Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

# MASTER THESIS

**TITLE: A Tool for Rapid MAC Protocols Prototyping and Designing for Wireless Sensor Networks**

**MASTER DEGREE: Master in Science in Telecommunication Engineering & Management**

**AUTHOR: Luis Miguel Amorós Martínez**

**DIRECTOR: Prof. Dr. Petri Mähönen; Xi Zhang, M. Eng. and Junaid Ansari, M.Sc.**

**DATE: September 25th 2012**

# PREAMBLE

This thesis presents the design, implementation and evaluation results of a toolchain which enables the user to rapidly prototype and design MAC protocols for Wireless Sensor Networks.

During this work, I have worked in collaboration with Noemi Arbós and while we worked on different parts, our thesis supplements each other. Therefore, many design and implementation blocks are tightly coupled to her thesis titled *Reconfigurable Medium Access Control Solutions for Resource Constrained Wireless Networks* [1].

First of all, I would like to dedicate this thesis to my family for its constant support, especially in those moments where I had to take important decisions. For that reason, Erasmus experience has been difficult because of the distance that separates us. Thanks for making me the person who I am now. In addition, I would like to dedicate it to my girlfriend too, because she has been an important element in my life during the last 6 years as well as an excellent work partner with an incredible patience. I thank also to M. Sc. Junaid Ansari and M. Sc. Xi Zhang for their supervising, ideas and support, even when situations were adverse. Finally, I dedicate this thesis to the incredible and crazy people I met in Aachen, without them this Erasmus experience would had never been the same.

"Gewisse Bücher scheinen geschrieben zu sein, nicht damit man daraus lerne,
sondern damit man wisse, dass der Verfasser etwas gewusst hat."

from "Maximen und Reflexionen", Johann Wolfgang von Goethe (1833, p. p.)

# Contents

# ABSTRACT

Wireless Sensor Networks (WSNs)consists of several resource constrained sensor nodes distributed over an specific geographical area. WSNs are typically energy constraint due to the fact the sensor nodes are battery powered.

Medium Access Control (MAC) protocols used in WSNs are usually designed to be power aware, i.e., they are more energy efficient than MAC protocols used for other ad-hoc wireless networks such as IEEE 802.11 [2]; to increase the lifetime of the nodes.

Traditional MAC protocol implementations are done for specific hardware platforms using a monolithic approach. Therefore, it is very difficult to port from one platform to another without modifying the whole implementation protocol. This reduces code reusage and increases the implementation efforts.

We have designed and implemented a toolchain which allows to design and prototype MAC protocols for WSNs in a simple manner. In addition, it allows non-specific sensors users to implement and execute them in sensor nodes without worrying about technical specifications of the platforms. The toolchain has been implemented in TinyOS using a component-based design. Special care has been taken to ensure hardware independence of the protocol implementations described in this thesis has been integrated with [1] to allow runtime reconfiguration of MAC protocols.

We have evaluated our toolchain against monolithic implementations in terms of memory consumption and execution time. The results show that the toolchain introduces an acceptable memory and execution time overhead, less than 5 %, compared to the monolithic approach and substantially eases the implementation efforts.

# 1

# INTRODUCTION

Smart environments are becoming more and more popular nowadays. They have provoked an evolution in many different aspects of the society, such as industrial, home or health-care applications [3]. To ensure the operation of smart environment, information about their surroundings as well as about their internal situation are needed.

Wireless Sensor Networks (WSNs) are often used to obtain the surrounding information as they are able to detect many different physical magnitudes, such as temperature, humidity, vibration, pressure, motion, etc. [4]. Mainly, a WSN consists of several nodes working in a cooperative way and distributed over a specific geographical area which sense the mentioned physical magnitudes, collect relevant data and send them to a main location in order to analyse and process the data obtained.

Sensor nodes are resource constrained with very limited processing and communication functions. For that reason, it is the coordinated and cooperative effort of these sensing devices that brings a significant impact on a wide range of applications in the areas of science and engineering, military settings, critical infrastructure protection and environmental monitoring.

Size and cost constraints on sensor nodes result in corresponding constraints on resources such as energy and memory consumption. Sensor nodes are usually battery powered, therefore, efficiently usage of the limited energy is the key to maximally extend their lifetime. Medium Access Control (MAC) protocols can improve the energy consumption efficiently and increase the overall lifespan of WSNs since they govern several sources of energy spending, such as the radio chip. For instance, the network usually consumes most of the energy while coordinating the access to the medium between all the transmitter nodes and ensuring reliable communication. The design of a power aware MAC protocol can lead to better energy efficiency for sensor nodes.

MAC implementations are usually closely coupled with the underlying hardware due to power-management schemes like sleep-scheduling policies and transmission power control which are specific for each radio stack layer. This fact restricts the portability of MAC implementations among different platforms. In order to address this shortcoming, we have developed a framework for enabling rapid MAC protocol prototyping and designing using *TinyOS*. *TinyOS* supports component based implementation and allows us to separate all basic MAC features such as *Carrier Sensing* or *Send Packet*. Once the basic MAC components are clearly separated and defined, it is easier to build a MAC solution by using them as if filling a puzzle with matched pieces. In addition to the basic blocks, some commonly used functions such as *Low Power Listening*, are also identified and encapsulated within reusable components resulting in a high degree of code reusability and MAC prototyping.

A Meta-language has been designed in order to allow the user design different

MAC implementations in a simple manner. This Meta-lanaguage uses a syntax very similar to popular languages, such as C, and it also expresses MAC specific syntax. Another language is defined to be understood by the sensor node which is called MetaNode-language. A parser has been designed and implemented to translate the user input Meta-language to MetaNode-language for sensor nodes to operate on. The sensor nodes executes the described MAC protocols by interpreting the code in MetaNode-language.

Overall, in this thesis we present a toolchain which is designed for the prototyping and designing of MAC protocols. It is implemented for TelosB [5] and Mica2 [6] as a proof of concept. This thesis is organized as follows. Chapter 2 describes the related work proposed by the community giving solutions for hardware independence, component-based design, as well as some introduction to typical MAC protocol implementations for WSNs. Chapter 3 explains the design of the framework based on component-based concept, the different languages used and the parser that is needed to make the translation between them possible. The implementation is presented in Chapter 4 along with the description of the problems encountered and solutions taken during the implementation. The evaluation and experimental results of our toolchain are presented in Chapter 5 to show the effectiveness and the possible disadvantages of our solution. At the end, we present some conclusions after finishing our work and point out possible future works.

# 2

# RELATED WORK

In the past few years, many Information Technology (IT) research communities are focusing in developing prototypes and designs of efficient MAC protocols for WSNs. However, most of the solutions found are specifically design for certain sensor platforms. In this chapter, a brief review of some approaches related to this topic are given below. Moreover, there is an introduction to MAC protocols specifically designed for WSNs.

## 2.1 COMPONENT-BASED DESIGN

*Component-based design* is becoming very popular as one of the techniques used in software design. This concept involves decomposing a system into independent and simple units with certain functionalities e.g. control radio hardware or switch on a led, as explained in [7] and [8]. The components should be observed as black boxes with defined inputs and outputs which can be developed and deployed independently. If the encapsulation is good, end users only need to know the specific interfaces provided by the components to be able to use them. There are many component-based platforms like COM/DCOM [9], .NET [10], EJB [11], TinyOS/nesC [12].

This approach has many advantages. It allows *dynamic composition*, i.e. applications made based on components that can be easily modified by adding/removing components, or changing the relationships between them. For example, as shown in FIGURE 2.1, a user can create an application connecting components to each other like pieces of a puzzle. Another important advantage is that components are reusable and extensible. Therefore, new components can be built by combining other components, thus decreasing the code size of the final application, which facilitates the growth of the maturity of the system. In addition, when an application requires changes, a modification in a single component does not have a great impact in the whole system because it can be easily done due to decoupling among components. Thereby, components can be inserted and deleted in the system in an easy way according to the needs in order to achieve the adaptation. Finally, it gives a clear structure of the system, because each functionality can be represented by an specific component. The business logic becomes more explicit as well due to the reasonable decomposition and encapsulation. According to these advantages a rapid MAC prototype can be realized by using this approach, as the components can be reused along the whole MAC implementation.

Component-based design has been used in order to achieve adaptive MAC protocols design through the *Decomposable MAC Framework* [13]. It has been demonstrated how MAC protocols are decomposed into independent elementary blocks based on

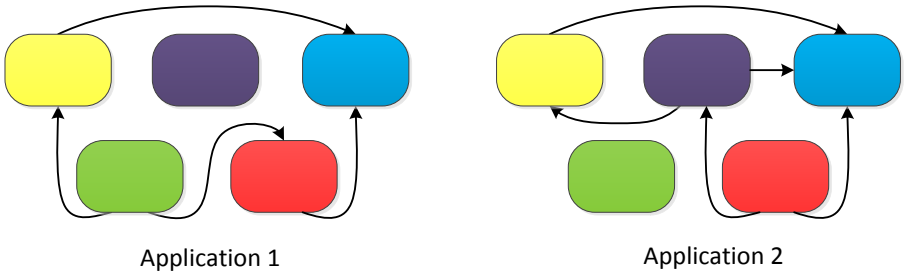


FIGURE 2.1: Different applications realized by a component-based design approach.

their services and how elementary blocks can be inserted and deleted as it is needed to achieve the adaptation.

Similarly appeared a component-based architecture for WSNs known as *MAC Layer Architecture* [14]. It consists of optimized and reusable components that implement a common set of features shared by existing MAC protocols. In order to design the components, MLA extract the most significant functionalities from Channel Polling MAC Protocols, Contention MAC Protocols and TDMA, such as *Radio Core* or *Channel Poller*. This architecture was designed with the aim that developers should be able to treat the MAC protocol as a single coherent entity, and hence be able to insert, replace or remove a MAC protocol with as little effort as possible.

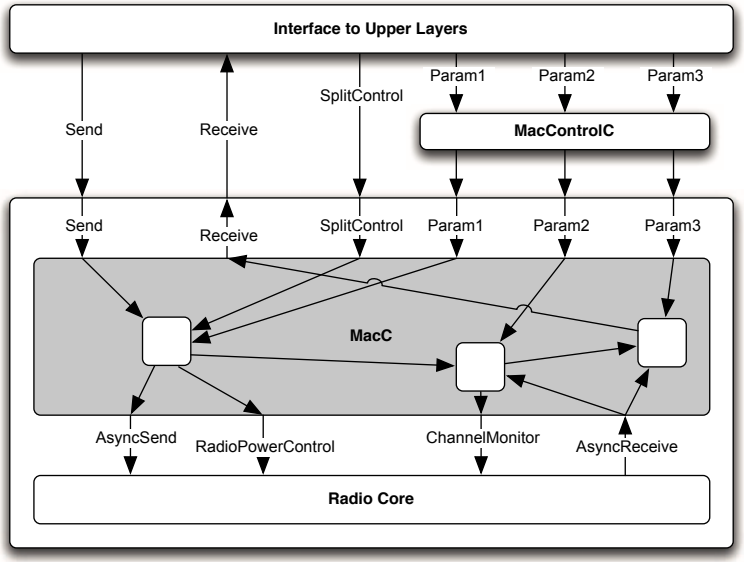


FIGURE 2.2: MLA architecture. [14]

As seen in FIGURE 2.2, each MAC protocol defines a *MacC* configuration that composes any reusable MLA components and any protocol specific components together. *MacC* should use a fixed set of low-level interfaces and produces: I/O interfaces like *Send* and *Receive*; and power control interfaces such as *SplitControl*. Following this scheme, it makes its operation as transparent to the user as possible. Thereby,

designing a new MAC protocol only implies creating a new *MacC* configuration accomplishing the new functionalities, instead of modifying more components.

Obaid Salikeen in [15] developed a component-based framework known as *MAC Protocol Designer (MAC-PD)* which encapsulated common tasks of MAC protocols in reusable components. A wide range of WSN MAC protocols are realized using these components. The main difference of this framework with MLA is the target user. MAC-PD is thought for MAC designers while MLA is more suitable for MAC developers. A MAC developer needs to have a good knowledge of the low-level platform language syntax whereas a MAC designer needs to only focus on designing the protocol without worrying about the implementation. According to that, MAC-PD framework provides a drag-and-drop based user-friendly Graphical User Interface (GUI) used to design and implement MAC protocols with mouse clicks and minimum amount of entered user data.

## 2.2 HARDWARE INDEPENDENCE

Since the beginning of the WSNs there have been lots of researches on defining a framework which is completely independent of the hardware platform in order to provide flexibility to the system. Thereby, the framework could be ported and executed in all the platforms available in the market without making any change.

David Culler, Joseph Polastre *et al*. [16, 17] have defined an unified link layer abstraction that can be instantiated over different platforms, in a similar fashion as the Internet Protocol stack. The architecture proposed is called *Sensor-net Protocol (SP)* and resides between network and link layer. It provides flexible link-layer interfaces for different WSNs applications residing on network layer, to optimize their performance and perform power aware operations independent of link layer.

*Unified Link-Layer API (ULLA)* [18] offers a common interface to retrieve link layer information as well as sensor measurement data independently of the deployed radio technology. It has considerably simplified the development process of link-aware protocols and applications. Using ULLA, applications can use a common interface to fetch link layer information regardless of the underlying hardware with negligible performance overheads. The main component in the ULLA architecture is ULLA Core in the middle of the FIGURE 2.3. It is an intermediate entity which connects Link Provider (LP) and Link User (LU). Link Provider is an interface which abstracts the sensor radio interface through its specific platform Link Layer Adapter (LLA). Basically, Link User (LU) is the application which is taking advantage of ULLA to read link layer information or sensor measures.

Similar to this work, Department of Computer Science and Engineering of Washington University developed *Unified Power Management Architecture (UPMA)* [19] which is another approach to provide a unifying abstraction for WSNs in order to provide portable solutions. *MAC Layer Architecture (MLA)* which has been mentioned in previous point, is an extended idea of UPMA specifically designed to facilitate power-efficient MAC protocol realizations. Two types of components are defined in this framework: High-level and Low-level. High-level or hardware-independent components allow different MAC protocol features to be composed together in a platform independent manner providing flexibility. The implementation of Low-level

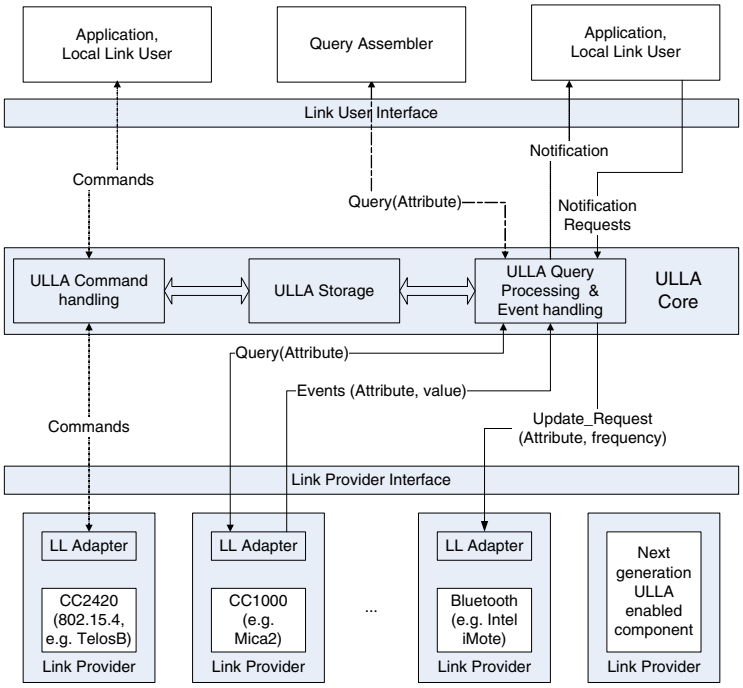


FIGURE 2.3: ULLA architecture. [18]

or hardware-dependent components is inherently platform specific, however, they export interfaces which support the development of fully platform independent high-level components. Thereby, porting the set of protocols developed in MLA to a new platform, it is only needed to provide new implementations of these low-level components alone.

## 2.3 SOLUTIONS FOR PROTOCOL PROTOTYPING AND DESIGNING

Many solutions have been proposed along the history of telecommunications regarding to the design of system and protocols. In this section, we are going to mention the most interesting and useful for the development of this thesis.

