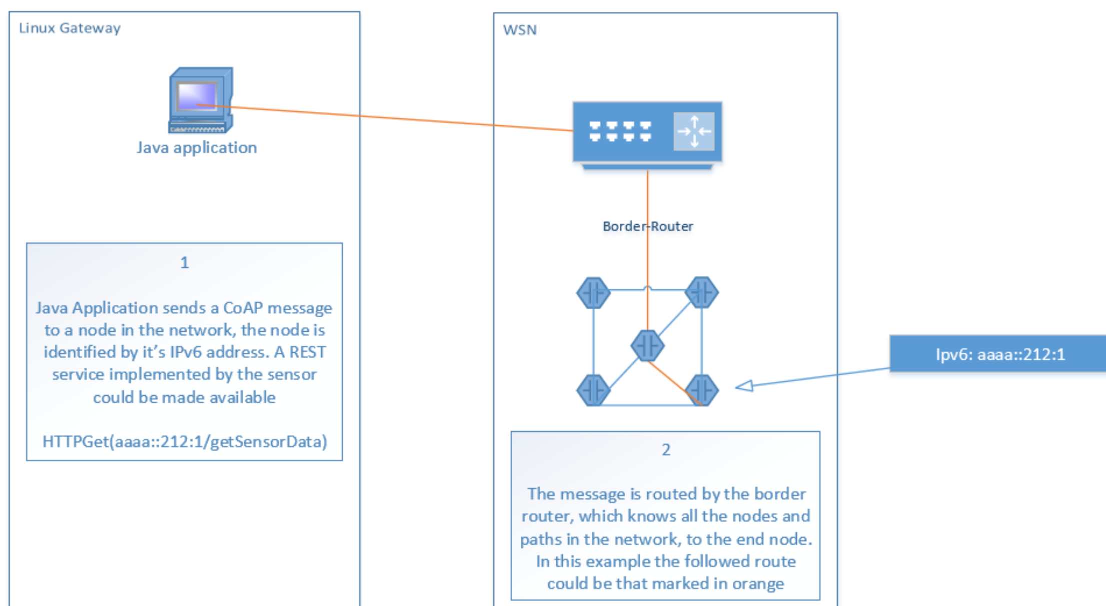


Figure 26 - Example of communication between the Linux Gateway and a WSN node

### 4.1.3 Linux Server

Arguably the most complex of all the components, this server is a virtual machine that was generously provided by the Instituto Superior de Engenharia do Porto for the duration of this thesis.

In order to keep costs low for whoever is building the system, Linux, a free Operating System that requires no purchase to use, was installed. The server is currently running the latest LTS version of Ubuntu, 14.04.2. Software installed includes:

- Openfire, the XMPP server that handles the trading of the messages is running in port :9090.
- Apache Tomcat is running on port :8080 and handles the incoming web service calls to the users database.
- MySql, the chosen database in charge of being storing Openfire tables

Jersey RESTful services were deployed in a project to Apache Tomcat, they contain endpoints for GET, PUT, POST and DELETE methods that perform operations in the database, although they are not operational at this point and serve no real function.

### 4.1.4 Android App

The Android application the main interface used by the User to access the data transmitted through the system. The earliest version of Android that can be used in this project is 4.2 and the target version is 5.0.

The Interface is simple, it has the following activities:

- A main page with some lists displaying live sensor values, with special attention for critical values
- A page with a list of all available sensors
- A page which presents with the sensor details
- A settings page to adjust things like automatic login

The app first connects to the Server to retrieve all offline messages flagged as critical. Then establishes a chat with the Linux Gateway through the server and starts receiving live messages from it. If the message is signalled with the critical flag it is displayed automatically in the main activity, if not it will just appear in the sensor details page.

## 4.2 Rejected approaches

The first big decision was in regard to the devices of network we would build for. The thesis proposal presented in the beginning of the year still included a WSN composed of various low

power sensors, a Java application, installed in a computer around the networks that would gather the sensors data and have an offline persistence ability, an online database, which holds important user data and receives and permanently stores sensor data and an android app that would display important values to its user about sensors he owns.

