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

MASTER THESIS

TITLE: Reconfigurable Medium Access Control Solutions for Resource Constrained Wireless Networks

MASTER DEGREE: Master in Science in Telecommunication Engineering & Management

AUTHOR: Noemí Arbós Linio

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 allows a user to design a MAC protocol, load it onto a sensor platform and perform runtime reconfiguration. This thesis uses the component based MAC design and Meta-language as its basis. The component based MAC design and Meta-language have been developed by Luis Miguel Amorós in his thesis *A Tool for Rapid MAC Protocol Prototyping and Design for Wireless Sensor Networks* [1] which was done in collaboration with my work. Owing to this reason the two thesis have a tightly coupling with the other.

I would thank very much my partner for all the work he does and its support during the last 6 years. I thank also M. Sc. Junaid Ansari and M. Sc. Xi Zhang for their supervising, ideas and support.

Finally, I would like to dedicate this thesis to my family for their support during my studies and specially during my Erasmus. I dedicate also this thesis to the people I met in Aachen, they are the reason that makes these 7 months an unforgettable experience.

"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

Preamble II

Contents IV

Abstract VI

1 Introduction 1

2 Related Work 3

2.1 Component Oriented Design . . . 3

2.2 Adaptability . . . 4

2.3 Runtime Reconfiguration . . . 4

2.4 MAC Schemes for WSNs . . . 6

2.4.1 Preamble-sampling Protocols . . . 6

2.4.2 Common Active Periods Protocols . . . 6

2.4.3 Hybrid Protocols . . . 6

3 System Design 7

3.1 MAC Functional Blocks . . . 7

3.2 MAC Meta-languages . . . 8

3.3 Sensor Interface . . . 9

3.3.1 Execution List . . . 11

3.3.2 Variable List . . . 13

3.3.3 MetaNode Compiler . . . 15

3.4 User interface . . . 16

3.4.1 Command Shell . . . 17

3.4.2 Serial Controller . . . 18

4 Implementation 19

4.1 MAC Functional Blocks . . . 19

4.2 Sensor Interface . . . 20

4.2.1 Execution List . . . 20

4.2.2 Variable List . . . 25

4.2.3 MetaNode Compiler . . . 27

4.2.4 ExecutionScheduler Component . . . 31

4.2.5 SENSOR NODE SERIAL CONTROLLER 31
4.3 USER INTERFACE 32
4.3.1 COMMAND SHELL 33
4.3.2 PARSER 37
4.3.3 SERIAL CONTROLLER 37

5 EXPERIMENTAL RESULTS AND EVALUATION 41
5.1 EXECUTION OVERHEAD 41
5.1.1 BASIC FUNCTIONALITIES 42
5.1.2 MAC PROTOCOLS 46
5.2 RECONFIGURATION COSTS 47

6 CONCLUSIONS 51

A ABBREVIATIONS 52

LIST OF TABLES 53

LIST OF FIGURES 54

BIBLIOGRAPHY 55

DECLARATION 58

ABSTRACT

Wireless Sensor Networks (WSNs) consist of several autonomous resource constrained sensor nodes distributed over a geographical area. The sensor nodes can measure, for instance humidity, temperature or vibration. Therefore, these networks can be deployed in many different types of dynamic environments.

Traditionally, Medium Access Control (MAC) protocols used in WSNs are implemented in a monolithic fashion with tight coupling to the underlying hardware. Although this approach of design and implementation can usually make full use of the capability of the underlying hardware, the MAC solution is static and provides satisfying performance only under the pre-defined conditions. However, as application requirements and network conditions may change, adaptability and reconfigurability of MAC protocols are desired.

In this thesis, we have designed and implemented a toolchain which enables runtime MAC protocol reconfiguration for WSNs. The toolchain has been implemented in TinyOS using component-based design and hardware independence, allowing users to develop MAC solutions for WSNs, execute and reconfigure them in many platforms. Finally, a user is able to interact with the sensor node through a user interface developed in Java. Furthermore, this toolchain has been enhanced by the features introduced in [1] to enable also simplifying the design of MAC protocols, allowing non-specific sensors users to implement and finally execute and reconfigure them in sensor nodes.