Specification and Description Language (SDL) [20] is a tool widely extended in the field of communications engineering in order to realize system design, prototyping, testing and verification. It was proposed by the International Telecommunication Union (ITU) inside its standard called Z.100. It can be used to specify and visualize a formal model in the form of state machines that can be executed for testing and verification purposes. However, it is very complex and difficult to use by a non-technical users.

Obaid Salikeen in [15] developed a component-based framework known as *MAC Protocol Designer (MAC-PD)* which enables to design MAC protocols through a GUI. However, the language generated by this toolchain was in eXtensible Markup Language (XML) format which was not very easy to remember the specific tags and not very understandable by a normal user.

Gwangwei Yang [21] designed and implemented a toolchain which was able to run any of the available MAC protocols for WARP boards using a language descriptor to describe the MAC protocol design which is processed by a meta-compiler. This language descriptor has a very easy syntax which was similar to some well-known programming languages, like C or C++.

According to previous contributions on this topic, our toolchain will be based in similar ideas as proposed by Yang, in order to provide an easy language that the user was able to use during the implementation of MAC protocols.

## 2.4 MAC PROTOCOLS FOR WSNS

As the wireless medium is highly affected by interferences it requires highly optimized medium access control protocols. As it is explained in [22], for WSNs is still more important to use this type of protocols due to sensors should work with batteries for a long time without human intervention. For that reason, the main design criteria is to extend the lifetime trying to keep the radio off when it is unnecessary to listen the medium. To achieve the desired optimization several solutions have been designed. The preamble-sampling, the common active periods and the hybrid protocols are three major categories which are widely used by WSNs.

### *2.4.1 Preamble-sampling Protocols*

In these type of protocols each node maintains its own active schedule independently. Normally, a node spends most of the time in sleep mode and only wakes up periodically to check if there is a transmission on the channel. Each data frame is preceded by a preamble long enough to make sure that the transmission is detected by the the receiver. The duration of the preamble has to be at least as long as the duration between two consecutive instants of node wakeups. To avoid collisions, these protocols use contention based approaches to listen the channel before transmitting the preamble.

#### *2.4.1.1 B-MAC*

B-MAC [23] is a CSMA/CA based preamble-sampling protocol. So, it is asynchronous and employs LPL and CCA operations. As it is explained before, the idea of this protocol is to keep most of the time the sensor in sleep mode in order to save energy and only wake it up for a short time to check if there is activity on the channel. In order to achieve that, as it can be seen in FIGURE 2.4, each sensor follows its own schedule and performs CCA in the active period; if the channel is clear it goes again to sleep and wakes up in the next cycle, otherwise it keeps its radio on to listen to the data packet. Moreover, when a node has a packet to transmit, first of all, it performs CCA to check that nobody is transmitting, then, it sends a preamble frame to assure that the receiver node turns its radio on and, finally, sends the data. This protocol saves a lot of energy in idle listening, but it also wastes a lot of energy in order to send the preamble frames. As the preamble frames are broadcast, the sensors that receives the preamble but are not the destination of the data packet are also wasting energy keeping its radio on during the preamble time.

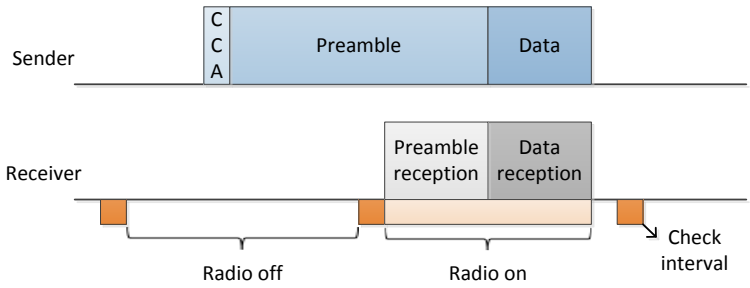


FIGURE 2.4: An illustration of a sender/receiver pair running B-MAC.

### 2.4.1.2 *MFP-MAC*

MFP-MAC [24] works in the same way that B-MAC but it splits the monolithic preamble into multiple micro frames and adds more information about the the contents of subsequent data frame inside them. This allows a node to make a decision of receiving or not in a timely manner without having to keep listening to the preamble until the data frame. As FIGURE 2.5 shows, this approach allows to save energy in two ways: avoiding listening to irrelevant data and reducing the duration of listening to the preamble. The node id of the receiver is included in these micro frames, so the nodes which are not the targets of the data can go back to sleep without listening the rest of the preamble and the data packet. Moreover, information about the total preamble length and the actual preamble time are included, allowing the receiver node to turn its radio off during the transmission of the rest of the preamble and turn it on again just to receive the data frame. MFP-MAC solves the problem of waste energy when receiving data, but sensors continue wasting energy by sending preamble frames during two consecutive instants of node wakeups.

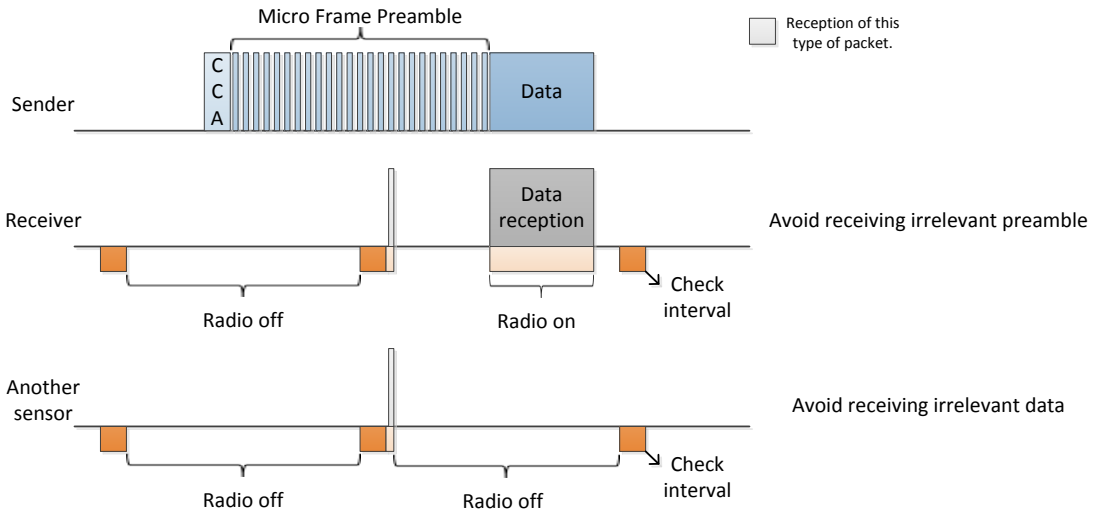


FIGURE 2.5: An illustration of a sender/receiver pair running MFP-MAC.

#### 2.4.1.3 X-MAC

X-MAC [25] works like B-MAC too, but the main design goals of X-MAC are energy-efficiency, low-overhead, low latency and high throughput for data. In order to achieve all of these goals, X-MAC follows the idea of the micro-frames from MFP-MAC and introduces a new concept: the receiver sends an ACK packet to stop sending the preamble. As FIGURE 2.6 shows, when the target node receives a micro frame preamble, as the receiver node id is included inside them, it knows that it is the target and sends an ACK packet to indicate that it has its radio on. The sender receives this packet, immediately, it stops sending preamble frames and sends the data frame. This approach allows avoid listening to irrelevant data like MFP-MAC and save energy when transmitting and receiving data.

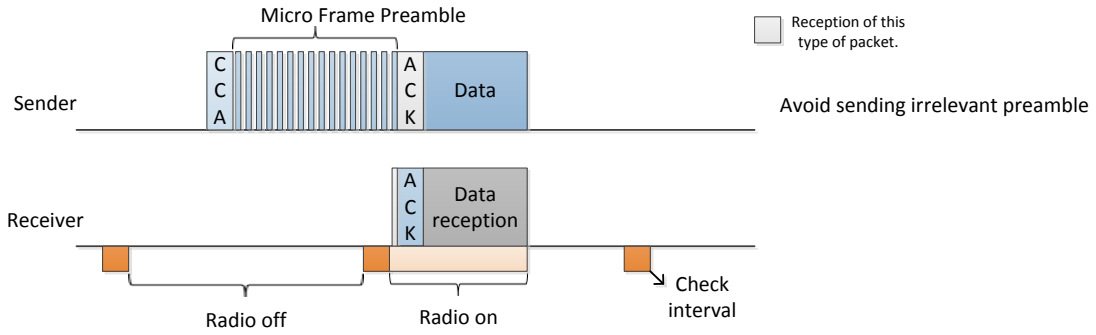


FIGURE 2.6: An illustration of a sender/receiver pair running X-MAC.

### 2.4.2 Common Active Periods Protocols

The sensors which use these protocols have common active/sleep periods. The active periods are used for communication using RTS/CTS/DATA/ACK handshake and the sleep periods are used to save energy by keeping the radio in off state. As the periods are common for all nodes, this approach requires a certain level of synchronization between all the nodes. For that reason, during the active period, sensors also send SYNC packets with this type of information. These protocols use contention based approaches to listen the channel before transmitting SYNC and RTS packets in order to avoid collisions too.

#### 2.4.2.1 S-MAC

The basic idea of S-MAC [26] is repeatedly put nodes in active and sleep periods, nodes turn their radio off in sleep periods to save energy and turn them on in active periods to exchange packets. As the active/sleep periods are shared by the nodes, it requires synchronization establishment and maintenance between them. S-MAC splits the active period in two parts: one to exchange SYNC packets with synchronization information and another to exchange data packets. The first time, a node listens to the channel for a duration of at least one active plus one sleep period to receive a SYNC packet, if it does not receive a packet then adopts its own schedule and disseminates

it. As it is shown in FIGURE 2.7, during the active time, a node can communicate with its neighbours following the RTS/CTS/DATA/ACK handshake. This scheme helps to reduce collisions, but in order to do that CCA is also used before sending SYNC and RTS packets. The primary goal in S-MAC design is reducing energy, but it also achieves good scalability and collision avoidance by using a combined scheduling and contention scheme. However, the use of SYNC frames, RTS/CTS control frames and ACKs increases the transmission overhead.

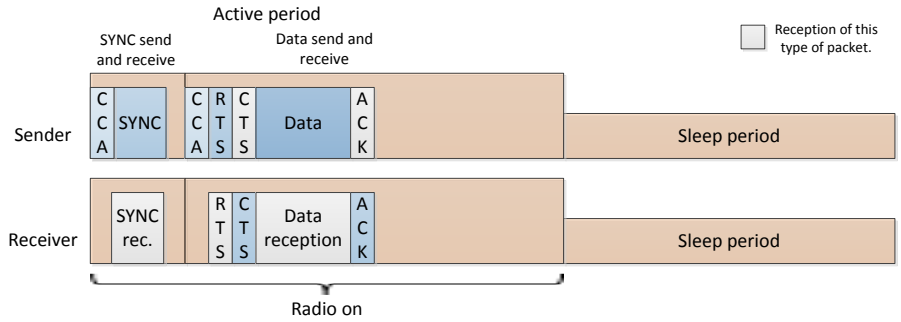


FIGURE 2.7: An illustration of a sender/receiver pair running S-MAC.

### 2.4.2.2 *T-MAC*

T-MAC [27] works identically to S-MAC, but it introduces the idea of adaptive duty cycle.The idea of T-MAC is to adapt dynamically the active time according to the current traffic. To accomplish that it uses a time-out mechanism like in FIGURE 2.8: after the node wakes up a timer starts (TA), if any RTS packet is detected, the node goes to sleep again. Otherwise, the node waits for the data packet and starts again the timer and so on. The duration of this timer should be long enough to span the contention duration and the RTS/CTS exchange. This scheme reduces the energy wasted by listening to the channel when there is no transmission. However, it also might lead to early sleeping problem: the nodes goes prematurely to sleep while another node still has some data for it.

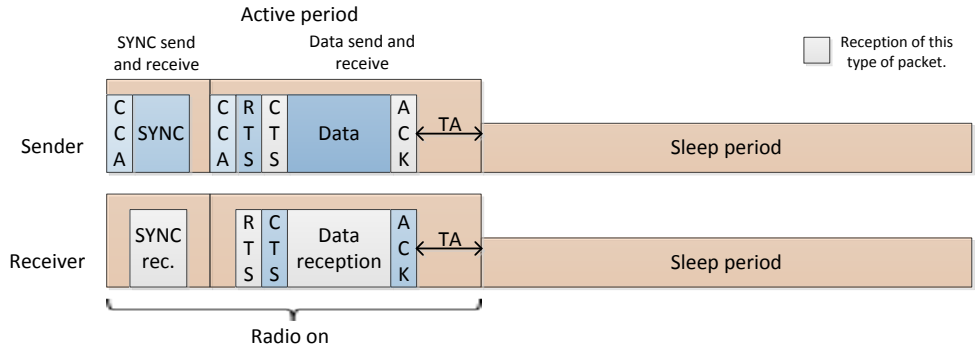


FIGURE 2.8: An illustration of a sender/receiver pair running T-MAC.

### *2.4.3 Hybrid Protocols*

Hybrid protocols combine two different MAC protocols to take advantage of their characteristics. They achieve high performance under variable traffic patterns by switching their behavior depending on the network conditions. For example, when there is a small number of nodes a protocol should use contention-based approaches, like preamble-sampling based protocols; however, when there is a large number of nodes the best choice is to use scheduled protocols, like common active periods based protocols.

#### *2.4.3.1 Z-MAC*

Z-MAC [28] increases the throughput in networks with variable traffic patterns following the idea of use CSMA inside large TDMS slots. Basically, this protocol runs CSMA in low traffic conditions achieving high channel utilization and low-latency. However, under high traffic conditions it switches to TDMA achieving high channel utilization and reducing collision among two-hop neighbors. In conclusion, by mixing CSMA and TDMA patterns, Z-MAC becomes more robust to clock synchronization or slot assignment failures, topology changes and time-varying channel conditions changes than a stand-alone protocol.

#### *2.4.3.2 Funneling-MAC*

Funneling-MAC [29] makes a spatial separation to mitigate funneling effect. This effect is caused by the many-to-one and hop-by-hop traffic pattern used in sensors networks and results in a significant increase in transit traffic intensity, collision, congestion, packet loss and energy drain in the regions closest to the sink. Accordingly, as it is shown in FIGURE 2.9, this MAC protocol uses a hybrid protocol TDMA/CSMA in these high traffic regions in order to increase channel utilization and reduce collision. However, it uses CSMA in regions which traffic is less intense to have better energy/throughput performance.

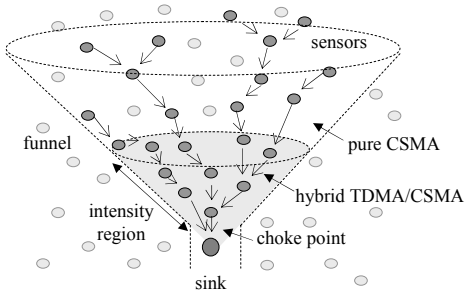


FIGURE 2.9: Funneling effect in sensors networks and Funneling-MAC solution. [29]

# 3

# SYSTEM DESIGN

In this chapter, the design of our toolchain for realizing prototype and design MAC protocols is described.

The design of our toolchain can be divided in three parts: MAC Components, the Languages used for MAC Protocol Design and the Meta-language Parser. Each MAC Component represents a functionality of a MAC protocol, such as MACRadioPowerControl is the responsible of turn on/off the radio, as explained in section 3.1 the components are used to form MAC protocols. Section 3.2 describes the Meta-language and the MetaNode-language which specify each MAC functionality; the Meta-language has been designed for the user and the MAC protocol implementation while the MetaNode-language is the language used by the sensor node. Finally, the Meta-language Parser is the element that allows the translation from Meta-language instructions into MetaNodes which the sensor node is able to understand and execute.

## 3.1 MAC COMPONENTS

One of the first steps in the design of our toolchain is to identify which are the most common functionalities in MAC protocols. The main goal is to create components that provides these functionalities and can be reused in different parts of a MAC protocol design.

As it has been mentioned previously in section 2.4, there are 3 different types of MAC protocols which are widely used in WSNs: preamble-sampling, common active periods and hybrid protocols. In order to take the most common basic functionalities, some MAC protocols of each type has been taken as example. For preamble-sampling, B-MAC, MFP-MAC and X-MAC have been selected; and for common active periods protocols, S-MAC and T-MAC have been selected. As hybrid protocols are combination of two different MAC protocols, we did not select any example as they usually combine previously mentioned types. In addition, we have included the protocol IEEE 802.11 [2] which is based in CSMA/CA, since it is commonly used in the area of wireless networks.