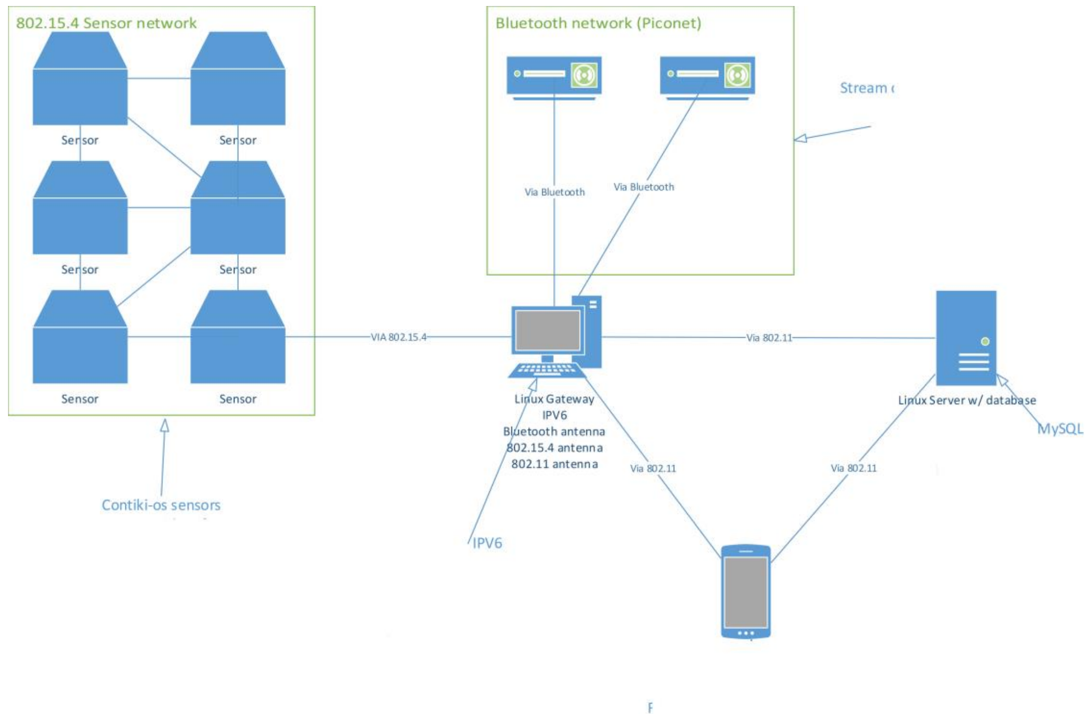


Figure 27 - General Architecture, as proposed initially.

This was the first proposed architecture diagram, however, the following decisions were made:

- To ignore the proposed Bluetooth network for now as it would take its time to implement, time that could be used to further study the WSN (A type of network students never programmed for during the course). Furthermore a Bluetooth network was already built in another project and had its analysis made in a four page paper, linked as annex.
- To include a node in the WSN to act as a border router.
- To drop the user database and the pretty Linux Gateway GUI, since it was mostly an extra feature that doesn't really make much of a difference in a research thesis such as this.
- To focus on the standards for the IoT and their implementation instead of a good looking system on top of basic programs.

# 5 System implementation

In this chapter it will be shown how the application is working as of now and bits of critical code will be presented. This is your guide to the implementation of the developed prototype and will clarify how things were built and how you can interact with the application. It also describes most proposed use cases. It is presented in a specific order: The flow of information since it is captured by the sensor until it arrives at the Android app.

## 5.1 Some considerations

A random function call is being used to simulate values from the sensors. Actual algorithms that will gather actual sensor information were developed and will be presented in this chapter, but since when running the program in a sensor network simulator such as COOJA no real sensor is attached to a real microcontroller loaded with the program, it's necessary to simulate those values somehow, and COOJA won't do that automatically.

It should be noted that the RPL algorithm, and subsequently the majority of the border router program was not written by the author of this report. That would constitute a thesis on its own. RPL is a standard for the IoT and is used more like a tool in this project. Still, this is a research project and explaining the way RPL was used is absolutely necessary to comprehend how the WSN works, so the implementation of the RPL algorithm Contiki comes bundled with will be presented in chapter 5.3.

## 5.2 Sensor Nodes

This is where the data gathering occurs. Each sensor node has code to extract values from its sensors. This code is obviously dependant on which sensor is going to be used, in this demonstration a temperature value is fetched from the sht11 sensor which tracks both humidity and temperature. A node will also have to make rest resources available for the Linux Gateway to call. So let's start by declaring and implementing the main process, which will run as soon as the nod is added to the network and powered on.

```
PROCESS(thesis_temp_sensor, "Temperature Sensor");
AUTOSTART_PROCESSES(&thesis_temp_sensor);

PROCESS_THREAD(thesis_temp_sensor, ev, data)
{
  PROCESS_BEGIN();

  sht11_init();
  SENSORS_ACTIVATE(sht11_sensor);
```

```
    rest_init();

    rest_activate_resource(&resource_sensor);
    rest_activate_resource(&resource_helloworld);
    rest_activate_resource(&resource_discover);

    PROCESS_END();
}
```

The `sht11_init()` line is required to use the sensor. According to its specification and library it powers up the device, which can be used an additional 11 milliseconds after the call.