The toolchain has been compared to monolithic implementations in terms of execution time and reconfiguration costs. The results show that the toolchain enables fast runtime reconfiguration of MAC protocols with an quantify execution time overhead. Our toolchain saves from 26 % up to 98 % the time needed to reconfigure a MAC protocol compared to the monolithic approach.

1

INTRODUCTION

Nowadays, smart environments represent an important evolution in building, industrial, home, utilities, health-care applications, traffic control, automation systems, etc [2]. A smart environment needs information about its surroundings as well as about its internal situation. This information can be provided by Wireless Sensor Networks (WSNs), which are responsible for detecting, monitoring, collecting and sending relevant data about the environment conditions.

A WSN is a network of several autonomous sensors distributed over a geographical area in an ad-hoc fashion which are used to monitor physical conditions, such as temperature, vibration, pressure, humidity, motion, sound or pollutants [3]. These sensors work together in a cooperative way to transmit the collected data through the network to a main location. Sensors nodes are resource constrained platforms in terms of energy, memory, computational speed and communications bandwidth. The most significant constrain is energy, due to the fact that sensor nodes usually operated on batteries.Since battery replacement can be cumbersome especially when sensor nodes are deployed in remote areas, efficient usage of energy to extend the sensor node lifetime becomes critical for sensor networks to be economically. In order to improve the energy consumption, WSNs use power aware Medium Access Control (MAC) protocols which govern efficiency the cost of radio communications to improve energy efficiency for sensor nodes.

Traditionally, these MAC protocols are implemented in a monolithic fashion with tight coupling to the underlying hardware which limits the adaptability and thus leads to underperformance when application requirements and/or network conditions change. Since, WSNs can be deployed in hostile and dynamic changing environment, the monolithically designed and implemented MAC protocol which is optimized for a specific scenario would not deliver the same expected performance when the scenario changes. For example, when a WSN would have to coexist with other networks on the same frequency band, reconfiguration of the MAC procedure would be necessary to allow the coexistence and fair sharing of spectrum resources. Another relevant scenario would be a WSN monitoring seismic activity where sensors nodes periodically send information about vibration [3]. However, when a earthquake is detected, a higher data rate is expected to detect possible greater replicas. The MAC protocol then needs to adapt itself to serve the new application requirements. Without having a flexible architecture for reconfigurable MAC protocol realizations, the existing MAC protocols perform poorly under the above mentioned situations.

In order to enable building MAC protocols on-the-fly without having to reprogram the target sensor nodes, we have designed and implemented a toolchain to reconfigure MAC protocols at runtime. If some modification is needed to adapt a MAC implementation into a new environment such as changing duty cycle from *Low Power Listening*,

there is no need to recompile source code. The development process has been done in collaboration with [1]. For that reason, the main components of the toolchain, such as the MAC Components or the languages used to design the MAC protocols, are tightly coupled with this thesis.

Overall, in this thesis we present a toolchain designed to enable rapid and flexible MAC protocol reconfiguration. It is implemented for TelosB [4] and Mica2 [5] as a proof of concept. This thesis is organized as follows. Chapter 2 describes the related work proposed for specific MAC protocols for WSNs, solutions for providing adaptive and on-the-fly modified protocols in general. Chapter 3 explains the design of the framework based on wiring engine concept which can be used to develop flexible adaptive applications in an efficient and reliable way. In Chapter 4, the implementation is explained. The evaluation and experimental results of our toolchain are presented in Chapter 5 to show the disadvantages of our solution. Finally, we present some conclusions and future work.

2

Related Work

In recent years, adaptation and reconfigurability are becoming important features in WSNs due to they are usually deployed in very hostile and dynamic changing environments. In this chapter, some approaches related with adaptation and reconfiguration in wireless communications are presented. In addition, MAC schemes most commonly used in WSNs are explained.

2.1 Component Oriented Design

Component oriented design consist of decomposing a system into independent units which can be developed and deployed independently providing basic functionalities of the application, as explained in [6] and [7]. The components are observed as black boxes with defined inputs and outputs and the end users only need to know these interfaces to be able to use them. This technique is becoming very popular to enable simple adaptation in software. Many component-based platforms exist, for instance COM/DCOM [8], .NET [9], EJB [10], TinyOS/nesC [11].