After studying each of them, it can be possible to extract some of the most basic functionalities which are illustrated in TABLE 3.1, such as *Send* and *Receive* which allow the nodes to communicate each other, or *Radio Control* which controls the radio power state (ON/OFF).

These functionalities will be the base to design more complex components such as *Carrier Sensing* which needs *Noise Floor Estimate*, *Radio Control* and *Timer* functionalities to determine if the medium is busy or nobody is transmitting. These little more complex functionalities are shown in TABLE 3.2.

| ***Basic Functionality*** | ***Protocols*** | | | | | |
|---|---|---|---|---|---|---|
| | *B-MAC* | *MFP-MAC* | *X-MAC* | *S-MAC* | *T-MAC* | *802.11* |
| Noise Floor Estimate | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Radio Power Control | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Random Generate | | | | | | ✓ |
| Receive Frame | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Send Frame | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Timer | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

TABLE 3.1: Summary of the basic functionalities of the most common MAC protocols.

| ***Basic Functionality*** | ***Protocols*** | | | | | |
|---|---|---|---|---|---|---|
| | *B-MAC* | *MFP-MAC* | *X-MAC* | *S-MAC* | *T-MAC* | *802.11* |
| Binary Exponential Backoff | | | | | | ✓ |
| Carrier Sensing | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Low Power Listening | ✓ | ✓ | ✓ | ✓ | ✓ | |
| Send Preamble | ✓ | ✓ | ✓ | | | |

TABLE 3.2: Summary of the complex functionalities of the most common MAC protocols.

## 3.2 LANGUAGES USED FOR MAC PROTOCOL DESIGN

Since our toolchain aims to provide an easy and efficient way to design MAC protocols and execute them in different platforms, we have designed a Meta-language in order to facilitate this work to the user.

Moreover, in order to execute the user design, we design a MetaNode-language. It describes the MAC protocol using a list of nodes which can be loaded in any platform.

Finally, we need a parser which checks format errors of the MAC protocols designs described using Meta-language and translates them to a list of MetaNodes which will be explained in section 3.3.

### *3.2.1 Meta-language Definition*

The Meta-language defined in this section is the one used by the user to describe the MAC implementations. For that reason, this language is designed to be as simple as possible. It follows the basic syntax of C or NesC.

Mainly, there are six basic operations which can be used in the Meta-language: *variable definition*, *expressions*, *event implementation*, *function calls*, *if-else structures* and *label-goto structures*. Each one of them will be explained in the following sections.

#### 3.2.1.1 *Variable definition*

The concept of variable exists in all programming languages. A *variable* is a storage location and an associated symbolic name which contains a value. The variable name is the usual way to reference the stored value; this separation of name and content allows the name to be used independently of the exact information it represents.

In our Meta-language, there is an important restriction: it is only possible to use numerical variables inside the code. This is due to all the functions that can be called need only numbers as parameters. All the user variables must be defined before being used in a function call, in an expression or in an if-else structure. After defining a variable, its default value is always 0. The basic instruction to define a new variable is the next one:

```
int variable_name;
```

In our toolchain, it only exists four types of variables:

- *System variables*: the ones already defined by the toolchain. These variables will be used to provide some information to the user, for example the *tos_node_id* of the sensor node. They will be defined in section 4.2. As they are predifined by the toolchain, the user will be able to use them in all the event implementations. However, the user cannot define new variable with the same name as the existing system variables anywhere inside the code.
- *Event attributes*: the ones that an event returns when it is triggered. These variables are used by the events to give some information to the user, for example if the medium is busy after executing carrier sensing. As the system variables, they will be defined in section 4.2. However, in this case, the user can only use them inside the events which have them as attributes. In the same way that in system variables, the user cannot define new variables with the same name as the event attributes anywhere inside the code.
- *Local variables*: the ones defined inside an event implementation. The context of these variables is the event inside they have been defined and they can only be used inside it. Two variables can have the same name only if they belong to two different events.
- *Global variables*: the ones defined by the user at the top of the code outside the events. These variables do not have context, but they belong to all of them. For that reason, they can be used inside all the event implementations and their value will be the same. However, the name cannot appear in another variable definition again inside the code.

#### 3.2.1.2 *Expressions*

Expressions are those operations which allow changing the value of previously defined variables. It has been designed three ways to change the value of a variable:

- *Assign a value*. It is possible to assign the value of a number, of another variable or the returned value of a function. The instructions to assign a value must follow the next structures:

```
variable_name = 3;
variable_name = other_variable_name;
variable_name = function_call();
```

- *Add a value*. In this case, the value specified is added to the actual value of the variable. As in the previous case, the value that will be added can be the value of a number, of another variable or the returned value of a function. The structures of these operations are the next ones:

```
variable_name = variable_name + 3;
variable_name = variable_name + other_variable_name;
variable_name = variable_name + function_call();
```

- *Subtract a value*. It is the same case as *add a value*, but subtracting it instead of adding. The structures of these operations are the next ones:

```
variable_name = variable_name - 3;
variable_name = variable_name - other_variable_name;
variable_name = variable_name - function_call();
```

#### 3.2.1.3 Event implementation

*NesC* [30] is an event-driven programming language, for that reason, it is necessary that our Meta-language follows the same scheme. An event is an action that is initiated somewhere inside the program and that is handled by a piece of code in another place inside the program. For example, once the program starts, the function *StartRadio* is called. After executing this action, an event called *StartRadioDone* is fired and the code to be executed after starting the radio should be inside this event.

The structure to implement an event is shown below. The events are predefined by the toolchain, in section 4.2 is provided the list of all the possible events that the user is able to implement. Between the parentheses the name of some attributes can appear to inform the user if there is an error, the data received in a packet, etc. The event implementation is described between the brackets, so the last bracket indicates that the event implementation is finished. Inside an event, all the operations are permitted except for another event implementation. It means that the user can declare variables, change variables' value, call functions or use if-else and label-goto structures.

```
EVENT_NAME(attributes){
        Operations to do when the event is fired;
}
```

#### 3.2.1.4 Function calls

The basic components explained in section 4.1 provide functionalities to the user in order to implement MAC protocols. The way to use these functionalities inside the code is through function calls that the user can call inside an event implementation. As the events, these functions are predefined by the toolchain and they are defined in section 4.2. The instruction to call a function has to follow the next structure:

```
function_call();
```

Between the parentheses should be included, if it is necessary, the parameters that the function may need. As the next examples shows, the parameters should be numbers or variables.

```
function_call(2, 3);
function_call(variable_name_one, variable_name_two);
```

#### 3.2.1.5 *If-else structures*

In the MAC protocol implementations which can be designed using our toolchain, *if-else structures* can be used. These structures work in the same way that in other languages like C, Java, etc. The restriction that our structure has is that is not possible to create *else if statements*, the Meta-language only allows *if*, *else* and *endif*. Just like in other *if-else structures*, a condition must be included in *if statement*. If this condition is true, the code inside *if* block will be executed. Otherwise, the code in *else* block will be executed, in case there is one. As the *else* block can be included or not, to indicate that the *if-else structure* is finished the *endif statement* must be always included. An example of how to use this structure is showed below.

```
if(variable_name == 2){
        Operations to do when the condition is true;
}else{
        Operations to do otherwise;
}endif;
```

The condition of *if statement* can be created using numbers or variables. However, the return value of a function cannot be included. If someone wants to use it, first of all, it is necessary to use an expression in order to store the return value into a variable. After that, this variable can be included inside the if condition. There are five possible conditions: equal "==", greater than ">", greater than or equal to ">=", less than "<" and less than or equal to "<=".

#### 3.2.1.6 *Label-goto structures*

In the Meta-language, *label-goto structures* that can be used work in the same way as in other languages like C. However, a restriction appears in our Meta-language description: the label and goto statements must be included inside the same event implementation. If they are in different event implementations, it will be impossible to execute the instructions which the label refers.

In the label statement a number must be indicated which will be the identifier of the label. On the other hand, in the goto statement, a number also has to be included which is the identifier of the referenced label to jump. The basic structure is showed below.

```
label 2;
...
goto 2;
```

In FIGURE 3.1 an example of how to use the Meta-language is given. In the first line, a global variable called *myVar* is defined. In the rest of the code, *BOOT_EVENT* and *RADIOSTART_EVENT* are implemented. When the protocol starts to run, the *BOOT_EVENT* is triggered and its operations are executed: assign value 1 to *myVar* and starts the radio. Once the radio is started, the *RADIOSTART_EVENT* is triggered. Then, if *myVar* has value 1, a packet is sent. Otherwise, a local variable called *anotherVar* is defined, its value is assigned to 1 and added to value of *myVar*. Finally, the *goto* sentence makes the code jump to the label definition and the code bellow the *label* sentence is executed again.

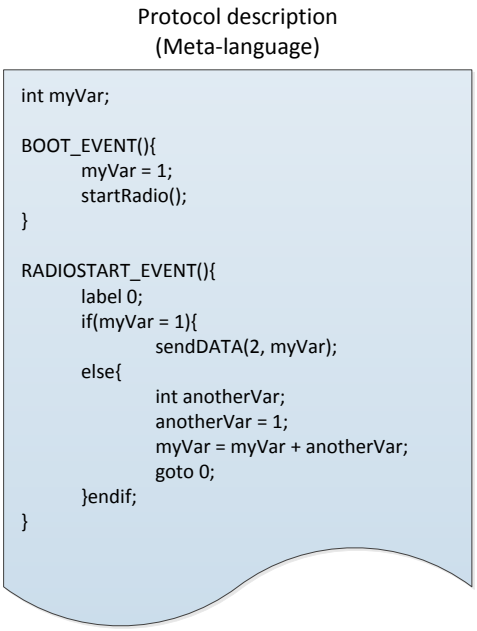


FIGURE 3.1: Example of MAC Protocol design using Meta-language.

### 3.2.2 *MetaNode-language Definition*

The *MetaNode-language* is the one used by the sensor node to execute the functionalities described in the MAC protocol definition (*Meta-language*). As this language is a translation from the Meta-language in order that a sensor node can understand the instructions, the basic operations that compose it are the same as the explained in section 3.2.1. However, our toolchain will use a Sensor Interface to execute the instructions allocated in an execution list where each instruction occupies one position, as explained in [1]. In order to be able to allocate each instruction in this list, we define a MetaNode to represent one of these instructions that appear in the MAC protocol description file. Therefore, the size of the execution list will be the same as the number of lines of the file, due to each MetaNode makes a reference to a line of the text file.

Thus, MetaNode-language will be the one which specifies the syntax of each MetaNode. First of all, we need to define the MetaNode structure which will be composed

by:

- *MetaLabel*: a numeric value which will specify the MetaNode type. For example, if it is a variable definition (tag DEF), an if statement (tag IF) or a function call (tag FUNC).
- *IdLabel*: a numeric value which will specify some characteristic of the MetaNode. For example, if it is a function call, the idLabel will specify at which function it refers. If it is a *label* or *goto statement*, it will specify the identifier of the corresponding label. In case of a *variable expression*, IdLabel will specify if the instruction is adding, subtracting or assign the value to this variable.
- *Parameters*: a set of attributes which will specify numeric values, variables or function calls that a certain MetaNode may require. The structure which allocates this parameters is called MetaParams and consists of two fields:
  - *Type*: it indicates if it is a numeric value (tag VALUE), a variable (tag VAR) or a function call (tag FUNC).
  - *Value*: it can indicate three different concepts depending on the type. In case of a numeric value indicates which value it is. If it is a variable, it indicates the name of this variable. Otherwise, the value indicates to which function it refers.

Once MetaNode structure is well specified, the translation from Meta-language to MetaNode-language for each one of the six available operations can be easily explained.

#### *3.2.2.1 Variable definition*

This type of MetaNodes is used to define new variables. As it is explained in section 3.2.1.1, a user can define two types of variables: local and global variables, depending on where they are defined, inside or outside an event implementation. To express it using MetaNodes, it is necessary to indicate that the value for MetaLabel is DEF and, in this case, none IdLabel is needed. Nevertheless, one Parameter of variable type (tag VAR) will be necessary to indicate the name of the defined variable.

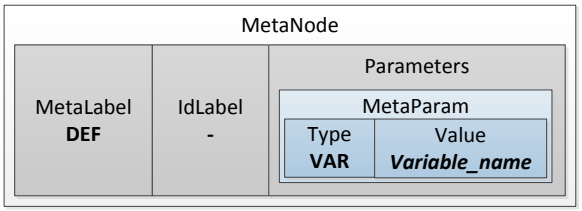


FIGURE 3.2: Structure of MetaNode in case of a Variable definition.

#### 3.2.2.2 *Expressions*

MetaNodes can also describe an expression to change the value of previously defined variables. In this case, they will use the tag EXP as MetaLabel value. As it is explained in section 3.2.1.2, there are three ways to change the value, thus to identify which of them is, Idlabel should be used: *assign a value* (tag EQUAL), *add a value* (tag ADD) or *subtract a value* (tag SUBTRACT).

The variable whose value will be changed must be specified as the first parameter of the list. In addition, it must be of variable type (tag VAR) and its value field must specify the name of the referenced variable. The second parameter can be defined in three different ways, depending on its type:

- *A number*: the type field is a numeric value (tag VALUE) and the value indicates what number it is.

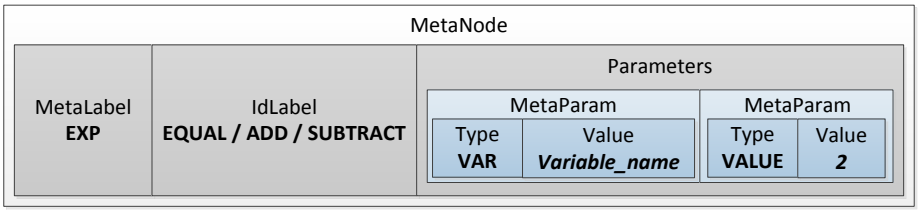


FIGURE 3.3: Structure of MetaNode in case of a Expression and its second parameter is a number.

- *Another variable*: the type field is a variable (tag VAR) and the value field indicates its name.

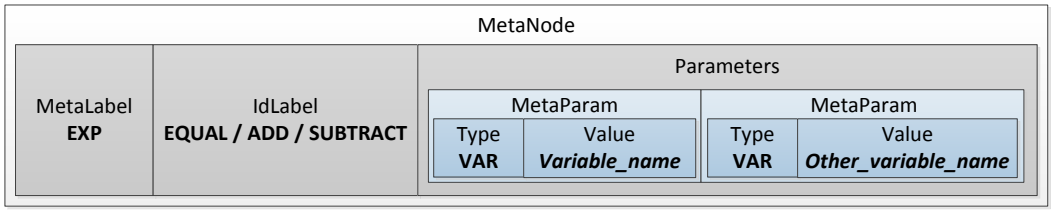


FIGURE 3.4: Structure of MetaNode in case of a Expression and its second parameter is another variable.

- *Returned value by a function call*: the type field indicates a function call (tag FUNC), the value field indicates the name of the called function and the rest of Paremeters are the parameters that the specific function might need. The way to specify the function parameters is the same as when making a function call which is explained in section 3.2.2.4.

#### 3.2.2.3 *Event implementation*

As it is explained in section 3.2.1.3, the operations which will be executed when an event occurs must be included inside the event implementation. In the MetaNode-

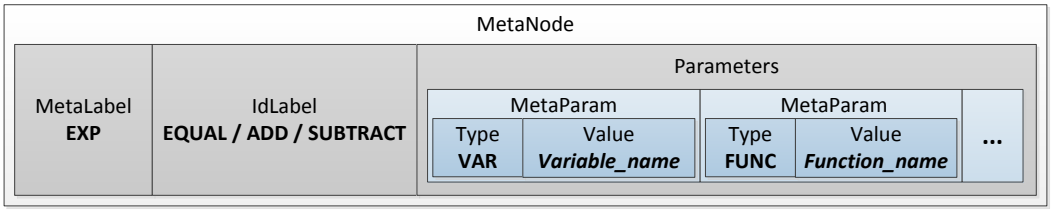


FIGURE 3.5: Structure of MetaNode in case of a Expression and its second parameter is a returned value by function call.

language, two types of MetaNode exist to define the limits of the event implementation.