The `rest_init()` line is necessary to setup the RESTful library. After this line is called you can activate REST resources by calling `rest_activate_resource(&resource_name);`

Since REST resources need to have logic implemented, each sensor has the following definitions:

```
RESOURCE(discover, METHOD_GET, ".well-known/core");
RESOURCE(sensor, METHOD_GET, "sensor");
RESOURCE(helloworld, METHOD_GET, "helloworld");
```

The first one is a standard for discoverable REST resources for CoAP clients. It is convention that this URI path will return all the available resources of a node, making them self-identifiable, the implemented logic of this resource is coded as follows:

```
void discover_handler(REQUEST* request, RESPONSE* response)
{
 char temp[100];
 int index = 0;
 index += sprintf(temp + index, "%s,", "</helloworld>;n=\"HelloWorld\"");
 index += sprintf(temp + index, "%s", "</sensor>;n=\"Sensor\"");

 rest_set_response_payload(response, (uint8_t*)temp, strlen(temp));
 rest_set_header_content_type(response, APPLICATION_LINK_FORMAT);
}
```

The name of these methods is also convention, it needs to be the resource variable given in the declaration ( `RESOURCE(discover, METHOD_GET, ".well-known/core")` ) followed by an underscore and the word handler.

The second one contains more or less the same signature but has a call to the data gathering method.

```
uint16_t temperature;
RESOURCE(sensor, METHOD_GET, "sensor");
Void sensor_handler(REQUEST* request, RESPONSE* response)
{
   temperature = ((sht11_sensor.value(SHT11_SENSOR_TEMP) / 10) -396) / 10;
```

```
    sprintf(temp,"Temp:%u\n", temperature);

    rest_set_header_content_type(response, TEXT_PLAIN);
    rest_set_response_payload(response, (uint8_t*)temp, strlen(temp));
  }
```

The mathematical operations applied to the temperature ((sht11_sensor.value(SHT11_SENSOR_TEMP) / 10) -396) / 10; are guidelines from the specification and manuals of the sensor.

## 5.3 Border Router

The Border router only coordinates the communication inside the WSN, it has no sensor data gathering or storage capabilities. This coordination is done mainly by implementing the RPL algorithm.

This is still a node running under Contiki, so it still follows the syntax explained earlier. The execution starts with the following protothread:

```
PROCESS_THREAD(border_router_process, ev, data)
{
 static struct etimer et;
 rpl_dag_t *dag;

 PROCESS_BEGIN();

 prefix_set = 0;
 NETSTACK_MAC.off(0);

/* Request prefix until it has been received */
 while(!prefix_set) {
  etimer_set(&et, CLOCK_SECOND);
  request_prefix();
  PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
 }

 dag = rpl_set_root(RPL_DEFAULT_INSTANCE,(uip_ip6addr_t *)dag_id);
 if(dag != NULL) {
  rpl_set_prefix(dag, &prefix, 64);
  PRINTF("created a new RPL dag\n");
 }
 /* Now turn the radio on, but disable radio duty cycling.
   * Since we are the DAG root, reception delays would constrain mesh    throughbut.
 */
 NETSTACK_MAC.off(1);

 #if DEBUG || 1
  print_local_addresses();
 #endif

 while(1) {
  PROCESS_YIELD();
  if (ev == sensors_event && data == &button_sensor) {
```

```
        PRINTF("Initiating global repair\n");
        rpl_repair_root(RPL_DEFAULT_INSTANCE);
      }
    }

  PROCESS_END();
}
```

Let’s look at some critical lines of code:

The timer that was declared in this line: `static struct etimer et;` is a timer commonly used in Contiki programs, and in this case it serves as a waiting time for the setting of the network prefix inside the while block.

The reason we turn off the radio with `NETSTACK_MAC.off(0);` is because the border router might accidentally join an existing DAG as a parent or child , or acquire a default router that will later take precedence over the SLIP fallback interface.

After the network prefix is received, we can put together a DAG network with our border router as the root node. As happens with any DAG, all the structure has one and only one root to which all the other nodes are directed towards, the following picture took from a COOJA simulation with the border router and some sensor nodes explains nicely the structure of the network.