The use of this approach provides many advantages. First, as components are totally independent, they can be interconnected in many different ways to create different applications based on the same components. This feature is called dynamic composition and it can be shown in FIGURE 2.1, where different applications are designed by interconnecting the existing components like in LEGO. Furthermore, as the components are reusable, it allows to reduce the memory consumption of the final application.

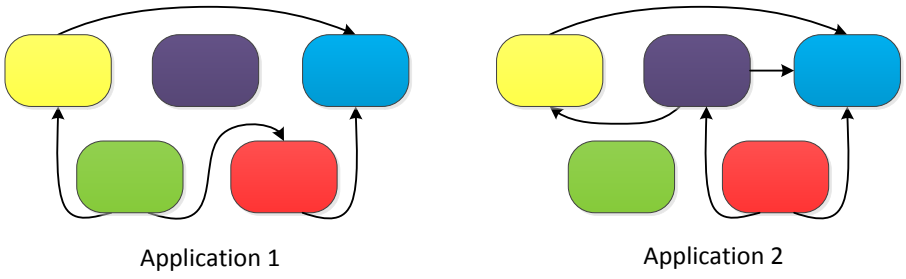

FIGURE 2.1: Different applications realized by a component-based design approach. [1]

Component oriented design has been used in order to achieve adaptive MAC protocols design through the *Decomposable MAC Framework* [12]. It has been demonstrated how MAC protocols are decomposed into independent elementary blocks based on

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

2.2 ADAPTABILITY

In dynamic environments, such as wireless networks or WSNs where medium conditions or network topology can change very frequently, MAC protocols require flexible adaptability in order to manage instability and unpredictability. There have been many approaches related with this topic.

A protocol called *Receiver-Based AutoRate Protocol (RBAR)* is presented in [13]. Its main goal is to allow modification of modulation schemes at runtime. It estimates channel quality, executes runtime adaptation mechanism in the receiver side and notifies transmitter to choose appropriate modulation schemes during RTS/CTS exchange. However, this approach is quite limited because it only allows parameter adaptation but it does not allow to change the functional behaviour of MAC protocols.

Hybrid MAC Protocols use different MAC protocols together as one MAC protocol. *Rate Adaptive Hybrid MAC Protocol (RAH-MAC)* [14] combines polling and contention MAC protocols in order to benefit from the advantages of each one of them.

Related to the idea of combining protocols, Meta-MAC [15] is a systematic and automatic method to dynamically combine any set of existing MAC protocols into a single upper layer. It achieves the performance of the best protocol without knowing in advance which of them will match the potentially changing and unpredictable network conditions. In addition, this optimization works without any centralized control or any exchange of messages, using only local network feedback information. Meta-MAC is introduced as a higher layer above existing MAC layer and its model compute the best decision of the protocol to be used. However, this technique has a large memory footprint, large implementation effort and redundancy in source code.

2.3 RUNTIME RECONFIGURATION

Classical method to modify a MAC protocol consists in changing source code, recompiling and redeploying to the specific platform. However, time is wasted when performing all this complex process. In order to improve it, some researches have proposed runtime reconfiguration approaches for MAC protocols which are currently running on the hardware platform.

Hermann S. Lichte and Stefan Valentin [16] show how to construct a compiler for the proposed language that generates most of the required implementation (model and MAC automaton) automatically. Basically, this approach pretends to define MAC pattern as an ordered sequence of frames assigned to roles. With this kind of patterns, MAC protocols can be easily described through a language called *MAC Pattern Description Language (MDPL)*. MDPL is very simple and intuitive for specifying MAC protocol behaviour which enables an efficient way of MAC protocol design and analysis. In addition, the developer only should care about the protocol design, the implementation is simplified by compiler. However, this solution is designed as domain-specific language which particularly concentrates on developing cooperative

relaying systems. It is not able to be used for those systems which cannot be expressed by patterns.

