

UNIVERSITÀ DEGLI STUDI DI PADOVA

---

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria delle Telecomunicazioni

Tesi di Laurea

**INDOOR LOCALIZATION IN  
WIRELESS SENSOR NETWORK  
USING FINGERPRINTING METHOD**

Relatore:  
Prof. MICHELE ZORZI

Laureando:  
MATTEO SALMISTRARO

Correlatori:  
Ing. ANGELO P. CASTELLANI  
Ing. PAOLO CASARI

---

ANNO ACCADEMICO 2009-2010



*Ai miei genitori*



# Contents

<b>Introduction</b>	<b>ix</b>
<b>1 Fingerprinting Method and WSN</b>	<b>1</b>
1.1 Fingerprinting Method . . . . .	1
1.1.1 Measurement Phase . . . . .	1
1.1.2 Localization Phase . . . . .	2
1.1.3 Other proposed methods . . . . .	2
1.1.4 Algorithms for the determination of the position . . . . .	3
1.2 WSN . . . . .	5
1.2.1 Main Characteristics of a Mote . . . . .	5
<b>2 The IoT</b>	<b>9</b>
2.1 Operative Systems for motes . . . . .	10
2.2 TinyOS . . . . .	10
2.2.1 Application Specific and Event Driven . . . . .	10
2.2.2 Developing an Application using TinyOS . . . . .	11
2.2.3 Example . . . . .	11
2.2.4 Configuration . . . . .	12
2.2.5 Modules . . . . .	13
2.2.6 Configuration as Components . . . . .	14
2.2.7 Tasks . . . . .	15
2.3 WSN Networks Aspects . . . . .	15
2.3.1 Network Architecture . . . . .	15
2.3.2 IEEE 802.15.4 Medium Access Control (MAC) . . . . .	18
2.3.3 IEEE 802.15.4 Physical (PHY) . . . . .	22
2.3.4 IEEE 802.15.4 and TinyOS . . . . .	25
2.3.5 Tools for Mote-PC communication . . . . .	29
2.4 6lowPAN . . . . .	31
2.4.1 Addresses . . . . .	31
2.4.2 Adaptation Layer . . . . .	31
2.4.3 Header Compression . . . . .	31
2.5 CoAP . . . . .	32
2.5.1 Transaction Messages . . . . .	33

2.5.2	Synchronous Transactions . . . . .	33
2.5.3	Asynchronous Transactions . . . . .	34
2.5.4	Transaction ID (TID) . . . . .	34
2.5.5	Methods . . . . .	35
2.5.6	Codes . . . . .	36
2.5.7	Content Type . . . . .	36
2.5.8	CoAP implementation in TinyOS . . . . .	37
2.6	Server-side programming . . . . .	40
2.6.1	HTTP - Hyper Text Transfer Protocol . . . . .	40
2.6.2	GWT - Google Web Toolkit . . . . .	41
2.6.3	Java Servlet . . . . .	41
2.6.4	Fandango . . . . .	44
2.6.5	Java Database Connectivity . . . . .	46
2.6.6	Gateway - Serial Tunneling . . . . .	48
2.6.7	Gateway - SENSEI Gateway . . . . .	50
2.7	TinyNET . . . . .	51
2.7.1	Architecture . . . . .	52
2.7.2	Wiring with Standard Applications . . . . .	52
2.7.3	The Routing Protocol . . . . .	53
<b>3</b>	<b>Implementation</b>	<b>55</b>
3.1	Fingerprinting Application . . . . .	55
3.2	Localization . . . . .	60
3.3	Mobile - Interrogation Phase . . . . .	61
3.4	BQuery Components . . . . .	72
3.5	Mobile - Gateway Communication . . . . .	75
3.6	Gateway . . . . .	79
3.6.1	Socket . . . . .	80
3.6.2	Plug-in . . . . .	81
3.7	Fandango . . . . .	83
<b>4</b>	<b>Performance Evaluation</b>	<b>89</b>
4.1	Test Environment . . . . .	89
4.2	Results . . . . .	92
<b>5</b>	<b>Conclusions</b>	<b>95</b>
	<b>Bibliography</b>	<b>97</b>
	<b>Acknowledgments</b>	<b>99</b>







# Introduction

Wireless Sensors Networks (WSN) have recently become quite a hot research topic, with many groups around the world spending efforts on it. The basic idea behind WSNs is that many useful services can be provided by a wireless network of inexpensive and power-efficient elements called motes (i.e., tiny electronic devices with memory, a processing unit, sensors and a radio interface). However, many services are location-based, in that they require that each mote has some degree of knowledge of its own position. This knowledge may be gathered by means of automatic localization algorithms. These algorithms should be computationally light and easy to be run, in order to obey the WSN concept whereby nodes should be as energy-efficient as possible (thus spend little power in either computational or transmission tasks).

In this work, we have implemented a method for indoor localization, namely network fingerprinting, which has been employed in 802.11 networks, but has found little room in WSNs to date. One of the major obstacles toward its implementation in WSN is network-related: every node needs to effectively interrogate its neighbors to gather power measurements, and then must effectively route these measurements toward a server, which will return the most likely position of the mote. Both operations must be carried out in little time, and must create little overhead traffic in the network. In this thesis, we have addressed these issues by implementing a localization module based on fingerprinting, where all phases of the procedure are designed to be lightweight. We will describe the approach, the tools that supported the development of the system, and the benefits and weakness of the localization algorithm. Experimental results have been obtained using part of the testbed deployed in the department of information engineering.



# Chapter 1

## Fingerprinting Method and WSN

### 1.1 Fingerprinting Method

When we create programs for WSN we must face two challenging problems: we must write power and computationally efficient algorithms. The scenario is the following: we have a network of known motes (anchor nodes) and a new mote (mobile node) comes in the network: using the Fingerprinting method it must be able to discover its geographical position using the power readings of the messages sent by the anchor nodes. This power reading is called *RSSI* (received signal strength indicator).

We can basically divide this method in two distinct phases: Measurement and Localization.

#### 1.1.1 Measurement Phase

In this phase we must take the “fingerprint” of many locations in the area: standing in one point of the building we ask for messages to the anchor nodes. Using the RSSI measurements we can form the signature of the location, the signature is an array containing the RSSI reading for each Anchor that can receive the request. We gather these arrays for many points in each room of the building and all the measurements are stored in a database in a server. This phase must be repeated each time we’ve changes in the building: new furniture, new walls, etc. At the end of this procedure we have a table in the database containing many measures structured as follows:

<b>x</b>	<b>y</b>	<b>z</b>	<b>rss_i</b>	<b>id_anchor</b>
0.00	0.00	0.00	-12	123

Where the (x,y,z) values are the coordinates of the measured position. The **rss\_i** field contains the average of many measures taken in the same position from the same node, in fact during this phase is very important to get as much measurements

as possible for weakening the effects due to random channel variation.

### 1.1.2 Localization Phase

The mobile node asks for messages to the anchor nodes, they answer, the mobile measures the rssi and sends the data acquired to the server. The server takes that new “fingerprint” and search for the most similar set of measurements stored in its database. When that set is found the server answers to the mobile. It’s now clear the most significant advantage of fingerprinting: we can relay on a server for heavy computations. However there are some challenging problems: when the anchors reply too many collisions can take place. When this event occurs we assist at the destruction of many messages: thus the mobile node will get not enough readings for a correct localization. Avoiding this with traditional methods like random back-offs can make the localization phase a long process. Secondly, an other problem is the communication with the server, sending large amount of data can easily consume the low power resources on the mobile node, so it’s important to send to the server enough measurements, allowing a correct localization, on the other hand is desirable to not send too much data. The dimensioning of this trade-off is an other problem. The last problem is the communication in an heterogeneous network like ours: we have messages between motes that uses wireless protocol (in our case IEEE 802.15.4), communication with computers (when a mote request its location sending measurements) and possibly communication between computers (if the gateway that “translates” the message coming from motes and the database don’t reside on the same PC).

### 1.1.3 Other proposed methods

Fingerprinting method is not the only way for solving the indoor localization problem, the basic idea is the same: a mobile node measures the signal coming from other nodes and estimates its current position. Other techniques use other kind of measurements on the received signal instead of using the RSSI measurement, for a more exhaustive list of the used methods please refer to [2].

1. *TOA - Time of Arrival*: This method uses a simple idea: the distance between two nodes is directly proportional to the propagation time, if the mobile node has knowledge of this time, it can calculate the distance that the signal has run, if it is able to collect three measurements from three different nodes it can triangulate its position. This system has a very important drawback, in order to compute the propagation time the receiver has to precisely know the instant in which the message was sent by the Anchor, so all the nodes in the network must be synchronized. For solving this issue many algorithms were proposed (CN, RWGH).
2. *TDOA - Time Difference of Arrival*: The basic idea in this case is to measure the difference of the arrival time of the messages coming from the anchor

nodes. In this case we use couples of anchor, say  $(x_i, y_i, z_i)$  the coordinates of the first and  $(x_j, y_j, z_j)$  the coordinates of the second. The position  $(x, y, z)$  of the mobile node must obey at the following equation:

$$R_{i,j} = \sqrt{(x_i - x)^2 + (y_i - y)^2 + (z_i - z)^2} - \sqrt{(x_j - x)^2 + (y_j - y)^2 + (z_j - z)^2} \quad (1.1)$$

where  $R_{i,j}$  is the range differences between the anchor nodes, this value can be estimated and it is the constant of our problem. In other words the point  $(x, y, z)$  must lie in a hyperboloid with constant range difference  $R_{i,j}$ . One can estimates the position using the intersections of these hyperboloids.

3. *RTOF - Roundtrip Time of Flight*: Using this method we have to measure the time of flight from the mobile to the anchor and back. Using this method we have less stringent synchronization requirements respect to the TOA method, but we have to face a new issue: the elaboration time of the anchor can be difficult to estimate, in particular if the localization service is only one among many services, thus on an Anchor node the task that elaborates the response can be interrupted from other tasks.
4. *POA - Phase of Arrival*: This method is also called “Received Signal Phase Method”. The idea is measuring the phase of the received signal for estimate the position. Suppose that the Anchor nodes transmit pure sinusoidal signals with zero phase offset and the same frequency  $f$ . We will receive

$$s_r(t) = e^{j2\pi f(t+t_0)} \quad (1.2)$$

but  $t_0$  is related to the distance between the nodes, more precisely  $t_0 = D_i/c$  where  $c$  is the speed of light. If we re-organize (1.2) we obtain

$$s_r(t) = e^{j(2\pi ft + \phi_0)} \quad (1.3)$$

where  $\phi_0 = 2\pi ft_0$ . Using that formulas we can obtain the distance information from the phase, so we can apply both TOA or TDOA methods to phase.

At the end of this overview a question arise: why have we chosen fingerprinting instead of these methods? Because they reach good performance when there is an high probability to find a LOS (Line of Sight) between anchor and mobile nodes, so they are suitable for outdoor localization not for indoor localization, which is an environment that suffer by multipath and other problems. RSSI based method such as fingerprinting has shown good performance in such environment so we have chosen this method.

#### 1.1.4 Algorithms for the determination of the position

In the Localization phase, once we have obtained the “fingerprint” of the unknown location, we have to search for the most similar stored fingerprints and using them

we can guess the position of the mobile node. We present various methods for doing that:

1. *Probabilistic Methods*: This method considers positioning as a classification problem. Assuming that we have  $n$  location candidates for the mobile node and  $s$  is the vector containing the RSSI measurements obtained during the localization phase. Defining  $P(L_i|s)$  as the probability of the mobile to be in the location  $L_i$ , the following decision rule can be obtained:

$$\text{Choose } L_i \text{ if } P(L_i|s) > P(L_j|s) \quad \forall i, j = 1, 2, 3, \dots, n \quad j \neq i \quad (1.4)$$

we can now write  $P(L_i|s) = P(s|L_i)P(s)/P(L_i)$ , using Bayes rule, if we suppose that no prior knowledge about the position of the node is available, ( $P(L_i) = P(L_j) \forall i, j$ ) we can elaborate 1.4 and we obtain

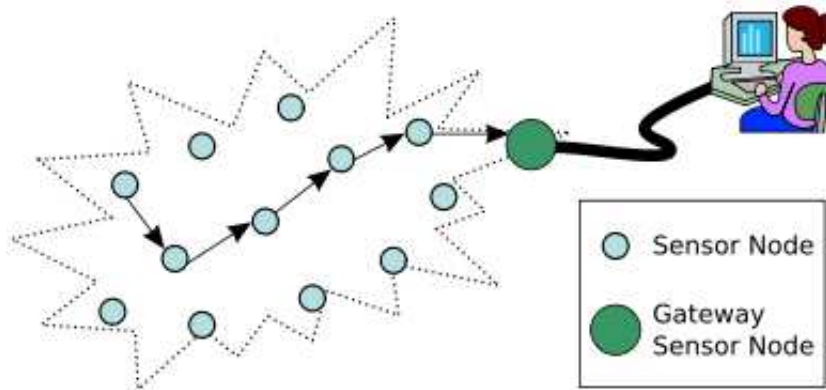
$$\text{Choose } L_i \text{ if } P(s|L_i) > P(s|L_j) \quad \forall i, j = 1, 2, 3, \dots, n \quad j \neq i \quad (1.5)$$

Using that formula and an appropriate statistical model of the probability one can devise the position of the mobile.

2. *k-NN (k nearest neighbor)*: In this method we search  $k$  known fingerprint, that were obtained during the Measurement phase, that are the closest to our measurements made during the Localization phase. After that we can average these locations obtaining an estimation of the mobile location. The distance is usually defined as euclidean distance between RSSI readings. We have used this method since it is the less computational complex algorithm. It is also the easier to implement since no probability models are required. We have used the *1-NN* version since we are interested in a coarse positioning while the hardware on the nodes and the transmission powers used do not allow finer determination of the position.
3. *Neural Networks*: In this system a neural network is used, it is trained during the Measurements phase and after that, during the Localization phase, the knowledge that the network has obtained during the first phase is used .
4. *SVM (Support Vector Machine)*: is a new and promising technique for data classification, machine learning and statistical analysis.
5. *SMP (Smallest M-Vertex Polygon)*: Suppose a total of  $M$  anchor nodes that reply to the request of the mobile node. Suppose that every anchor replies more than one time. Using one or more of the replies of one anchor we can estimate a position (say  $(x_i, y_i, z_i)$ ), we can repeat this for every node finding a M-vertex polygon. We can repeat this process choosing other replies from the anchors, so we can find more than one M-vertex polygon, once we have created enough polygons we can choose the polygon with shortest perimeter, averaging its coordinates we can estimate the position of the mobile.

## 1.2 WSN

We have already discussed about WSN in the introduction, now we are going to discuss the topic in a more detailed manner. WSN stands for “Wireless Sensors Networks”, it is a common acronym used to indicate a wireless network of nodes called motes, each mote is an autonomous electronic device. These networks can be used for environmental control, data acquisition, etc. Every mote is connected to the others via wireless communication systems.



### 1.2.1 Main Characteristics of a Mote

There are many kinds of motes, but we can recognize some common drawbacks

- **Low bitrate of the wireless connections**
- **Low computational power**
- **Low energy consumption:** a mote must work for much time without requiring battery change.

but we can list some common advantages as well:

- **Fault Tolerance:** usually WSN consist of many nodes, so if one or more nodes fails, if there are dynamic algorithms that could overcome the problem, the network can continue its duty.
- **Cooperation between nodes:** a mote has a ridiculous amount of computational power but if many nodes can cooperate they can work out heavy computations.

Motes has very different architecture but we can indicate some common characteristics at this level too: a node basic architecture is illustrated in Fig. 1.2.1.

- **Memory:** The memory is usually divided in RAM (Random Access Memory) and ROM (Read Only Memory). The RAM contains the data that the program

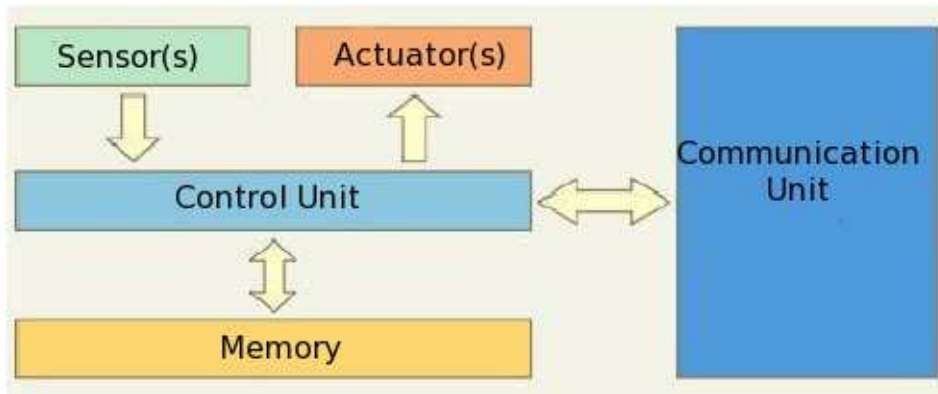


Figure 1.1: Basic Architecture of a Node, from [3]

elaborates. The ROM contains the instructions that the nodes have to follow (the program logic). The term ROM is usually used in publications but can be misleading, the ROM content can be modified (for node re-programming), thus it's possible to write in ROM but this isn't usually done during normal node activities.

- **Control Unit:** It's the processor of the node, its main tasks are the control of the resources of the node and to work out the computations needed by the program. On current nodes are used various kinds of energy efficient processors (usually microcontrollers).
- **Communication Unit:** It's the unit that is in charge to transmit the data to other nodes. Usually the bitrate is very low compared to computer networks bitrate but this is necessary because the transmission and the receipt are energy expensive operations.
- **Sensor(s):** They allow the node to gather information from the "external world" such as temperature, humidity, etc.
- **Actuator(s):** They allow the node to modify the "external world", for example we can have a mechanical arm that can open a window if the temperature in the room rises too much.

After the examination of a general architecture we can now introduce the architecture of the node that has been used in this work: TelosB [4].

TelosB main characteristics:

1. IEEE 802.15.4/ZigBee compliant, RF transceiver
2. 2.4 to 2.4835 GHz, a globally compatible ISM band



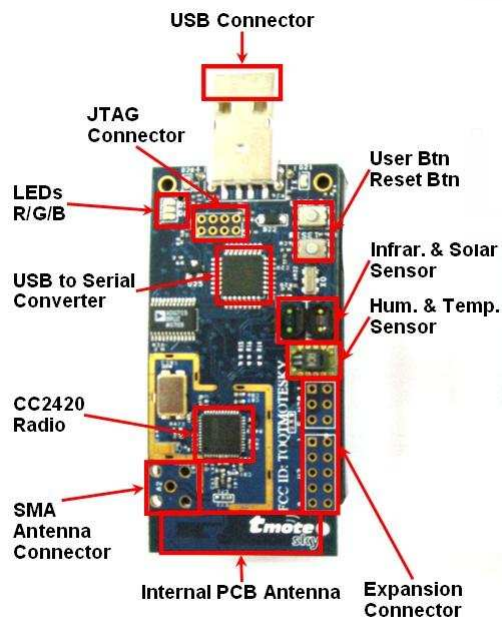


Figure 1.2: MoteIV TelosB

3. 250 kbps data rate
4. 1Mhz TI MSP430 microcontroller with 10kB RAM
5. 45Kbyte ROM from program storing
6. Integrated onboard antenna
7. Data collection and programming via USB interface
8. TinyOS compatible
9. Integrated Temperature, light and humidity sensor



## Chapter 2

# The IoT

In this section we are going to discuss the tools that we have used to create the software needed for this work, this application is a prototype of what, in the future, will be a IoT *Internet Of Things* application. This term indicates the possibility, in the near future, to interconnect machines and allow them to communicate with each other, enabling the so-called *Machine-to-machine communications*. In Fig. 2.1 we can see an example of an IoT application, in which the motes can communicate with a gateway, which, in turn, communicate using classical internet services with a server, but it can also communicate with PCs or other systems connected to the network. In this chapter we will analyze the tools involved in such an application,

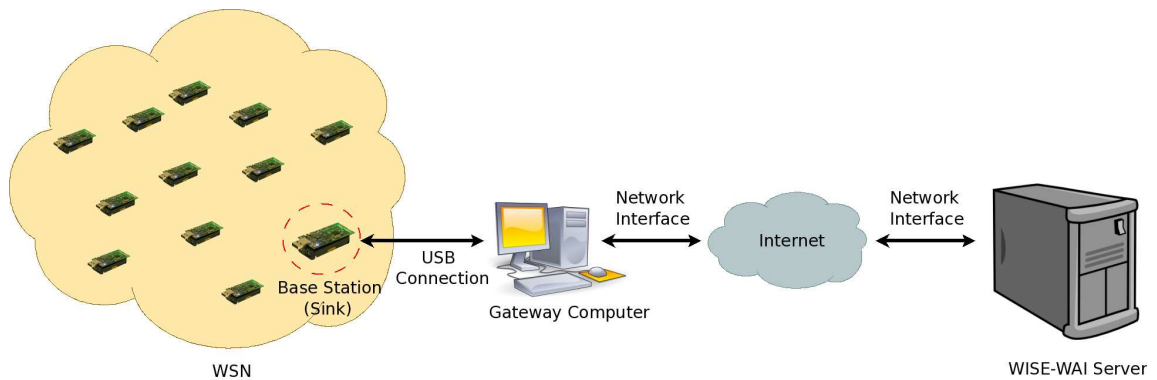


Figure 2.1: Example: The IoT Application

we will examine how to program the motes, how to allow the gateway to interact with the motes and with the Internet. We will also examine the tools needed to create the web-based interface of our application.

## 2.1 Operative Systems for motes

In the past chapter we have examined the constraints that we have to deal with when we use a mote. In such a constrained environment the operative system has to fulfill the following tasks:

1. **Work with limited resources:** It does not have to add too much overhead to the mote computations.
2. **Concurrency Handling:** The mote is created to control the “external world”, which generates asynchronous events, thus it is necessary for the mote to carry out more than one task at once.
3. **Reusability:** Once we have written a program for a kind of mote, no or few changes are needed to the code to run it on an other kind of mote.
4. **Simple hardware control:** We cannot deal directly with the enormous ensemble of devices that a mote can use, thus it’s necessary to provide easy tools for commanding the hardware. On the other hand that tools cannot be too general, because every device has its own features: for example it is useful to power-off some devices when unused. With a program that is aware of the hardware we can have a less portable but more efficient application.
5. **Low power consumption:** In WSNs the deploy phase is very tedious so it is unreasonable to often redo it only for recharging the motes, the programs used have to consume the least possible power so we have not to frequently re-deploy the network.

TinyOS [6] is a OS (Operative System) that tries to solve these problems.

## 2.2 TinyOS

TinyOS is an Open-Source OS, created by the Berkley University (California). It’s written in nesC [7] (Network Embedded System C). nesC is a C dialect that has been developed specifically for TinyOS.

### 2.2.1 Application Specific and Event Driven

Application Specific and Event Driven are the main concepts that have driven the TinyOS development.

**Application Specific:** The application and the operative system are compiled together, thus only the parts of the OS that are used in the application are compiled and installed on the mote. For example, if we don’t need the led support for our application, the parts of the OS that administrate the leds aren’t compiled.

**Event Driven:** A TinyOS program is a connected set of components (we will return on this concept further in this work) every component asks for services to other components. A component could wait for the reply, but this is a waste of resources: if an other component needs the CPU, a context-switch must take place, furthermore in such a difficult environment we cannot use a complex scheduler that allows us to keep track of the state of the process. TinyOS uses a simple solution: when a component asks for services it ends its execution and it restart when the data are ready (i.e. when an event take place), thus the request for the service and its reply are decoupled. When we say “event driven” we intend an OS when the execution is driven by the events.

### 2.2.2 Developing an Application using TinyOS

We have already said that a program in TinyOS is a connected set (graph) of components. It’s time to better clarify that statement. A component is an autonomous entity with its own memory, statically allocated at compile time. Every components asks for services to other components using *commands* and reply to the request coming from the other components signaling *events*. Events and commands that are consistent between them are grouped in *interfaces*. A classical example of interface is `Read`, we will use it to explain better the interactions between nodes, but first it’s necessary to introduce how an interface is used by a mote. A component can **provides** or **uses** an interface. When a component **provides** an interface it means that it can execute the commands of the interface (thus it has to implement them). When a component **uses** an interface it means that it can query for data other components that **provides** that interface, thus it has to implement the events that are the action performed on the provided data.

### 2.2.3 Example

First of all we can take a look to the `Read` interface

```
interface Read<val_t> {  
  
    command error_t read();  
  
    event void readDone( error_t result, val_t val );  
  
}
```

Now suppose that we have two components, the first `AnchorC`, **uses** the interface, the second, `HamamatsuS10871TsrC` is the component that wraps the hardware device measuring the infrared light, thus it **provides** the `Read` interface. If the `AnchorC` wants to get the reading it must issue the command `read()`, it returns immediately

a value that inform the component if the request is started. When the operations worked out by `HamamatsuS10871TsrC` component are terminated it signals the event to `AnchorC` that will elaborate the value of `result` and `val` in the implementation of the event `readDone()`. In this case `result` contains the error state of the request (it is `SUCCESS` if no error occurred during the request) and `val` contains the actual read value.

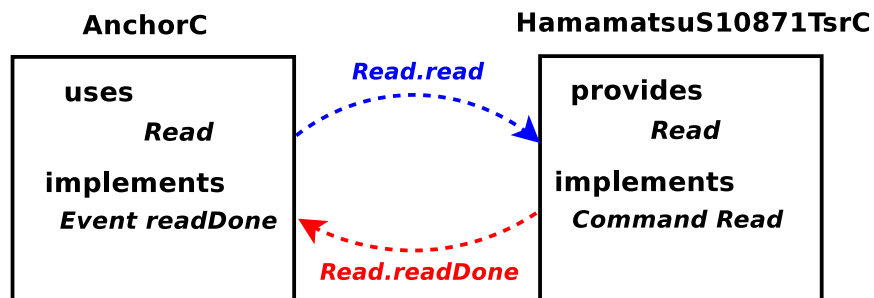


Figure 2.2: Example: Reading a value

## 2.2.4 Configuration

We have introduced the interface concept but we have not explained which connect a user with a provider: this task is fulfilled by the *configuration*. This operation is called wiring, thus in the configuration (`AnchorAppC`) we will find the following lines of code:

```

configuration AnchorAppC{
implementation
{
components new HamamatsuS10871TsrC() as InfraRed;
components AnchorC;
.....

AnchorC.readInfra->InfraRed.read;
}
  
```

the first and the second line create the components that have to be connected. The last line connect the interface `readInfra` used by `AnchorC` with the interface `Read` provided by `InfraRed` which is an *alias* (defined with the keyword `as`) for `HamamatsuS10871TsrC`. Even `readInfra` is an *alias* of the `Read` interface. The alias are used for avoid confusion (for example when in a module we have more than one `Read` interface). A *configuration* is written in a particular file that has the same

name of the configuration. The *configuration* can only contain wiring and export instructions (which will be explained later), hence when is the implementation of commands and events located? It's located in *modules*.