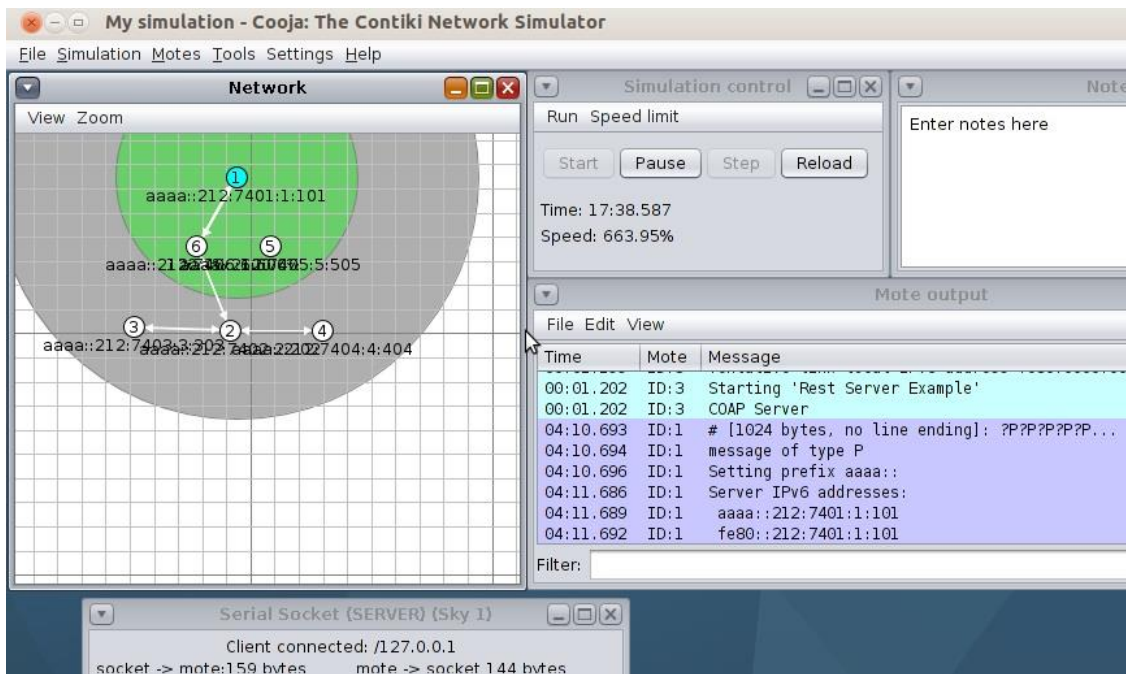


Figure 28 - DAG netowrk captured in COOJA with radio environment

Node 1 is the border-router. In green we have the range of the border router. All the ranges of the nodes are the same, so to ping node number 4 for example, the border router has to use a route of nodes that are collectively within range of one another and make up a path between

the nodes 1 and 4. Since this is a DAG the structure of the network is like a tree and the border router sees nodes directly connected to it as his direct sons and nodes connected to those as sons of his sons, and so on.

In short. To get to node N, the border router does a search to see which of his sons (and his son's sons, and so forth) has a pointer to that node, and directs his request through that path.

## 5.4 Linux Gateway

The Linux Gateway starts with an implementation of the CoapClient interface of jcoap. AS an interface, it needs to implement the following methods:

```
@Override
  public void onConnectionFailed(CoapClientChannel channel, boolean notReachable, boolean
resetByServer) {
    System.out.println("Connection Failed");
  }

  @Override
  public void onResponse(CoapClientChannel channel, CoapResponse response) {
    System.out.println("Received response " + new String(response.getPayload()));
  }
```

The first thing Linux Gateway does is to query all the nodes for their CoAP services. This is done by doing a CoAP request to a conventional .well-known/core path.

```
BasicCoapClient client = new BasicCoapClient();
client.channelManager = BasicCoapChannelManager.getInstance();

For(Sensor s : sensorNodes){
                  try {
      clientChannel = channelManager.connect(this, InetAddress.getByName(s.SERVER_ADDRESS),
s.PORT);
      CoapRequest coapRequest = clientChannel.createRequest(true, CoapRequestCode.GET);
      coapRequest.setUriPath("/well-known/core");
      clientChannel.sendMessage(coapRequest);
      System.out.println("Sent Request");
    } catch (UnknownHostException e) {
      e.printStackTrace();
  }
}
```

The responses for these requests are treated on the onResponse method, which adds to an ArrayList<String> of the sensors the paths of the available resources.