The first one indicates in the MetaLabel which event will be implemented by the following MetaNodes. The second one indicates that this implementation is finished by using the tag END_EVENT as MetaLabel. Both of these Metanodes does not use any IdLabel or Parameters. Thereby, every MetaNode which is included between these tags will be executed after the event is fired.

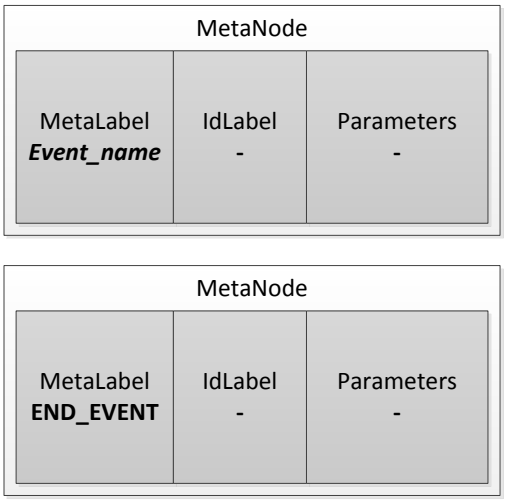


FIGURE 3.6: Structure of MetaNode in case of an Event implementation.

#### *3.2.2.4 Function calls*

In case of translating a function call, it will be necessary to use a MetaNode with the tag FUNC as MetaLabel. The name of the specific function must be included as IdLabel and, if the function needs some parameters, they must be included in the Parameters list. As it is explained in section 3.2.1.4, the parameters used in a function call can be numbers or variables. If the parameter is a number, the type field must be a numeric value (tag VALUE) and the value field must be the number. Otherwise, the type field must indicate a variable and its name must be indicated in the value field.

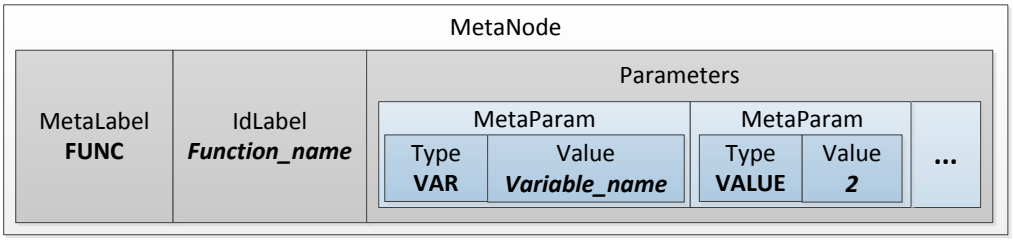


FIGURE 3.7: Structure of MetaNode in case of a Function call.

#### 3.2.2.5 *If-else structures*

In order to define *if-else structures* in MetaNode-language, three kinds of MetaNodes must be used. First, it must be defined the *if condition* which is the main one. Second, the *else condition* to execute in case of the result of first condition is false. Finally, an *EndIf statement* is used to indicate that all the conditional block is finished.

The *if condition* is specified by using the tag IF as MetaLabel and it must be accompanied by three Parameters. The first one with type VALUE will indicate which operation must be checked inside the condition. The five possibilities are differentiated with tags that suggest the operation type: EQUAL, GREATER, GREATER_OR_EQUAL, LESS or LESS_OR_EQUAL. Next Parameters must be the elements to be compared which can be numeric values or variables, as it is explained in section 3.2.1.5.

On the other hand, the *else condition* is specified by using the tag ELSE as MetaLabel while the *EndIf statement* uses the tag END_IF.

However, sometimes it is necessary to include if-else structures inside other, thus a way to identify the MetaNodes' hierarchical level. To achieve that, IdLabel is used to indicate the actual level. Thereby, the highest level will have a 0 as IdLabel, the following will be the number 1, and so on.

#### 3.2.2.6 *Label-goto structures*

Translating this kind of structures, implies using two MetaNodes, one used for the *label* statement and the other for the *goto* one.

In case of the MetaNode for the *label*, the tag LABEL must be used as MetaLabel and the numeric value which identifies this specific label must be indicated inside IdLabel.

On the other hand, the MetaNode used for the *goto* must use the tag GOTO in the as MetaLabel and indicate in the IdLabel the numeric value of the label which it refers too. Both MetaNodes do not need to specify any type of Parameter.

## 3.3 META-LANGUAGE PARSER

Once it has been specified how to define all the possible operations from the Meta-language in the MetaNode-language, FIGURE 3.10 shows an example of the translation from the code of FIGURE 3.1 into MetaNodes.

As it has been explained in section 3.2, the Meta-language used to describe MAC protocols using text files is not the same as MetaNode-language. The last one is the

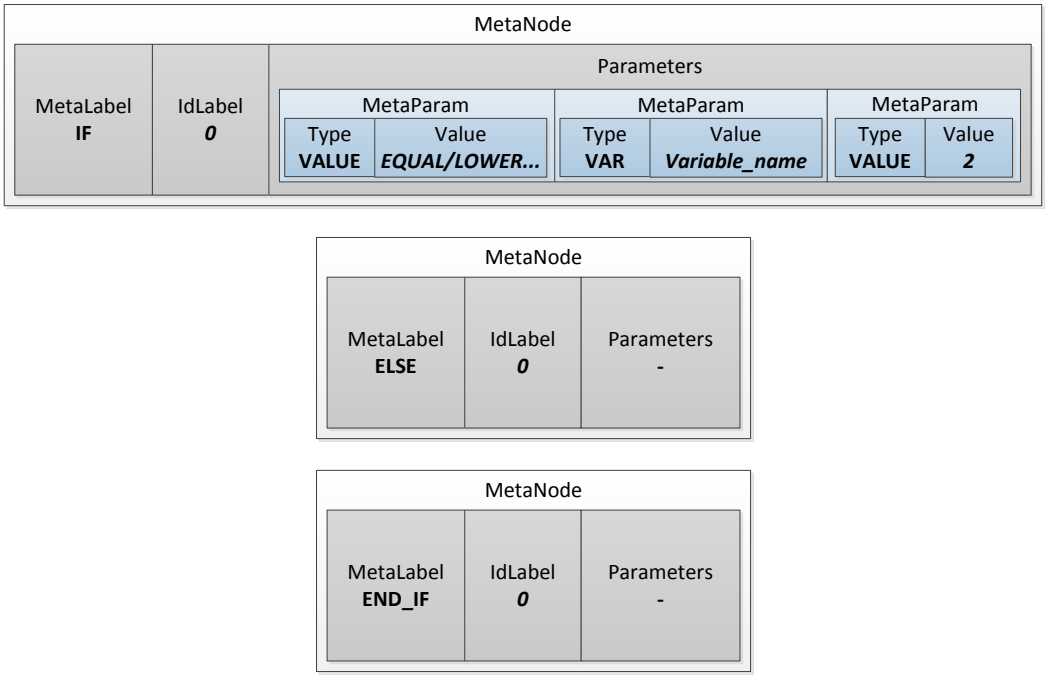


FIGURE 3.8: Structure of MetaNode in case of an If-else structure.

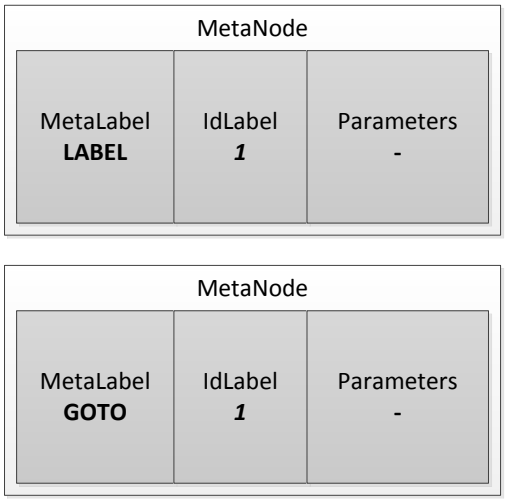


FIGURE 3.9: Structure of MetaNode in case of a Label-goto structure.

language used by the sensor node to execute MAC protocols by interpreting an execution list. Therefore, it is necessary a parser which is able to read the text file, translate the protocol description into an execution list formed by MetaNodes, allowing the sensor node to run it.

Furthermore, the parser will also check syntax and logical errors that may appear in Meta-language in order to send a feedback to the user. If there is no code error

inside the file, the execution list will be created.

As this can be a hard process, the Meta-language Parser will be implemented inside a User Interface designed in [1] which enables fast runtime reconfiguration of MAC protocols. This User Intarface is able to establish a communication with the sensor node. Thereby, the transfer of all execution list of MetaNodes generated by the parser can be transfer using it.

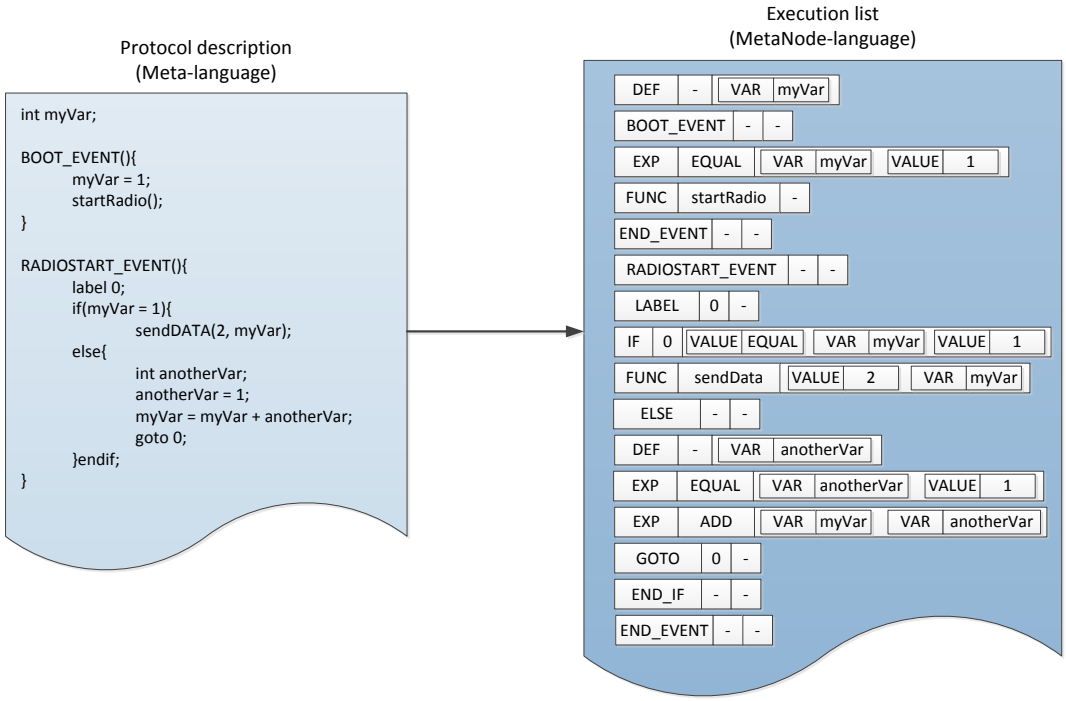


FIGURE 3.10: Example of translation of a MAC Protocol design from Meta-language to MetaNode-language.

# 4

# IMPLEMENTATION

This chapter describes the implementation details of the toolchain designed in order to allow a rapid MAC protocol prototyping and designing. As introduced in chapter 3, it can be divided in three different parts: MAC Components, the Languages used for MAC Protocol Design and the Meta-language Parser. In section 4.1, implemented MAC Components and their relationships are described. The commands and events which can be used in the Meta-language and the corresponding MetaNode-language are defined in section 4.2. Finally, the Meta-language Parser implementation is described in section 4.3.

## 4.1 MAC COMPONENTS

Our framework has been designed with the aim to simplify the creation of new MAC protocols. In order to achieve this goal, the framework follows a component structure, starting with very simple and independent blocks which can be used to build more complex blocks quickly. The possibility of using several existing blocks in the development of other new blocks promotes the code reusability inside the framework. Moreover, each block is independent and provides an interface to facilitate its use, this approach allows designers to focus on the MAC protocol without worrying about the complexities and dependencies between blocks. In this section, the MAC components implemented following the basic and complex functionalities of section 3.1 will be explained. These components have been implemented using the idea of component-based design and hardware independence. Therefore, they can be used in the design of any MAC protocol and deployed in any type of hardware.

In TABLE 4.1, a list of Basic and Complex Components and their composition are defined. Basic Components are the simplest ones built using only TinyOS components. Complex Components are built combining existing components of the framework and TinyOS components.

As shown in TABLE 4.1, MACNoiseFloorEstimator and MACRadioPowerControl are basic blocks which provides basic MAC operations like sensing the medium or switching the state of the radio. However, MACCarrierSensing is a complex block which provides operations of detecting activity in the channel and is built using the Basic Components MACTimer and MACNoiseFloorEstimator. Complex Components can be built using a combination of basic and other Complex Components. For instance, MACLowPowerListening is built using MACTimer, MACRadioPowerControl and MACCarrierSensing.

FIGURE 4.1 shows the final structure of our MAC protocols implementation and the relationships between them. The arrows indicate that the complex component has been built using the indicated basic or complex component.

| ***Basic Component*** | ***Composition*** |
|---|---|
| MACNoiseFloorEstimator | QueueC, HplCC1000C or CC2420ControlC |
| MACRadioPowerControl | ActiveMessageC, StateC |
| MACRandomNumGenerator | RandomC |
| MACReceive | AMReceiverC, ActiveMessageC |
| MACSend | AMSenderC, ActiveMessageC |
| MACTimer | TimerMilliC |
| ***Complex Component*** | ***Composition*** |
| MACCarrierSensing | MACTimer, MACNoiseFloorEstimator, StateC |
| MACLowPowerListening | MACTimer, MACRadioPowerControl, MAC-CarrierSensing, StateC |
| MACBinaryExponentialBackoff | MACRandomNumGenerator |
| MACReceivePacket | MACReceive |
| MACSendPacket | MACSend |
| MACSendPreamble | MACTimer, MACSend, StateC |

TABLE 4.1: MAC components library and their composition.

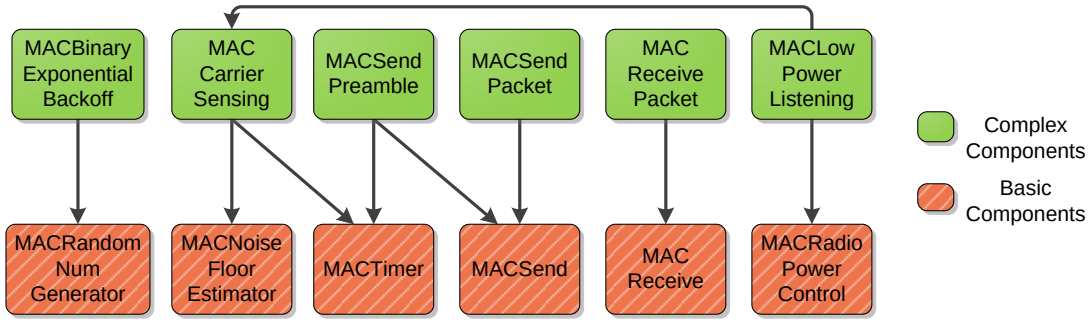


FIGURE 4.1: Diagram of reusable components.

TABLE 4.2 shows the components used in each MAC protocol implementation. Components MACRadioPowerControl, MACCArrierSensing, MACReceivePacket and MACSendPacket are fundamental for all protocols. Other components, like MACLow-PowerListening and MACSendPreamble are common for preamble-sampling protocols. Some of the Basic Components are used not in the main implementation of the protocol, but used inside complex blocks. For example, MAC protocols that use MACLowPowerListening, use also MACRadioPowerControl since MACRadioPowerControl is used to implement MACLowPowerListening block.

| ***Component*** | ***Protocols*** | | | | |
|---|---|---|---|---|---|
| | *B-MAC* | *MFP-MAC* | *X-MAC* | *S-MAC* | *T-MAC* |
| MACNoiseFloorEstimator | ✓ | ✓ | ✓ | ✓ | ✓ |
| MACRadioPowerControl | ✓ | ✓ | ✓ | ✓ | ✓ |
| MACRandomNumGenerator | | | | | |
| MACReceive | ✓ | ✓ | ✓ | ✓ | ✓ |
| MACSend | ✓ | ✓ | ✓ | ✓ | ✓ |
| MACTimer | ✓ | ✓ | ✓ | ✓ | ✓ |
| MACCarrierSensing | ✓ | ✓ | ✓ | ✓ | ✓ |
| MACLowPowerListening | ✓ | ✓ | ✓ | | |
| MACBinaryExponentialBackoff | | | | | |
| MACReceivePacket | ✓ | ✓ | ✓ | ✓ | ✓ |
| MACSendPacket | ✓ | ✓ | ✓ | ✓ | ✓ |
| MACSendPreamble | ✓ | ✓ | ✓ | | |

TABLE 4.2: Component reusability for different MAC protocol implementations.