### 2.2.5 Modules

Modules are the principal way to create components, they're written in a file that has the same name of the module and they can be divided in two parts, the first part is the *definition* of the module, here we can find the used and provided interfaces, continuing our example we could find something like:

```
module AnchorC
{
  ...
  uses interface Read<uint16_t> as readInfra;
  ...
}
```

In the second part (the *implementation*) we can find the implementation of the commands and the event, (in our example we only use interfaces so we only implement the events).

```
implementation
{
  ...
  event void readInfra.readDone(error_t result, uint16_t val)
  {
    if(result==SUCCESS)
    {
      elaborate val
    }
    else
    {
      deal with the occurred problem
    }
  }
  ...
}
```

In this section, in the body of a command or an event, we can also issue the command using the keyword `call`, in our example, for starting the reading process, we should write `call readInfra.read();`.

## 2.2.6 Configuration as Components

An other way to create a component is using the Configurations: this is how we can create libraries for others programmers. Suppose that we have two components, one creating data (**CreatorC**) and an other encrypting them (**EncryptC**). We would like to create a component (**DataEmitterC**) containing both of them but with an interface for allowing something else to get the encrypted data. For this purpose we can create the interface **DataReady**, in Fig. 2.3 we can see the example, all the interfaces indicated (**DataOut**,**DataIn**, etc.) are alias of **DataReady**. When the data are ready **CreatorC** (which provides **DataOut**) alert **EncryptC** (which uses **DataIn**) and it, which provides **EncData**, alert the other components. We want to wrap all this complexity in one component that provides only one interface **EncDataOut**.

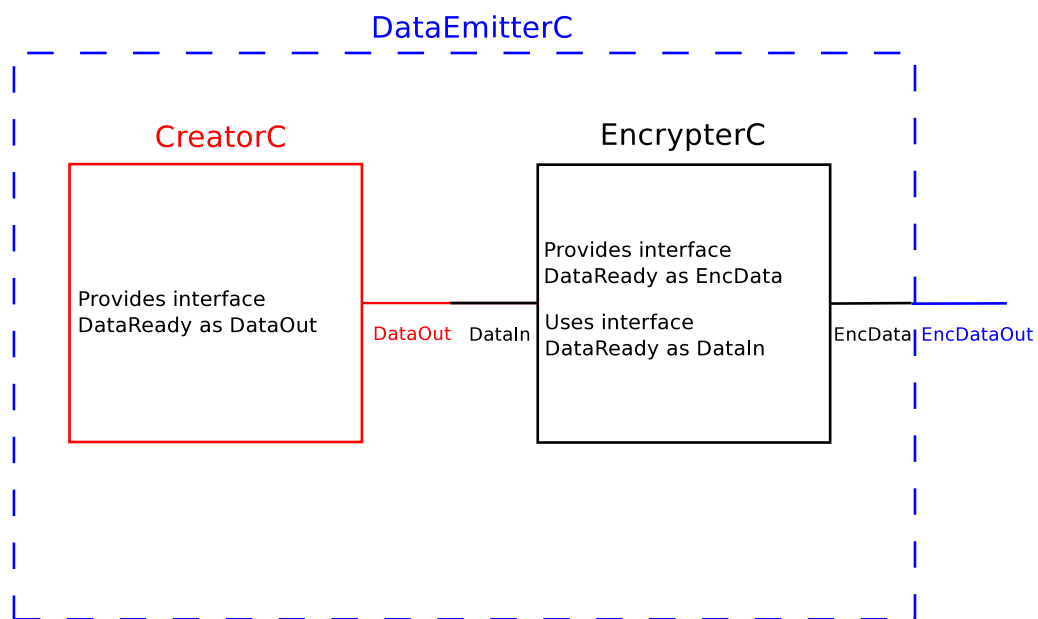


Figure 2.3: Encrypter example

We can now take a look at the code of **DataEmitterC**

```
configuration DataEmitterC
{
    provides interface DataReady as EncDataOut;
}
implementation
{
    components CreatorC, EncrypterC;
```



```
    EncrypterC.DataIn->CreatorC.DataOut;  
    EncDataOut=EncrypterC.EncData;  
}
```

As we can see the configuration can't implement commands or events so the only thing that it can do is *exporting* an interface belonging to one of its components. This task is fulfilled using the *export* operator which is =.

### 2.2.7 Tasks

Events and commands have problems when they have to deal with long computations. TinyOS gives us a tool for dealing with this problem: the *tasks*, they are a deferred function call, in other words when we issue a task (with the keyword `post`) the scheduler isn't obliged to execute it immediately, it can be executed when the scheduler is free from other works.

A more detailed introduction to TinyOS programming can be found in [8].

## 2.3 WSN Networks Aspects

In WSN the low power consumption is one of the basic constraints, thus a power-efficient protocol for wireless communication is needed. One of the most promising standards in this scenario is IEEE 802.15.4, which is developed for WPAN (Wireless Personal Area Networks). WPAN are formed by devices with constraints similar to the motes, so this protocol could be a good choice. It is important to say that IEEE 802.15.4 isn't ZigBee. ZigBee relies on IEEE 802.15.4 implementing the upper layers, see Fig. 2.4.

IEEE 802.15.4 is developed by IEEE 802.15.4 Task Group, which has defined the PHY (Physical) and MAC (Medium Access Control) layers.

### 2.3.1 Network Architecture

This standard divides the connected devices into two categories: FFD and RFD. FFD stands for *full-function devices*, while RFD stands for *reduced-function devices*. FFD can coordinate one or more RFD, a RFD can be associated to only one coordinator at time. RFD is intended for very simple tasks and can communicate only with the associated coordinator, on the other hand the FFD can communicate with RFDs and FFDs too. With this particular architecture the resources needed by the RFDs are very low. A WPAN is usually formed by at least two devices in the same POS (*Personal Operating Space*). The POS is the area that a FFD can cover (usually 10 meters or similar distances). One of the FFD present in the area must be the coordinator. In one POS the network can be organized in a peer-to-peer topology or

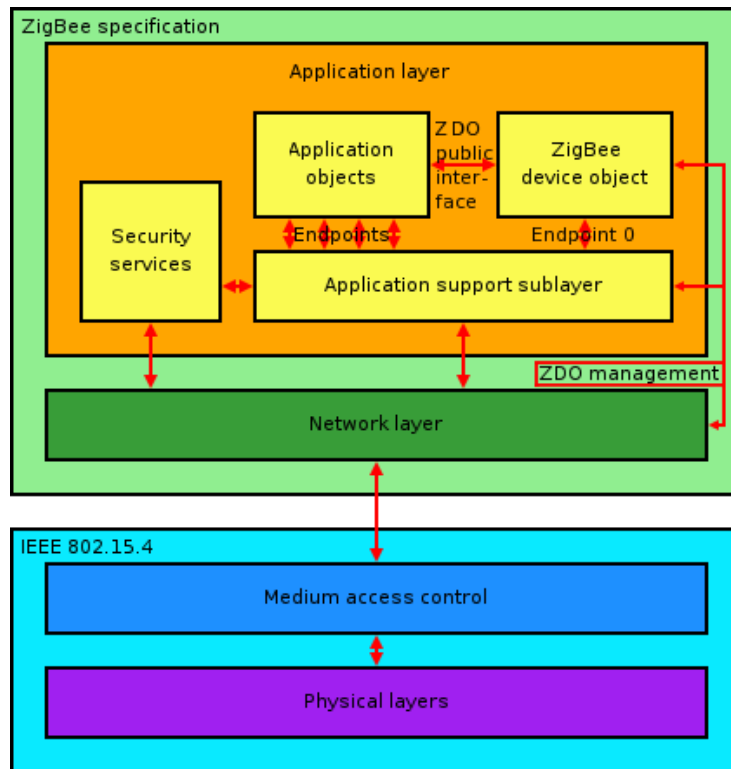


Figure 2.4: ZigBee and IEEE 802.15.4

in a star topology. In the star topology we have only one coordinator and the other nodes communicate with it. The coordinator of the PAN chooses a PAN identifier that is a unique identifier shared by all the nodes coordinated by it. Every star has its own identifier and its own coordinator, if a simple node wants to communicate with a node belonging to another star it has to pass through its coordinator which sends the messages to the coordinator of the receiver. In the peer-to-peer topology we have in one PAN more than one FFD and these nodes can communicate with each other. The peer-to-peer topology allows the creation of a particular type of network: cluster-tree network. This system has two levels of coordinators: the first level is PAN coordinator and it is unique in the tree. The coordinator communicates with the CLHs (Cluster Heads), each CLH is the coordinator of a particular cluster identified by a CID (Cluster Identifier). The PAN coordinator also acts as coordinator of the cluster with CID=0.

IEEE 802.15.4 provides two kinds of address for the nodes: the first one is a unique 64-bit address, the second is a 16-bit address assigned by the coordinator during the association event.

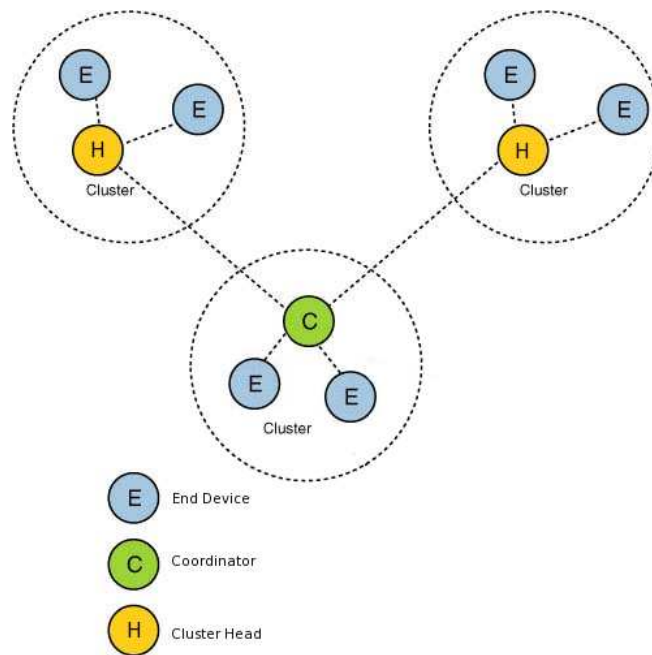


Figure 2.5: An Example of IEEE 802.15.4 Cluster Tree



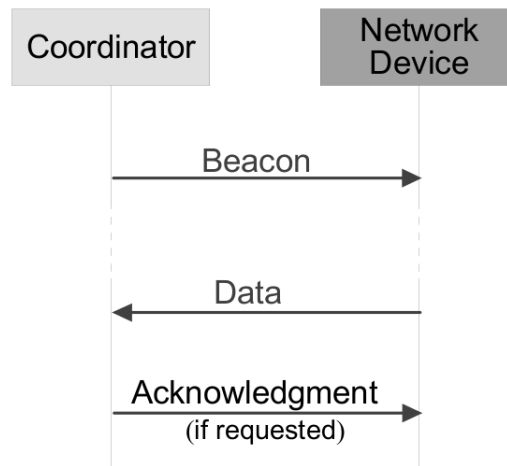


Figure 2.7: Device - Coordinator beacon enabled communication, from [25]

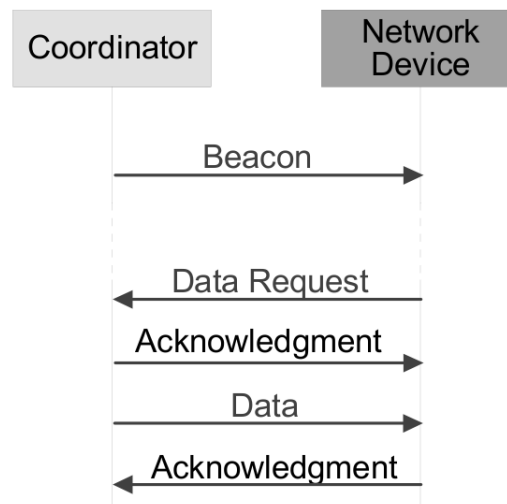


Figure 2.8: Coordinator - Device beacon enabled communication, from [25]

a Coordinator, thus it uses the Peer-to-peer data transfer option of the standard ([25] p. 21), this option is provided in the standard but it isn't described, giving freedom to the implementer.

In the previous described modes four kinds of frame are used:

- Beacon Frame, used by a coordinator to transmit beacons.

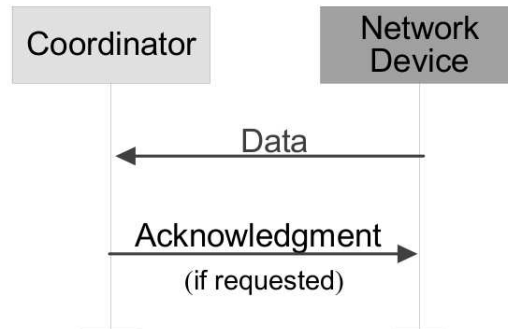


Figure 2.9: Device - Coordinator non-beacon communication, from [25]

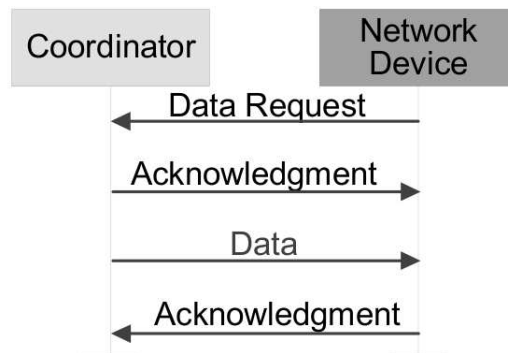


Figure 2.10: Coordinator - Device non-beacon communication, from [25]

- Data Frame, used for all data transfers.
- Acknowledgment Frame, used for confirming successful frame reception.
- MAC Command Frame, used for handling all MAC commands transfers, for example the Association with the coordinator.

We can now give a brief overview of the CSMA-CA unslotted algorithm used in 802.15.4, presented in Fig. 2.11. The algorithm must be performed every time a node wants to send a message, for ensuring a low collision probability. A *transmission attempt* is the time that elapses between the arrival of the packet from the upper layers and its actual dispatch (or discard in case of failure). For each *transmission attempt* the node maintains two variables: NB and BE (other variables are used in the slotted version). NB is the **N**umber of **b**ackoff, i.e. the number of times the CSMA-CA algorithm was required to backoff while attempting the current transmission. BE is the backoff exponent, which is related to how many backoff periods a device shall wait before attempting to assess a channel.

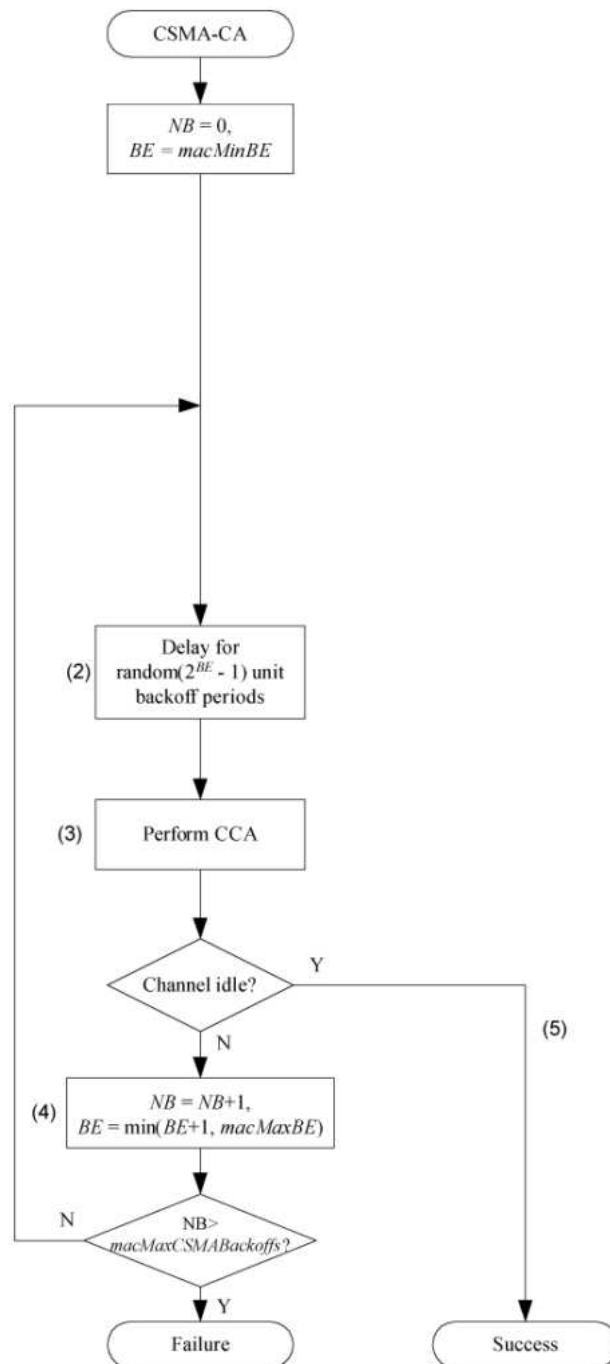


Figure 2.11: CSMA-CA algorithm, adapted from [25]

The algorithm initialize BE and NB and goes to [step (2)]. The MAC sublayer waits until a random number of complete backoff periods in the range 0 to  $2^{BE}-1$

are elapsed and then requests that the PHY perform a CCA [step (3)] which starts immediately. CCA stands for *Clear Channel Assessment* and it is the procedure used to determine if the channel is used or not from other nodes. CCA can use three method for determining the state of the channel:

1. **CCA Mode 1:** Energy above threshold. CCA reports a busy medium detecting energy above a determined threshold (ED threshold, energy detection threshold).
2. **CCA Mode 2:** Carrier sense only. CCA reports a busy medium only upon the detection of a signal compliant with the standard. This signal may be above or below the ED threshold.
3. **CCA Mode 3:** Carrier sense with energy above threshold. CCA reports a busy medium using a logical combination of
  - Detection of a signal with the modulation and spreading characteristics of the standard.
  - Energy above the ED threshold.

The logical operator may be AND or OR.

If the channel is assessed to be busy [step (4)], the MAC sublayer increments both NB and BE by one, ensuring that BE shall be no more than *macMaxBE*. If the value of NB is less than or equal to *macMaxCSMABackoffs*, the CSMA-CA algorithm returns to [step (2)]. If the value of NB is greater than *macMaxCSMABackoffs*, the CSMA-CA algorithm terminates with a channel access failure status.

In 2.7 we can see that if a message is correctly received usually an ACK (Acknowledgment) is sent, this option can be deactivated, setting the 5<sup>th</sup> bit in the FCF (Frame Control Field) to 0, see Fig. 2.12, for further details on FCF field see [25] p.139.

Bits: 0-2	3	4	5	6	7-9	10-11	12-13	14-15
Frame Type	Security Enabled	Frame Pending	Acknowledge request	Intra PAN	Reserved	Destination addressing mode	Reserved	Source addressing mode

Figure 2.12: FCF field, from [12]

### 2.3.3 IEEE 802.15.4 Physical (PHY)

IEEE 802.15.4 supports 4 kind of PHY layer:

- An 868/915 MHz direct sequence spread spectrum (DSSS) PHY employing binary phase-shift keying (BPSK) modulation



- An 868/915 MHz DSSS PHY employing offset quadrature phase-shift keying (O-QPSK) modulation
- An 868/915 MHz parallel sequence spread spectrum (PSSS) PHY employing BPSK and amplitude shift keying (ASK) modulation
- A 2450 MHz DSSS PHY employing O-QPSK modulation

The CC2420 chip implements the last PHY, reaching a 250 Kbit/s bitrate, in Fig. 2.13 we can see the rates and the modulation used.

PHY (MHz)	Frequency band (MHz)	Spreading parameters		Data parameters		
		Chip rate (kchip/s)	Modulation	Bit rate (kb/s)	Symbol rate (ksymbol/s)	Symbols
868/915	868–868.6	300	BPSK	20	20	Binary
	902–928	600	BPSK	40	40	Binary
868/915 (optional)	868–868.6	400	ASK	250	12.5	20-bit PSSS
	902–928	1600	ASK	250	50	5-bit PSSS
868/915 (optional)	868–868.6	400	O-QPSK	100	25	16-ary Orthogonal
	902–928	1000	O-QPSK	250	62.5	16-ary Orthogonal
2450	2400–2483.5	2000	O-QPSK	250	62.5	16-ary Orthogonal

Figure 2.13: IEEE 802.15.4 supported rates, modulations and frequencies

The binary data are spreaded and then passed to a O-QPSK modulator which modules the signal allowing its transmission via the wireless media, see Fig.2.14.

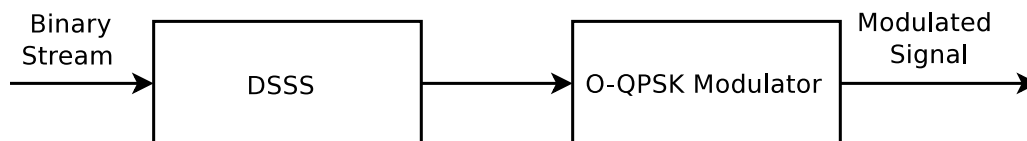


Figure 2.14: IEEE 802.15.4 modulator architecture

The DSSS is obtained through a spreading sequence that maps 4 data bit ( $b_0b_1b_2b_3$ ) in 32 bits ( $c_0c_1 \dots c_{30}c_{31}$ ), thus a byte is mapped in 64 bits, refer to [25] p. 47 for the complete list of the spreading sequences. A sequence of 4 bit is used to select one

of 16 sequences to be modulated, this is the explanation of the “16-ary orthogonal” that appears in Fig. 2.13 in the “Symbols” column.

The obtained stream is modulated using a O-QPSK modulator, O-QPSK stands for *offset quadrature phase-shift keying* it can be indicated also by S-QPSK (Staggered QPSK), the even chips  $c_0, c_2, \dots$  are mapped in the  $I$  channel, the odd chips  $c_1, c_3, \dots$  in the  $Q$  channel. In a normal QPSK modulator (Fig. 2.15) we can see the usual bitmap, then the pulse shaping and the modulation with two carriers in quadrature, in Fig. 2.16 we can see a O-QPSK modulator, which is simply a QPSK modulator with a delay block in the  $Q$  branch of the modulator.  $T$  is the symbol period (in our case it is the chip period) thus we add a delay of an half of a chip period obtaining a shift of the  $I$  and  $Q$  channels (an example is provided in Fig. 2.17).

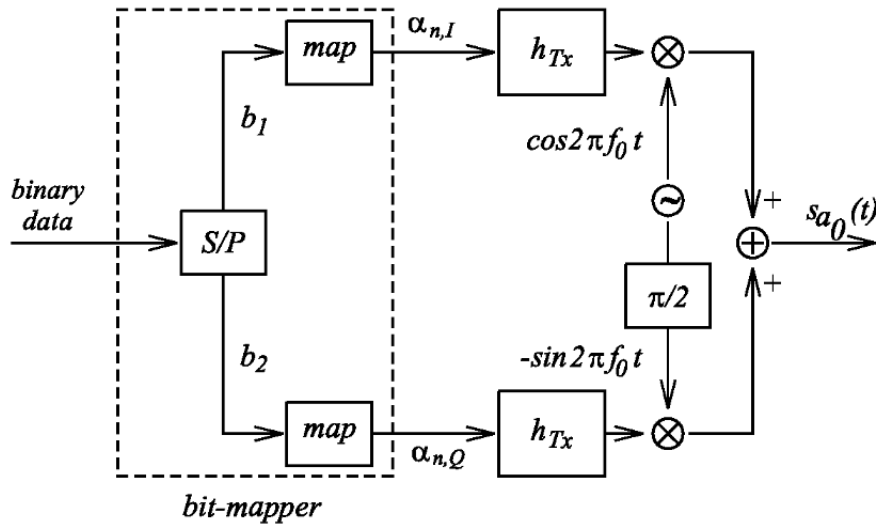


Figure 2.15: QPSK modulator, from [26]

The standard defines many channels that divide the bands, the channels that divide the band that we are using (2.4 GHz band) are 16 and they are numbered with a index ( $k$ ) going from 11 to 26, the center frequency of each channel can be computed as follows:

$$F_c = 2450 + 5(k - 11) \text{ Mhz} \quad (2.1)$$

In Fig.2.18 we can see the channels.