After this is done, the Linux Gateway just does the same thing but using a different UriPath, the new path is dependent on the responses the call to the well-known/core call produced. So if the call returned a /sensor resource available on the sensor we can now query it and expect a result which gives us the sensor data: coapRequest.setUriPath("/sensor");

After a timeout period defined by the user, for example 5 minutes, we establish a chat with our mobile app using XMPP.

The first thing that needs to be done for this to happen is to build a TCP connection to our Linux Server. In this case, the following settings were used

```
XMPPTCPConnectionConfiguration config = XMPPTCPConnectionConfiguration.builder()
        .setUsernameAndPassword(myUsername, myPassword)
        .setServiceName(myServiceName)
        .setHost(myIP)
        .setPort(myPort)
        .build();
```

The serviceName, host ip and host port will be different for every user. These are chosen when you first install Openfire in the Linux Server and have to be manually introduced in the Linux Gateway for it to work.

After these are set, we can connect:

```
AbstractXMPPConnection conn1 = new XMPPTCPConnection(config);

conn1.connect();

conn1.login();
```

We get the roster, which represent the apps we are sending the values to:

```
Roster roster = Roster.getInstanceFor(conn1);
if (!roster.isLoaded()) {
        roster.reloadAndWait();
}

Collection<RosterEntry> entries = roster.getEntries();
```

And create the chat connection, keep in mind that the roster only contains the online users, so if someone is not using the mobile app the chat will not start, as is intended:

```
If(!entries.isEmpty()){
    ChatManager chatmanager = ChatManager.getInstanceFor(conn1);

        chatmanager.addChatListener(new ChatManagerListener() {
          @Override
          public void chatCreated(Chat chat, boolean createdLocally) {
            setChat(chat);
            chat.addMessageListener(new ChatMessageListener() {
              @Override
              public void processMessage(Chat chat, Message message) {
                if (message.getType() == Message.Type.chat || message.getType() ==
    Message.Type.normal) {
                  if (message.getBody() != null) {
                    System.out.println("Message: " + message.getBody());
                  }
```

```
                        }
                    }
                });
            }
        });
        Chat newChat = chatmanager.createChat(entries.iterator().next().getUser());

    sendValues();
}
```

Messages received by the Linux Gateway are currently not being processed for anything, and are only being printed to the console, as this block suggests:

```
if (message.getBody() != null) {
    System.out.println("Message: " + message.getBody());
}
```

The messages are being sent with the following code inside sendValues();

```
For(Sensor s : sensors){
        For(String value : s.Values){
                newChat.sendMessage(s.Name() + "." + value);
        }
}
```

## 5.5 Linux Server

There was no programming component in the Linux Server. All that is required is to setup some programs, so this chapter will focus on what implementation choices were made. Connection to this server is made through VMWare vSphere Client since it is a virtual machine made available by ISEP.

Here we can see that Openfire is indeed installed in that machine and the admin console is available at port :9090, the default admin console port.

You can also notice the port where the Linux Gateway and Mobile app will connect through, it is the Client to Server port :5222. If an encrypted connection is required port :5223 can be used.