In next subsections, the MAC Components which have been finally implemented are explained depending on their type: basic or complex.

### *4.1.1 Basic Components*

As it has been mentioned before, Basic Components are the simplest ones due to they are built using only the components provided by TinyOS. There are 6 Basic Components implemented which allow to create the Complex Components, which are explained in more detail in next subsections.

#### *4.1.1.1 MACTimer*

MACTimer component provides an interface similar to that of TinyOS Timer: it provides the basic timer functionalities, like *isRunning()* which return a boolean indicating if the timer is active or not, and adds new ones more specific, like *timeRemaining()* which returns the remaining time of an active timer.

A MACTimer instance can be instantiated once with *startOneShot(duration)* or periodically with *startPeriodic(duration, bound)*, where duration is the duration in milliseconds and bound is the number of periodic cycles. At the end of each cycle the event *fired()* is triggered.

#### *4.1.1.2 MACRadioPowerControl*

MACRadioPowerControl component is used to switch the radio on and off. It has been implemented using the TinyOS SplitControl interface. This component is very important due to the radio have to be switched on in order to send and receive messages and also have to be switched off to extend the battery lifetime of the sensor nodes.

The radio can be switched on using *startRadio()* and switched off using *stopRadio()*, when the state is correctly changed the event *radioStartDone()* or *radioStopDone()*, re-

```
interface MACTimer{
      command void startOneShot(uint16_t duration);
      command void startPeriodic(uint16_t duration, uint8_t bound);
      command void stop();
      command bool isRunning();
      command uint32_t timeRemaining();
      command uint32_t getDuration();
      command uint32_t getNow();

      event void fired();
}
```

FIGURE 4.2: Interface definition for *MACTimer* component.

spectively, is triggered. The *startRadio()* command turns on the voltage regulator and the crystal oscillator of the sensor's radio and makes sure that they are stable before the user uses it. To inform about the actual state of the radio is used the command *getRadioPowerState()* and in the case of the control state, for example if the radio is already turning on, is used *getSplitControlState()*.

```
interface MACRadioPowerControl{
      command void startRadio();
      command void stopRadio();
      command uint8_t getRadioPowerState();
      command uint8_t getSplitControlState();

      event void radioStartDone();
      event void radioStopDone();
}
```

FIGURE 4.3: Interface definition for *MACRadioPowerControl* component.

#### 4.1.1.3 *MACRandomNumGenerator*

MACRandomNumGenerator component provides one command that returns a random number within the specified range. It is based on TinyOS Random interface.

```
interface MACRandomNumGenerator{
      command uint16_t randomNum(uint16_t range);
}
```

FIGURE 4.4: Interface definition for *MACRandomNumGenerator* component.

#### 4.1.1.4 *MACNoiseFloorEstimator*

MACNoiseFloorEstimator component provides the basic commands to manage the actual noise floor and to read the RSSI value .

The actual noise floor can be consulted using *getNoiseFloor()* and it is a median between 10 samples saved in a queue. This samples can be updated by the user using *setNewNoiseFloorSample()*. TinyOS Queue interface is used in order to manage these noise floor samples.

The TinyOS ReadNow or Read interface is used to read the RSSI value depending on the platform. Because platform which use the radio chip CC1000, like Mica2 sensors, provides ReadNow interface through the HplCC1000C component. In this case, it is also mandatory to use TinyOS Resource interface in order to request the access to RSSI resource and be able to read the RSSI value once the it is granted. On the other hand, the ones with radio chip CC2420, like TelosB sensors, provides only Read interface through CC2420ControlC component.

These two ways of reading the RSSI value have been combined in one interface and it is selected the appropriated one automatically. Thereby, the user has not take into account the platform when designing the MAC protocol. It is used *readRssi()* to read the value and it is returned when the *readRssiDone(value)* event is triggered.

```
interface MACNoiseFloorEstimator{
      command void setNewNoiseFloorSample(int16_t noiseSample);
      command float getNoiseFloor();
      command void readRssi();

      event void readRssiDone(int16_t value);
}
```

FIGURE 4.5: Interface definition for *MACNoiseFloorEstimator* component.

#### 4.1.1.5 *MACSend*

MACSend component uses TinyOS AMSend, AMPacket and Packet interfaces and provides an unique interface to manage messages and send tasks. This interface is not available to the final user because it needs parameters of type message_t. Thereby, MACSend components is an intermediate interface between another high level components and the TinyOS AMSend interface. For example, MACSendPacket and MACSendPreamble components use it to manage the send tasks.

```
interface MACSend{
      command error_t send(am_addr_t addr, message_t* msg, uint8_t
         length);
      command error_t cancel(message_t* msg);
      command error_t resend();
      command void* getPayload(message_t* msg, uint8_t length);
      command uint32_t getPacketTransmissionTime(uint8_t
         payloadLength);

      event void sendDone(message_t* msg, error_t error);
}
```

FIGURE 4.6: Interface definition for *MACSend* component.

#### 4.1.1.6 *MACReceive*

MACReceive component uses TinyOS Receive, AMPacket and PAcket interfaces and provides an unique interface for all of them. The final user can not use this component due to it returns a parameter of type message_t. Thereby, similar to MACSend component, it is an intermediate interface between another high level components and the TinyOS AMReceive interface. For example, MACReceivePacket uses it to manage all the messages that the sensors receive.

```
interface MACReceive{
      command void setAddressRecognitionEnabled(bool
         enableAddressRecognition);

      event void receive(uint8_t type, uint16_t preambleDuration,
         uint16_t preambleTime, uint16_t destnode, uint16_t
         srcnode, uint16_t counter);
}
```

FIGURE 4.7: Interface definition for *MACReceive* component.

### 4.1.2 *Complex Components*

As it has been mentioned before, Complex Components are built combining existing components provided by the toolchain (Basic Components) and TinyOS components. There are 6 Complex Components implemented which are explained in more detail in next subsections.

#### 4.1.2.1 *MACCarrierSensing*

Carrier sensing functionality is very important for MAC protocols to detect any activity in the medium. For instance, a sender performs carrier sensing to conclude whether or not to initiate data transmission in order to reduce collisions, or a receiver performs it to know if there is data transmission in the channel and wake ups or goes again to sleep in order to save energy.

MACCarrierSensing component provides this utility using our Basic Components MACTimer and MACNoiseFloorEstimator. Carrier sensing is performed by calling *readRssi()* command of MACNoiseFloorEstimator and comparing these RSSI values with the actual Noise Floor get through *getNoiseFloor()* command. To prevent false positives, the RSSI values have to be greater than the noise floor value three times to consider that there a transmission in the channel.

To perform carrier sensing *startCS(time)* should be called. When three RSSI values are greater than the noise floor *firedCS(mediumBusy)* event is triggered and *mediumBusy* is TRUE. Otherwise, when the time expires this event is triggered but *mediumBusy* is FALSE. The command *stopCS()* is used to stop the current carrier sensing and *isIdle()* is used to know if carrier sensing is been performed in this moment. Finally, *updateThreshold()* allows using only one command to read one RSSI value and update the noise floor value using *readRSSI()* and *setNewNoiseFloorSample()* commands of MACNoiseFloorEstimator component.

```
interface MACCarrierSensing{
      command error_t startCS(uint16_t time);
      command bool isIdle();
      command void updateThreshold();
      command void stopCS();

      event void firedCS(bool mediumBusy);
}
```

FIGURE 4.8: Interface definition for *MACCarrierSensing* component.

#### 4.1.2.2 *MACLowPowerListening*

Preamble-sampling based protocols divides one period in active and sleep times. When an sleep time ends, the channel is listened performing carrier sense, if the channel is free the radio is switched off again. Otherwise, starts the active time and the radio keeps turned on until this period is finished.

MACLowPowerListening components provides the functionalities to duty cycle the radio and performs the carrier sense automatically. It uses the Basic Component MACRadioPowerControl to switch the radio on and off, MACCarrierSensing to performs carrier sense and MACTimer. The component duty cycle the radio depending on two parameters: sleep interval and duty cycle. The first one indicates the total time between two checks of transmissions on the channel. Duty cycle indicates in percentage the time that the radio will keep turned on. For instance, if sleep interval is 100 ms and duty cycle is 30 %, the radio will keep turned on during 30 ms and turned on again after 70 ms.

This component provides two commands to change the sleep interval and the duty cycle, *setSleepInterval(interval)* and *setDutyCycle(dutyCycle)* respectively. Moreover, to consult the actual values are used *getSleepInterval()* and *getDutyCycle()*.

In order to starts the cycle is used *startLpl()*, if during the carrier sense a transmission in the channel is detected *signalDetected()* event is triggered and the cycle is stopped. Also is provided the command *stopLpl()* to stop the cycle. It will be necessary for example to send a packet. In this case, when the cycle is stopped and the radio is turned on is triggered *lplStopped()* event.

```
interface MACLowPowerListening{
      command void setSleepInterval(uint16_t interval);
      command void setDutyCycle(uint16_t dutyCycle);
      command uint16_t getSleepInterval();
      command uint16_t getDutyCycle();
      command error_t startLpl();
      command void stopLpl();

      event void signalDetected();
      event void lplStopped();
}
```

FIGURE 4.9: Interface definition for *MACLowPowerListening* component.

#### 4.1.2.3 *MACSendPacket*

MACSendPacket component provides a simple interface for sending different types of packets, for example a data packet, an ACK, an RTS, etc. This component is built using the basic component MACSend, so, it only has to include the user parameters and the packet type in the corresponding message and call the command *send()* of MACSend interface. Moreover, when the event *sendDone()* of this interface is triggered, it has to trigger another event to the user.

MACSendPacket provides five commands to send a packet, each one of them refers to a different type of packet:data, ACK, RTS, CTS and synchronization. When the packer is correctly sent, an event is triggered. In the same way as for the commands, five event exist depending of the type of packet sent.

```
interface MACSendPacket{
      command error_t sendDATA(am_addr_t destnode, uint16_t
         counter);
      command error_t sendACK(am_addr_t destnode);
      command error_t sendRTS(am_addr_t destnode);
      command error_t sendCTS(am_addr_t destnode);
      command error_t sendSyn(uint16_t timeToSleep);

      event void sendDATADone(am_addr_t destnode, uint16_t counter,
         error_t error);
      event void sendACKDone(am_addr_t destnode, error_t error);
      event void sendRTSDone(am_addr_t destnode, error_t error);
      event void sendCTSDone(am_addr_t destnode, error_t error);
      event void sendSYNCDone(error_t error);
}
```

FIGURE 4.10: Interface definition for *MACSendPacket* component.

#### 4.1.2.4 *MACSendPreamble*

MAC protocols which use duty cycle strategies and no-synchronization need a way to indicate the receivers that there is a data transmission and they should keep their radio turned on. This strategy is to send a long preamble before sending the data packet.

MACSendPreamble provides an interface to send these type of long packets. It uses the Basic Component MACTimer and, as MACSendPacket component, MACSend to manage the send tasks. This component allows the user to send three type of preambles: monolithic preamble which is the one used in B-MAC protocol, micro frame preamble or MFP which is the one used in MFP-MAC protocol and a short preamble, called strobbed, which is the one used in X-MAC protocol. Each one of them needs different parameters due to they have different behaviours. Nevertheless, the *preambleLength* parameter is always needed in order to determine the maximum time that the preamble has to be sent.

Three commands are provided in order to select the type of preamble ans start to send it: *sendPreambleMonolithic(preambleLength)*, *sendPreambleMFP(destnode, preambleLength)* and *sendPreambleStrobbed(destnode, preambleLength)*. The command *stopPream-*

*ble()* is used to stop sending the preamble before the time expires. Finally, when the time expires, the event *preambleSendDone()* is triggered.

```
interface MACSendPreamble{
      command error_t sendPreambleMonolithic(uint16_t
         preambleLength);
      command error_t sendPreambleMFP(am_addr_t destnode, uint16_t
         preambleLength);
      command error_t sendPreambleStrobbed(am_addr_t destnode,
         uint16_t preambleLength);
      command void stopPreamble();

      event void preambleSendDone(am_addr_t destnode, error_t
         error);
}
```

FIGURE 4.11: Interface definition for *MACSendPreamble* component.

#### 4.1.2.5 *MACReceivePacket*

MACReceivePacket provides a simple interface to notify trough events that a message is received. This component is built using the basic component MACReceive, so when the event *receive()* of this basic component is triggered, the parameter of the packet which indicates the type is read and is triggered another event indicating the type of message and the value of the important parameters.

Following this idea, eight events are defined: receive a monolithic preamble, a MFP preamble, a strobbed preamble, a data packet, an ACK, a RTS, a CTS or a synchronization packet. The command *setAddressRecognitionEnabled(enableAddressRecognition)* is used to enable or disable address recognition. When the parameter is FALSE, it is disables, therefore the component will trigger receive events when a transmission is detected without filtration. Otherwise, the component will trigger receive events only for the messages with destination address equal to the node address or broadcast.

#### 4.1.2.6 *MACBinaryExponentialBackoff*

MACBinaryExponentialBackoff components has been included to use it in collision avoidance schemes. In case of collision, they delay the retransmission of packets depending on the number of collisions accumulated. Specifically, a random number of slot times between 0 and 2c - 1 is chosen, being c the number of collisions. So, for the first collision each sender will wait 0 or 1 slot times. After the second collision, the senders will wait anywhere from 0 to 3 slot times inclusive and so on.

So, this component select the range of slot times depending on the number of collisions like is explained before and then, it selects one of them randomly using the basic component MACRandomNumGenerator. In order to provide this functionality, the component has the command *getBinaryExponentialBO(numCollision)*.

```
interface MACReceivePacket{
      command void setAddressRecognitionEnabled(bool
         enableAddressRecognition);

      event void receivePreambleMonolithic();
      event void receivePreambleMFP(uint16_t preambleDuration,
                           uint16_t preambleTime, uint16_t
                              destnode);
      event void receivePreambleStrobbed(uint16_t destnode,
         uint16_t srcnode);
      event void receiveDATA(uint16_t destnode, uint16_t srcnode,
         uint16_t counter);
      event void receiveACK(uint16_t destnode, uint16_t srcnode);
      event void receiveRTS(uint16_t destnode, uint16_t srcnode);
      event void receiveCTS(uint16_t destnode, uint16_t srcnode);
      event void receiveSYN(uint16_t time);
}
```

FIGURE 4.12: Interface definition for *MACReceivePacket* component.

```
interface MACBinaryExponentialBackoff{
  command uint16_t getBinaryExponentialBO(uint16_t numCollision);
}
```

FIGURE 4.13: Interface definition for *MACBinaryExponentialBackoff* component.

## 4.2 LANGUAGES USED FOR MAC PROTOCOL DESIGN

Once MAC Components have been implemented, we need to define how their commands and events are called in the Meta-language and in the MetaNode-language. For the MetaNode-language, as in TinyOS is complicated to manage String variables, instead of using Strings, numerical values will be used to describe the functionalities. The structures of MetaNode and MetaParam are shown in FIGURE 4.14 and 4.15. The MetaNode contains two numerical values of type *uint8_t* which are the MetaLabel and the IdLabel and an array of MetaParams. The size of the array is five, due to in the worst case is the maximum number of parameters that may be needed. The MetaParam struct contains an *uint8_t* field which is the type and an *int16_t* which is the value.

```
typedef struct MetaNode{
      uint8_t metaLabel;
      uint8_t idLabel;
      MetaParam parameters[5];
}MetaNode;
```

FIGURE 4.14: Struct definition for *MetaNode*.

In order to translate from Meta-language to MetaNode-language as well as being able to execute the operations described by MetaNodes, it is necessary to define the

```
typedef struct MetaParam{
      uint8_t type;
      int16_t value;
}MetaParam;
```

FIGURE 4.15: Struct definition for *MetaParam*.

numerical values that corresponds to each case for MetaLabel, IdLabel of the MetaNode structure and for the type and value of the MetaParam structure.