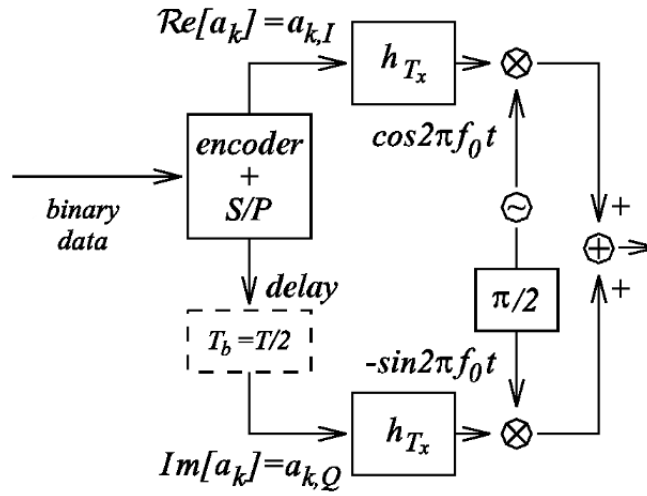


Figure 2.16: O-QPSK modulator, from [26]

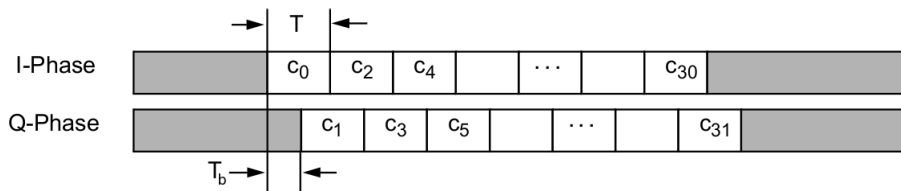


Figure 2.17: Bit allocation and offset in the I and Q channels

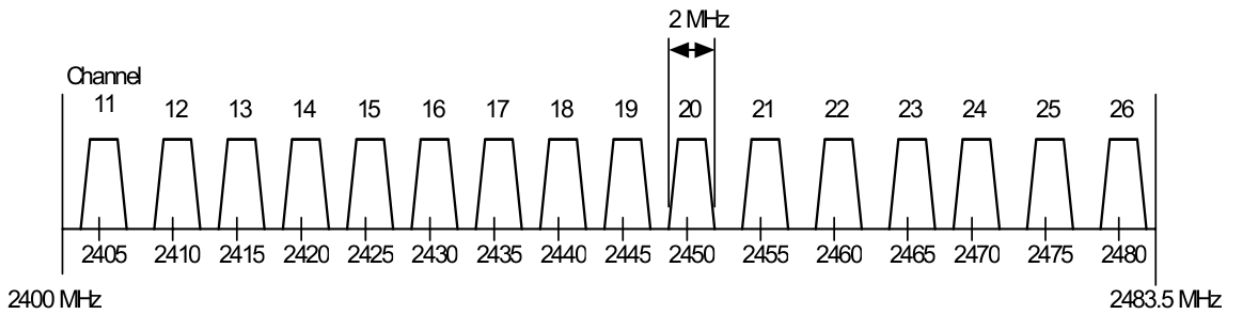


Figure 2.18: IEEE 802.15.4 channels for the 2.4 Ghz band

### 2.3.4 IEEE 802.15.4 and TinyOS

TinyOS, for wireless communication, relies on *Active Messages*, which are contained in a IEEE 802.15.4 data frames. They use the 16 bits short address for identification, this address is the TOS\_NODE\_ID which identifies a mote in a network and it is decided at compile time. The address 0xFFFF is the broadcast address. Usually in literature

when we indicate an *Active Message* header the 802.15.4 header is included.

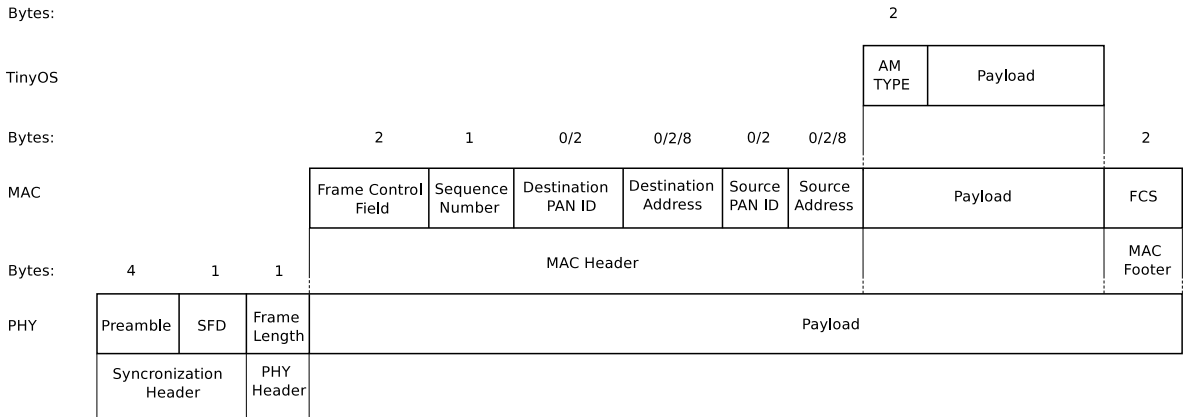


Figure 2.19: IEEE 802.15.4 Frame and TinyOS field

As can be seen from Figure 2.19 TinyOS simply adds one more field: the `AM_TYPE` field which is a identifier of the specific application that generates the packet, so the operative system signals the receipt of the message only to the applications that are interested in that packet. In the figure one can notice that some fields can have various possible dimensions, in TinyOS the source and destination addresses are setted to 2 bytes, note that only one of the PAN ID (destination) is setted, the other is not included in the frame. The destination PAN ID in TinyOS is called `TOS_AM_GROUP`, it allows more than one different network to share the same channel without interfering. In TinyOS for sending a packet is necessary to statically allocate a buffer of type `message_t`, this structure has space for the header and the payload. In the configuration file is necessary to declare a component of type `AMSenderC` in which we define the `AM_TYPE` of the packet that we want to send through it. This component provides an interface of type `AMSend`. We have used the `getPayload()` and `Send()` commands and the `sendDone()` event of this interface. `getPayload()` provides a pointer to the payload part of the message buffer, `Send()` issues the procedure for the dispatch of the packet. `sendDone()` is signaled when the packet is transmitted. When a mote receive a packet controls if it is addressed to it. If so a `Receive()` event is signaled to the `Receive` interface that is bound to that particular `AM_TYPE` (if such an interface does not exist the packet is discarded). The `Receive` event sends to the bound component a pointer to the payload of the packet thus is possible to access its various fields. It's also possible to access the fields of the header. The `Receive` interface is provided by the component `AMReceiverC`. The last aspect to be clarified is how the developer can define the payload of a packet. Clearly it is possible to modify or read bit by bit the payload area but this would be an error-prone behavior, the best practice in this case is to create a data structure describing the payload, such as:

```
typedef nx_struct{
    nx_uint16_t ActionCode;
    nx_uint16_t id;
    nx_uint8_t channel;
    nx_uint16_t nTimes;
}toBSpck_t;
```

These structures are written in a header file with extension `.h`. All the commands and events previously described that interact with the payload return a `void*` pointer, thus it is possible to use a cast for gaining access to the various part of the payload. Referring to the presented example above we can use the following instructions to fill the payload, (`buffer` is the `message_t` variable storing the packet).

```
toBSpck_t *payload =call AMSendBraccio.getPayload(&buffer, sizeof(toBSpck_t));
payload->ActionCode=1;
payload->id=2;
etc.
```

When receiving the packet we can do the same thing for inspecting the packet.

An other aspect to be clarified is how TinyOS allows the modification of the channel, it is possible to define the channel to be used at compile time using the `CC2420_DEF_CHANNEL` flag. During the execution of the program is possible to do the same thing using the `CC2420Config` interface provided by the `CC2420ControlC` component. The interface provides many functionality, such as enable/disable ACK, enable/disable address recognition, etc., however we are interested in few methods and events:

```
interface CC2420Config {

    command void setChannel( uint8_t channel );

    command error_t sync();

    event void syncDone( error_t error );

    command uint8_t getChannel();

    other events/commands
}
```

The `setChannel` command is the first utilized for the channel change, we have simply to invoke it providing the number of the new channel. For starting the actual channel change we have to invoke the `sync` command. When the change is complete the event `syncDone` is signaled, hence the system can start to send messages safely. `getChannel` gives the channel on which the node is currently sending and receiving data.

TinyOS allows deep modifications on the CSMA-CA mechanism, using the `RadioBackoff` interface provided by the `CC2420CsmA` component. Examining it we can note that the TinyOS implementation of 802.15.4 slightly differs from the standard, in Fig. 2.11 we can see that the max delay increases from attempt to attempt, conversely in TinyOS implementation are defined two backoffs time, *InitialBackoff* and *CongestionBackoff*. *InitialBackoff* is the longest backoff period, requested on the first attempt to transmit a packet. The long initial backoff period prevents the node from capturing the channel from other nodes by sending subsequent packet transmissions. *CongestionBackoff* is a shorter backoff period used when the channel is found to be in use. It has to be stressed however that these two backoff are random in the standard TinyOS implementation. We can modify these two values using the `RadioBackoff` interface.

```
interface RadioBackoff {

    async command void setInitialBackoff(uint16_t backoffTime);

    async command void setCongestionBackoff(uint16_t backoffTime);

    async command void setCca(bool ccaOn);

    async event void requestInitialBackoff(message_t * ONE msg);

    async event void requestCongestionBackoff(message_t * ONE msg);

    async event void requestCca(message_t * ONE msg);
}
```

We can analyze for example `requestInitialBackoff()` event (the behavior of `requestCongestionBackoff` is the same). When `CC2420CsmA` has to handle a message which is about to be sent, sets the backoff at its random value following the standard TinyOS algorithm, after that it signals the `requestInitialBackoff()` event, which is captured by every module wired to it, the event has only one parameter, a pointer to the message, allowing the modification of the algorithm on a per-packet basis. In the event the `setInitialBackoff()` command is called to modify the backoff interval. We can make an example for sake of clarity, we will set the `InitialBackoff` value at 0 if the type of the message is 12, the value will remain unmodified otherwise.

```
async event void RadioBackoff.requestInitialBackoff(message_t * msg)
{
    if(call AMPacket.type(msg)==12)
        call RadioBackoff.setInitialBackoff(0);
}
```

The `requestCca` event and `setCca` command work in the same way: if the value passed to the command is `TRUE` the CCA will be performed as usual, if `FALSE` is passed the CCA will not be performed and the channel will be assumed clear, thus the transmission will start after the backoff.

TinyOS provides a way to affect the Ack mechanism of the system. It has to be stressed that the CC2420 chip provides automatically Ack messages, however as stated in [27], this leads to false acknowledgments when the radio chip receives a packet it acknowledges its reception, but the microcontroller could never actually receive the packet. TinyOS by default disables hardware Ack and uses software Ack generated when the MCU receives the packet. The Ack mechanism can be affected by the `PacketAcknowledgments` interface provided by the `AMReceiverC` component.

```
interface PacketAcknowledgments {  
  
    async command error_t requestAck( message_t* msg );  
  
    async command error_t noAck( message_t* msg );  
  
    async command bool wasAked(message_t* msg);  
  
}
```

The first and the second allows respectively to set the Acknowledge Request bit to 1 or 0 in the FCF field in the header of the message passed as argument. The last command allows to determine if an Ack message was received in response to a previously sent message.

### 2.3.5 Tools for Mote-PC communication

The interfaces presented in the past section can be used for mote-mote communication but also for mote-PC communication. The only difference, for the user, is that instead of using `AMSenderC` and `AMReceiverC` components `SerialAMSenderC` and `SerialAMReceiverC` has to be used. The creators of TinyOS have developed tools for allowing the creation of programs running on PC but able to interact with motes connected via the USB interface which is also the mean used for programming the motes. These tools can be divided into two categories: *Translation* and *Connection* tools, for this work we have developed programs written in JAVA for the Mote-PC interaction, thus we will describe the procedures referring to this particular language. The *Translation* tool is `mig`: the packets sent via serial port need a data structure for their specification, but these structures cannot be handled by JAVA, it is necessary to write a class that describe the packet, allowing the JAVA program to handle these data. This translation work is automatically worked out by `mig` that translates the data structures into JAVA classes. These classes have getter and setter methods for the manipulation of the payload. The class created is a specialization of `Message`, a class provided in the JAVA TinyOS API.

The *Connection* tools are classes that allow the JAVA program to directly send and receive packets. The first one is the `MessageListener` interface, it has to be implemented by the application main class (the class implementing the `main()` method). This interface requires that the

```
public void messageReceived(int to, Message message)
```

method is overwritten. This method is invoked each time a packet (of every kind) is correctly received from the serial port. `message` is the received packet, for its manipulation it is necessary to do a cast to the correct class, in this case the JAVA operator `instanceof` may help. `to` is the receiver address. `MoteIF` is a class that represents a mote, we have an instance of it for each connected node to the PC. We have, now, to connect the class that represents the mote with the actual mote via serial port. This is done using the `PhoenixSource` class.

```
PhoenixSource phoenix;
phoenix = BuildSource.makePhoenix(source, PrintStreamMessenger.err);
mote= new MoteIF(phoenix);
```

where `source` is a String object formatted in the following way:

```
serial@device:speed
```

in our case, supposing that we are using the `/dev/ttyUSB0` port and a `telosb` we can write

```
serial@/dev/ttyUSB0:115200
```

115200 baud is the speed of the serial connection with the TelosB, but for standard architecture (TelosB, micaZ, etc.) we can simply write the name of the architecture:

```
serial@/dev/ttyUSB0:telosb
```

Finally, it is necessary to connect the `MoteIF` instance with the `main` class, this is done using the

```
registerListener(Message m, MessageListener l);
```

method belonging to the `MoteIF` class. `m` is the message that the actual mote will send, `l` is the object implementing the `MessageListener` interface, referring as usual to our example and supposing that we are inside the `main` class we should write:

```
mote.registrerListener(new toBSpck(), this)
```

where `toBSpck` is the class translated from the data structure and `this` is the pointer at the current object. The `MoteIF` class is also used for sending packets to the Basestation, indeed it provides the

```
send(int to, Message m)
```

method, where `to` is the receiver address and `m` is a `Message` object, which usually belong to a specialization of the `Message` class and it is created using `mig`.



## 2.4 6lowPAN

In growing WSN environment would be useful to have a network-layer protocol like IPv4 or better IPv6 and possibly a transport-layer protocol like UDP (TCP would be too expensive in WSN). One advantage in doing this would be, for example, a simpler communication with normal networks. We can't use standard IPv6 since the overhead coming from the header would dramatically decrease the available bit-rate and in nodes aren't available enough memory for the buffers needed for the storage or creation of the packets. 6lowPAN is an IETF specification for the communication in WSN via a compressed version of IPv6. 6lowPAN has to define basically three functions: the mapping between a IPv6 address and the 802.15.4 addresses, an adaptation layer for the fragmentation of the packets and finally mechanisms for header compression.

### 2.4.1 Addresses

Each IEEE 802.15.4 device has an unique 64 bit identifier, and this address can be used for the creation of a IPv6 address (see [11]), but we can do the same thing using the `TOS_NODE_ID` (the 16 bit short address), creating a 48 bits pseudo address as specified in Fig. 2.20, where the PAN ID is specified during the association phase by the coordinator. In current TinyOS implementation the PAN ID (`TOS_AM_GROUP`) is decided at compile time because there is no coordinator nor association event.

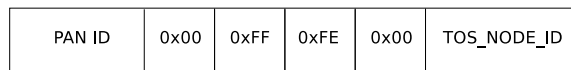


Figure 2.20: Pseudo 64 bit address

### 2.4.2 Adaptation Layer

IPv6 specifies a MTU (Maximum Transmission Unit) of 1280 bytes. The MTU indicates the maximum dimension of an IPv6 packet. Obviously this dimension is far bigger than the dimensions that a IEEE 802.15.4 frame can transmit, usually the maximum dimension of the payload of a packet can be decided at compile time, with the flag `TOSH_DATA_LENGTH`, by default it is setted at 28 bytes, in our work we used a value of 112 bytes, however we cannot increase the size too much: this will create collisions problems, thus an adaptation layer for fragmentation and reassembly of fragments is needed.

### 2.4.3 Header Compression

In many application an header compression is convenient, for example in WiMAX, after the registration of a node to the main node of the network, in the packets are

carried only the identifier of the connection but not the complete IP address, it will be added by the coordinator if the packets have to be sent through a normal IP network. The main differences between the well known approaches and 6lowPAN is that usually more than one connection using IP is assumed between nodes, and often the two nodes (normal node and the node that carries out the header compression and decompression) are in direct connection. In 802.15.4 we haven't multi-stream connections and usually we have multi-hop network thus the compression must preserve the field necessary for the routing.

For further details one can consult the Master Thesis of the author of the implementation of 6lowPAN in TinyOS [11].

## 2.5 CoAP

CoAP [28] stands for *Constrained Application Protocol*, which is a specialized protocol conforming the REST constraints. REST stands for *Representational State Transfer* and indicates a family of protocols, for example HTTP, based on client-server interactions: the client requests informations to the server, it replies with an appropriate response. The main concepts in REST protocols are:

- **Resource:** It can be any kind of meaningful concept that can be addressed.
- **Representation:** It is the resource current status.

The client requests the representation of a resource and the server replies with it.

The main purpose of CoAP is to give a RESTful (conforming to the REST constraints) protocol for WSN. This protocol relies on the UDP and 6lowPAN layers already developed. The main challenge is avoiding the fragmentation of large packets performed by the 6lowPAN layer. CoAP is not only a compressed version of HTTP for WSN: it is a full RESTful protocol which has a common subset of command with HTTP, allowing the creation of automatic proxies for translation of HTTP messages in CoAP messages and vice versa.

We can summarize the main features of CoAP as follows:

- It is a constrained RESTful protocol suitable for Machine to Machine (M2M) interactions.
- It allows the creation of automatic proxies for compression/decompression of HTTP messages.
- It provides communication reliability on an unreliable transport protocol like UDP.
- It supports synchronous transactions (like HTTP) but also asynchronous transactions.
- It provides low complexity and overhead.

Usually in M2M interactions client and server reside on the same node thus we will refer to both server and client using the word *end points*. The client requests an action (indicated by a *Method Code*) on a resource (identified by an URI, *Uniform Resource Identifier*). Requests and responses are transmitted using the *Transaction Messages* and *Transaction IDs* (TIDs).

### 2.5.1 Transaction Messages

CoAP supports 4 kinds of messages:

- **Confirmable (CON)**: The receiver of this message must reply with an ACK. The ACK can be empty, indicating only the correct receipt of the message, or it can contain the response.
- **Non - Confirmable (NON)**: These messages do not require ACKs, they can be used for delivering repeated informations, such as periodic readings, where the lost of few packets can be tolerated.
- **Acknowledgment (ACK)**: They confirm the receipt of a Confirmable message, carrying its TID.
- **Reset (RST)**: It indicates that a Confirmable message has been received correctly but some context informations are missing for allowing the correct elaboration of the response (we will see an example further).

### 2.5.2 Synchronous Transactions

In the synchronous transactions Confirmable messages are used, the reply is directly carried in the ACK. In Fig. 2.21 we can see two examples: the first is a request of a resource present on the server, in the second the resource is not available on the server.

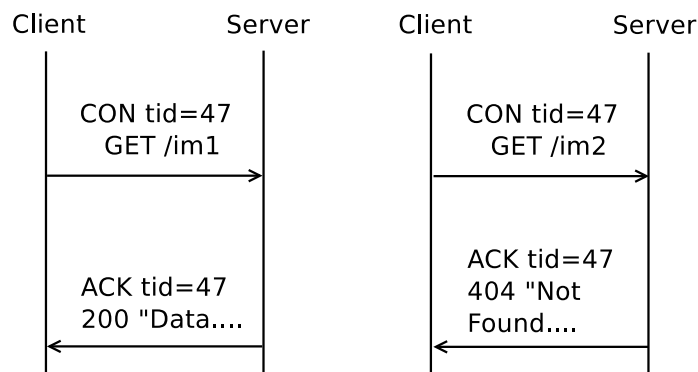


Figure 2.21: Synchronous Transactions

### 2.5.3 Asynchronous Transactions

In the asynchronous transactions request and response are decoupled and both carried in Confirmable messages. The need of this kind of transaction is immediately obvious if one think that the server can be a sensor with very low resources, the elaboration of the description for the requested resource can take much time, therefore the Server when receive the request send an empty ACK message indicating the correct receipt of the message. When the response is ready a new confirmable message is sent initiating a new transaction with a new TID. For allowing the client to match the response to the request the URI present in the request is echoed in the response (this was not necessary in the previous case), see Fig. 2.22.

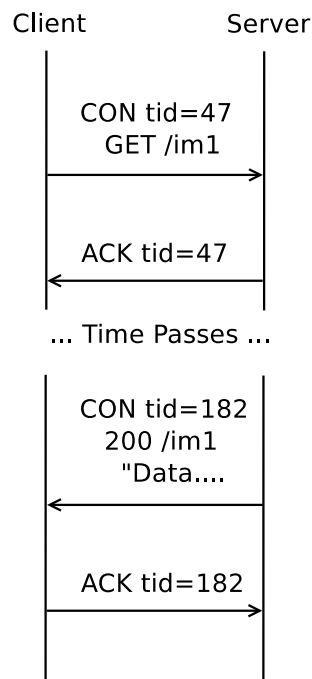


Figure 2.22: Asynchronous Transaction - Success

Asynchronous transactions add a new kind of failure: it happens when the Client is no longer aware of the request (for example due to a reboot). In this case it has to use the RST message, Fig.2.23.

### 2.5.4 Transaction ID (TID)

The Transaction ID is a 16-bit unsigned integer kept by the CoAP end-point for each sent Confirmable and Non-Confirmable message. Each end-point keeps only

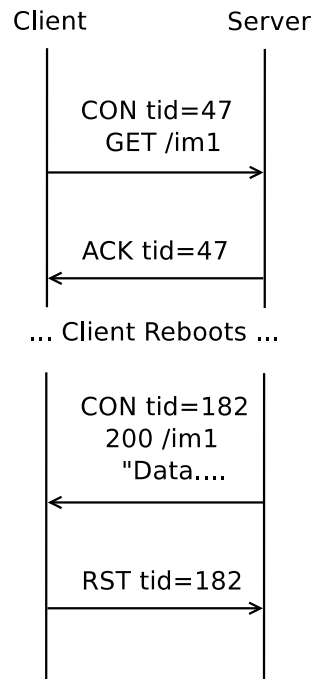


Figure 2.23: Asynchronous Transaction - Reset

one TID variable, which is modified each time a new CON or NON message is sent and it is used to match the ACK which the sent request. This choice allows a low memory occupation but the echoing of the URI in the response is needed in the Asynchronous Transactions. If an ACK is not received for a CON message its retransmission must contain the same TID, thus the TID must not be reused within the potential retransmission window.

### 2.5.5 Methods

CoAP supports the basic HTTP methods: GET, POST, PUT, DELETE. If an endpoint receives a message containing an unknown method it must generate a message with a “405 Method Not Allowed” Response Code.

- **GET:** It retrieves the description of the resource identified by the URI carried by the request. If the request is successful a 200 (OK) Response Code is sent with the response.
- **POST:** It is used to request the server to create a new resource under the provided URI. If the resource already exist a 200 (OK) Response Code is sent, otherwise a 201 (Created) is sent in the response. In the response must be sent a description of the updated resource too.
- **PUT:** It requests the update of the resource indicated in the URI provided.

The data used for the update are contained in the request body. If the update is successful a 200 (OK) code is returned, if the resource does not exist is created and a 201 (Created) code is returned. If an error occur an appropriate error code is sent.

- **DELETE:** It requests the deletion of the indicated resource. On success 200 (OK) code is sent.

### 2.5.6 Codes

CoAP uses a subset of the HTTP status codes, which are encoded in a 8-bit unsigned integer code. In the following table we list the CoAP codes and their correspondent HTTP codes. Some of these method are already been seen in the previous section.

CoAP Code	HTTP Code Equivalent	Description
40	100	Continue
80	200	OK
81	201	Created
124	304	Not Modified
160	400	Bad request
164	404	Not found
165	405	Method not allowed
175	415	Unsupported Media Type
200	500	Internal Server Error
202	502	Bad Gateway
204	504	Gateway Timeout

### 2.5.7 Content Type

MIME is an Internet standard [14] for the specification of the type of the content, it stands for *Multipurpose Internet Mail Extensions*, despite its name it is used also in HTTP and not only in e-mail transfer. In HTTP the content type is indicated by a string, for example **application/xml**. Sending in the packet this kind of strings can be burdensome for the nodes, therefore a mapping between a subset of the MIME codes and an 8 bit identifier is provided by the standard. In the following table are listed the supported MIME types and their mapping. The values from 201 to 255 are reserved for vendor specific, application specific or experimental use, the values from 0 to 200 are maintained by IANA.

CT	MIME Type
0	text/plain (UTF-8)
1	text/xml (UTF-8)
2	text/csv (UTF-8)
3	text/html (UTF-8)
21	image/gif
22	image/jpeg
23	image/png
24	image/tiff
25	audio/raw
26	video/raw
40	application/link-format
41	application/xml
42	application/octet-stream
43	application/rdf+xml
44	application/soap+xml
45	application/atom+xml
46	application/xmpp+xml
47	application/exi
48	application/x+bxml
49	application/fastinfoset
50	application/soap+fastinfoset
51	application/json

### 2.5.8 CoAP implementation in TinyOS

The SigNET group is currently developing an implementation of CoAP for TinyOS, in Fig. 2.24 we can see a sketch of the modules and the interfaces developed.

The main module is `CoAPP` which provides the actual implementation of the interfaces, while the configuration `CoAPC` wire it with the UDP/6lowPAN modules, making the module easily usable since only the wiring of the `CoAPClient` and `CoAPServer` interfaces is needed. `CoAPClient` is used when a node wants to send requests to a server, the interface is structured as follows:

```
interface CoAPClient {

    command coap_tid_t request (
        coap_absuri_t* absuri,
        coap_method_t method,
        coap_content_t* content,
        bool acked
```

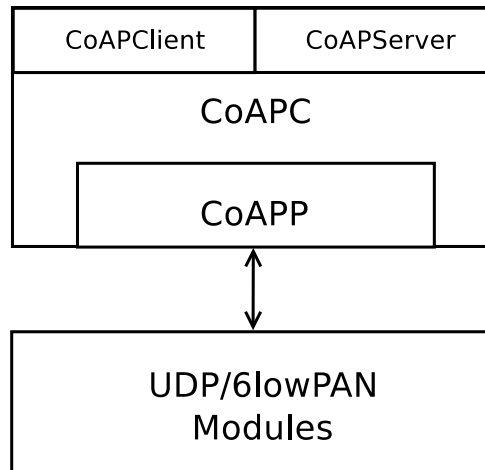


Figure 2.24: CoAP implementation in TinyOS

```

);

event void response (
  coap_status_t status,
  coap_content_t* content
);

}
  
```

the `request` command allows the node to make a request on a particular resource on a particular server (indicated in `absuri`) using a particular method (indicated in `method`). The request can carry informations passed through the `content` parameter. The boolean parameter `acked` allows the user to send Confirmable (`acked=TRUE`) or Non-Confirmable requests. We can examine the two data structures: `coap_absuri_t` and `coap_content_t`.

```

typedef struct {

  ip6_addr_t* host;
  uint16_t port;
  coap_uri_t *uri;

} coap_absuri_t;
  
```

With the `coap_absuri_t` data structure, a complete set of informations about the requested resource is described: the IP address of the server (`host`), the UDP port and the uri are provided.

```

typedef struct {
  
```



```
coap_char_t* data;
coap_length_t len;
coap_contenttype_t format;

} coap_content_t;
```

The `coap_content_t` structure provides the informations needed for the elaboration of the body of the message: a pointer to the actual payload is present (`data`) with its length (`len`) and its format (encoded using the MIME types provided by the standard). `coap_length_t` and `coap_contenttype_t` are 8 bit unsigned integer. If a Confirmable request is sent when the ACK arrives the `response` event is signaled, providing the Response Code (`status`) and the arrived data (`content`). For the sake of completeness we can now examine the `CoAPServer` interface even if we have not used it.

```
interface CoAPServer {

event void request (
coap_rid_t rid,
coap_absuri_t* uri,
coap_method_t method,
coap_content_t* content,
bool toack
);

command error_t response (
coap_rid_t rid,
coap_status_t status,
coap_content_t* content
);

}
```

The event `request` is signaled when a request arrives, if the boolean parameter `toack` has `TRUE` value the server can response to the request using the `response` command. The parameters are the same seen in the previous interface the only difference is the `rid` parameter which stands for `Request ID`, it is a data structure providing various informations about the current transaction, such as the `TID`, the IP address of the client that has sent the request etc. These information are vital for the correct delivery of the response (if needed) thus they must be provided to the `response` command.