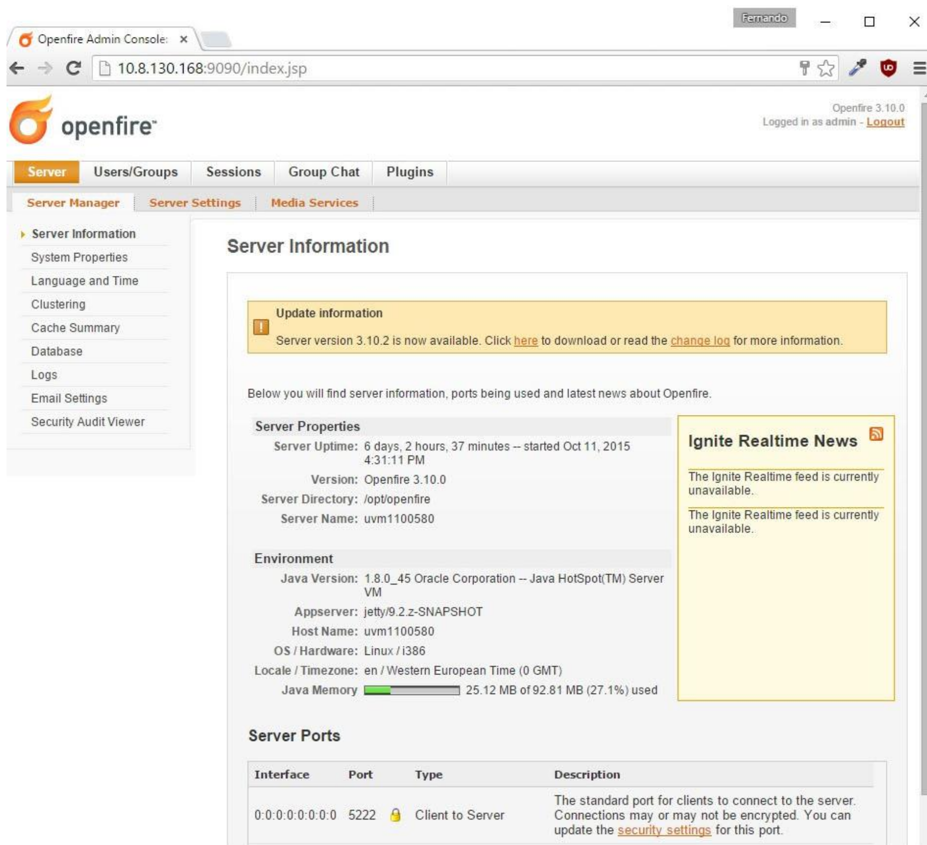


Figure 29 - Openfire admin console, displaying information about the installation

Openfire was installed using a mySql database so that functionality can be extended on top of the existing structure. Apache Tomcat was also installed and is running on port :8080, but as of now it isn't associated with any functionality since the option to have a user database was dropped from the project for now. It should be noted that the rest services were indeed developed and are running in the Tomcat server, but they consist of only CRUD operations to a User table in mySql.

/manager | Apache Tomcat/8.0.21

10.8.130.168:8080/manager/html

The Apache Software Foundation
http://www.apache.org/

**Tomcat Web Application Manager**

| Message: | OK |
|---|---|

| Manager | | | |
|---|---|---|---|
| List Applications | HTML Manager Help | Manager Help | Server Status |

| Path | Version | Display Name | Running | Sessions | Commands |
|---|---|---|---|---|---|
| / | None specified | Welcome to Tomcat | true | 0 | Start Stop Reload Undeploy / Expire sessions with idle ≥ 30 minutes |
| /docs | None specified | Tomcat Documentation | true | 0 | Start Stop Reload Undeploy / Expire sessions with idle ≥ 30 minutes |
| /examples | None specified | Servlet and JSP Examples | true | 0 | Start Stop Reload Undeploy / Expire sessions with idle ≥ 30 minutes |
| /host-manager | None specified | Tomcat Host Manager Application | true | 1 | Start Stop Reload Undeploy / Expire sessions with idle ≥ 30 minutes |
| /manager | None specified | Tomcat Manager Application | true | 1 | Start Stop Reload Undeploy / Expire sessions with idle ≥ 30 minutes |
| /restThesis | None specified | Restful Web Application | true | 0 | Start Stop Reload Undeploy / Expire sessions with idle ≥ 30 minutes |

Figure 30 - The crud REST services running in the Linux Server

Ultimately this shows how extensible this project is, that you can use whatever exists already and build on top of it to expand functionality, just as it was intended.

## 5.6 Mobile App

In terms of implementation, the main Activity contains only lists and basic UI components. The real effort is in the connection to the Linux Gateway through the server.

If before we gave no importance to the receiving of messages, now the exact opposite is being done. The app has no logic to transmit messages to the Linux Gateway, and instead only consumes them. Firstly it's necessary to build a chat listener to receive messages sent by the Linux Gateway and then it's necessary to register a message listening method to handle incoming messages.

```
ChatManager chatmanager = connection.getChatManager();
 connection.getChatManager().addChatListener(new ChatManagerListener()
 {
   public void chatCreated(final Chat chat, final boolean createdLocally)
```

```
 {
  chat.addMessageListener(new MessageListener()
  {
    public void processMessage(Chat chat, Message message)
    {
      //logic of the message receiving goes here
    }
  });
 }
});
```

The logic of receiving messages is too long to insert here but it distributes messages across ArrayLists<Message> inside Sensor instances. It also has logic to distribute critical messages to a special List which is presented in the main page.

# 6 Conclusion

The quest to build a project that studies low-level programming and IoT technologies proved itself difficult. There is work to be done in the establishment of standards and the area needs more scientific development and in-depth studies of technologies to become easier to pick up.