In TABLE 4.3 the events that the user can implement inside the code are defined. In the first column, it is shown how they have to be used in the Meta-language and which are the attributes for each one. The MetaNode-language tag is the name used in TinyOS to define them while the MetaNode value is the numerical value used in the MetaLabel field.

| ***Meta-language*** | ***MetaNode-language tag*** | ***MetaNode value*** |
|---|---|---|
| *BOOT_EVENT()* | BOOT_EVENT | 6 |
| *SENDPREAMBLE_EVENT( destAddr, error)* | SENDPREAMBLE_EVENT | 7 |
| *SENDDATA_EVENT(destAddr, dataCounter, error)* | SENDDATA_EVENT | 8 |
| *SENDACK_EVENT(destAddr, error)* | SENDACK_EVENT | 9 |
| *SENDRTS_EVENT(destAddr, error)* | SENDRTS_EVENT | 10 |
| *SENDCTS_EVENT(destAddr, error)* | SENDCTS_EVENT | 11 |
| *SENDSYNC_EVENT(error)* | SENDSYNC_EVENT | 12 |
| *RECEIVEMONOLITHIC_EVENT()* | RECEIVEMONOLITHIC_EVENT | 13 |
| *RECEIVEMFP_EVENT( preambleDuration, preambleTime, destAddr)* | RECEIVEMFP_EVENT | 14 |
| *RECEIVESTROBBED_EVENT( destAddr, srcAddr)* | RECEIVESTROBBED_EVENT | 15 |
| *RECEIVEDATA_EVENT( destAddr, srcAddr, dataCounter)* | RECEIVEDATA_EVENT | 16 |
| *RECEIVEACK_EVENT(destAddr, srcAddr)* | RECEIVEACK_EVENT | 17 |
| *RECEIVERTS_EVENT(destAddr, srcAddr)* | RECEIVERTS_EVENT | 18 |
| *RECEIVECTS_EVENT(destAddr, srcAddr)* | RECEIVECTS_EVENT | 19 |
| *RECEIVESYNC_EVENT( timeToSleep)* | RECEIVESYNC_EVENT | 20 |
| *LPLSIGNALDETECT_EVENT()* | LPLSIGNALDETECT_EVENT | 21 |
| *LPLSTOP_EVENT()* | LPLSTOP_EVENT | 22 |
| *FIREDCS_EVENT(mediumBusy)* | FIREDCS_EVENT | 23 |
| *READRSSI_EVENT(value)* | READRSSI_EVENT | 24 |
| *RADIOSTART_EVENT()* | RADIOSTART_EVENT | 25 |
| *RADIOSTOP_EVENT()* | RADIOSTOP_EVENT | 26 |
| *TIMER0FIRED_EVENT()* | TIMER0FIRED_EVENT | 27 |
| *TIMER1FIRED_EVENT()* | TIMER1FIRED_EVENT | 28 |
| *TIMER2FIRED_EVENT()* | TIMER2FIRED_EVENT | 29 |
| *TIMER3FIRED_EVENT()* | TIMER3FIRED_EVENT | 30 |
| *TIMER4FIRED_EVENT()* | TIMER4FIRED_EVENT | 31 |
| *TIMER5FIRED_EVENT()* | TIMER5FIRED_EVENT | 32 |
| *TIMER6FIRED_EVENT()* | TIMER6FIRED_EVENT | 33 |
| *TIMER7FIRED_EVENT()* | TIMER7FIRED_EVENT | 34 |
| *TIMER8FIRED_EVENT()* | TIMER8FIRED_EVENT | 35 |

TABLE 4.3: Definition and numerical value of events.

The rest of possible values that can be included in MetaLabel field are used for function calls, end of an event implementation, if-else structures, label-goto structures, variable definitions and expressions which are defined in TABLE 4.4.

| ***Operation*** | ***Meta-language*** | ***MetaNode-language tag*** | ***MetaNode value*** |
|---|---|---|---|
| Function call | *function_name();* | FUNC | 0 |
| If-else structure | *if(variable_name == 2){*<br>*}else{*<br>*}endif;* | IF<br>ELSE<br>END_IF | 1<br>2<br>3 |
| Label-goto structure | *label id;*<br>*goto id;* | LABEL<br>GOTO | 4<br>5 |
| End of event implementation | } | END_EVENT | 36 |
| Variable definition | *int variable_name;* | DEF | 37 |
| Expression | *variable_name = 2;* | EXP | 38 |

TABLE 4.4: Definition and numerical values of different operations.

MetaNodes of function type, as it is explained in section 3.2.2.4, should include also to which certain function or command refers in the IdLabel field. TABLE 4.5 defines how they should be called in Meta-language and which parameters should be provided, the name used in TinyOS and their values.

| ***Meta-language*** | ***MetaNode-language tag*** | ***MetaNode value*** |
|---|---|---|
| *sendPreambleMonolithic( preambleLength);* | SENDMONOLITHIC | 0 |
| *sendPreambleMFP(destination, preamble-Length);* | SENDMFP | 1 |
| *sendPreambleStrobbed(destination, preambleLength);* | SENDSTROBBED | 2 |
| *stopPreamble();* | STOPPREAMBLE | 3 |
| *sendData(destination, counter);* | SENDDATA | 4 |
| *sendAck(destination);* | SENDACK | 5 |
| *sendRts(destination);* | SENDRTS | 6 |
| *sendCts(destination);* | SENDCTS | 7 |
| *sendSync(timeToSleep);* | SENDSYNC | 8 |
| *sendCancel();* | SENDCANCEL | 9 |
| *setSleepInterval(interval);* | LPLSETSLEEPINT | 10 |
| *setDutyCycle(dutyCycle);* | LPLSETDUTYCYCLE | 11 |
| *getSleepInterval();* | LPLGETSLEEPINT | 12 |
| *getDutyCycle();* | LPLGETDUTYCYCLE | 13 |
| *startLpl();* | LPLSTART | 14 |
| *stopLpl();* | LPLSTOP | 15 |
| *goSleepTime(time);* | LPLSLEEP_MFP | 16 |
| *startCS(time);* | CSSTART | 17 |
| *csIsIdle();* | CSISIDLE | 18 |
| *updateTreshold();* | CSUPTADETRESH | 19 |
| *stopCS();* | CSSTOP | 20 |
| *getBinaryExponentialBO( numCollisions);* | GETBINEXPBO | 21 |
| *startRadio();* | RADIOSTART | 22 |
| *stopRadio();* | RADIOSTOP | 23 |
| *getRadioPowerState();* | RADIOGETPOWERSTATE | 24 |
| *getSplitControlState();* | RADIOGETCONTROLSTATE | 25 |
| *randomNum();* | RANDOMNUM32 | 26 |
| *timerStartOneShot(id, duration);* | TIMERSTARTONESHOT | 27 |
| *timerStartPeriodic(id, duration, bounds);* | TIMERSTARTPERIODIC | 28 |
| *timerStop(id);* | TIMERSTOP | 29 |
| *timerIsRunning(id);* | TIMERISRUNNING | 30 |
| *timerTimeRemaining(id);* | TIMERTIMEREMAINING | 31 |
| *timerGetDuration(id);* | TIMERGETDURATION | 32 |
| *timerGetNow(id);* | TIMERGETNOW | 33 |
| *setAddressRecognitionEnabled(boolean);* | SETADDRRECOGNITION | 34 |
| *led0On();* | LED0ON | 35 |
| *led0Off();* | LED0OFF | 36 |
| *led1On();* | LED1ON | 37 |
| *led1Off();* | LED1OFF | 38 |
| *led2On();* | LED2ON | 39 |
| *led2Off();* | LED2OFF | 40 |
| *ledSetNum(num);* | LEDSET | 41 |
| *setNewNoiseFloorValue(value);* | NEWNOISEFLOORSAMPLE | 42 |
| *getNoiseFloor();* | GETNOISEFLOOR | 43 |
| *readRssi();* | READRSSI | 44 |

TABLE 4.5: Definition and numerical value of function calls.

Another important element that has to be defined is the numerical value for each of the operations used in expressions and in *if conditions*. As it is explained in section 3.2.2.2, expressions allow the user to assign, add or subtract a value to a variable and the specific operation should be included in IdLabel field. In the case of the *if-else structures*, as it is explained in section 3.2.2.5, the *if condition* can be *equal to*, *greater than*, *greater than or equal to*, *less than* or *less than or equal to*. Therefore, all these possibilities are defined in TABLE 4.6.

| ***Condition*** | ***Meta-language*** | ***MetaNode-language tag*** | ***MetaNode value*** |
|---|---|---|---|
| Assign a value or condition equal to | == | EQUAL | 0 |
| Add a value | + | ADD | 1 |
| Subtract a value | - | SUBTRACT | 2 |
| Greater than | > | GREATER | 3 |
| Greater than or equal to | >= | GREATER_OR_EQUAL | 4 |
| Less than | < | LESS | 5 |
| Less than or equal to | <= | LESS_OR_EQUAL | 6 |

TABLE 4.6: Numerical values for expressions and if-else structures conditions.

Regarding to MetaParam structure, the numerical value for type field should be defined due to it varies depending on if it is a numerical value, a variable or a function call. This definition is shown in TABLE 4.7.

| ***Parameter Type*** | ***MetaNode-language tag*** | ***MetaNode value*** |
|---|---|---|
| Numerical value | VALUE | 0 |
| Variable | VAR | 1 |
| Returned value of a function | FUNC | 2 |

TABLE 4.7: Numerical values for expressions and if-else structures conditions.

Finally, the MetaParam value field has to be defined which will have different meanings depending on the three different cases:

- *A number*: the value indicates the number is. As the numerical value, in this case, can be positive or negative; value field of MetaParam is an integer instead of an unsigned integer.
- *A variable*: as it is explained before, Strings are complicated to manage in TinyOS. For that reason, each variable (system, attribute, local or global) will have a numerical value as identifier. System variables and event attributes have a number which is predefined. However, the variables defined by the user in the MAC protocol will have a number assigned by order. For instance, if *variable1* is the first defined will have the value 18, then when a *variable2* is defined will have the value 19, and so on.
  System variables can be used in all the code while event attributes can be only

used inside those events which have them as attributes. TABLE 4.8 defines the names of these variables. The variables between *success* and *ecancel*, both included, are the possible values of *error* when is returned by the events.

| ***Meta-language*** | ***Identifier*** | ***Type*** | ***Information*** |
|---|---|---|---|
| *tos_node_id* | 0 | System var. | It indicates the own id of the sensor node. |
| *destAddr* | 1 | Attribute | Indicates the node id of the destination sensor node. |
| *error* | 2 | Attribute | Indicates if the function/event is success/fail/ebusy/ealready/ecancel. |
| *dataCounter* | 3 | Attribute | Indicates the value of the counter received. |
| *srcAddr* | 4 | Attribute | Indicates the node id of the source sensor node. |
| *preambleDuration* | 5 | Attribute | Indicates in milliseconds the duration of the preamble. |
| *preambleTime* | 6 | Attribute | Indicates in milliseconds the time since it has started to send preamble. |
| *mediumBusy* | 7 | Attribute | Indicates if the medium is free (false) or other sensor nodes are transmitting (true). |
| *timeToSleep* | 8 | Attribute | Indicates the time until the sensor node switches the radio off. |
| *rssiValue* | 9 | Attribute | Indicates the value of the RSSI measured. |
| *true* | 10 | System var. | Indicates a *true* like a boolean variable. |
| *false* | 11 | System var. | Indicates a *false* like a boolean variable. |
| *success* | 12 | System var. | Indicates that there was no error. |
| *fail* | 13 | System var. | Indicates that there was a fail. |
| *ebusy* | 14 | System var. | Indicates that the component is occupied. |
| *ealready* | 15 | System var. | Indicates that that the function is already started. |
| *ecancel* | 16 | System var. | Indicates that the function has been cancelled by the user. |
| *broadcast_addr* | 17 | System var. | Indicates the broadcast id. |

TABLE 4.8: System variables and event attributes.

- *Returned value by a function call*: the value indicates the function called, thus its value is assigned like in TABLE 4.5.

## 4.3 META-LANGUAGE PARSER

One of the most important components of the toolchain is the Meta-language Parser which is the responsible of reading text files written using Meta-language and translate it into MetaNodes.

As it has been mentioned before, this can be a hard process. Therefore, the Meta-language Parser will be implemented inside a User Interface designed in [1] which enables fast runtime reconfiguration of MAC protocols. This User Intarface has been implemented in Java due to TinyOS provides some native libraries in order to establish a communication with the sensor node using Java. For that reason, the Meta-language Parser will be implemented in Java too.

Mainly, the process of parsing consists of four basic steps: init, read line, identify node and parse node. They are explained below:

- *Init*: Before reading the text file, it is important to reset the variables used in parsing process, such as number of line read, *If* counters, the context, etc.
- *Read line*: Once the variables are initiated, the text file is read line by line.
- *Identify node*: It consists in identify to which instruction refers the line read previously according to some string patterns. There are some keywords which allow us to carry out this process easily. If line begins with "*int* ", it means a *variable definition*. In case of beginning with "*label* " or "*goto* ", it refers to a *label definition* or *goto statement* respectively. Regarding to *if statements*, it is easy to recognize them due to they start with "*if(*". However, *else statements* and *endif statement* are recognizable because they are exactly like "*}else{*" and "*}endif;*" respectively. In case the line starts in capital letters, it implies an *event implementation*. If the line contains an "=" symbol but it does not start with "*if(*", it is an *expression*. By elimination, if line does not match to none of the cases explained above, it is a function call.
- *Parse node*: As the line has been identified, each line will be parsed to a certain MetaNode filling all the fields according to its type and the Meta-language description. As a MetaNode object in Java is necessary, we will use MIG (Message Interface Generator) which is a tool to generate code that processes TinyOS messages. Thereby, MIG will convert TinyOS MetaNode structure into a totally compatible Java MetaNode object by reading the header file where it is defined using next command:

```
>> mig java -target=null -java-classname=MetaNode
    MACDefinition.h MetaNode -o MetaNode.java
```

  For each node, a verification of errors is carried out such as the node complies the Meta-language syntax, variables used have been previously defined or functions called are the ones provided by toolchain. In case of error, a message will be sent to the shell so that it displays it to the user. Most of procedures to parse a node are trivial, but others are quite peculiar, thus we will explained them exhaustively in next subsections.

Once the node is parsed, the process starts again by reading next line and so on.

### *4.3.1 Parsing a Variable definition*

User is able to define as many variables as he desires, thus many variables should be managed by the toolchain. As it is explained in section 4.2, Strings are complicated to manage in TinyOS. Therefore, each variable (system, attribute, local or global) will have a numerical value as identifier. However, the user is able to define variables using Strings as name when writing the text file. For that reason, some kind of structure is needed to store the translation between the String name defined by the user and the numerical identifier generated by the toolchain. An object called *Variable* has been designed with this purpose. It contains the name in String format, the numerical identifier and the context definition.

Once a variable definition is parsed, a new *Variable* object is instantiated and filled all its fields. The name is the one used in the text file, the numerical identifier is the identifier of the last defined variable incremented by one unit and the context of the definition will be the number of the event where the variable is being defined or DEF in case of being a global variable. This new *Variable* is stored in a list called *VariableBuffer*. System variables and attributes are also stored inside this buffer due to sensor node needs a numerical value to access them too.

Moreover, there is an additional field defined in *Variable* object, a boolean type which indicates if the variable exists or it is not needed any more. This field is very useful when user tries to modify MAC protocol at runtime and removes a line where a variable is being defined. The toolchain will set this field to false such that when the *nodeList* is checked, an error will appear because the variable will not appear in *VariableBuffer* as existing. If the user redefine the variable again, the toolchain only needs to set this field to true, an everything will work as before again, remaining the same numerical identifier and all the references to this variable intact.

### *4.3.2 Parsing an If-Else statement*

Other nodes which are difficult to parse are *if-else statements* due to they may include other nodes inside itself.

The process begins recognizing an *if statement*. After verifying there is no error related to the syntax, i.e. parentheses or curly brackets missing; an IF MetaNode is created and added to *nodeList*. Before returning to the process of parsing the rest of lines of text file, as *if-statements* usually include other instructions inside is necessary to parse them before closing with an END_IF MetaNode.