## 2.6 Server-side programming

In this section we will introduce the tools used for the server-side programming.

### 2.6.1 HTTP - Hyper Text Transfer Protocol

In this work we have used HTTP for the communication between Gateway and Server, thus an overview is needed. HTTP is a RESTful protocol belonging to the *Application* layer of the ISO/OSI stack, it uses the client-server paradigm. Its most popular usage is for web browsing. The interaction between client and server starts with a request from the client, the server replies with a response containing the data required (usually a web page, images, or other kind of media). The current version of HTTP (HTTP/1.1) allows more requests/responses in the same connection while previous versions didn't allow it. The **request** is a message formed by four parts:

1. The **Request** line that indicates the **METHOD** that we want to use, the resource that we want to request and the HTTP version that we are using.
2. An arbitrary number of headers (even zero) containing the parameters needed by the request.
3. An empty line
4. The body (optional)

We can send data using **GET**, including them in the URL. It can be done using URL formatted in the following way:

```
www.mysite.com/search?param1=val1&param2=val2
```

where **param** is a parameter (**name**, **surname**, **age**, **RSSI measurements**) and **val** is the value of that parameter. An other way to send data to a server is the **POST** method (which is the way that we have chosen). The main advantage of using this method is that the data are sent in the body of the request so we can send more data than using **GET** method. The values included in the body are couple of type **parameter - value**. The **Response** has also an header and a body. The header contains various informations about the server and the requested resource, the body contains the description of the resource. For a more detailed introduction to HTTP and other Internet service [17] is a good reference. In Figure 2.25 we can see the request and response messages needed for retrieving a web page from a server, in this case we are using **get** for sending data, it is important to notice that in this case the request has no body, conversely in 2.26 we use the **POST** method for sending data, thus the request has a body.



Figure 2.25: HTTP session with GET method examined using WireShark [19]

## 2.6.2 GWT - Google Web Toolkit

GWT [9] is a powerful tool for the creation of complex web-oriented applications. The basic idea within GWT is simple: when developing web-based application we have to deal with many languages: usually we have to create a server-side part (code that runs on the server) using technologies as ASP, PHP, JSP and a client-side part (code that runs on the browser) using AJAX. So why can't we use the same language for doing all these things? GWT allows the programmer to do this using JAVA, then the client-side part is translated into JavaScript and HTML automatically. The server-side part remains in Java since JSP is a technology supported by most of the web servers. The only drawback is that when developing the client part we can't use the standard Java classes, we can only use the GWT classes but their behavior is similar to the standard Java classes.

### 2.6.3 Java Servlet

Servlet can be used in other environment, not only in Web with HTTP, in this short introduction we will only refer to the Web case. The interactions between a Servlet and a client (browser) are illustrated in in Figure 2.27. This technology can be used in the server-side part of a GWT application.

The main idea behind servlets is simple and similar to PHP or ASP philosophy: we have a program that runs on a server, which responds to the requests of the clients (Request) interacting with various resources (database, files, etc.), through standard



Figure 2.26: HTTP session with POST method examined using WireShark [19]

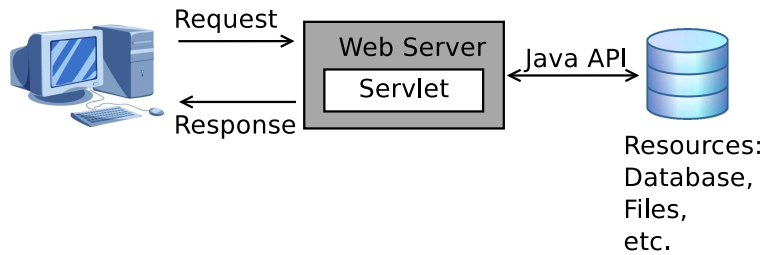


Figure 2.27: Java Servlet

JAVA API and generates dynamically a HTML page which is sent to the client using the **Response**. The strength of this approach is that a developer with knowledge of HTML and JAVA can easily develop dynamic web pages, secondly in the program we can use the standard Java APIs that allows us to access to complex resources, for example database, with simple commands. We can use a single servlet for generating various kind of web pages (we can build a different web pages depending on which

parameters are sent in the `Request`) and Servlets can cohabit with normal web pages on a server, but for sake of simplicity we can think that in a web server there is a Servlet for each web page that the user can display. Each Servlet is a Java class that inherits its method from the class `HttpServlet`, the important methods for our discussion are:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
```

and

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
```

The first is invoked when the client uses the `GET` method, the second when `POST` method is used, they are very similar so we will analyze both at the same time. The methods have two parameters:

```
HttpServletRequest request
```

which contains the data that the client has sent to the server through the `Request`, and

```
HttpServletResponse response
```

which contains the data that the server wants to send to the client (usually a Web page). Basically the behavior of a servlet is simple:

1. When a request is made `doPost()` or `doGet()` are invoked
2. The method elaborates the data present in `request`
3. The object `response` is filled with the data that have to be sent to the client

For accessing to the data the `req` has the method

```
String getParameter(String name)
```

it returns the value associated to the parameter with name `name`, all the values returned are strings so for some of these data is necessary to convert them in the correct type using the conversion functions provided by Java. Now we have the data and we can elaborate them: we need a way to respond to the client, we have to write the response in the `response` object. We first need to set the type of the response using the `setContentType()` method, if, for example, we want to reply using `text/html` format we will write

```
response.setContentType("text/html");
```

after that we can obtain a `PrintWriter` object to write in `response`, this object provides the standard `print()` and `println()` methods that are used for the standard I/O in Java. After the use the object must be closed.

```

out= resp.getWriter();
out.println(content);
out.close();

```

When `doGet()` or `doPost()` is ended the content of `response` will be sent to the client. For more detailed information refer to [16].

### 2.6.4 Fandango

Fandango (see Fig.2.28) is an application developed by the SigNET group using GWT. The main characteristics of Fandango is the integration with Google Maps: we can see all the motes deployed in the department and we can issue performance tests (such as testing the routing performance) or control the motes, for example we can inspect the software installed on one mote (see Fig.2.29).

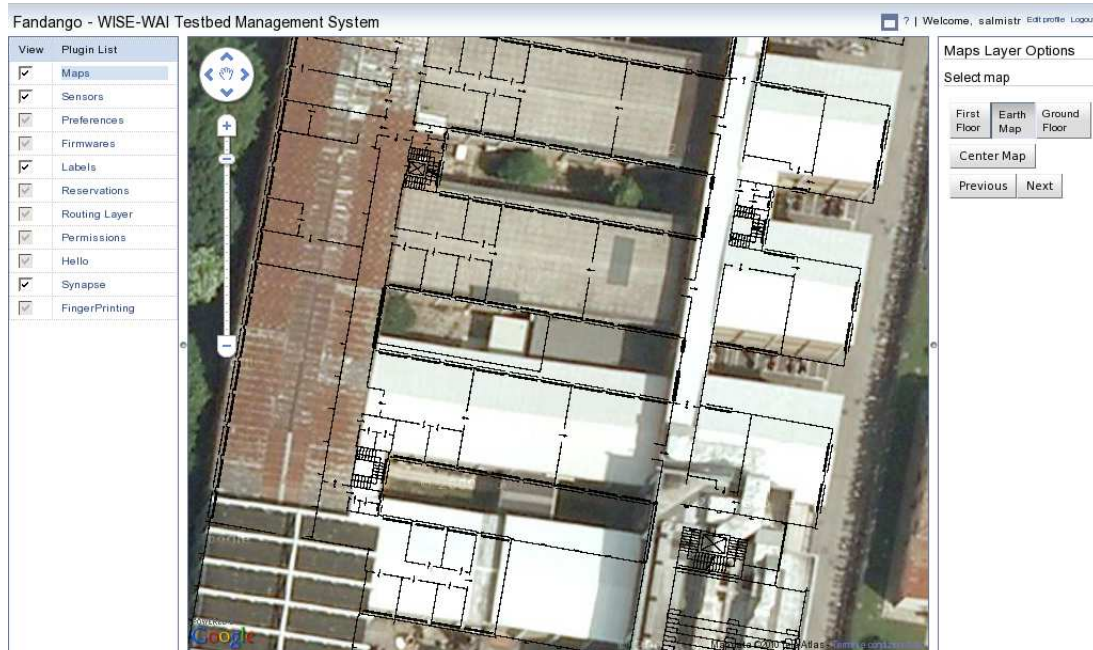


Figure 2.28: Fandango Application

In this work we have developed a plug-in for Fandango, thus we will explain only the parts that the reader needs to understand our work. In Fandango every plug-in consists of two parts, the first is the *layer* which is the part that creates the GUI for the user and displays data, it is the client part that we have previously introduced

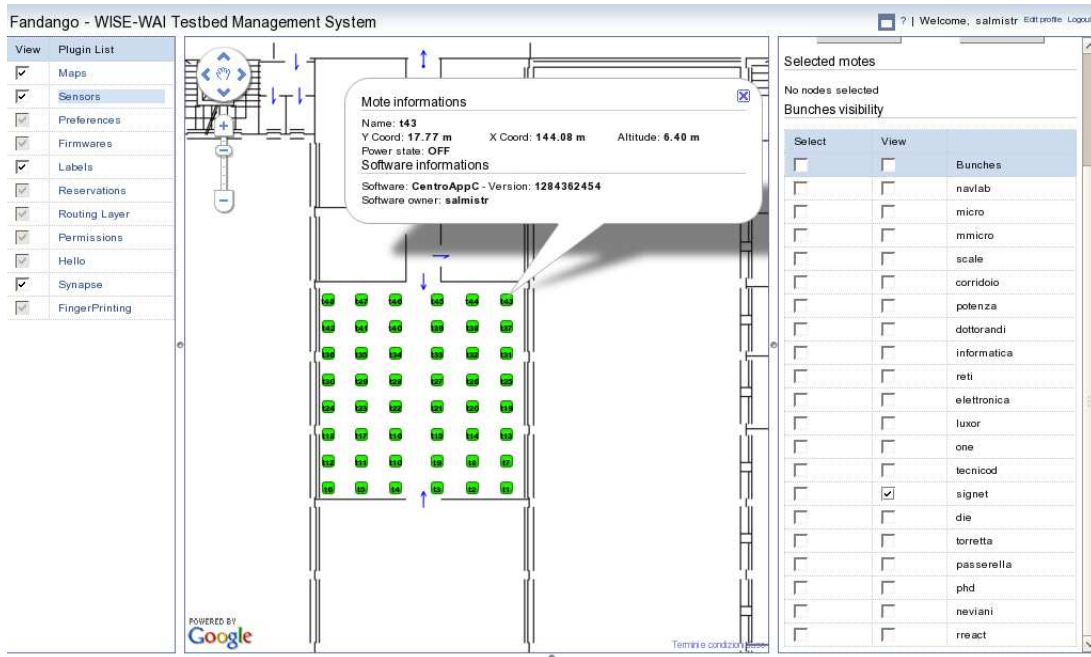


Figure 2.29: Fandango Application: Inspecting the software installed on a mote

in the discussion about GWT. The second is the *server*, it resides on the server and elaborates the request coming from the *layer*. *Layer* part and *Server* part can communicate using the standard RPC implemented in GWT ([18]), but in Fandango is implemented an other way to do it: the *Notification Service*. With RPC the client can ask for actions to the server. With the *Notification Service* is the server that asks for actions to the client. The basis of this system is the *NotificationDispatcher*, a singleton class that takes the data from the server and dispatches them to the layers. On the server side a class extending the *NotificationPlugin* class must exist. This class will send data to the client part via the *NotificationDispatcher*. The first operation that is necessary to do this is the registration. *NotificationDispatcher*, contains an array of the registered plug-ins, we have simply to add our own object to the array, once the operation is completed, the object can send data to the *NotificationDispatcher* using these instruction

```
NotificationDispatcher.getNotifier().notifyData(
FingerPrintingLayerServiceImpl.this,
"fingerprintingNotification",
status
);
```

where `FingerPrintingLayerServiceImpl.this` is the current object (the one previously registered), `"fingerprintingNotification"` is a String that indicates the



name of the notification and `status` is the object containing the data that we want to send to the layer.

The layer is an extension of the class `AbstractLayer` allowing the `layer manager` to manage it. The `layer manager` is the object that controls all the layers and it is passed as argument to all the layers in the constructor method. In our particular case the layer has to subscribe to the notification service using

```
lm.subscribe(this, "fingerprintingNotification");
```

where `lm` is the `layer manager` and the `subscribe` method allows the current layer (`this`) to subscribe to a particular *Notification Service* (in this case is the `"fingerprintingNotification"` service). `this` (the current object) must belong to a class which extends the `HasNotificationService` interface. When the server issues the `notifyData()` method the `onNotificationReceived()` method that has to be implemented in the `layer` part is invoked.

```
public void onNotificationReceived(String name, ArrayList<Datatype> data) {
    if(name.equals("fingerprintingNotification"))
    {
        elaborate data
    }
}
```

The presence of this method is guaranteed by the `HasNotificationService` interface which obliges the layer to overwrite the method. The `name` parameter is necessary when a layer registers to more than one *Notification Services*, since all the services issue the same method `onNotificationReceived()` and it needs to know which server has fired the event.

For a more detailed introduction to Fandango programming please refer to [10].

### 2.6.5 Java Database Connectivity

In this subsection we will introduce the concept of database (in a not rigorous manner) and the techniques that can be used to connect a Java program with a database. Basically a (relational) database is a collection of structured data, the data are grouped in tables, which have fixed number of columns and arbitrary number of rows. Each column have a name and a type (Integer, floating point, etc.). Between the data in the table relations exist and they can be exploited for data retrieval. It is possible to create instruction for the manipulation of the data in a *declarative* way: we have simply to define what data we want to manipulate and the manipulation, the database will execute the requested operations in a transparent way. These instruction are written using SQL (*Structured Query Language*) [20] and are called query, the SQL allows the the manipulation of the data in the database and also the database structure too. The Java program that we have written has simply to



create the correct query and to elaborate the extracted data (if requested). The data modification or retrieval process will be worked out by the database.

A JDBC (*Java Database Connectivity*) driver is a set of classes that allow the user to easily interact with the database, the issues coming from using a particular database instead of an other are all solved internally by these classes. They provide the user with a standard set of classes and a standard SQL syntax for the manipulation of the database. A JDBC driver can belong to one of the following categories:

- **Type 1:** This driver makes use of the JDBC-ODBC technology. ODBC is a standard set of API, independent from the operative system and from the language used. However ODBC needs the specific driver of the database to operate. This adds latency to the system because there are more than one level between application and database.
- **Type 2:** This driver translate the requests made by the application in request made using the database standard API. This system is more efficient than the Type 1 but it needs the presence on the computer of the database API whose license can be expensive.
- **Type 3:** This system uses the network sockets, it needs the presence of a **Database Server**, a program that waits for requests, performs the request and provides the required data. This is the most common type of driver. The request are translated in a database independent network protocol. It has many drawbacks: first is necessary to know the IP address of the server and on which port it is listening. Secondly a server or a similar application is needed.
- **Type 4:** It is similar to the Type 3, but the net protocol used is vendor specific allowing better performance.

For a more deep description [21] is a good starting point.

We can now examine the code needed to interact with a database. In our case the chosen database is MySQL [22] so we will examine the class needed for this particular database, however the only difference in using other databases is simply in the driver needed, the classes are the same. First, we need to load in memory the correct driver, the string needed and the `.jar` archive containing the driver can be found at [22].

```
Class.forName ("com.mysql.jdbc.Driver").newInstance ();
```

After that the driver is loaded, we need to establish a connection with the database (it is important that the database server is running). For getting access to the server we simply need

```
Connection conn = DriverManager.getConnection (url, user, passw);
```

where `url` is the url of the database server, `user` is the username, `passw` is the password for get access to the database. For manipulating or extracting data from the database we need a `Statement` object

```
Statement st=conn.createStatement();
```

the `Statement` class has two important methods:

```
ResultSet executeQuery(String s);
```

which is used for data extraction. The string is the query , a typical query string could be

```
SELECT * FROM database_name.table_name
```

The second is

```
int executeUpdate(String s);
```

In this case the string can be a query that manipulate data o the database structure, for example

```
DELETE FROM database_name.table_name WHERE condition
```

In the first case the output is a `ResultSet` object. It contains the data extracted using the query, it allows the user to examine row by row the data, accessing each field within a row. For a more detailed description of the `ResultSet` class see [23].

### 2.6.6 Gateway - Serial Tunneling

One of the major drawback when using WSN is the difficulty to exchange data with a standard IP network, the TinyOS toolchain comes with a helpful program: *SerialTun*, it can be found at `TOSROOT/tos/support/sdk/c/6lowpan/serial_tun`, its main task it to take *6lowpan* packets and translate them into normal IPv6 packets and vice versa. In Fig. 2.30 we can see the architecture of a standard `SerialTun` utilization.

When using the tool we must have a mote connected with the PC via USB. This mote is simply a IEEE 802.15.4 adapter: it takes packets coming from other motes and sends them to the PC via USB. It also takes the packets coming from the PC and sends them to other motes. The main tasks are worked out by the Serial Tunneling application: it takes the *6lowpan* packets, reassembles them if needed and finally creates a normal IPv6 packet with the same payload, it is able to do the inverse operation. It creates a virtual interface (`tun` interface) on the computer running the application, this interface is used like a standard ethernet port, allowing the user to use it in a very intuitive way, for example in Fig. 2.31 we can see the traffic

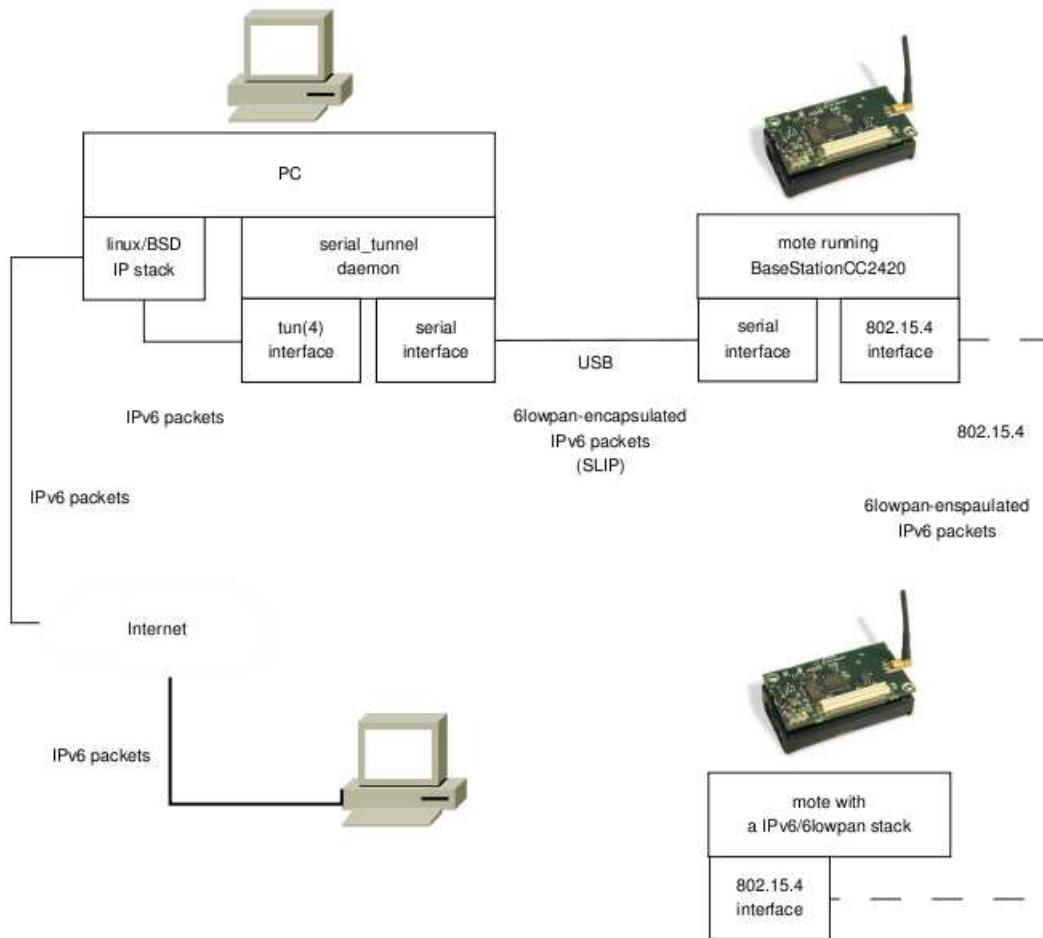


Figure 2.30: SerialTun utilization, from [11]

generated by Serial Tun captured using WireShark [19]. The first packets (1) are messages captured by the Base Station, which doesn't make any kind of control on the packets. Serial Tun is incapable to correctly process them so they become malformed data which are discarded by the system. The last packet (2) is a correct compressed UDP packet transported over 6lowPAN, Serial Tun can manage this kind of packet so the packet is correctly decompressed. As can be see all the data shown are consistent with the UDP standard.

Now we have the possibility to exchange data between two different networks, but we want to be more flexible: the data inside the UDP packet are understandable only by a particular application created to deal with motes, secondly we want to use in the PC-part of the network a more reliable connection since we haven't to deal

No.	Time	Source	Destination	Protocol	Info
N/A	0.000000	N/A	N/A	N/A	Raw packet data[Malformed Packet]
N/A	8.387995	N/A	N/A	N/A	Raw packet data[Malformed Packet]
N/A	8.435210	N/A	N/A	N/A	Raw packet data[Malformed Packet]
N/A	15.65298	N/A	N/A	N/A	Raw packet data[Malformed Packet]
N/A	16.68798	N/A	N/A	N/A	Raw packet data[Malformed Packet]
N/A	26.15400	N/A	N/A	N/A	Raw packet data[Malformed Packet]
N/A	27.03996	N/A	N/A	N/A	Raw packet data[Malformed Packet]
N/A	30.00298	N/A	N/A	N/A	Raw packet data[Malformed Packet]
N/A	32.61904	2001:638:709:1234::ffe:42	2001:638:709:1234::ffe:12	UDP	Source port: 32949 Destination port: 32180

Figure 2.31: Serial Tun working

with bit-rate or energetic constraints. A response to this problem is the *SENSEI-Gateway*.

### 2.6.7 Gateway - SENSEI Gateway

The SENSEI-Gateway allows users to interact with nodes as ordinary web pages. We have exploited this work to create a plug-in for the gateway that allow the nodes to request resources to normal web pages. The SENSEI-Gateway has been developed during the SENSEI Project [5], a general architecture can be seen in Fig. 2.32.

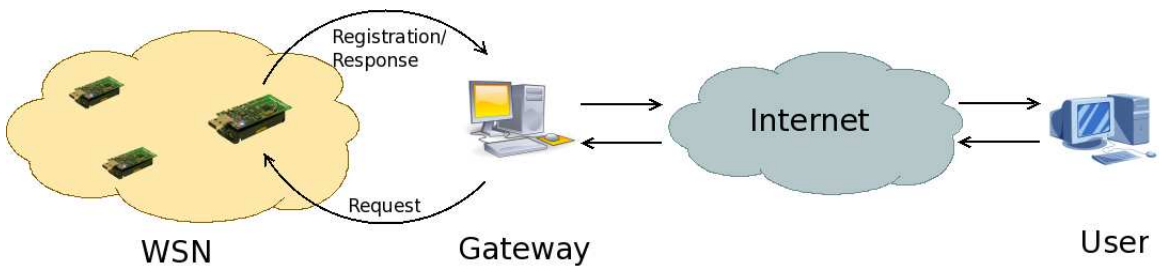


Figure 2.32: SENSEI general architecture

When a node is booted it registers itself to the gateway. In the registration it also inform the gateway of the *resources* belonging to it. With the word *resource* we intend all the informations that the node can provide, for example if it is equipped with humidity sensors the humidity measurement will be a resource, each resource is mapped to a particular URI. After this part the nodes and the resources are published: from now on every user that accesses to the gateway through the Internet can request a resource belonging to a particular node simply inserting the URL associated to that particular resource in a web browser. The browser requests the resource to the gateway, it transforms that HTTP request in a request in the

particular protocol used in the particular WSN. A gateway can handle more than one type of WSN, we can have a WSN using 6lowPAN, one using IEEE 802.15.4 or ZigBee or other systems. The strength of the SENSEI - Gateway is giving a uniform way to request informations to motes, all the issues regarding the different kinds of motes or protocol used are managed inside the gateway. In the future we will have this kind of systems for interacting with the WSN, we will use new technologies using well known interfaces.

The gateway is made of plug-in, each plug-in is an independent thread that is bound to a particular UDP port, allowing the mote to interact with it, but it can also publish web resources allowing a normal browser to request data to the plug-in. We have used only the first feature. In this section we will explain how a plug-in for the gateway can be developed. The gateway is written in C++, so the plug-in has to be created using the same language. A plug-in that has to communicate with the motes is made of two parts: the real plug-in and a `BWSocket` (*Binary Web Server Socket*). The `BWSocket` elaborates requests coming from the motes, after the tunneling phase, when the 6lowPAN header translation has been already worked out by the *Serial Tunneling* program. Its main duty is to give to the developer a way to easy interact with data contained in the header or other data such as the URL. The motes can request a resource using a URL and that information is stored in the UDP payload. It is handled like raw data by the tunneler so it's the `BWSocket` that extrapolates it. Exploiting the C++ object model which allows inheritance and polymorphism we have developed a specialization of `BWSocket`: `LocalizationBwsSocket`. When the `BWSocket` receives a request invokes a method of the connected plug-in. The socket can access to the URL so if we are developing a complex plug-in handling more than one service, we can, at this level, decide which is the requested service thus invoking the correct method of the plug-in, if it has a method per service. Once the request is arrived to the plug-in the data can be sent using, for example, HTTP services to every connected device to the Internet. The `BWSocket` can handle the responses generated by the plug-in as well, allowing the plug-in to respond to a request coming from the motes. This data can be accessed by the mote.