The hardware limitations lead to weird programming trends and standards which are not in line with the much more mature software development seen in companies nowadays. It's almost as if building an IoT solution is to make the best of a bad situation as there is no widely accepted set of standards that connect the IoT to the rest of the internet or to the development process, i.e. there is a lack of software development methodologies like scrum that work well in this kind of development; there are no real debug tools in Contiki; etc.

This, however, is not caused by incompetence or even lack of interest of the subject. This is what every programming field passes through on its way to greatness. It is more of a privilege than a curse to build this thesis in a time where new, completely revamped versions of IoT operating systems are coming out.

The final thought is that, despite all this, it is absolutely possible to build a solid IoT solution or project with the existing standards and technologies, that Contiki itself is a great operating system for what it's intended and that the 6LoWPAN stack covers a lot of problems that exist in communication between nodes. The fact that many companies are building PaaS's with a high level of abstraction for this field means that the industry is evolving to fix the previously reported problems, and that in years' time development for the IoT will be much quicker and smoother and bring about a re-reawakening of the IoT, complete with high-level programming of low-power devices.

It was also possible to contribute to the field with the proposed characteristics of this system, as well as build an extensive report of the state of the art of all these technologies. Information about them is not scarce, even though there is very little consensus on the various competing technologies, which often try to occupy one another's fields.

# 7 Works Cited

[1] “oreilly,” [Online]. Available: http://www.oreilly.com/openbook/opensources/book/appa.html.

[2] mff.cuni.cz. [Online]. Available: https://d3s.mff.cuni.cz/~ceres/sch/osy/text/ch01s03s06.html.

[3] “thelinuxdesk,” [Online]. Available: https://thelinuxdesk.wordpress.com/tag/monolithic-kernel/.

[4] J. N. Herder, “minix3,” [Online]. Available: http://www.minix3.org/theses/herder-true-microkernel.pdf.

[5] Contiki.org. [Online]. Available: http://www.contiki-os.org/.

[6] M. O. F. a. T. K. *, “mdpi,” [Online]. Available: http://www.mdpi.com/1424-8220/11/6/5900/htm.

[7] c2. [Online]. Available: http://c2.com/cgi/wiki?EventDrivenProgramming.

[8] J. Hill. [Online]. Available: http://www.cs.berkeley.edu/~culler/cs294-f03/papers/tinyos.pdf.

[9] B. G. T. V. Adam Dunkels, “dunkels,” [Online]. Available: http://www.dunkels.com/adam/dunkels04contiki.pdf.

[10] alignan, “github,” [Online]. Available: https://github.com/contiki-os/contiki/wiki/An-Introduction-to-Cooja.

[11] “stanford.edu,” [Online]. Available: http://tinyos.stanford.edu/tinyos-wiki/index.php/TinyOS_Documentation_Wiki.

[12] “TinyOs.net,” [Online]. Available: http://www.tinyos.net/tinyos-1.x/doc/tutorial/lesson5.html.

[13] T. Reusing. [Online]. Available: http://www.net.in.tum.de/fileadmin/TUM/NET/NET-2012-08-2/NET-2012-08-2_02.pdf.

[14] “OASIS-open.org,” [Online]. Available: https://www.oasis-open.org/standards.

[15] J. Rogerson. [Online]. Available: http://www.techradar.com/news/phone-and-communications/mobile-phones/4g-and-lte-everything-you-need-to-know-926835.

[16] V. Beal, “webopedia,” [Online]. Available: http://www.webopedia.com/TERM/4/4G_LTE.html.

[17] M. Christiano. [Online]. Available: http://www.allaboutcircuits.com/technical-articles/zigbee-vs-bluetooth-and-bluetooth-smart/.

[18] “science.smith,” [Online]. Available: http://www.science.smith.edu/~jcardell/Courses/EGR328/Readings/WSN_Zigbee%20Ovw.pdf.

[19] J. Schonwalder, “Utwente,” [Online]. Available: https://www.utwente.nl/ewi/dacs/colloquium/archive/2010/slides/2010-utwente-6lowpan-rpl-coap.pdf.

[20] “ti.com,” [Online]. Available: http://www.ti.com/lsds/ti/wireless_connectivity/6lowpan/overview.page.

[21] J. Olsson. [Online]. Available: http://www.ti.com/lit/wp/swry013/swry013.pdf.

[22] D. Gascón, “libelium.com,” [Online]. Available: http://www.libelium.com/802-15-4-vs-zigbee/.