For that reason, we have designed a complex procedure to interpret and parse all the instructions included in the *if-else statements* before returning to the normal procedure of parsing the text file as FIGURE 4.16 shows.

Initially, it is necessary to initialize some variables: *counterIf* must be incremented by one unit due to an If node has been parsed, it will indicate the depth level of the If when some *if-else statements* are defined inside others (chained *if-else statements*); *ifsOpened* must be incremented by one unit too because it enables toolchain to detect how many ifs has been opened but not successfully closed, in case this variable is

different of 0 after the parsing process is finished; *found* must be set to false, in case of being true, it indicates the process is already finished.

After that, the value of *found* is verified. In case of being true, the process of parsing nodes would conclude. Otherwise, a new line must be read and interpreted. If line does not start with character "*}*", it means it is a normal node so it should be identified first, parsed and added to *nodeList*.

In case the line starts with this character, it has to be checked if an *else* or *endif statement* is defined depending if the line matches with "*}else{*" or "*}endif;*" respectively. Then, the specific MetaNode for each case must be created and inserted into *nodeList*. However, some variables modifications are needed in *endif statement* case before creating the MetaNode: *found* should be set to true to indicate that the whole *if-else* block has already finished, and the counters *ifsOpened* and *counterIf* should be decremented by one unit. If the line does not match with any of these two strings, an error should be shown by the shell indicating that there is an error in a certain line of the text file and the process stops.

After that, the process go back to the verification of found variable and continue parsing the rest of nodes in case the variable remains being false. This procedure allow us to parse some chained *if-else statements*, e.g. one if-else statement defined inside another one; due to there is a branch of the flowchart where allows to identify a node and start the same process for this other *if-else statement*. Once this process has finished, it will continue with the main process.

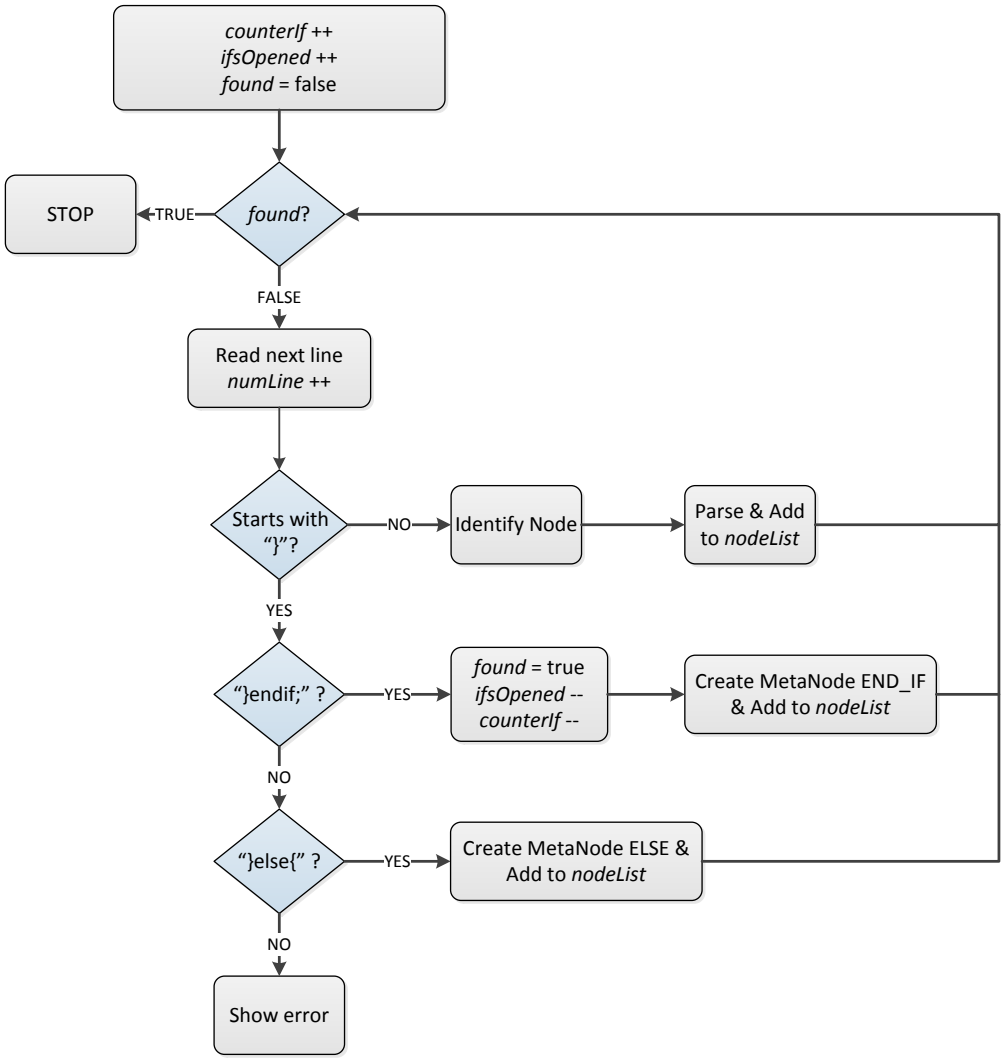


FIGURE 4.16: Flowchart ot the procedure of parsing nodes included inside an If-else statement.

### 4.3.3 *Parsing an Event implementation*

A similar case to *if-else statements* occurs when trying to parse an event implementation as it may include other instructions inside itself.

The process begins recognizing an event. After verifying there is no error related to the syntax, i.e. parentheses or curly brackets missing or that event has been already implemented; a MetaNode with its name as MetaLabel is created and added to *nodeList*. Before returning to the process of parsing the rest of lines of text file, as events usually include other instructions inside is necessary to parse them before closing with an END_EVENT MetaNode.

For that reason, we have designed a complex procedure to interpret and parse all the instructions included in the event implementation before returning to the normal procedure of parsing the text file as FIGURE 4.17 shows which is very similar to the

one explained in subsection 4.3.2.

Initially, it is necessary to initialize some variables: *eventsOpened* must be incremented by one unit because it enables toolchain to detect how many events has been opened but not successfully closed, in case this variable is different of 0 after the parsing process is finished; *context* should be set with the same tag as the MetaLabel of the event MetaNode to indicate that next variables to be defined will belong to this context; *found* must be set to false, in case of being true, it indicates the process of parse this event is already finished.

After that, the value of *found* is verified. In case of being true, the process would conclude and the Parser will continue to process the rest of the code. Otherwise, a new line must be read and interpreted. If line does not start with character "*}*", it means it is a normal node so it should be identified first, parsed and added to *nodeList*. An error will be shown if the node is another event implementation due to this instruction is not allowed in that location.

In case the line starts with this character, some variables modifications are needed before creating the END_EVENT MetaNode and adding to *nodeList*: *found* should be set to true to indicate that the whole event implementation has already finished, and the counter *eventsOpened* should be decremented by one unit.

After that, the process go back to the verification of found variable and continue parsing the rest of nodes in case the variable remains being false.

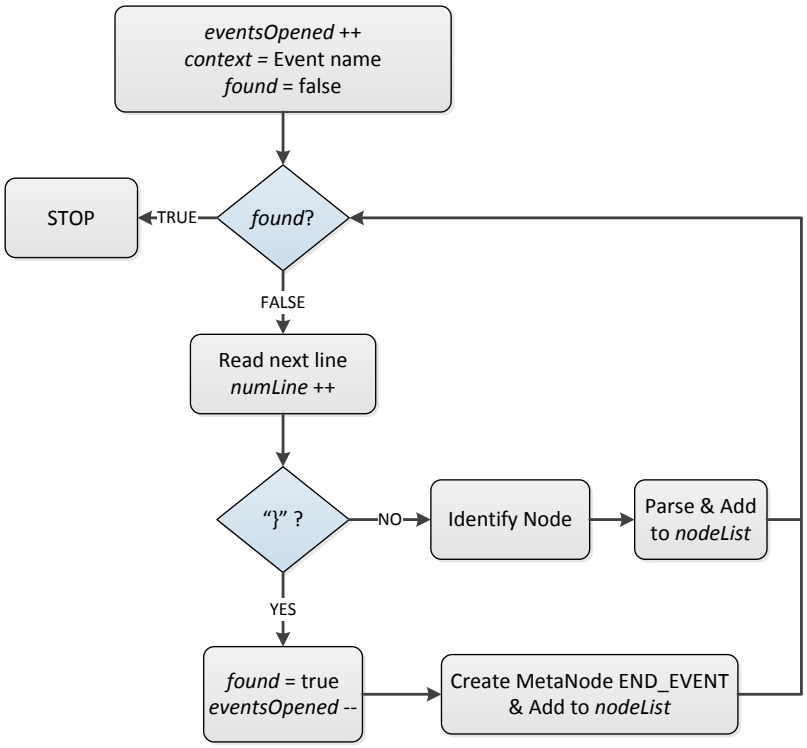


FIGURE 4.17: Flowchart ot the procedure of parsing nodes included inside an event implementation.

# 5

# EXPERIMENTAL RESULTS AND EVALUATION

The target of the toolchain is to make easier the process of prototyping and designing MAC protocols for WSNs by abstracting the designer away the complexities of TinyOS environment. However, it is also necessary to make sure that the behaviour of the designed MAC protocols is comparable to their monolithically implemented counterparts.

In this chapter, the experimental results and evaluation of different aspects for some MAC protocols are presented. Metrics such as memory consumption or execution time are evaluated in order to determine the disadvantages and advantages of the MAC protocols implemented with the toolchain compared to the monolithically implemented ones. The monolithic approach consists of designing MAC protocols using component-based design in a static manner.

All the experiments were carried out on two different platforms, such as TelosB sensors [5] which use the radio chip Texas Instruments CC2420 [31] and Mica2 sensors [6] which use Texas Instruments CC1000 [32] as radio chip.

The main metrics used to perform the evaluation of our toolchain are:

- *Memory consumption*: the targets of the toolchain are resource constraint embedded platforms. For that reason, we are interested in the memory overhead that the toolchain imposes to guarantee the proper execution of the MAC protocols implementations designed by the toolchain.

- *Execution time*: it refers to the time that the toolchain needs to execute a certain function. In addition, this term can be used when measuring the time needed by a certain MAC protocol to perform all the operations needed to send a packet, such as making Carrier Sensing, Sendig Preamble, etc.

## 5.1 MEMORY CONSUMPTION

Since our toolchain targets to run on resource constraint embedded platforms, memory consumption is an important metric. We have measured both RAM and ROM in terms of memory consumption. We refer ROM consumption to the number of ROM bytes occupied when a TinyOS project is deployed in a platform, while RAM consumption is the number of bytes of RAM memory reserved to allocate variables or objects before starting the execution.

A shell script has been used in order to realize the memory footprint tests. This script analyses the executable files created after compiling TinyOS projects and extract a complete trace indicating the ROM and RAM memory consumed by each component. The next command should be used to run this script:

```
>> ./memory_usage -v [path to executable file]
```

TABLE 5.1 shows the memory footprint results in terms of RAM and ROM consumption for a TelosB sensor node for different MAC protocols. The tested MAC protocols were B-MAC, MFP-MAC, X-MAC as example of preamble-sampling protocols, S-MAC and T-MAC as example of common active periods protocols. In addition, we measure the memory footprint for a simple application which does not use any MAC protocol; it only starts the radio, send one packet after that and stops the radio cyclically.

The results indicate that the reconfigurable toolchain increases approximately between a 20 % and a 40 % the ROM consumption for the MAC protocols, and a 85 % for the simple application. For the monolithic implementations the memory consumption varies: as more complex the MAC protocol is, higher is the ROM consumption, because more code lines are needed to implement all the functionalities. However, for the implementations using our toolchain, the consumption is always the same, in some cases the overhead is high, but as more complex the protocol is, it includes less overhead.

In addition, the MetaNodes obtained after the parsing process are stored inside an execution list which is inside the sensor node. Moreover, the variables are also stored in a variable list in order to do all the operations needed, as explained in [1]. As TinyOS does not allow Dynamic Memory Allocation, these lists have to be predefined with an static size depending on the RAM of the platform. According to that, this can be a reason of the high memory consumption obtained by our toolchain in terms of RAM.

| ***Protocols*** | ***Monolithic Implementation*** | | ***Toolchain*** | |
|---|---|---|---|---|
| | *ROM* | *RAM* | *ROM* | *RAM* |
| *B-MAC* | 25112 bytes | 1242 bytes | 31282 bytes | 6836 bytes |
| *MFP-MAC* | 25734 bytes | 1242 bytes | 31282 bytes | 6836 bytes |
| *X-MAC* | 25304 bytes | 1242 bytes | 31282 bytes | 6836 bytes |
| *S-MAC* | 24326 bytes | 1306 bytes | 31282 bytes | 6836 bytes |
| *T-MAC* | 22096 bytes | 1140 bytes | 31282 bytes | 6836 bytes |
| *No-Protocol* | 16978 bytes | 1102 bytes | 31282 bytes | 6836 bytes |

TABLE 5.1: Memory footprint in bytes of the implementation for different MAC protocols on TelosB sensor nodes.

Although preamble-sampling protocols like B-MAC, MFP-MAC or X-MAC are simpler than common active periods protocols like S-MAC or T-MAC, the results show that for the last case the monolithic implementations with reusable components have smaller ROM consumption. This effect is caused by the fact of the implementation of preamble-sampling protocols requires the use of more components. For instance, in TABLE 4.2, we can see that these protocols use two components more than S-MAC and T-MAC, in particular MACLowPowerListening and MACSendPreamble.

TABLE 5.2 shows the memory footprint for a Mica2 sensor node for different MAC protocols. However, in this case, the test can be only carried out with B-MAC, MFP-MAC and X-MAC, because Mica2 hardware limitations. As it is explained in [1],

Mica2 has only 4 kBytes of RAM, thus it only allows to execute an execution list of 130 MetaNodes as maximum, while 200 MetaNodes are needed to implement S-MAC or T-MAC at least. In this case, ROM consumption increases approximately a 30 % for the MAC protocols and a 98 % for the simple application. RAM results for the reconfigurable toolchain show lower numbers compared to TelosB results due to the size restriction of the execution list, but it increases also a lot compared with the monolithic approach.

| ***Protocols*** | ***Monolithic Implementation*** | | ***Toolchain*** | |
|---|---|---|---|---|
| | *ROM* | *RAM* | *ROM* | *RAM* |
| *B-MAC* | 21366 bytes | 1445 bytes | 28306 bytes | 3745 bytes |
| *MFP-MAC* | 21686 bytes | 1445 bytes | 28306 bytes | 3745 bytes |
| *X-MAC* | 21708 bytes | 1499 bytes | 28306 bytes | 3745 bytes |
| *No-Protocol* | 14278 bytes | 999 bytes | 28306 bytes | 3745 bytes |

TABLE 5.2: Memory footprint in bytes of the implementation for different MAC protocols on Mica2 sensor nodes.

## 5.2 EXECUTION TIME

As it has been explained before, once the MAC protocol description is translated from Meta-language to MetaNodes by the Parser, the execution list which is running in the sensor node is the responsible of the MAC execution. For that reason, we need to measure which is the time needed by our toolchain to execute an instruction of a MAC protocol and compare it with the results obtained using monolithic approach. Both results have been compared in terms of absolute time and the overhead added by the toolchain.

This experiment consists of two different parts. First, it has been measured which is the time needed by our toolchain to execute simple MAC operations, such as starting the radio or sending preamble. After that, an evaluation has been done while measuring which is the time needed when performing a whole MAC operation. It consists of measuring the total time needed by a MAC protocol to send a packet performing all the operations that this specific protocol may require.

As the work realized during this thesis is related with the one done in [1], this experiment is important to see how is the performance of both implementations joined.

Before starting to discuss about the results obtained, we need to specify which are the two setups used to make the measurements.

In FIGURE 5.1, it is shown the experimental setup used to measure the execution time in monolithic approaches. In this case, a very small resistor of 3.33 $\Omega$ is connected in series with the sensor node. Moreover, the sensor is powered up by a power supply. As batteries used by sensor nodes provide 3.0 V, this power supply is set to this voltage. Finally, an oscilloscope measures which is the voltage across the resistor. Using techniques, such as turning on different LEDs when the function starts and finishes, allows us to easily mark the execution time of a function in the oscilloscope screen because LEDs generate a peak of current consumption when they are turned on.