## 2.7 TinyNET

In this section we will introduce TinyNET, a modular framework developed by the SigNET group [13], it allows an easy connection between the application and the radio stack. Using TinyNET a routing algorithm has been developed, such system is used in our work. Usually in TinyOS applications the various modules communicates directly with the modules that wrap the radio hardware, this approach is good only if the application that we are developing is very simple or the various application that share the radio device are aware of the fact that they are not alone.

### 2.7.1 Architecture

In Fig. 2.33 we can see a simplified architecture of TinyNET, which can be divided into two layers, the first is the *Application Layer*, the second is the *Network Layer*. The *Application Layer* is similar to every TinyOS application, we have modules created by the user of the framework that interact with the *Network Layer* using particular interfaces. Some interfaces are Utility Interface, they allow the start/stop of the RF subsystem and channel selection. These tasks are now centralized, with the normal approach would be difficult, for example, allowing an application to get the exclusive possession of the radio stack, now this is possible. The *Network Layer* contains the elements that need to get access to all the packets received by the node. It is where we insert the modules that implement the protocol part of our application, allowing independent developing of the two parts and the reusing of the written code. Using TinyNET a routing protocol has been implemented and this system has been used in the thesis for the delivery of some kinds of messages.

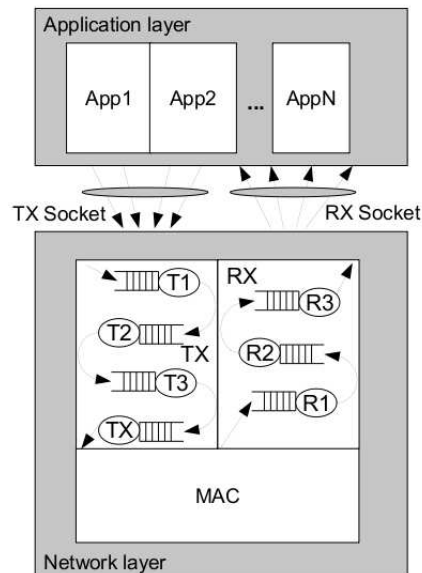


Figure 2.33: TinyNET simplified architecture

### 2.7.2 Wiring with Standard Applications

When wiring with a standard application (i.e. an application developed for the standard radio stack) we have to deal simply with three components: `BaseSingleNetC`, `AMSend2TXC` and `Receive2RXC`. `BaseSingleNetC` is the components that provides

RFSwitch, Packet and AMPacket, the first is an alias for the SplitControl interface which allows the power on/power off of the radio stack, the other two are interfaces for manipulating the data contained in the header or in the payload of a `message_t` variable. These interfaces are the same provided by `ActiveMessageC` so if a program does not need a deep control of the radio stack (for example disabling acknowledgment) the substitution is very simple. We want a centralized control of the network stack, so every component that uses `BaseSingleNetC` has to be wired at the same component. Each instance of `BaseSingleNetC` is connected to `BaseNetC` using parametrized interface (see [8]). This system add flexibility: using parametrized interfaces when `BaseNetC` receives a command it can signal the event to every module connected or to only the module that has issued the command. The other two components (`AMSend2TXC` and `Receive2RXC`) substitute the normal `AMSenderC` and `AMReceiverC` components respectively, providing the same interfaces. As can be seen is a relatively simple task adapting a standard application to TinyNET.

### 2.7.3 The Routing Protocol

The routing protocol used has been developed using TinyNET. It is based on hop count (HC) that is the number of intermediate node that the packet has to travel through for reaching the destination. Every node that has an hop count of  $n$  relays on a node with HC of  $n - 1$ . The destination is the *Sink Node*, for our work it is the *BaseStation* connected to the Gateway Computer. The procedure starts when the **Sink** is booted, it sends periodically advertisement packets (*Hello*) Packet with HC set at zero. The other nodes when booted are set with an arbitrary high HC. Each node receive from **Sink** or from other nodes **Hello** with a certain HC (for example  $n$ ), they compare that HC plus one to its HC, if it is lower the node sets its HC at  $n + 1$  and uses the sender as its next hop. When the *Sink* is started it sends **Hello** with HC set to 0, each node that receives that packets gets an HC of 1 and the procedure is repeated recursively. When a node has to send a packet to the sink it selects as destination of the AM message the next hop stored in its memory. This system allows node to *Sink* communication but with simple modifications can allow the *Sink* to node communication. Each node when receives a packet stores in a cache the source and the previous hop of each received packet, this allow the *Sink* to send responses to the messages of the nodes. The protocol needs more data for the routing process than those that can be found in the packet. It needs: source, sink address, next hop, previous hop. The source and the destination are written in the packet header, at the source address is added 0x8000 for signaling that the packet has to be elaborated by the routing algorithm. The other two fields (previous and next hop) are added at the end of the payload area.





## Chapter 3

# Implementation

In this section we will discuss the implementation of the application, following the same steps of the past section

### 3.1 Fingerprinting Application

The first step when using the fingerprinting method is to get the “fingerprint” of the environment, in simple words we have to measure the RSSI of the received packets in many points of the building. For doing this we have developed a JAVA application, which uses one or more Basestation (BS) to send and receive messages from the other motes. When doing this phase of the process we perfectly know our position, thus we can safely assume that we can also determine which motes are in coverage, in other words we know the motes that can correctly receive our request. The GUI of the application (Fig. 3.1) allows us to insert the data needed for the measurement, like coordinates ( $X, Y$ , `refSys`), the `TOS_NODE_ID` of the BS and its `Reference`. `Reference` is the identifier of the particular node, it is not defined at compile time, it is defined when the mote is built, these value can be recovered using the `motelist` program. The other informations needed are the motes to be interrogated, that are to be inserted in the `TextField` provided.

We can now explain how the application works: it basically parse the string containing the list of the nodes, for each entry the application sends a packet to the BS, it is a `toBSpck` packet, the fields of the payload are listed in Fig.3.2 while the messages exchanged in the process are shown in Fig. 3.3.

The `toBSpck` payload is formed by four element:

1. `actionCode`: it allows the user to specify options for the queried node.
2. `id`: The `TOS_NODE_ID` of the anchor node.
3. `ch`: The channel in which the anchor have to start to transmit.
4. `nTimes`: The number of messages that the anchor have to send per channel.

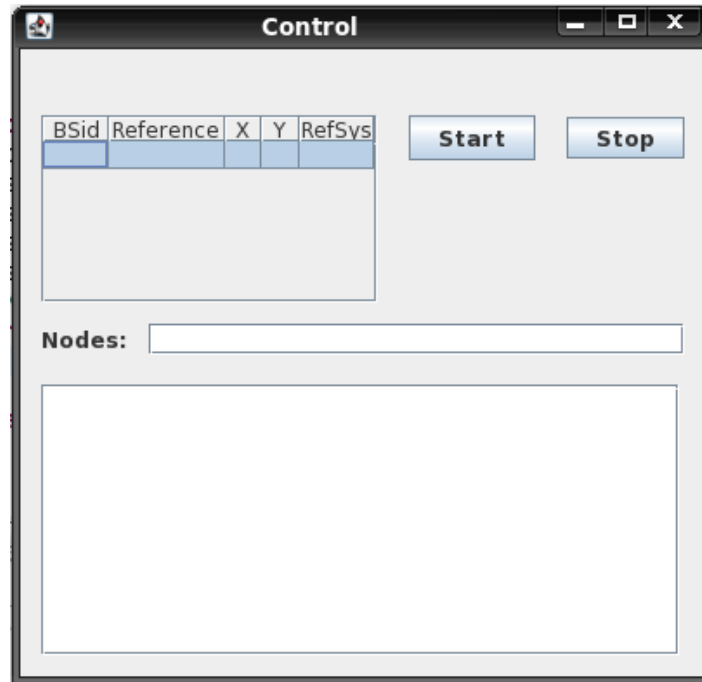


Figure 3.1: Fingerprinting Application GUI

2 byte	2 byte	1 byte	2 byte
ActionCode	id	ch	nTimes

Figure 3.2: Payload of the toBSpck packet

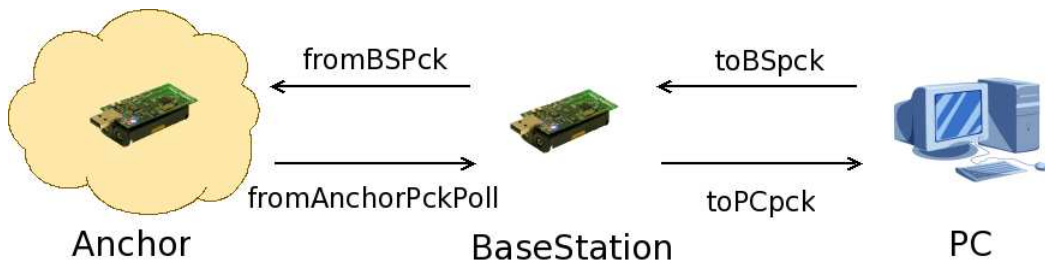


Figure 3.3: Fingerprinting Application messages exchange

After receiving the message the BS immediately create a new packet to be sent to the interrogated node. This packet is a `fromBSPck` which is the same packet used in the localization phase. Basically in this message are copied all the information written in the previous packet, an exception is the `id` which will be written in the header `destination` field. After receiving the message the anchor starts to reply

to the BS, it sends `nTimes` messages for each channel from `channel` to 26. Now a problem arises: how does the BS know that it is time to change the channel? First of all we can give a look to the payload of the `fromAnchorPckPoll` packet in Fig. 3.4. The fields of the payload have various meanings:

2 byte	1 byte	4 byte	4 byte	2 byte	2 byte	2 byte	2 byte
<code>rssiMeasByAnchor</code>	<code>lqiMeasByAnchor</code>	<code>counter</code>	<code>mirror</code>	<code>temp</code>	<code>hum</code>	<code>infra</code>	<code>lig</code>

Figure 3.4: Payload of the `fromAnchorPckPoll`

1. `rssiMeasByAnchor`: it's the RSSI measured on the request message.
2. `lqiMeasByAnchor`: it's the LQI measured on the request message. The LQI (Link Quality Indicator) is an indicator of the reliability of the channel.
3. `counter`: It is a counter incremented by the anchor each time it sends a message.
4. `mirror`: It is where the payload field of the request is copied, it is used for diagnostic purposes.
5. `temp`: Measurement of temperature of the anchor.
6. `hum`: Measurement of humidity of the anchor.
7. `infra`: Measurement of infrared light intensity of the anchor.
8. `lig`: Measurement of visible light intensity of the anchor.

Using the `counter` field, which is set to zero each new request, by the anchor, the BS can determine if the received packet is the last for the present channel and can start to switch to another channel. The change happens when `counter%nTimes=0`, where % indicates the modulo operator. The problems arise when the last packet of a channel is lost: the BS has a timer which is reset every new arrived response, if the response doesn't arrive in time a packet, in which the channel is set to zero, is sent to the PC from the BS, the program stores the "failed" node in an array and when the application has finished to query all the nodes it starts to query the failed nodes starting from the channel in which the last reply didn't arrive. If even in this case the procedure fails the program will query again the "failed" nodes, this process continues until all the nodes are successful or the user stops the procedure. The `toPCpck` packet is very similar to the `fromAnchorPckPoll` packet, it has only more fields for the measurements (RSSI, LQI) made by the BS, the indication of the current channel and the `TOS_NODE_ID` of the current BS:

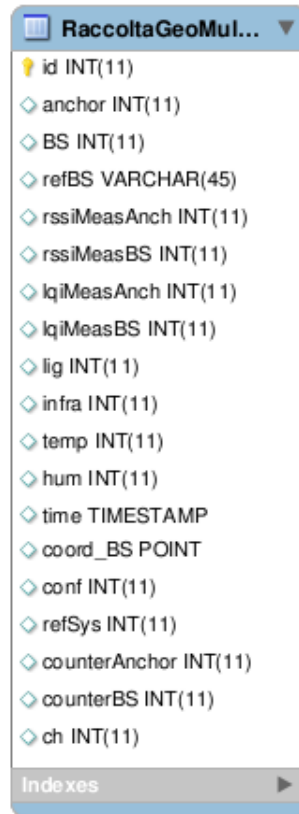
```
typedef nx_struct toPCpck{
nx_int16_t rssiMeasByAnchor;
nx_int16_t rssiMeasByBS;
nx_uint8_t lqiMeasByAnchor;
nx_uint8_t lqiMeasByBS;
nx_uint8_t ch;
nx_uint16_t anchor;
nx_uint16_t bsID;
nx_uint16_t temp,hum,lig,infra;
nx_uint32_t counterBS;
nx_uint32_t counterAnchor;
}toPCpck_t;
```

We list only the new fields.

1. `rssiMeasByBS`: it's the RSSI measured on the response message.
2. `lqiMeasByBS`: it's the LQI measured on the response message.
3. `counterAnchor`: It is a counter incremented by the anchor each time it sends a message.
4. `counterBS`: It where the payload field of the request is copied, it is used for diagnostic purposes.
5. `ch`: It is the channel on which the message has been received by the BS.

Thanks to these data the application can fill the table created for holding all the measurement. The fields of the table are presented in Fig. 3.5. In the table we can find the following fields:

1. `id`: It is the primary key of the table, an auto-incrementing field.
2. `anchor`: The `TOS_NODE_ID` of the Anchor node.
3. `BS`: The `TOS_NODE_ID` of the BS node.
4. `refBS`: The reference of the BS node.
5. `rssiMeasAnchor`: RSSI measured by the anchor on the request message.
6. `rssiMeasBS`: RSSI measured by the BS on the response message.
7. `lqiMeasAnchor`: LQI measured by the anchor on the request message.
8. `lqiMeasBS`: LQI measured by the BS on the response message.
9. `lig`: The anchor node visible light intensity measurement.
10. `infra`: The anchor node infrared light intensity measurement.



Field Name	Data Type
id	INT(11)
anchor	INT(11)
BS	INT(11)
refBS	VARCHAR(45)
rssMeasAnch	INT(11)
rssMeasBS	INT(11)
lqiMeasAnch	INT(11)
lqiMeasBS	INT(11)
lig	INT(11)
infra	INT(11)
temp	INT(11)
hum	INT(11)
time	TIMESTAMP
coord_BS	POINT
conf	INT(11)
refSys	INT(11)
counterAnchor	INT(11)
counterBS	INT(11)
ch	INT(11)

Figure 3.5: RaccoltaGeoMultiCh table

11. `temp`: The anchor node temperature measurement.
12. `hum`: The anchor node humidity measurement.
13. `time`: The timestamp of the insertion of the record.
14. `coord_BS`: The BS coordinates (X,Y) when the measurement was made.
15. `conf`: A field for allowing the storing of more then one measurement campaign of the same points.
16. `RefSys`: The BS refSys when making the measurement.
17. `counterAnchor`: The counter used by the anchor node.
18. `counterBS`: The counter used by the BS node.
19. `ch`: The channel on which the measurement was made.

After the fingerprinting phase we have to use these data, but we do not need all these data, we only need the RSSI measurement average for each channel for each

position for each Anchor node: doing this all the times in a query would not be computational efficient, we have decided to store these data in an other table (Fig. 3.6): The content of the field is similar to the previous table:



Figure 3.6: RefLoc table

1. **id**: It is the primary key of the table, an auto-incrementing field.
2. **anchor**: The TOS\_NODE\_ID of the Anchor node.
3. **rssiMeasBS**: *Average* RSSI measured by the BS on the response message.
4. **coord\_BS**: The BS coordinates (X,Y) when making the measurement.
5. **RefSys**: The BS refSys when making the measurement.
6. **ch**: The channel on which the measurement was made.

The table is filled using the following query:

```
insert into wsn.RefLoc(anchor,rssiMeasBS,coord_BS,refSys,ch)
(
  select anchor,avg(rssiMeasBS),coord_BS,refSys, ch
  from wsn.RaccoltaGeoMultiCh group by ch,coord_BS,refSys,anchor
)
```

where `wsn` is the database name that we are using.

## 3.2 Localization

In the localization phase we have a Mobile node that request to other nodes responses on various channels: from our experiments we have discovered that using measurements on different channels instead of a great number of data gathered on a single channel lead to better results. This phase is the Interrogation or Measurement Phase. After that phase the mobile sends to the Gateway the measurements. The

gateway in turn sends them to the Server that replies with the coordinates of the node. When a TelosB mote is setted on a channel it cannot receive or send data on others channels, thus it is necessary to define a *Standard Channel* on which are sent the request of the Mobile node and on which the Anchor are listening. We have decided that the *Standard Channel* is the channel 26 .

### 3.3 Mobile - Interrogation Phase

The interrogation phase in our system is the most delicate moment, we have to interrogate more nodes as possible (using the *Standard Channel*) and protect the message from collisions, in other words we need to have many answers, but the messages do not have to collide. This is a completely new problem in Fingerprinting: this system was used in PC networks, Mobile and Anchor were respectively laptops and Access Points (AP), in [24] the authors have to deal with 7 AP, while we have to face the problem to interrogate more Anchor nodes (in the SigNET laboratory we have 48 nodes). Furthermore the Mobile do not know its location thus is impossible for the Mobile to decide the nodes that have to reply. The decision is taken by the Anchor nodes, they have two elements for making this decision: the first is the distance, the more distant is the anchor from the mobile the more unreliable the RSSI reading on its packet will be. The second is the density of the Anchor nodes, we can suppose that an Anchor node can estimate how many neighbors it has intercepting messages. We have developed a little module that randomly create messages for simulating this behavior, in a real implementation Fingerprint will be only one among many services provided by the WSN, thus for this phase the nodes would have only to intercept packets created by other applications. Using this information a node can decide if it can reply or not to the Mobile, if they are close the probability of reply will be high, it will decrease while the Mobile is moving away from the Anchor. The probability will decrease if we are in a dense environment (many nodes per area unit) and will conversely increase in uninhabited areas. These speculation may be wrong: suppose that we are in a very dense area but almost all the nodes are busy and cannot reply: only the Mobile can determine if it has received enough responses. It is necessary to allow the mobile to force the Anchor to reply with higher probability. Furthermore it would be desirable to allow the Mobile to determine if collisions have taken place, in this case it will force the Anchors to reply with lesser probability. The first problem can be easily solved: in the request packet we can insert a parameter (we will call it **alpha**) that modifies the probabilities, if the Mobile receive few answers the probabilities can be increased. The second problem can be solved using **alpha** for decreasing the probabilities, if we are able to create a system for the determination of the collisions. TinyOS doesn't allow this, we have analyzed all the radio stack of TinyOS and we have found that the test for the correctness of the packet is carried out by the radio hardware. The RF hardware mainly consists of the antenna and the CC2420 system on a chip [12]. The CC2420 implements protocols and RF functions, it automatically

computes the CRC and verifies the correctness of the packet. Luckily the CC2420 does not discard the packet if it is incorrect, the decision of how handle the problem is delegated to the OS. In Fig.3.7 we can see a sketch of the components involved in the receiving of a packet in the standard TinyOS implementation. The various modules has different tasks, for example `CC2420ActiveMessageC` sends the packets to the correct `Receive` interface using the `AM_TYPE`. The module that we have modified is the `CC2420ReceiveP` module, this module after receiving the packet controls if the first bit of the last byte of the frame is set to 1, if this is true then the CRC test was correct, otherwise the packet contains error. If we are in the first case the event is signaled to the chain of components listed in Fig.3.7, conversely no action is performed. In Fig.3.8 we can see how the data elaborated by the CC2420 are presented to the system, we can see how even in this phase the system try to optimizer the memory resources: the last two bytes are occupied by the FCS (Frame Check Sequence) which is the CRC. The user isn't interested in this value, thus it is substituted by three more important values: RSSI (1 byte), CRC correct/incorrect (1 bit), correlation calculated over the preamble (used for calculating the LQI, 7 bits).



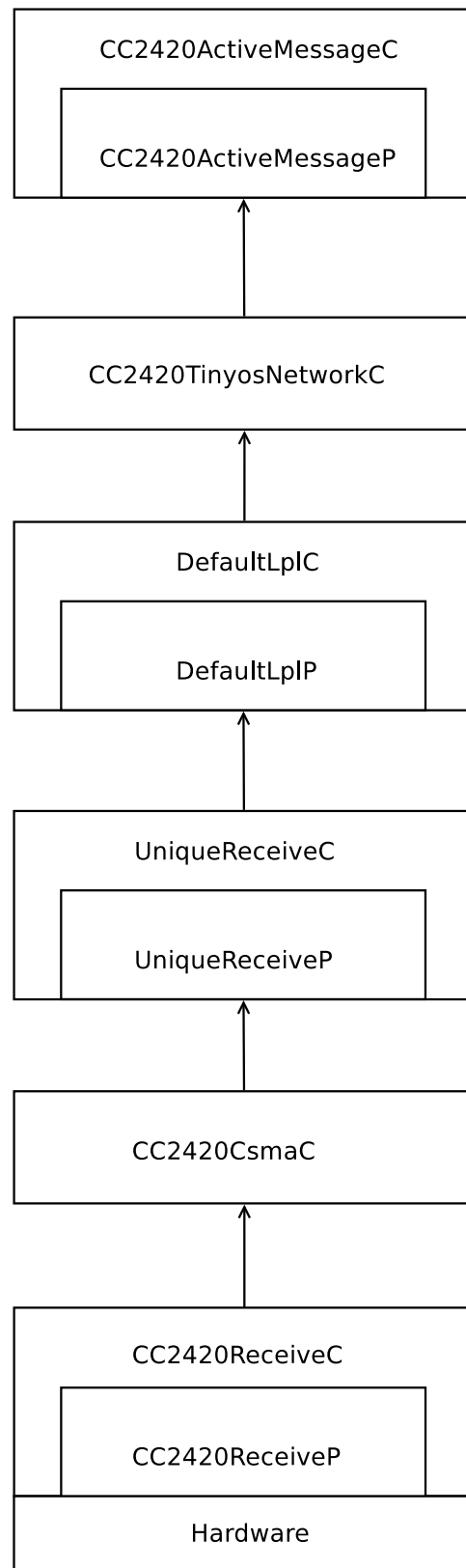


Figure 3.7: Modules involved in the packet receipt

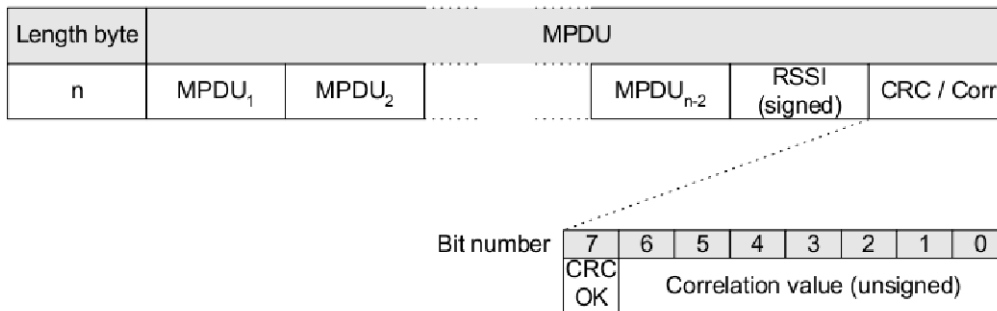


Figure 3.8: Data elaborated by the CC2420 chip, from [12]

Returning to our problem we have simply added a new interface to `CC2420ReceiveC`, a `Notify` interface:

```
interface Notify<val_t> {
    command error_t enable();
    command error_t disable();
    event void notify( val_t val );
}
```

We are only interested in the `notify` event, when the CRC is incorrect, the `notify` event is signaled, it is caught by one of our modules. We are assuming that errors happen only due to collisions and not for thermal noise, this assumption is good because the algorithm select the nearest nodes thus an error caused by thermal noise is unlikely.