Guangwei Yang [17] developed a toolchain which is able to run any of the available MAC protocols for wireless networks using a language descriptor to describe the MAC protocol design which is processed by a meta-compiler. Moreover, as it is shown in FIGURE 2.2, this toolchain introduces few elements which allows runtime reconfigurations, such as the wiring engine and the command shell. The wiring engine is designed based on a linked list data structure in order to build MAC protocols by wiring functional blocks in a logic way. The other element, command shell, takes raw data from user input and interprets commands to decide which action has to be executed. User is able to load a certain MAC protocol in the same way as he can add some reconfigurations sending commands through this shell without the need of compile again. This toolchain was initially designed and implemented for WARP boards, but, after a time, it was ported to some sensor node platforms. The results of the performance tests carried out for WSNs were acceptable. However, there was a decrease of the toolchain efficiency in comparison with the WARPs results, due to the high overhead introduced.

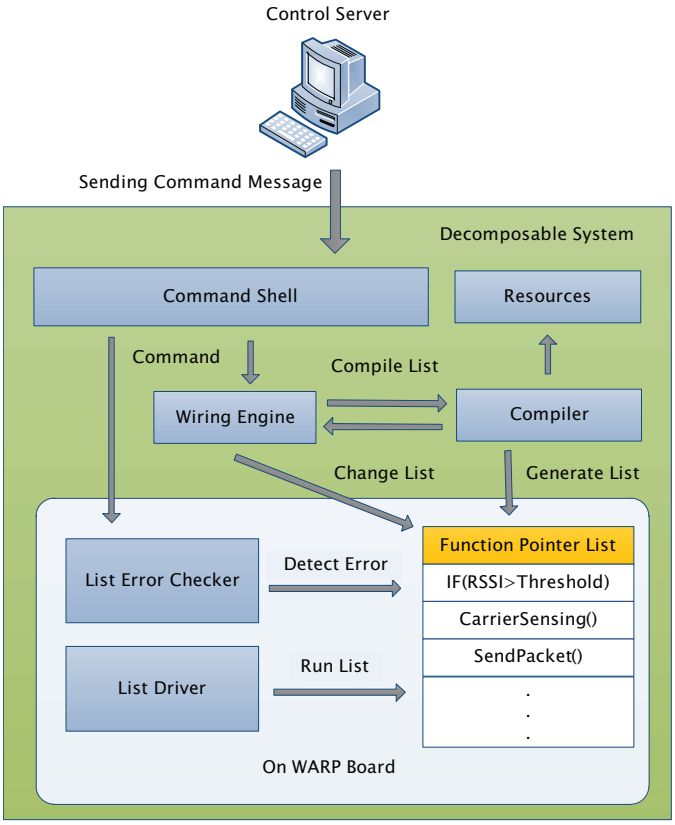

FIGURE 2.2: MAC runtime reconfiguration toolchain structure. [17]

Our toolchain will be based on the idea of the reconfiguration through a shell from Yang's work [17] and the decomposable MAC framework from Salikeen's work [18].

2.4 MAC SCHEMES FOR WSNS

The sensor nodes of WSNs work with batteries for a long time without human intervention. For that reason, as it is explained in [19], WSNs uses optimized medium access control protocols which their main design criteria is to extend the lifetime trying to keep the radio off when it is unnecessary to listen the medium. The three MAC schemes categories most commonly used in WSNs are the preamble-sampling, the common active periods and the hybrid protocols. Each one of them is explained in the next sections.

2.4.1 Preamble-sampling Protocols

Exists many protocols basic on preamble-sampling, such as B-MAC [20], MFP-MAC [21] or X-MAC [22]. These protocols try to save a lot of energy performing idle listening. The sensor node spends most of the time in sleep mode and wake ups periodically to check if there is a transmission on the channel. Therefore, a sender has to send a preamble long enough to assure that the transmission is detected by the the receiver. Moreover, preamble-sampling protocols use contention based approaches to listen the channel before transmitting the preamble in order to avoid collisions.