[23] “coap,” [Online]. Available: http://coap.technology/.

[24] T. Jaffey. [Online]. Available: http://www.eclipse.org/community/eclipse_newsletter/2014/february/article2.php.

[25] "slideshare," [Online]. Available: http://www.slideshare.net/jvermillard/hands-on-with-coap-36793005.

[26] P. J. Mueller. [Online]. Available: http://muellerware.org/papers/oopsla-1996/oopsla96-10.html.

[27] B. SmartWorx. [Online]. Available: http://bb-smartsensing.com/basics-of-mqtt/.

[28] "hivemq," [Online]. Available: http://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment.

[29] xmpp.org. [Online]. Available: http://xmpp.org/about-xmpp/technology-overview/.

[30] Wroot. [Online]. Available: https://community.igniterealtime.org/docs/DOC-2285.

[31] Flow. [Online]. Available: https://community.igniterealtime.org/blogs/ignite/2015/03/29/smack-410-released.

[32] amqp.org. [Online]. Available: https://www.amqp.org/product/features.

[33] "Pivotal," [Online]. Available: https://www.rabbitmq.com/features.html.

[34] "Xively," [Online]. Available: xively.com.

[35] "ThingWorx," [Online]. Available: http://www.thingworx.com/.

[36] D. M. H. F. Mário Alves, "CISTER," [Online]. Available: https://www.cister.isep.ipp.pt/docs/mobile_iot__smart_hop_over_rpl/947/view.pdf.

[37] "MQTT-OASIS-Webinar," [Online]. Available: https://www.oasis-open.org/committees/download.php/49205/MQTT-OASIS-Webinar.pdf.

# Annex 1

Figure 31 - First idea of project to develop

# Annex 2

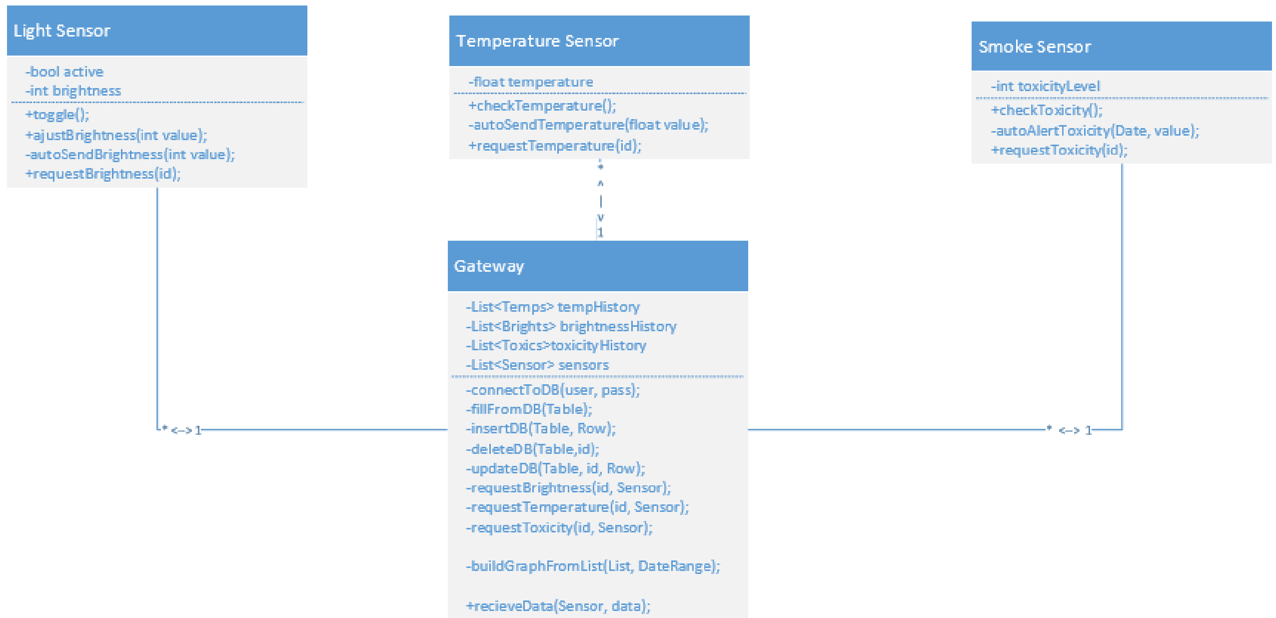


Figure 32 - Early idea of sensor's data members and methods