We can now describe the synchronization algorithm used, starting from the basic idea. The first version of the algorithm was naive and inefficient: in the request message we insert the channel and the anchor nodes, using the guidelines discussed above, decided if reply or not to the request. In order to reduce the number of collisions the time from the sending of the request is divided in slot, more precisely 10 slots because from our experiments resulted 10 slots is a good compromise between number of queried anchors and time. The main problem in this approach is the overheads introduced for the channel change and creation of the packet. We have estimated that the change of the channel (jointly with other minor operations) require 160 ticks. The tick is the measurement unit of the `Alarm` timer provided by TinyOS, which is the most precise of all the timers provided by the OS, thus we have decided to use it, a tick is  $\frac{1}{32768}$ s. Therefore the change of the channel require approximately 5ms. For the creation of the packet, sending, receipt, elaboration

(copy of the meaningful data in a buffer) we have estimated an overhead of more or less 360 ticks (11 ms), the actual measurement requires 10 slot, 1 slot is 150 ticks, we need 1500 (46 ms) ticks for the actual measure but we have an overhead of 520 ticks (16 ms), the situation is sketched in Fig. 3.9, thus we need 12 slots and the time for the change of the channel. The first idea for improving the system is to

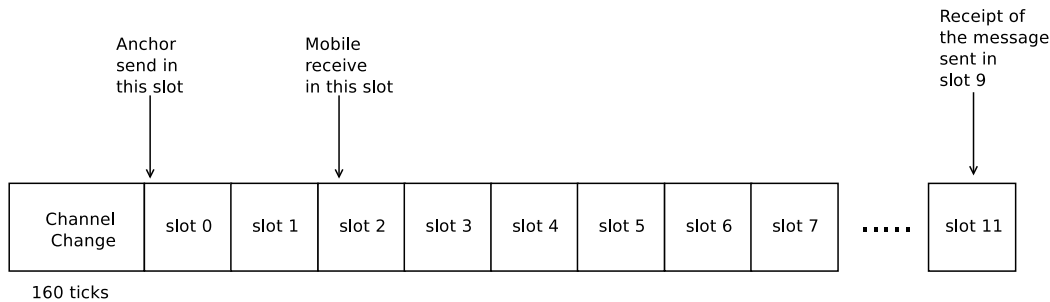


Figure 3.9: Problems due to computational complexity and channel change

use a single message for requesting responses in more than one channel. We call the time in which a node remains in a channel *macroslot*. After the first response the anchor knows that it has to send the reply and it can anticipate this moment, until the duration of a *macroslot* is higher than the time needed for channel change. We cannot solve the problem coming from the channel change overhead because it is an hardware constraint but we can attenuate the impact of the second problem. In Fig. 3.10 we can see the time evolution of the interaction between the Anchor and the Mobile (we analyze two macroslot). As we can see only the first macroslot consists of 12 slots, the others consists of 10 slots (plus the time needed for the channel change of the mobile). This system allows a good overhead compensation but there is a great issues to deal with. In the past system if too many Anchors reply we have only lost a macroslot, however, in this case, each Anchor doesn't know the slot chosen by the other anchors, thus two or more anchors can choose the same slot causing collisions, we have decided to split the algorithm into two phases: the first one is the *Discover* phase, it takes place in the first macroslot when we discover the neighbors, after the first macroslot (when all the nodes are switched in the 2<sup>nd</sup> channel) the mobile sends an other packet: `silPck`

```
typedef nx_struct silPck{
    nx_uint16_t talk[NSLOT];
}silPck_t;
```

this packet contains an array of the `TOS_NODE_ID` received in the *Discover*, phase. The Anchors that have not replied in the *Discover* phase are still in the *Standard Channel* thus they do not receive the packet. The others are already in the second channel, and they wait for this message. If a node finds its `TOS_NODE_ID` in the *i*-th position of the array it will send all the other responses in the *i*-th slot belonging to the following macroslots, which are called *Deterministic* phase. If its `TOS_NODE_ID`

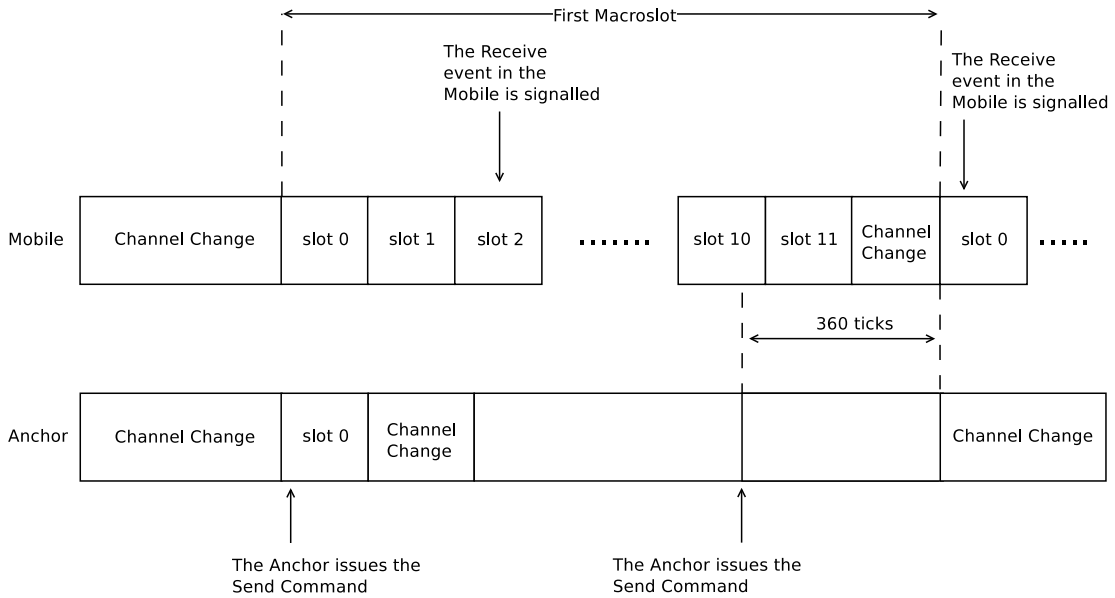


Figure 3.10: First solution

is not listed in the array it returns on the **Standard Channel** waiting for others request. A problem can occur: due to collisions with other messages the Anchors cannot receive the packet. This is a less unlikely situation of which one can think: we are in the ISM band, even if we suppose that all our network is synchronized others services, such as Wi-Fi (IEEE 802.11), can interfere. For solving this problem we have used an other timer **TFault**, which is a simple **TimerMilliC** timer (less precise than the **Alarm** timer but suitable for our purposes). We start it when the node sends its first reply, if it fires before the arrival of the **silPck** we assume that it has been lost, and the node reset its state returning to the *Standard Channel*. An other problem that we have to deal with was that unsynchronization due to backoff and carrier sense can occur between Mobile and Anchor. We have overcome the problem disabling backoff and carrier sense using the interface **RadioBackoff**. An other problem was the Ack messages that may add delays, they have been disabled using the **PacketAcknowledgement** interface. In Fig. 3.11 and 3.12 we can see a sketch of the Anchor-side part of the algorithm, in Fig. 3.13 we can see the Mobile-side part of the algorithm, **dSlot** is the length, in ticks, of the slots (in our case **dSlot=150**). **nSlot** is the number of slots that forms the macroslot in the **Deterministic** phase, in our case **nSlot=10**.

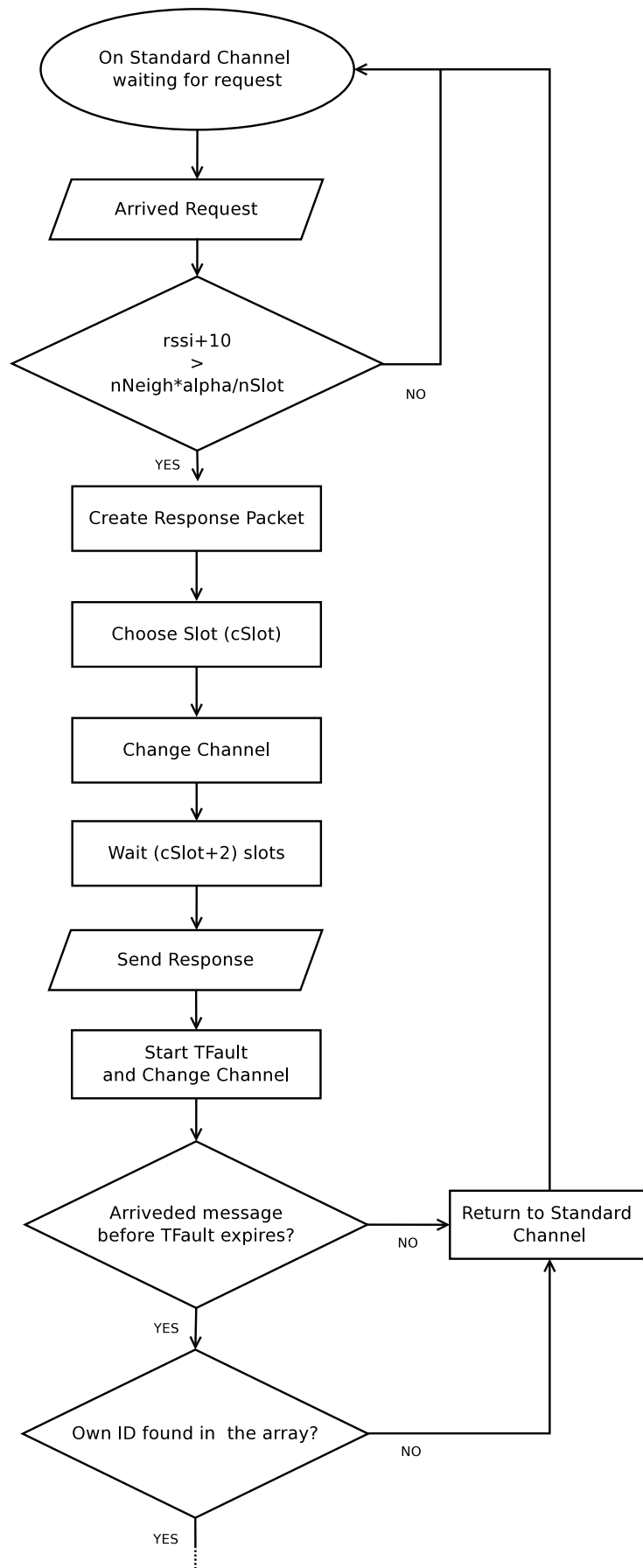


Figure 3.11: Anchor - Side part of the Algorithm, first part

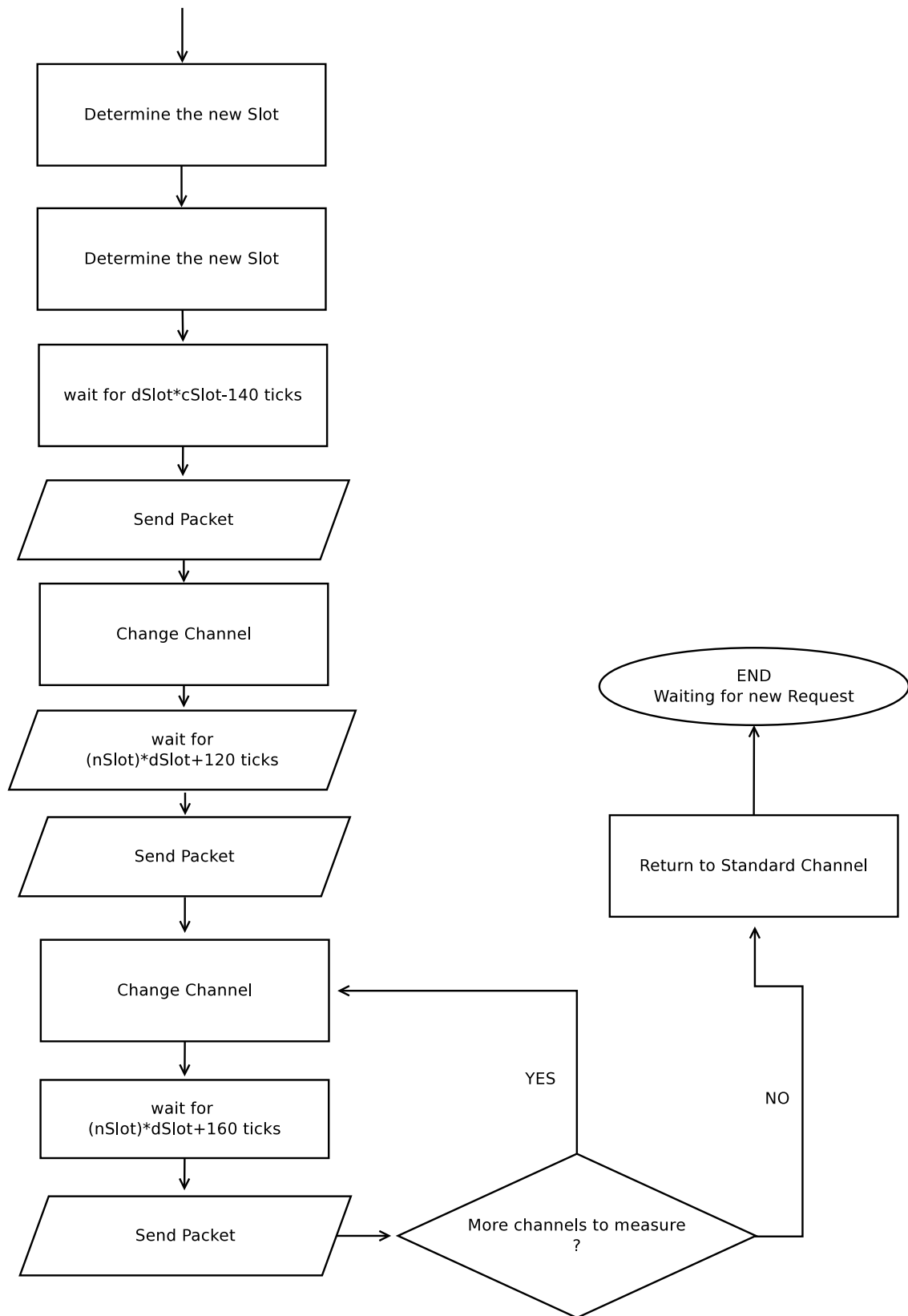


Figure 3.12: Anchor - Side part of the Algorithm, second part

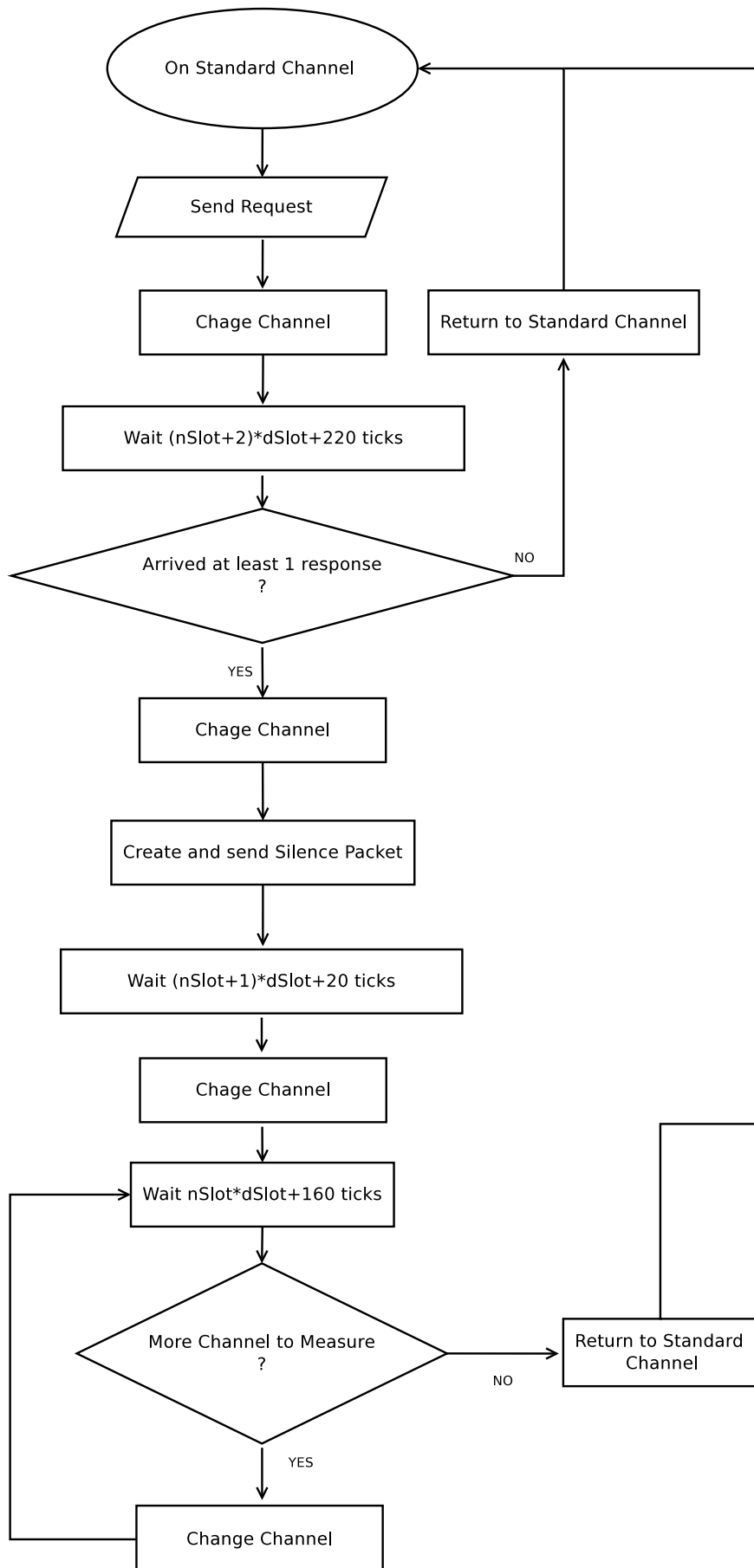


Figure 3.13: Mobile - Side part of the Algorithm

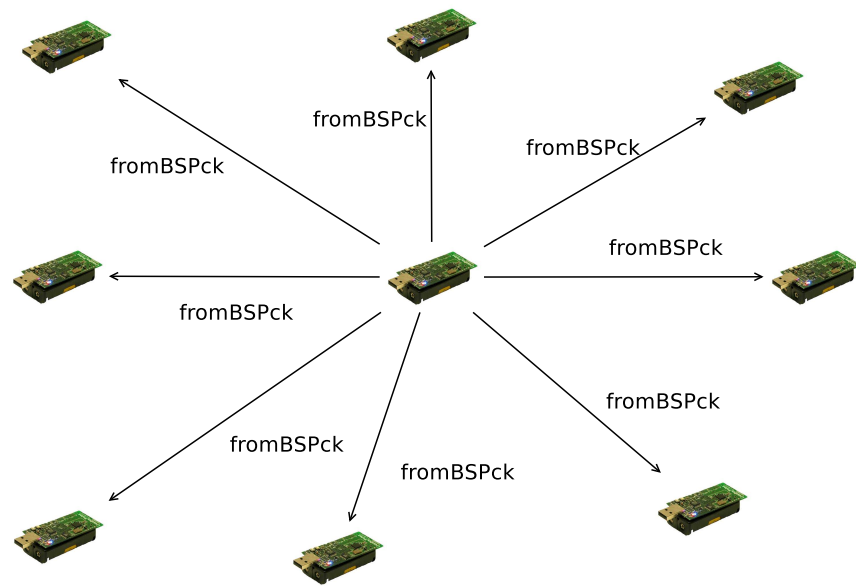


Figure 3.14: Mobile - Anchor Nodes interactions - first message exchange, Discover phase

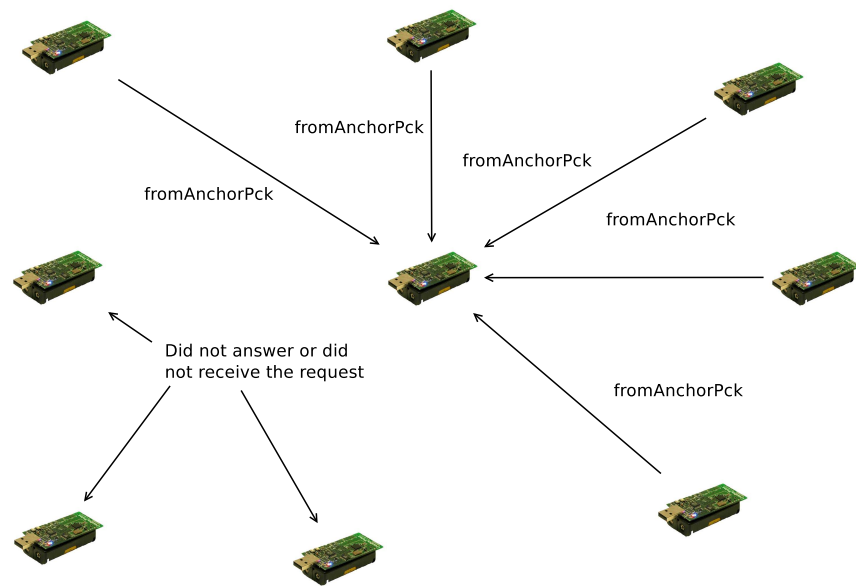


Figure 3.15: Mobile - Anchor Nodes interactions - first Anchors response, Discover phase

In Fig. from 3.14 to 3.17 are summarized the messages exchange between Mobile (the central mote) and Anchors (the other motes), we can examine the `fromBSPck`



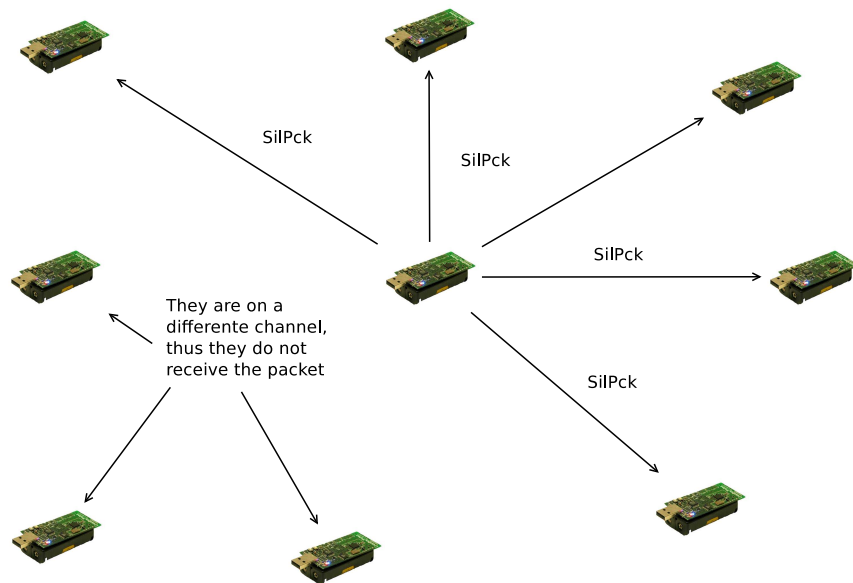


Figure 3.16: Mobile - Anchor Nodes interactions - start of the Deterministic phase

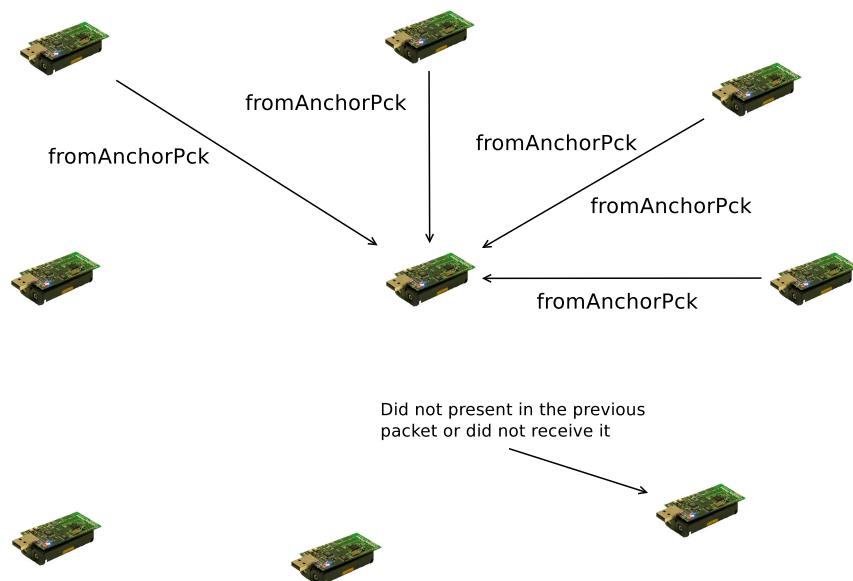


Figure 3.17: Mobile - Anchor Nodes interactions - first responses in the Deterministic phase

packet.

```
typedef nx_struct fromBSPck{
    nx_uint16_t ActionCode;
```

```

nx_uint8_t alpha;
nx_uint32_t payload;
nx_uint8_t nSlot;
nx_uint8_t nCh;
nx_uint8_t ch[MAX_CHAN];
nx_uint16_t dimPayloadRsp;
nx_int8_t minRssi;
nx_int8_t maxRssi;
}fromBSPck_t;

```

1. **ActionCode**: It is used as option for informing the anchor if the request comes from the mobile or from the BS (it can be used in the **Fingerprinting Application** as well).
2. **alpha**: It is the values that allow the mobile to control the number of responses.
3. **payload**: It is a payload which is copied in the response for diagnostic purposes, it is a counter.
4. **nSlot**: The number of slots that will be used by the anchors.
5. **nCh**: The number of channels that will be used by the anchors ( $nCh \leq MAX\_CHAN$ ).
6. **ch**: The list of the channels that will be used. **MAX\_CHAN** is a constant which indicates the maximum number of channel that can be used.

The last three fields are not useful for our algorithm, they are included since they are needed in the library that we have created for implementing the algorithm, see the next section for further explanations. The last packet to be examined is the **fromAnchorPck** packet:

```

typedef nx_struct fromAnchorPck
{
    nx_uint8_t py[RESPONSE_PAYLOAD];
}fromAnchorPck_t;

```

**RESPONSE\_PAYLOAD** is the maximum payload, in bytes, of the packet. The payload can be freely defined by the user, while this algorithm has been developed as a general library usable for other purposes.

### 3.4 BQuery Components

BQuery is the library that we have created during this work, in Fig.3.18 we can see the modules and the interfaces that form the library. This system is capable

of querying up to 10 anchors in about 0.43 seconds. It is possible, for the user, to define the dimension and the structure of the response payload, the system is capable of auto-compensate the dimension of the payload up to 30 bytes. A payload with a dimension between 20 and 30 byte requires an overhead of about 9 ms in the time required for the measurement. BQueryP and BQueryC implement the algorithms

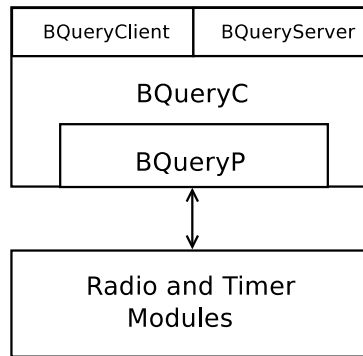


Figure 3.18: BQuery Library

previously described, they provide two interfaces: BQueryClient and BQueryServer.

```

interface BQueryClient {

    command error_t request(bquery_cli_req_param_t *param);
    event void response(bquery_cli_resp_t* resp, uint8_t nResp, uint8_t nColl);

}
  
```

`request` issues the start of the algorithm mobile-side (client-side) part sending the `fromBSPck` packet. The issuing can fail, due to an ongoing anchor(server)-side part of the algorithm. In that case the command returns `FAIL` if the request starts properly the command returns `SUCCESS`. We can analyze the `bquery_cli_req_param_t` data structure, used for passing all the data needed by the client:

```

typedef struct
{
    uint8_t nSlot;
    uint8_t* ch;
    uint8_t nCh;
    void* data;
    uint8_t ldata;
    int8_t minRssi;
    int8_t maxRssi;
    uint8_t alpha;
}bquery_cli_req_param_t;
  
```

The nomenclature is the same seen in the `fromBSPck` thus we analyze only the new fields:

- `data`: Optional data that can be copied in the `payload` field of the `fromBSPck`.
- `ldata`: Length of the optional data.