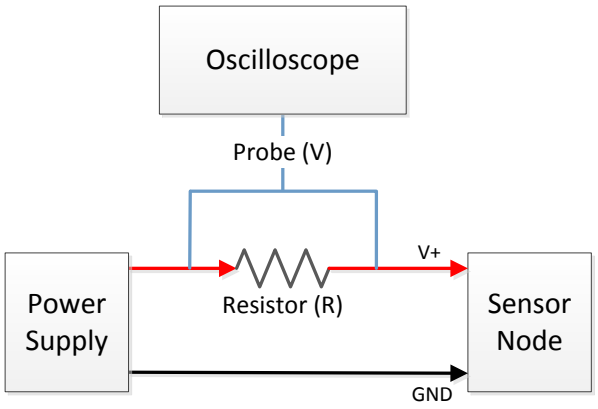


FIGURE 5.1: Block diagram of the experimental setup for monolithic execution time measurements. [1]

As the sensor node should be connected to a PC in order to receive the MetaNodes obtained after the parsing process, the setup shown in FIGURE 5.1 cannot be used to measure the execution time for our toolchain. In FIGURE 5.2 it is shown which is the setup used to measure this time. In this case, the same resistor is connected in series with the sensor node. However, the sensor node is connected to the PC trough the serial port. Thereby, the sensor node is powered up by the PC and it can receive from and send packets to it. As the power provided by the PC is 5 V, we cannot compare the results in terms of voltage or current. However, since we are only interested in the time domain, no modification is needed. Finally, an oscilloscope measures with is the voltage across the resistor by subtracting the voltage provided by the PC and the voltage in the sensor node.

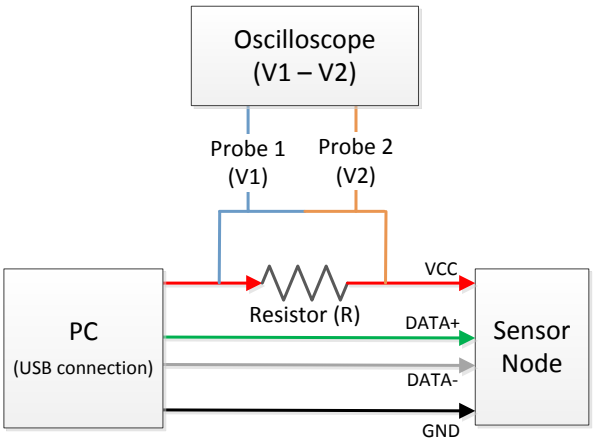


FIGURE 5.2: Block diagram of the experimental setup for toolchain execution time measurements. [1]

In order to have statistically significant results, 10 samples have been taken for each execution time experiment and the average is presented.

### 5.2.1 Basic Functionalities

In order to measure which is the time needed by our toolchain to execute most common basic functionalities, we have selected these five: start the radio, stop the radio, read the actual RSSI value, perform carrier sensing and send a packet. A comparison between the execution time measured for our toolchain and the time needed by the monolothic approach has been done in [1]. Most important results are summarized in TABLE 5.3 and TABLE 5.2.

TABLE 5.3 shows the results obtained in a TelosB sensor node, while TABLE 5.2 shows the results for a Mica2 platform. The time added oscillates between 100 µs and 250 µs for TelosB, and between 65 µs and 120 µs for Mica2 platform. This delay introduced by toolchain can fluctuate depending on the position where the function calls and their events are located inside the execution list, as well as the number variables that these functions calls take as parameters. Mainly, the delay depends on the search process described in [1].

| ***Basic Functionality*** | ***Monolithic Time*** | ***Toolchain Time*** | ***Overhead Time*** | ***Overhead*** |
|---|---|---|---|---|
| Start Radio | 2.76 ms | 2.94 ms | 180 µs | 7 % |
| Stop Radio | 283 µs | 387 µs | 104 µs | 37 % |
| Read RSSI | 486 µs | 622 µs | 136 µs | 28 % |
| Carrier Sensing (50 ms) | 50.25 ms | 50.35 ms | 100 µs | 0.2 % |
| Send Packet (11 bytes payload) | 8.74 ms | 8.99 ms | 250 µs | 2.86 % |

TABLE 5.3: Time results for basic functionalities executed in a TelosB sensor node. [1]

| ***Basic Functionality*** | ***Monolithic Time*** | ***Toolchain Time*** | ***Overhead Time*** | ***Overhead*** |
|---|---|---|---|---|
| Start Radio | 2.49 ms | 2.57 ms | 82 µs | 3 % |
| Stop Radio | 188 µs | 262 µs | 74 µs | 39 % |
| Read RSSI | 469 µs | 534 µs | 65 µs | 14 % |
| Carrier Sensing (50 ms) | 50.23 ms | 50.35 ms | 120 µs | 0.24 % |
| Send Packet (11 bytes payload) | 22.83 ms | 22.90 ms | 70 µs | 0.31 % |

TABLE 5.4: Time results for basic functionalities executed in a Mica2 sensor node. [1]

### 5.2.2 MAC protocols

Similarly to the previous case, a comparison of the execution has been done in [1] between the monolithic approach and the toolchain when performing complete MAC operations.

For this experiment, we have measured the time needed by a MAC protocol to send a packet performing all the operations needed, such as Carrier Sensing, Sending Preamble or making RTS-CTS-DATA-ACK exchange. In this case, an example of preamble-sampling MAC protocol, B-MAC; and an example of common active period,

S-MAC; have been used. In addition, an application which does not use any MAC protocol has been used for this experiment.

The results obtained are summarized in TABLE 5.5 for TelosB and in TABLE 5.6 for Mica2 platforms. As it can be observed, the maximum overhead introduced is 4 % in case of not using any protocol, which is not quite representative. In the other cases, the results are around or below 1 % of overhead. In case of Mica2 platforms, as it has been mentioned before, due to hardware restrictions an implementation of common active period protocols are not possible.

| ***MAC Protocol*** | ***Monolithic Time*** | ***Toolchain Time*** | ***Overhead Time*** | ***Overhead*** |
|---|---|---|---|---|
| *No protocol* | 10.53 ms | 10.96 ms | 427 µs | 4.06 % |
| *B-MAC* | 161.78 ms | 163.36 ms | 1.577 ms | 0.97 % |
| *S-MAC* | 195.84 ms | 198.57 ms | 2.727 ms | 1.39 % |

TABLE 5.5: Time results for different protocols executed in a TelosB sensor node. [1]

| ***MAC Protocol*** | ***Monolithic Time*** | ***Toolchain Time*** | ***Overhead Time*** | ***Overhead*** |
|---|---|---|---|---|
| *No protocol* | 26.67 ms | 26.88 ms | 211 µs | 0.79 % |
| *B-MAC* | 190.49 ms | 191.13 ms | 637 µs | 0.33 % |

TABLE 5.6: Time results for different protocols executed in a Mica2 sensor node. [1]

# 6

# Conclusions and Future Work

In this thesis, we present a toolchain which enables easy and fast MAC protocol prototyping and designing. In order to allow simple MAC protocol design, we have taken into account a component oriented approach in defining the MAC components which provide basic functionalities. The components have been implemented following a hardware-independent approach. Therefore, a user can design a MAC protocol without having knowledge of the platform for deployment. Furthermore, the users are also not required to know the specific programming environment of the target platform, such as NesC [30], since we have designed a Meta-language specifically for MAC design using our toolchain. The Meta-language has simple syntax and is C-like. The Meta-language is translated by a corresponding parser to MetaNode-language which is understood by the target platform. The MetaNode-language is transferred to the sensor node through a User Interface as described in [1] which is a collaboration work with this thesis.

We have evaluated the toolchain in terms of memory consumption and execution time. We have compared the results obtained to monolithic MAC protocols implementations using two different platforms with different radio chips: TelosB and Mica2. In both cases, the results show that the toolchain introduces an acceptable memory and execution time overhead, which in most of the cases is below 5 %. However, it allows to design MAC protocols in a simple manner by abstracting the designer away the complexities of TinyOS environment. Our toolchain helps to achieve the rapid reconfiguration of MAC protocols proposed by the work implemented in [1].

In conclusion, our toolchain provides two main contributions for MAC protocol development. First, our Meta-language provides a simple way to design new MAC protocols without having any knowledge about TinyOS programming. Second, as basic MAC components has been designed and implemented following the hardware independence approach, the toolchain can be executed in any platform.

In order to achieve a better performance of the toolchain, the memory consumption can be reduced. One solution could be to think another structure very similar to the existing MetaNode, in order to represent the same information but using less than 17 bytes which is the actual size of this structure. Taken into account all the possible values that each MetaNode field may require, the total size of the structure should be 13 bytes. According to that, in case of TelosB platforms, 1200 bytes of RAM would be free allowing us to increase the execution list size in 90 elements. In the same manner, in case of Mica2 platforms, 520 bytes of RAM would be free and the execution list can be enlarged in 40 elements. Thereby, more complex MAC protocols could be executed. Furthermore, more MAC Components could be implemented in order to extend the Meta-language with more functionalities to elaborate hybrid or centralized protocols as well as introducing functionalities of reliability like checksums.

Another important improvement that can be introduced to our toolchain is to implement a Graphical User Interface (GUI). Similarly to [15], a GUI which supports dragging and dropping components can be implemented to make easier the design of MAC protocols.

# A

# ABBREVIATIONS

**ACK** Acknowledgement

**CCA** Clear Channel Assesment

**CTS** Clear To Send

**GUI** Graphical User Interface

**I/O** Input and output

**IT** Information Technology

**LPL** Low Power Listening

**MFP** Micro-Frame Preamble

**MAC** Medium Access Control

**MAC-PD** MAC Protocol Designer

**MDPL** MAC Pattern Description Language

**MLA** MAC Layer Architecture

**OS** Operating System

**RAH-MAC** Rate Adaptive Hybrid MAC Protocol

**RAM** Random-Access Memory

**RBAR** Receiver-Based AutoRate Protocol

**ROM** Read-Only Memory

**RSSI** Received Signal Strength Indicator

**RTS** Request To Send

**SDL** Specification and Description Language

**SNR** Signal to Noise Ratio

**SP** Sensor-net Protocol

**ITU** International Telecommunication Union

**ULLA** Unified Link-Layer API

**UPMA** Unified Power Management Architecture

**WSN** Wireless Sensor Network

# List of Tables

# LIST OF FIGURES

# BIBLIOGRAPHY


[1] N. Arbós, "Reconfigurable Medium Access Control Solutions for Resource Constrained Wireless Networks," M.S. thesis, Institute for Networked Systems, RWTH Aachen University, September 2012.

[2] IEEE 802.11 Working Group, "IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements / Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications," Tech. Rep., (Institute of Electrical and Electronics Engineers (IEEE), 1999.

[3] D.J. Cook, S.K. Das, and John Wiley, *Smart Environments: Technologies, Protocols, and Applications*, Wiley, November 2004.

[4] K. Sohraby, D. Minoli, and T. Znati, *Wireless Sensor Networks: Technologies, Protocols, and Applications*, Wiley, April 2007.

[5] Crossbow Technology Inc., "Datasheet: TelosB Mote Platform," May 2004.

[6] Crossbow Technology Inc., "Datasheet: Mica2 Mote Platform," May 2004.

[7] D. G. Messerschmitt, "Rethinking Components: From Hardware and Software to Systems," *Proceedings of IEEE*, vol. 95, no. 7, pp. 1473–1496, July 2007.

[8] H. Liu, M. Parashar, and S. Hariri, "A Component Based Programming Framework for Autonomic Applications," *Proceedings of the International Conference on Autonomic Computing (ICAC 2004)*, 2004.

[9] D. Box, *Essential COM*, Addison-Wesley Professional, January 1998.

[10] J. Prosise, *Programming Microsoft .NET*, Microsoft, May 2002.

[11] R. Monson-Haefel, B. Burke, and S. Labourey, *Enterprise JavaBeans*, O'Reilly Media, 4th edition, June 2004.

[12] P. Levis and D. Gay, *TinyOS Programming*, Cambridge University Press, 2009.

[13] J. Ansari, X. Zhang, A. Achtzehn, M. Petrova, and P. Mähönen, "Decomposable MAC Framework for Highly Flexible and Adaptable MAC Realizations," *Dyspan2010*, pp. 222–248, September 2010.

[14] K. Klues, G. Hackmann, O. Chipara, and C.Lu, "A Component-Based Architecture for Power-Efficient Media Access Control in Wireless Sensor Networks," *SenSys*, November 2007.

[15] O. Salikeen, "Enabling Flexible Medium Access Design for Wireless Sensor Networks," M.S. thesis, Department of Wireless Networks, RWTH Aachen University, December 2010.

[16] D. Culler, P. Dutta, C. T. Ee, R. Fonseca, J. Hui, P. Levis, J. Polastre, S. Shenker, I. Stoica, G. Tolle, and J. Zhao, "Towards a Sensor Network Architecture: Lowering the Waistline," *Proceedings of the International Workshop on Hot Topics in Operating Systems (HotOS)*, 2005.

[17] J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shenker, and I. Stoica, "A Unifying Link Abstraction for Wireless Sensor Networks," *SenSys*, 2005.

[18] K. Rerkrai, J. Riihijärvi, M. Wellens, , and P. Mähönen, "Unified Link-Layer API Enabling Portable Protocols and Applications for Wireless Sensor Networks," *Proceedings of the IEEE International Conference on Communications (ICC)*, 2007.

[19] K. Klues, G. Xing, and C. Lu, "Towards a Unified Radio Power Management Architecture for Wireless Sensor Networks," *Wireless Communications and Mobile Computing - Distributed Systems of Sensors and Actuators*, vol. 9, pp. 313–323, March 2009.

[20] International Telecommunication Union Telecommunication Standarization Sector (ITU-T), "Z.100: Specification and Description Language (SDL)," Tech. Rep., International Telecommunication Union, August 2002.

[21] G. Yang, "A Toolchain for the Design and Implementation of Adaptive Medium Access Control Protocols," M.S. thesis, Department of Wireless Networks, RWTH Aachen University, December 2010.

[22] A. Bachir, M. Dohler, T. Watteyne, and K. K. Leung, "MAC Essentials for Wireless Sensor Networks," *IEEE communications surveys and tutorials*, vol. 12, no. 2, 2010.

[23] J. Polastre, "Sensor Network Media Access Design," Tech. Rep., Computer Science Division, EECS Department, University of California, Berkeley, 2003.

[24] A. Bachir, D. Barthel, M. Heusse, and A. Duda, "Micro-Frame Preamble MAC for Multihop Wireless Sensor Networks," *ICC Istanbul*, vol. 7, pp. 3365 – 3370, June 2006.

[25] M. Buettner, G. V. Yee, E. Anderson, and R.Han, "X-MAC: A Short Preamble MAC Protocol for Duty-Cycled Wireless Sensor Networks," Tech. Rep., Department of Computer Science, University of Colorado, Boulder, November 2006.

[26] W. Ye, J. Heidemann, and D. Estrin, "An Energy-Efficient MAC Protocol for Wireless Sensor Networks," *Procedings of the IEEE Infocom*, vol. 3, pp. 1567–1576, June 2002.

[27] T. van Dam and K. Langendoen, "An Adaptive Energy-Efficient MAC Protocol for Wireless Sensor Networks," *SenSys*, pp. 171–180, 2003.

[28] I. Rhee, A. Warrier, M. Aia, J. Min, and M. L. Sichitiu, "Z-MAC: A Hybrid MAC for Wireless Sensor Networks," *IEEE/ACM Transactions on Networking*, vol. 16, pp. 511–524, June 2008.

[29] G. Ahn, E. Miluzzo, A. T. Campbell, S. G. Hong, and F. Cuomo, "Funneling-MAC: A Localized, Sink-oriented MAC for Boosting Fidelity in Sensor Networks," *SenSys*, pp. 293–306, November 2006.

[30] D. Culler E. Brewer D. Gay, P. Levis, *nesC Language Reference Manual*, July 2009.

[31] Chipcon AS SmartRF, "Datasheet: CC2420 - 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver," June 2004.

[32] Chipcon AS SmartRF, "Datasheet: CC1000 - Single Chip Very Low Power RF Transceiver," April 2002.

# Declaration

I declare that this thesis does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any university and that to the best of knowledge it does not contain any materials previously published or written by another person except where due reference is made in the text.

Aachen, September 25, 2012
Luis Miguel Amorós