`minRssi` and `maxRssi` are the thresholds that are used by the Server to choose if it must reply or not. If `minRssi` and `maxRssi` have the same value the criteria previously described in Fig. 3.11 is used. If `minRssi` and `maxRssi` differ then the node replies (in a random slot) only if the received RSSI lies between the two thresholds. The `response` event is signaled when the process is terminated, it provides the number of response arrived (`nResp`) and the number of collisions (`nColl`) and an array with `nResp` elements containing the data gathered from each response. Each array elements have the following structure:

```
typedef struct
{
    int8_t rssi;
    uint8_t lqi;
    uint8_t delta;
    uint8_t ch;
    uint16_t sour;
    uint8_t slot;
    uint8_t data[RESPONSE_PAYLOAD];
    uint8_t ldata;
}bquery_cli_resp_t;
```

- `rssi`: Rssi measurement on the received response.
- `lqi`: Lqi measurement on the received response.
- `delta`: Difference between the start of the slot and the receipt moment.
- `ch`: The channel on which the response has been received.
- `sour`: The `TOS_NODE_ID` of the anchor that has sent the response.
- `slot`: The slot on which the response arrived.
- `data`: A copy of the payload of the `fromAnchorPck` packet received.
- `ldata`: Length of the payload of the `fromAnchorPck` packet received

The `BQueryServer` interface is structured as follows:

```

interface BQueryServer {

    command error_t start(uint16_t nNeigh);

    command void stop();

    event void buildResponse (input_handler_t *data, char* out, uint8_t *lout);

    event void updateResponse(input_handler_t *data, char* out, uint8_t *lout);

}

```

The first two commands start and stop the server functionalities, the `start` command requests the number of neighbors nodes (`nNeigh`). The event `buildResponse` is signaled when the `fromBSPck` arrives allowing the Server to build the response packet (`fromAnchorPck`) in a user-defined way. `updateResponse` is signaled each time a new response is sent allowing the user to modify the response payload on a per-macroslot basis. `out` points the buffer where the payload of `fromAnchorPck` packet will be written, `lout` points the variable storing the length of the passed buffer, when the payload is created the value of the variable can be changed to the actual amount of bytes written. The `input_handler_t` structure holds the data needed from the user's code to create a coherent response:

```

typedef struct
{
    message_t *mess;
    uint8_t rSlot;
    void* payload;
}input_handler_t;

```

`mess` points to the complete received message, `payload` points to its payload and `rSlot` is the slot chosen by the Client in the Discover phase.

### 3.5 Mobile - Gateway Communication

The communication between Mobile and Gateway takes place after the measurement phase: the Mobile node sends all the readings to the Gateway, this, in turn, sends the reading to the Server which perform the computation. When the coordinates are ready they are sent to the mobile via the Gateway. In this section we will analyze the three elements needed to do this: the CoAP and TinyNET components on the mote and the SerialTun component in the Gateway. After making the measurements the node make the average of the two RSSI for each node in each channel. These values are sent to the Gateway using the payload formatted like in Fig. 3.19. First are

inserted the 4 values of the 4 channels on which the Mobile made the measurements, these values goes from 11 to 26, (16 values, 4 bit per value) and they are written in the packet minus 11, thus the range goes from 0 to 15. After the channels are written the (maximum) 10 `TOS_NODE_ID` of the queried anchor nodes. Finally we have to write the (maximum) 40 values of RSSI (4 values per Anchor), they are written in order of arrival, `RSSI1-1` stands for RSSI on the first channel of `Anchor1`, `RSSI1-10` stands for RSSI on the first channel of `Anchor10`, this pattern is repeated for the others channels. The module that make the actual transmission of the informations

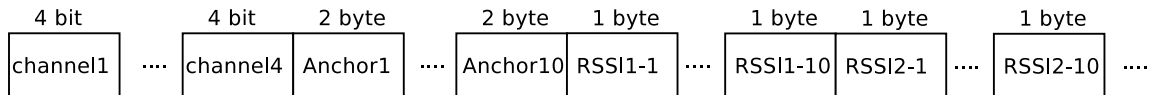


Figure 3.19: Payload of the request sent to the Gateway

is divided into 3 part, a configuration (`PostC`), a module (`PostP`) and an interface (`Post`). These three part were developed as a library for allowing others to use it in a completely transparent way. The `Post` interface has a command and an event:

```
interface Post {

    command void doPost(uint8_t *payload, uint8_t length);

    event void endPost(coap_status_t result, void* payload, uint8_t length);

}
```

The `doPost` command needs a pointer to the buffer where the payload is stored and its `length` (in byte). The `endPost` event is signaled when a response arrives or after the timeout. `result` is the Response Code received. When the response arrives in time, its payload is stored in a buffer referenced by `payload` having the provided `length`. `PostP` implements the command and the event provided, the actual transmission and receipt are carried out by the `CoAPClient` interface. The `doPost` command copies the provided data in an internal buffer, after that it sets the URL to which the request will be made to `/fan/loc/mote`. Finally it uses the `request` of the `CoAP` component to send the message created. We have decided to develop this component to make the initialization procedure transparent to the user, this task is accomplished in the `booted` event, implemented in `PostP`, in which we call two command of the `IP` and `IPControl` interfaces:

```
call IP.setAddressAutoconf(&addr_src);
call IPControl.start();
```

The first wants in input an IPv6 with the last 2 bytes set to zero, in whose space will be put the `TOS_NODE_ID` creating the IPv6 address of the node. The `start`

command allow the IP stack to begin its duties. The IPv6 address to which the request is sent is created in the same way of the source address, the `TOS_NODE_ID` of the BS must be 32766. The `TOS_NODE_ID` is important for the routing protocol but not for the actual receipt of the message, because the BS simply does not make any kind of address recognition: each message received will be sent via USB interface to the PC. The `IPControl` and `IP` interfaces are provided by the `IP2_6LPC` component which implement 6lowPAN protocol and convergence layers. A sketch of the connections of this component with the others is presented in Fig. 3.20. We can now

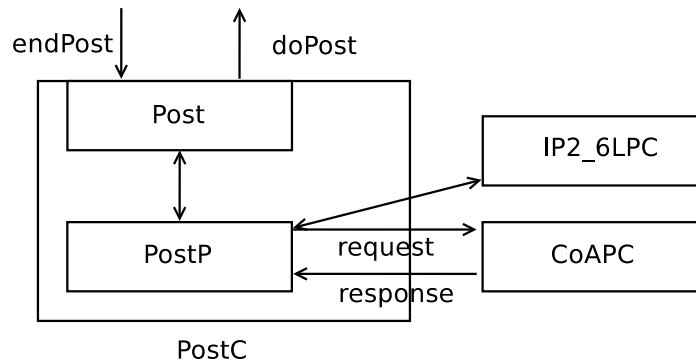


Figure 3.20: PostC wiring

describe how the TinyNET connections (and modifications) were made. In the standard 6lowPAN implementation we have that the `IP2_6LPC` configuration connect IP, UDP and radio components, we have created `IP2_6LPC` a similar configuration in which the radio components are substituted with the TinyNET components. The component that implements the IPv6 stack is `IP2_6LPP` we had to modify its connection with the Radio Stack. The situation before the TinyNET insertion is presented in Fig. 3.21, in Fig.3.22 we have the situation after the insertion. This operation were very simple thanks the plug and play design of TinyNET. With this wiring a problem persists: every message sent or received through the Radio Interface will be managed by TinyNET, this is the desired situation for the `Hello` packets and for the IP packets, but it isn't desirable for the packet used in the measurement phase of the algorithm, while we do not want that the `Request` packets are routed and the utilization of TinyNET would add delays to the algorithm (this would lead to add more unused slots). We can analyze Fig. 3.23, problems came from `NetPacketC` and `MAC` components. `NetPacketC` disturb the sending process: despite the algorithm is directly connected with the radio stack it is triggered automatically when it is starting the packet dispatch. It modifies the source `TOS_NODE_ID` adding 8000 to its value, signaling that the packet has to processed by the routing protocol. We have added the possibility to disable it: if the `AM_TYPE` of the message belongs to the list of the values that have not to be routed it does not modify the address, in this list are present all the message used by the algorithm except the IP and `Hello` packets. The second problem comes from the `MAC` component, the receive interfaces

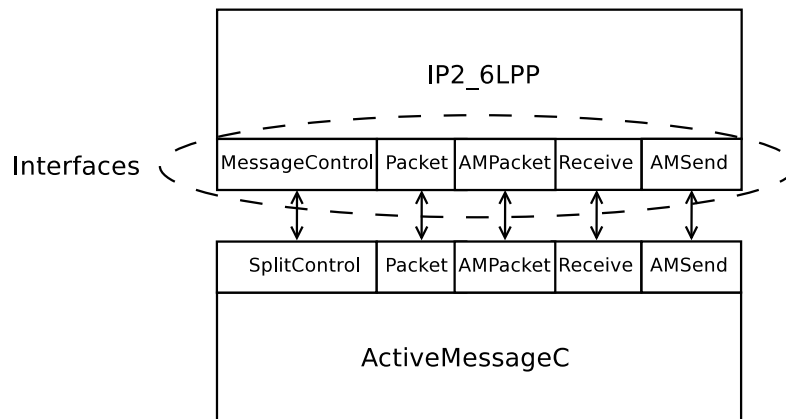


Figure 3.21: Before TinyNET insertion

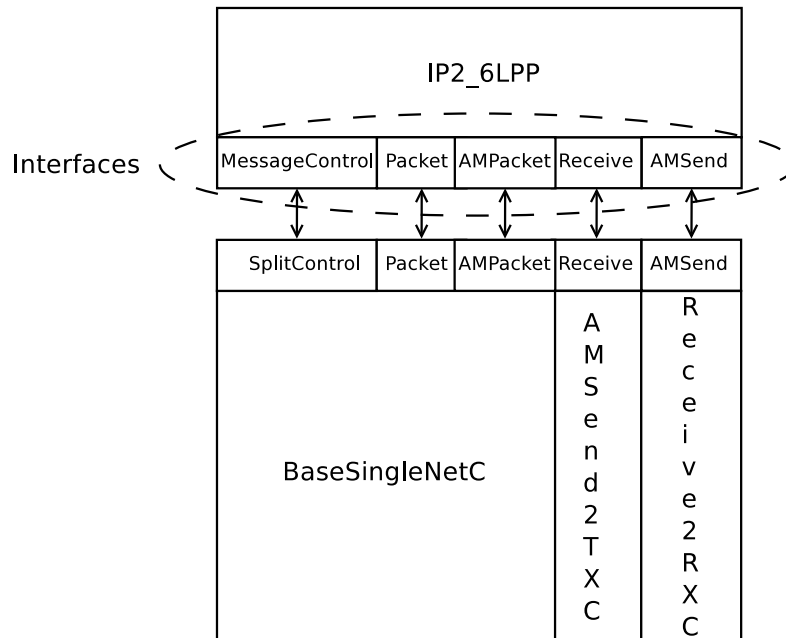


Figure 3.22: After TinyNET insertion

of our measurements modules are directly connected to the radio stack interfaces, but the problems could rise from the latency added by TinyNET, we have used the same expedient previously described: when it is signaled the **Receive** event to the **MAC** component, if the **AM\_TYPE** belong to the list the component does not signal any event or make any action. We have tried others solutions but they would require deeper changes to the framework. The last modification was made to the **SerialTun** application: when developing the system we have noticed that usually the requests arrived to the gateway, they were worked out but the reply (correctly sent by the



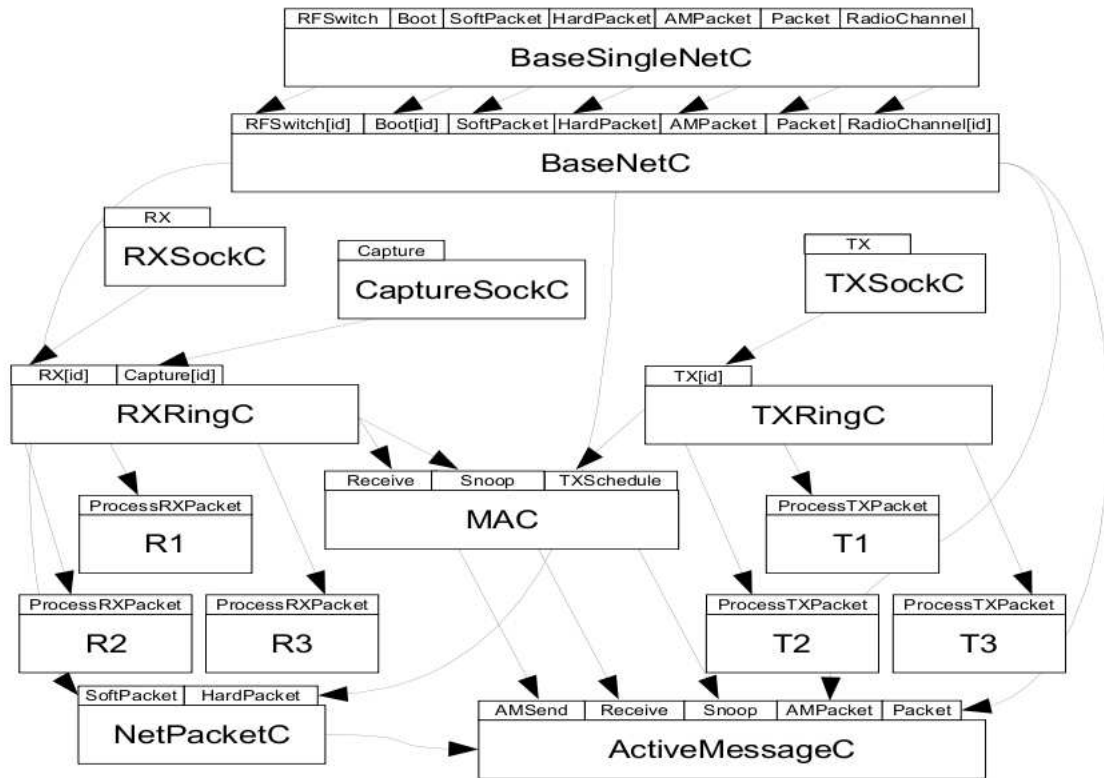


Figure 3.23: TinyNET Architecture

Gateway) never reached the Mobile. We have developed a simple network sniffer that captures and shows the in transit packets, we discovered that the destination address in the response packet was set to `0xFFFF` which is the broadcast address, we have discovered that the problem lies in the implementation of SerialTun: regardless of the IPv6 source address of the packet created by the Gateway the corresponding 6lowPAN packet has broadcast address. We have corrected this problem, now SerialTun translates correctly IPv6 addresses in `TOS_NODE_IDS`.

### 3.6 Gateway

We have already discussed how a plug-in for the gateway is made: there are two part, a socket (an object derived by the `BWSocket` class) and the actual plug-in that communicates with motes through the socket and with the user using standard Internet technologies. In Fig. 3.24 we can see the interactions that Sockets and Plug-in have with the rest of the system, the solid lines represent the data flow from the mote to the server, the others the inverse flow. The Gateway is developed using the `Sockets` library [15].

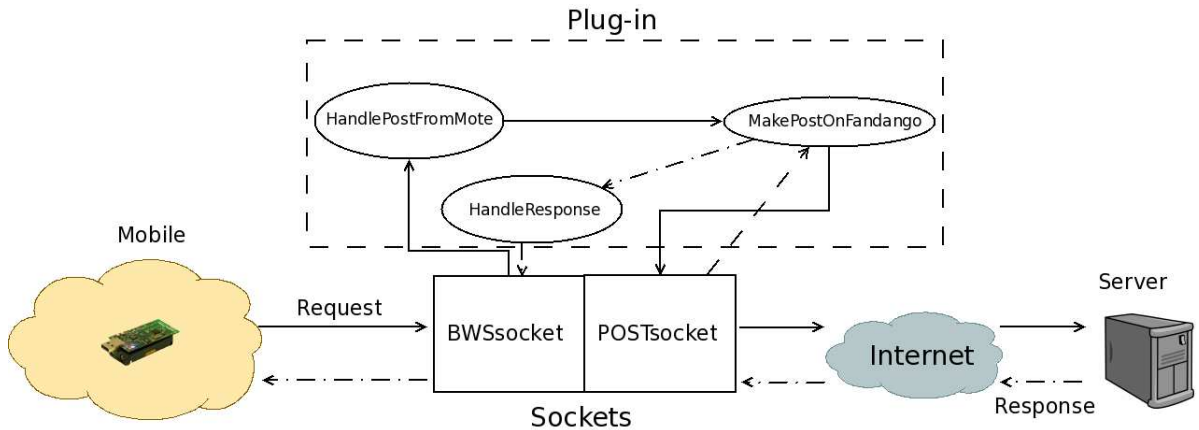


Figure 3.24: Plug-in interactions

### 3.6.1 Socket

The class performing the socket operations `LocalizationBwsSocket` is specified into two files: `LocalizationBwsSocket.hpp` and `LocalizationBwsSocket.cpp`. In the first file are specified the methods of the class, in the second these methods are implemented. For our specific problem the only method that is overwritten is

```
HandleBwsRequest(in6_addr a, port_t p, bws_header hdr, char* buf, int len)
```

this method is called by `OnRawData()`, a method belonging to `BWSocket`, this method verifies if the received UDP packet is correct, if so `HandleBwsRequest()` is called. The parameters that are passed to the method are:

- `in6_addr a`: The IPv6 address of the sender.
- `port_t p`: It is the source UDP port, it can be used by the plug-in to reply to the request on the right port.
- `bws_header hdr`: It is the structure that contains the data belonging to the compressed header of the message, for example in this structure we can find a pointer to the URL used in the request.
- `char* buf`: Pointer at the buffer that contains the actual payload of the request
- `int len`: Length of the payload

These data are passed to the appropriate method of the plug-in. In this case we have only one method, thus we don't need to examine the packet, in more complex application we can examine the packet and invoke a particular method given a particular request. In this case we invoke the method

```
void HandlePostFromMote(in6_addr a, port_t p, uint8_t tid, uint8_t content,
char* buf, int len)
```

It has to be stressed in this moment that the socket have a pointer to the plug-in object, but also the the plug-in has a pointer to the socket, so each object can reach the other without any problems. This method needs two others parameters obtainable by the header:

- `uint8_t tid`: it is the TID, already described in the CoAP section.
- `uint8_t content`: it is the content type identifier, already described in the CoAP section.

An important thing has to be underlined: even if the the URL is part of the header it is stored in the payload of the UDP packet so `*buf` points at the start of the URL which is followed by the actual payload, so it is needed that the socket adds to the pointer the length of the URL (this information is contained in `hdr`).

### 3.6.2 Plug-in

The plug-in is a specialization of the class `Thread` [15]. When the object is created before its actual execution the method `Init()` is called. Its only task is to add the current plug-in to the `plugin_handler` defined in the `main()` method of the application. We can access to it thanks to the keyword `extern`, we only need to reference to the object in the plug-in using that keyword and we can get access to the object defined in the `main()`. The main method is the `Run()` method, it is called by the thread when it is ready to run, it stays in a loop until the execution of the process is terminated. It instantiates the `SocketHandler` (`h`) which is an object that generates events for the actual socket. After that the `LocalizationBwsSocket` is generated and connected with the `SocketHandler`. The plug-in and the `LocalizationBwsSocket` are connected setting the respective pointer, after that the socket is bound to the port chosen for this service, in this case the port is 32180. After that it enters in the following loop:

```
while (IsRunning() && h.GetCount() && !halt)
{
    h.Select(1,0);
}
```

`IsRunning()` returns true if the current plug-in (`Thread`) is already running. `GetCount()` is a method of the `SocketHandler` object, returns how many sockets are controlled by it. `halt` is an other `extern` variable, it is setted by the main program that instantiates and controls all the plug-in. `Select(long sec, long usec)` is a method that returns after `sec` seconds and `usec`  $\mu$ s or if an event is detected, so at least every 1 seconds the conditions are checked, if they are all true the program continues, otherwise the plug-in is terminated and the object destroyed. We can now examine

the `HandlePostFromMote()` method, its prototype was already been discussed, so we can focus on the implementation. It has to examine the data contained in the payload of the header, and make a `POST` on a servlet running on the Fandango server. After unpacking the data it pass them to an other method:

```
void makePostOnFandango(const string& ip, unsigned long port,
int *ch, int *nodes, int *rssi, int n)
```

1. `ip`: It is the IP source address
2. `port`: It is the UDP source port
3. `*ch`: It is an array containing the four channel used by the mobile for its measurements.
4. `*nodes`: It is an array containing the `TOS_NODE_ID` of the Anchors that have been used for the localization.
5. `*rssi`: It is an array containing the RSSI measurements of the incoming messages.
6. `n`: It is the number of Anchors that have been used for the localization.

This method uses an other socket provided by [15]: the `PostSocket` which has to be instantiated in the same way used for the `BWSsocket`. The only difference is that a buffer (`*buf`) of length `len` has to be provided, in this buffer the data contained in the response coming from the server will be stored:

```
void SetDataPtr(unsigned char *buf,size_t len)
```

After that we can add all the fields at the `POST` request using the following method.

```
void AddField(const std::string& name,const std::string& value);
```

`name` is the name of the field, while `value` is its value. Using this name we will access to its value. We can now issue the `POST` using the following lines of code (where `h` is a `SocketHandler`) and `*body` is the pointer to the buffer previously passed.

```
sock.Open();
h.Add(&sock);
h.Select(1,0);
while (h.GetCount())
{
h.Select(1,0);
}
HandleResponse(body);
```

The code is very simple, the socket is opened and the program remains in the loop until a response from the server arrives, when it arrives it is passed to the method `HandleResponse()`.

```
void HandleResponse(char* body)
```

It is the dual of the `HandlePostFromMote()` method, it simply takes the data coming from the server, creates a new payload and sends it to the mote that issued the request, the correct address and port are present in the response. The method fulfill this duty using :

```
int BwsSendResponse(in6_addr a, port_t p, uint8_t tid, uint8_t code,
uint8_t content, char* buf, size_t len)
```

`code` is the code associated to the result of the request, for example it can be `OK` or `FAILURE` or other codes. The packet sent to the mobile contains three coordinates, `X`, `Y`, `refSys`, the last is an unsigned integer defined in one byte. The other two are double, but their precision is setted to the cents, we have studied the possible data to be sent and we have decided that a signed integer defined on 1 byte for the integer part and an unsigned integer defined on 1 byte for the decimal part can be used, thus we use 2 bytes for `X` and 2 bytes for `Y`, see Fig. 3.25.

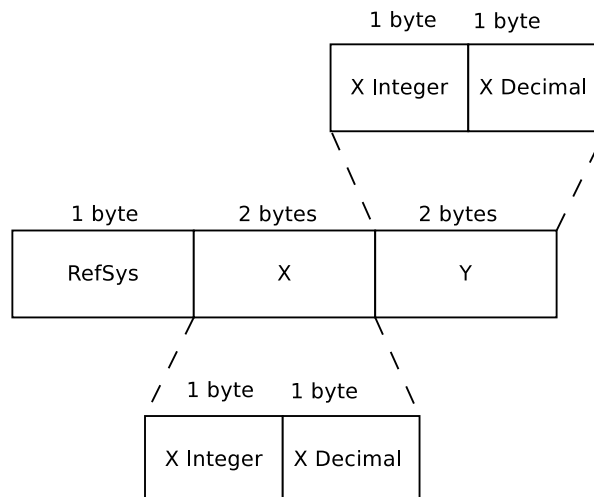


Figure 3.25: Response received by the mote

Now our plug-in is ready, the main problem is how to run it. This is a very simple task, we have to modify the main program (`GWMain.cpp`) adding the instantiation of the plug-in.

### 3.7 Fandango

In this work we have implemented a plug-in for Fandango, it receives the RSSI measurements through the POST made by the Gateway, creates the query needed

to find the position of the mobile, draws a “X” where the mobile is supposed to be and replies to the request sending to the mote the coordinates. In Fig. 3.28 we can see a sketch of the interactions between the various part of the Plug-in.

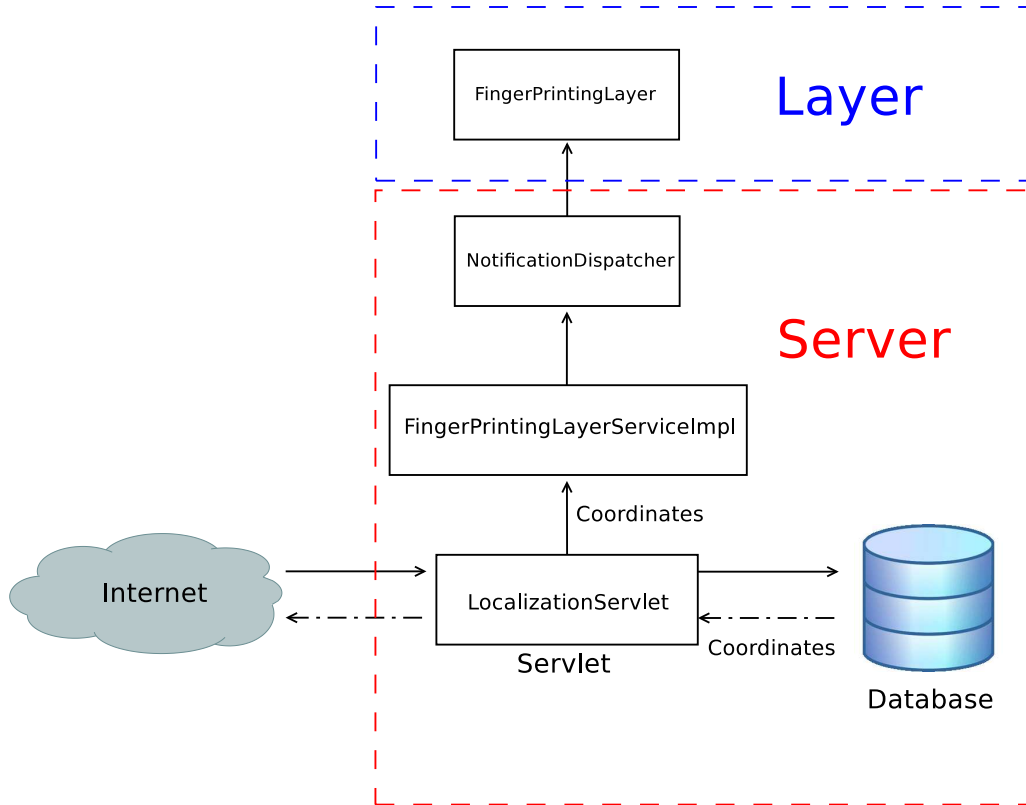


Figure 3.26: Plug-in interactions

The plug-in exploits the Servlet, GWT e JDBC technologies already illustrated. It is necessary to underline the fact that the Gateway does not keep any informations about the mobile (IP address and UDP port), they are sent within the POST message, the Servlet has to copy these informations in the response. This choice was made for allowing the Gateway to serve more than one request at once. The response is a string in which the information required are written using the standard URL encoding:

```
param1=val1&param2=val2....
```

In Fig. 3.27 are illustrated the interactions between the methods of the classes that form the plug-in. `doPost()` simply pass the data to `processRequest()` which performs the query, extracts the data, fill the `HttpServletResponse` object and invoke the `messageReceived()` method of the `FingerPrintingLayerServiceImpl` class. The `LocalizationServlet` can access to the used instance of the

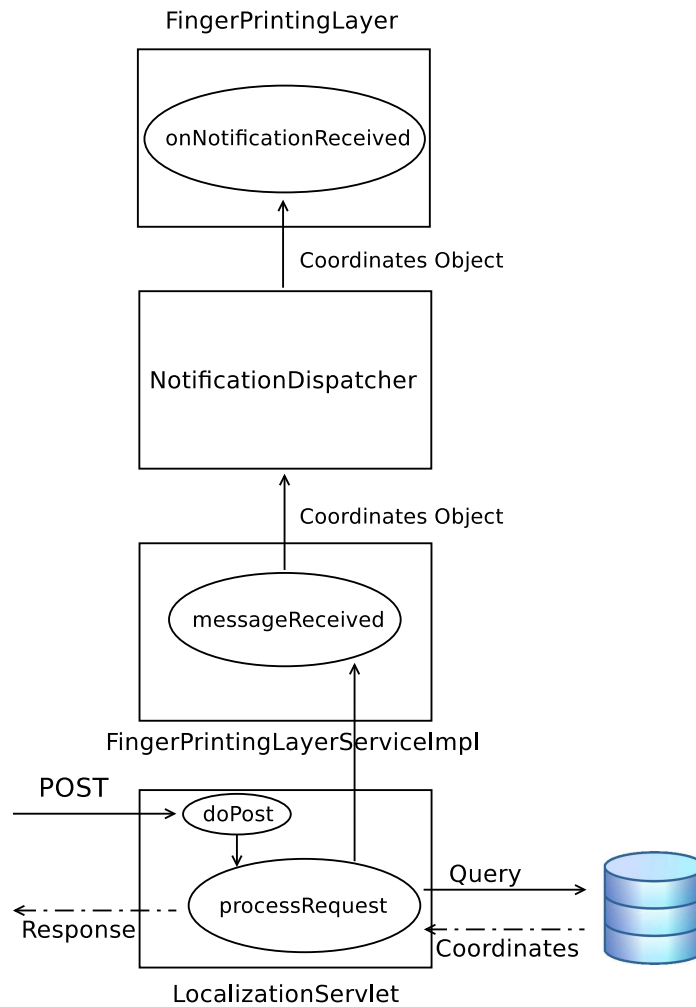


Figure 3.27: Plug-in methods and interactions

`FingerPrintingLayerServiceImpl` class inserted into the `NotificationDispatcher` using the `signal` pointer initialized as follows:

```

NotificationPlugin[] plug = NotificationDispatcher.getNotifier().getPlugins();
for (int i = 0; i < plug.length; i++) {
    if (plug[i] instanceof FingerPrintingLayerServiceImpl) {
        signal = (FingerPrintingLayerServiceImpl) plug[i];
        break;
    }
}

```

The first line allows the object to get access at all the plug-ins registered to the `NotificationDispatcher`, then we examine all the array element, when we found an

element which is instance of the class `FingerPrintingLayerServiceImpl` we have found the instance in which we were interested and we can terminate the search. The query used in the servlet is dynamically created, performing the 1-NN algorithm, it selects all the records of interest, which are the records in which the source and the channel are present in the measures. After that, using the `if` statement of the query (see [20] pp. 230 and following) at each stored RSSI measurement is subtracted the measurement done on the same channel from the same source (we refer at these values as *delta*). After that the square sum of the *delta* is computed, the sum is grouped by the coordinates of the point where the fingerprint was taken, the point with lowest “distance” is chosen as estimated position of the mobile.

After that the `FingerPrintingLayerServiceImpl` object can signal the event to the layer part using the classes examined previously. One thing has to be stressed, the Notification system can pass data using classes that implement the `Datatype` interface. We have written the `Coordinates` class for this purpose. The object is

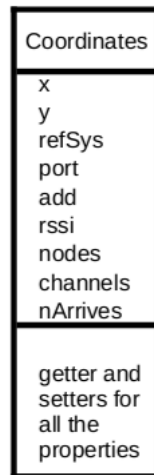


Figure 3.28: The `Coordinates` class

passed to the `FingerPrintingLayer`, which has simply to draw the chosen image on the map, for doing this we use the `OverlayX` object, a specialization of the class `Overlay`, provided by the GWT API, it is an object that contains an image and the coordinates needed for its correct positioning. The final GUI that the user can see is presented in Fig. 3.29, on the right are presented various informations about the request (Source IP address, Source Port, RSSI readings), on the map a red “X” is drawn.



Fandango - WISE-WAI Testbed Management System

View Plugin List

- Sensei
- Reservations
- Synapse
- FingerPrinting
- Sensors
- Labels
- Preferences
- Routing Layer
- Firmwares
- Maps
- Permissions
- Hello



FingerPrinting Layer Options

Source Address: 2001:638:709:1234::ffe:1  
Port: 61622

Source	Ch 15	Ch 18	Ch 22	Ch 26
200	-12	-10	-11	-14
186	-5	-7	-3	-9
188	-15	-18	-14	-16
205	-17	-14	-19	-12
185	-8	-6	-10	-7

Figure 3.29: The GUI of the Fingerprinting plug-in



## Chapter 4

# Performance Evaluation

In this chapter we will discuss the performance evaluation of the proposed system.

### 4.1 Test Environment

The tests were carried out in the sigNET laboratory, this room is approximately 10m x 10m. The room has an high density of nodes since 48 nodes are present. In Fig. 4.1 we can see the motes (the green circles), each node has a nick name (the alphanumeric string in the circle, such as `t1`). The red rectangles are the ALIX, which are small computers used for programming and debugging the motes. When the user wants to program one or more mote he has to send commands to the WISEWAI-Server, it sends these commands to the ALIXs which perform the actual operation: programming the node, power off/on of the node, capturing packets sent via the serial port, etc. We have used, for our measurements, a modified mote (Fig. 4.2) with a SMA connector that allow the connection of the mote with an external antenna. We have decided to use an external antenna (Fig. 4.3) since the patch antenna provided by the mote would be too much affected from the orientation of the mote respect to the room.

On each room with motes is defined a local cartesian coordinates system, every room has its own origin point, which is called `refSys`. Every `refSys` is indicated with its own coordinates in the more general cartesian coordinates system of the department (Fig. 4.4). Therefore if we want to calculate the position of a point in the building we have to add the coordinates in the `refSys`, say `X`, to the coordinates of the `refSys` point in the general coordinates system, say `refX` obtaining: `realX=X+refX`. We can do the same with the other coordinates. However we needs the coordinates expressed in Latitude and Longitude for displaying the symbol for indicating the position of the mobile in the map. Each `refSys` has also its Latitude `refLat` and Longitude `refLong` and the rotation `rot` of its axis respect to the coordinates system of the Earth. Having the coordinates `X`, `Y`, of the point respect to the `refSys` we can use these simple formulas, obtaining `pointLatRad` and `pointLonRad` which are the point latitude and longitude expressed in radians.

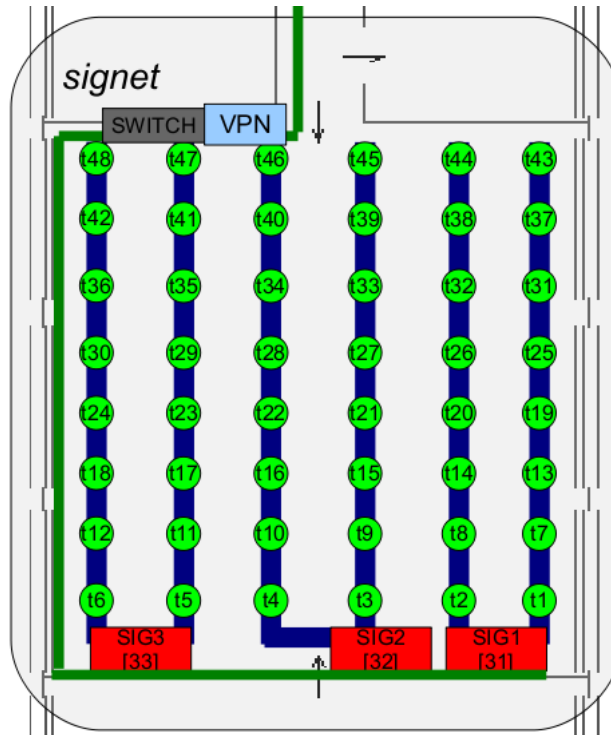


Figure 4.1: The room and the motes deployed

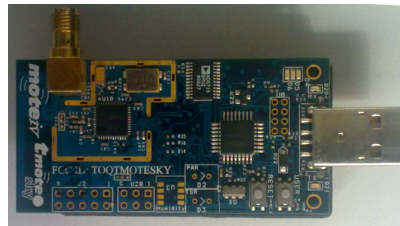


Figure 4.2: The modified mote

```

rotX = x * cos(rot) - y * sin(rot);
rotY = x * sin(rot) + y * cos(rot);
pointLatRad = rotY / EARTH_RADIUS+refLat;
pointLonRad = rotX / (EARTH_RADIUS * cos(pointLatRad))+refLong;

```

The tests were carried out in the sigNET lab, using 18 positions, i.e. we have obtained in the measurement phase 18 fingerprints, the positions are sketched in Fig. 4.5 using the “X”, this is called *configuration 1*. After these first tests we have used an higher number of points: 60 positions (*configuration 2*), measuring other 3 points in the rows between the previous measured positions, in each row we have that two points are distant 50 cm.



Figure 4.3: The antenna

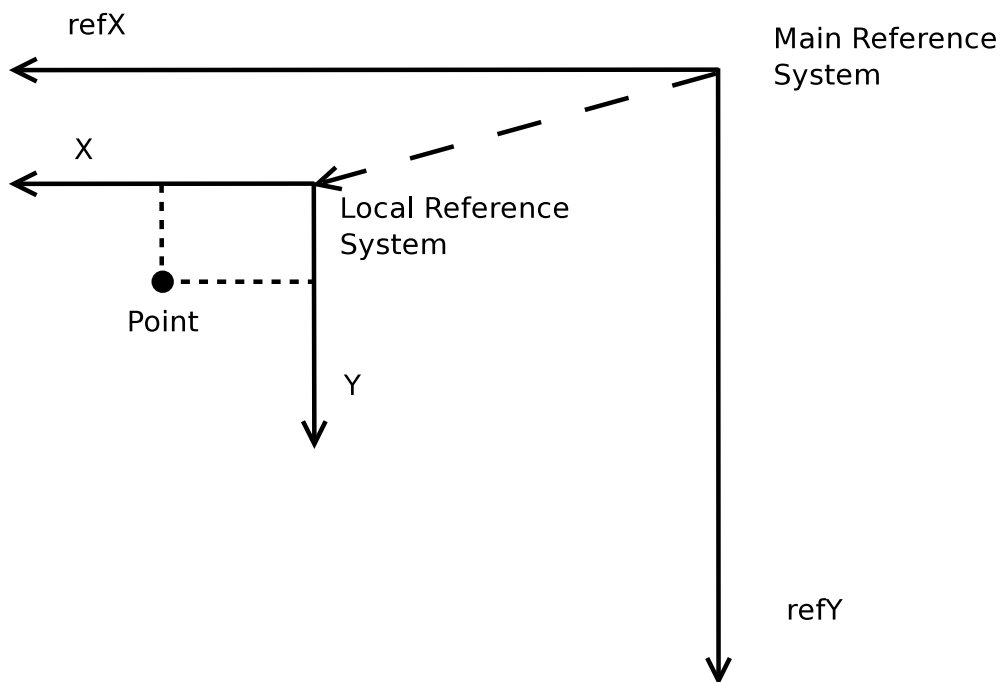


Figure 4.4: The coordinates system

We have carried out various tests, using all the notes available or a subset, thus simulating more or less dense areas. All the nodes send packets using a power level of  $-1$  dBm, (0.79 mW). For each kind of area we have made localization test positioning the mote exactly on the point of which we have already had a fingerprint. In this case we say that the measurement is *correct* if the system identifies the point or if

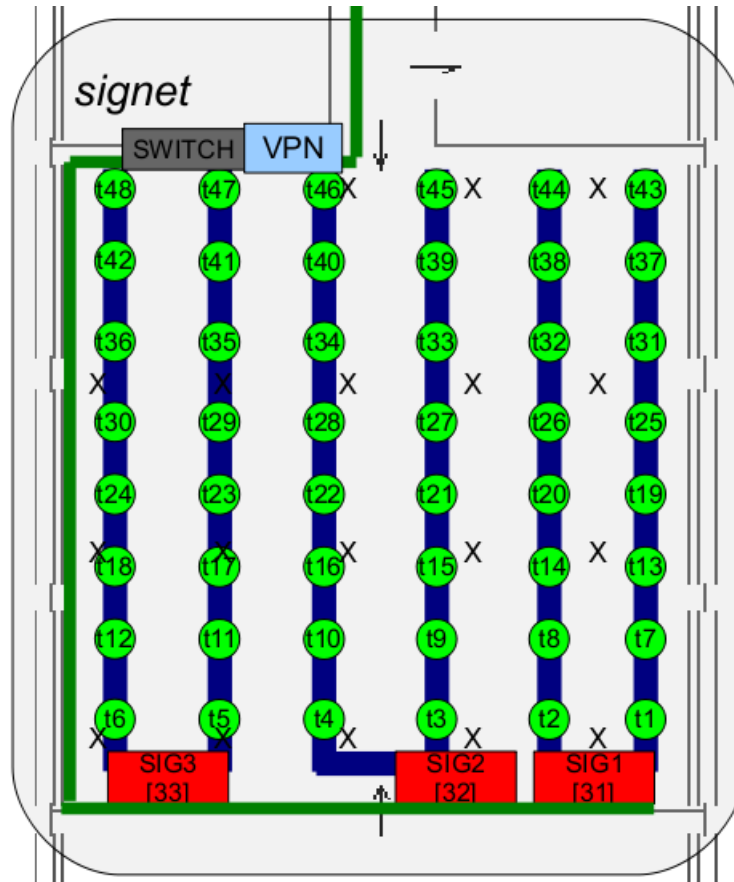


Figure 4.5: The positions on which we have measured the power, in *Configuration 1*.

the identified point has a distance from the actual position of about 2 meters, we say *wrong* otherwise. We have made other tests staying away from a fingerprinted point, the response is *correct* when it is individuated the closest position or if the identified position has a distance from the closest position of about 2 meters, otherwise the response is *wrong*. It has to be stressed, however, that the tests were carried out in a real environment, with people moving in the room, without disabling the wireless network and other wireless services.

## 4.2 Results

We report the results of our tests in the following table, the letters without \* are tests made in the first configuration, the others are made in the second configuration. We have also made a test lowering the power emitted by the Achors.

Test	Correct Answers (%)
A	97.5%
B	84%
C	72%
D	76%
E	80%
F	70%
G	55%
H	90%
B*	96%
G*	92%
A**	85%

- A) Few people moving, exactly on the measured point, all the Anchors are active.
- B) Many people moving, exactly on the measured point, all the Anchors are active.
- C) Few people moving, exactly on the measured point, half of the Anchors are active, the active nodes are disposed like in a chessboard.
- D) Many people moving, exactly on the measured point, all the Anchors are active, different antenna from which used in the Measurement phase.
- E) Many people moving, exactly on the measured point, all the Anchors are active, different antenna and Mobile node from which used in the Measurement phase. The Mobile used in the measurement phase is manufactured by XBOW, the Mobile used during the localization is manufactured by MoteIV.
- F) Few people moving all the Anchors are active, near the border between the zone of two measured positions.
- G) Many people moving all the Anchors are active, near the border between the zone of two measured positions.
- H) Like experiment A) in which after the Measurement phase we have waited 7 days for making the Localization phase, in that time the laboratory where used as usual, people moved chairs, laptop, etc.
- B\*) Many people moving, exactly on the measured point, all the Anchors were active.
- G\*) Many people moving all the Anchors are active, near the border between the zone of two positions.
- A\*\*) Few people moving, exactly on the measured point, all the Anchors are active, the Anchors transmit at -5 dBm.

As can be seen the tests in the first configuration shown good result in all the case except the last, which is the more realistic case, thus we have used the second configuration reaching good performance. The problems in the first configuration are probably due to the fact that we are near the motes that reply to the request, thus a slightly change in the position can dramatically change the received power, we can see 4.6 for a change in the position of 1 meter (from  $(0 - 1)$  to  $(-1 - 1)$ ) we can assist to a great change of the received RSSI, using a smaller sampling interval attenuates this problem. Probably using anchor nodes with other types of antennas the sampling interval can be increased, but we have not enough modified nodes for carrying out meaningful tests. However fingerprinting an area like the sigNET lab, with 60 positions, takes about 1 hour. The last test demonstrates that using anchors capable of emitting more power would lead to better performance.

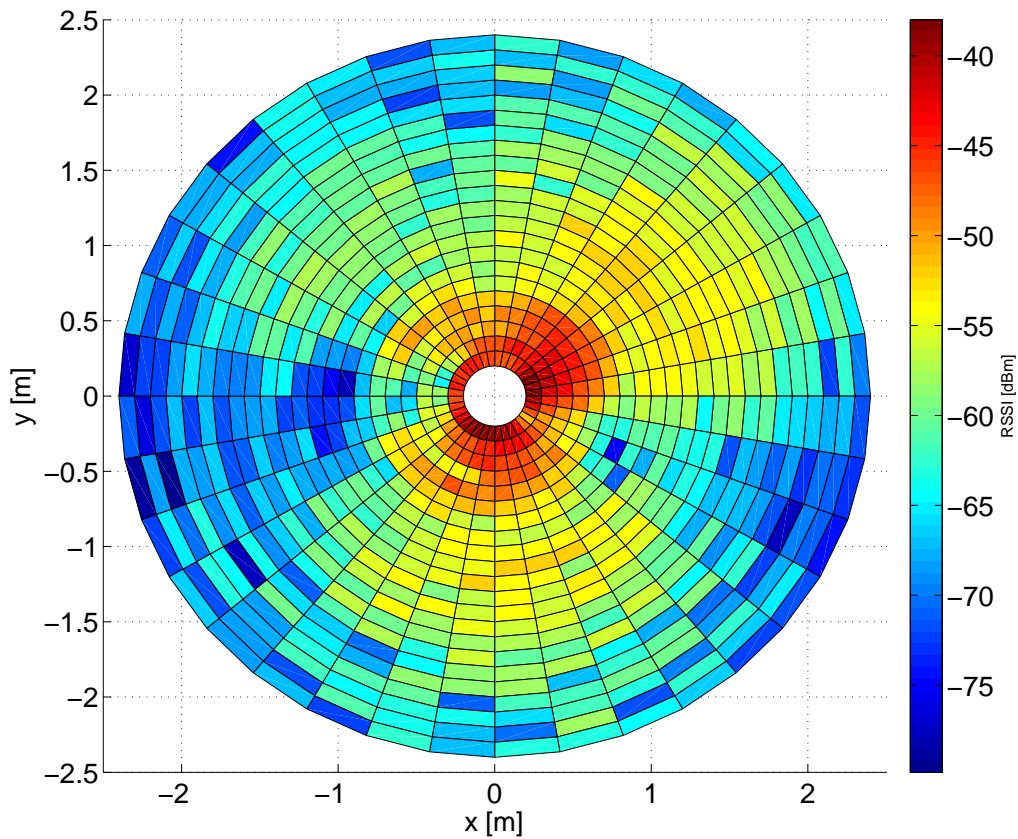


Figure 4.6: Empirical radiation diagram of the patch antenna of the TelosB mote, courtesy of Andrea Bardella, sigNET lab.



## Chapter 5

# Conclusions

From the experimental results we can obtain some interesting conclusions:

- Fingerprinting in WSN is an applicable method, while we have shown that in realistic environment we can reach an acceptable performance when we require coarse localization.
- The performance degradation due to the difference from the mote that has made the Measurement phase and the mote that is making the Localization phase is acceptable, thus systems for the dynamic compensation of these differences are not needed.
- Performance decrease due to human actions in the area is very high, thus systems for human position estimation and compensation are needed.
- When a node is positioned between two Measured points the performance decrease is great, thus a great density of measured points are needed.

The performance gap between our solution and the proposed solutions using WI-FI technologies is due basically to lower power usage and worse antennas. The main problem in our opinion in the Fingerprinting method is the Measurement phase in which an operator is required to move the antenna that is making the measurements, a method for automatic Measurement can be very interesting. However Fingerprinting has shown good resistance to changes of the environment, thus the fingerprints gathered during the Measurement phase can remain valid for much time.

Our work has interesting side effects: we have developed systems for allowing a node to access to resources on a standard web server using the CoAP system developed in the SigNET laboratory. Secondly we have developed a module for allowing a mote to query other motes in a very quick way. The simulations were carried out in a very dynamic environment, with people walking, moving chairs, laptop, etc.

Future work can be the exploration of others algorithms on the server-side part, in [24] the authors used k-NN method and Probabilistic method, showing that the

first one shown a slightly better performance than the second, but in the article is not defined the probabilistic model used, thus, since good models for RSSI readings were developed, their use may improve the results. Other methods like Neural Networks can be used, all these changes can be freely done without modifying the architecture of the system.

In our opinion Fingerprinting can be a good localization method in WSN while requiring low power consumption e simple algorithms, obviously this simple system leads to low performance in extreme environment, but for achieving good performance motes with better radio hardware, capable of emitting more power are needed.

# Bibliography

- [1] <http://cariparo.dei.unipd.it/> - WISE-WAI Project official site
- [2] H. Liu, H. Darabi, P. Banerjee, J. Liu “Survey of Wireless Indoor Positioning Techniques and Systems”, in *IEEE Trans. on Systems, Man and Cybernetics - Part C: Applications and Reviews*, Vol. 37, No. 6 November 2007 pp. 1067-1080
- [3] A. Zanella, M. Zorzi, ‘Reti di sensori: dalla teoria alla pratica’, in *Notiziario Tecnico Telecom Italia*, Year 15, No. 1, June 2006, pp. 47-59 .
- [4] [http://www.willow.co.uk/TelosB\\_Datasheet.pdf](http://www.willow.co.uk/TelosB_Datasheet.pdf) - TelosB datasheet
- [5] <http://www.sensei-project.eu/> - SENSEI official web site
- [6] <http://tinycos.net/> - TinyOS official web site
- [7] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, D. Culler “The nesC Language: A Holistic Approach to Networked Embedded Systems” - [www.cs.berkeley.edu/~pal/pubs/nesc.pdf](http://www.cs.berkeley.edu/~pal/pubs/nesc.pdf)
- [8] P. Levis, D. Gay “TinyOS Programming” - *Cambridge University Press 2009*
- [9] <http://code.google.com/intl/it-IT/webtoolkit> - Google Web Toolkit Homepage
- [10] M. Visonà “Fandango - User and developer manual”.
- [11] Matus Harvan “Connecting Wireless Sensor Networks to the Internet – a 6lowpan Implementation TinyOS 2.0” - [www.eecs.jacobs-university.de/archive/msc-2007/harvan.pdf](http://www.eecs.jacobs-university.de/archive/msc-2007/harvan.pdf)
- [12] Chipcon “CC2420 Datasheet”  
- <http://inst.eecs.berkeley.edu/~cs150/Documents/CC2420.pdf>
- [13] A. Castellani, P. Casari, M. Zorzi - “TinyNET: A Tiny Network framework for TinyOS” - in *Proc. of the 2009 International Conference on Wireless Communications and Mobile Computing*
- [14] N. Freed, N. Borenstein - “Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types” - IETF RFC 2046

- [15] <http://www.alhem.net/> - Sockets Library Official Site
- [16] <http://www.jsptube.com/servlet-tutorials/simple-servlet-example.html>  
- Java Servlet Tutorial
- [17] James F. Kurose, Keith W. Ross “Computer networking: a top-down approach”  
- *Pearson Education 2009*
- [18] <http://code.google.com/intl/it-IT/webtoolkit/doc/1.6/DevGuideServerCommunication.html> - Google Web Toolkit RPC introduction
- [19] <http://www.wireshark.org/> - WireShark Network Analyzer Official web Site
- [20] Michael Kofler “The Definitive Guide to MySQL5” - *Apress 2005*
- [21] <http://java.sun.com/products/jdbc/driverdesc.html> - Sun Developer Network Tutorial on JDBC technology.
- [22] [http://www.mysql.com/?bydis\\_dis\\_index=1](http://www.mysql.com/?bydis_dis_index=1) - MySQL Official Web Site
- [23] <http://download-llnw.oracle.com/javase/1.4.2/docs/api/java/sql/ResultSet.html> - JAVA API, ResultSet description
- [24] Y. Zhao, H. Zhou, M. Li, R. Kong - “Implementation of Indoor Positioning System based on Location Fingerprinting in Wireless Networks” - in *Proc. of the 4th International Conference on Wireless Communications, Networking and Mobile Computing, 2008. WiCOM '08*
- [25] IEEE Computer Society - “IEEE Standard for Information technology — Telecommunications and information exchange between systems — Local and metropolitan area networks — Specific requirements Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)” - IEEE 8 September 2006
- [26] N. Benvenuto, G. Cherubini - “Algorithms for Communications Systems and their Applications” - *John Wiley and Sons, 2002*
- [27] D. Moss, J. Hui, P. Levis, J. Choi - “TEP 126 (TinyOS Extension Proposals) - CC2420 Radio Stack” - <http://www.tinyos.net/tinyos-2.x/doc/pdf/tep126.pdf>
- [28] Z. Shelby, B. Frank, D. Sturek - “Constrained Application Protocol (CoAP) draft-ietf-core-coap-01” - <https://datatracker.ietf.org/doc/draft-ietf-core-coap/>

# Acknowledgments

Desidero ringraziare tutti coloro che in questi anni mi hanno supportato e sopportato: i miei genitori, i miei amici, i miei compagni di corso.

Un ringraziamento particolare va senza alcun dubbio a mia madre e mio padre per la loro continua vicinanza.

Desidero inoltre ringraziare il Prof. Michele Zorzi, gli Ingg. Angelo P. Castellani e Paolo Casari per avermi dato l'opportunità di lavorare con loro e per avermi guidato nel percorso della Tesi.

Infine un ringraziamento particolare va ai ricercatori del sigNET che con la loro simpatia e con le loro capacità mi hanno aiutato nella parte finale del lavoro.

A tutti voi va un sentito grazie.