2.4.2 Common Active Periods Protocols

Some examples of common active periods protocols are S-MAC [23] and T-MAC [24]. The primary goal in these protocols is reducing energy, but they also achieve good scalability and collision avoidance by using a combined scheduling and contention scheme. As its name indicated, the sensors which use these protocols have common active/sleep periods. In active periods, sensors transmit data using RTS/CTS/-DATA/ACK handshake and in the sleep periods, the sensors save energy by keeping the radio in off state. As the periods are common, certain level of synchronization between all nodes is required. For that reason, during the active period, sensors send SYNC packets.

2.4.3 Hybrid Protocols

Hybrid protocols, like Z-MAC [25] or Funneling-MAC [26], combine two or more different MAC protocols to take advantage of their characteristics. They switch their behaviour depending on the network conditions allowing to achieve high performance. For instance, they use preamble-sampling based protocols when the number of nodes of the network is small and when this number increase, they change to common active periods based protocols.

3

SYSTEM DESIGN

In this chapter, the design of the toolchain for reconfigure MAC protocols is described. With our design, there is no need to modify a MAC protocol in a monolithic way of changing source code, recompiling and redeploying to the specific hardware platform. The developers can reconfigure protocols at runtime by simply sending commands through a shell.

The design of our toolchain can be divided in four parts: MAC Functional Blocks, MAC Meta-languages, Sensor Interface and User Interface. as explained in section 3.1, each MAC functional block represents a functionality of a MAC protocol, such as turn on/off the radio. Section 3.2 describes the Meta-language and the MetaNode-language; the Meta-language has been designed for the user and the MAC protocol implementation and the MetaNode-language is the language used by the sensor node. Thirdly, the Sensor Interface mechanism described in section 3.3 is designed for binding the MAC components together. This interface uses a MetaNode list to execute the MAC protocol implementation. The MetaNode list also allows the runtime reconfigurations by adding and removing nodes. Finally, we built a command shell for the user to load the MAC protocols implementations on the sensor node and do the runtime reconfigurations. In section 3.4, this shell is explained and also how it communicates with the sensor node in order to execute the user commands.

3.1 MAC FUNCTIONAL BLOCKS

The first step in the design on the toolchain is to identify the most common functionalities in MAC protocols. Then, following the component oriented design approach, a set of independent components which provides these functionalities can be designed.

As it is explained in [1], in order to identify these basic functionalities, some MAC protocols has been taken as example. For instance preamble-sampling protocols like B-MAC, MFP-MAC and X-MAC, and common active periods protocols like S-MAC and T-MAC. Moreover, the protocol IEEE 802.11 [27] which is based in CSMA/CA has been included, since it is commonly used in the area of wireless networks.

After studying each one of them, a set of basic and complex functionalities shown in TABLE 3.1 and TABLE 3.2 have been identified. Some examples of basic functionalities are *Send* and *Receive* which allow the nodes to communicate each other, or *Radio Control* which controls the radio power state. Complex functionalities are the ones that needs to call basic functionalities to be correctly executed, for instance, *Carrier Sensing* needs *Noise Floor Estimate*, *Radio Control* and *Timer* basic functionalities to determine if the medium is busy.

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. [1]

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. [1]

3.2 MAC META-LANGUAGES

Since our toolchain aims to provide an easy and efficient way to design and reconfigure MAC protocols and execute them in different platforms, we need a language to make this process easy to the user and another one that the sensor nodes are able to interpret. This toolchain will use the two languages explained in detail in [1].

The Meta-language is the one used by the user to design the MAC protocols. For that reason, this language is designed to be simple and follows the basic syntax of C or NesC. 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*.

The *MetaNode-language* is the one used by the sensor node to execute the functionalities described in the MAC protocol design. As this language is a translation of Meta-language used by the sensor to understand the instructions, it has the same basic operations mentioned in Meta-language. As it is explained before, our toolchain uses an Sensor Interface to execute the instructions allocated in an execution list. 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. Therefore, the size of the execution list will be the same as the number of lines of the file. The MetaNode structure is 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.

As it is explained in the implementation section of [1], 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. For that reason, the used implementation of MetaNode and MetaParam are as shown in FIGURE 3.1 and FIGURE 3.2.

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

FIGURE 3.1: Struct definition for *MetaNode*. [1]

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

FIGURE 3.2: Struct definition for *MetaParam*. [1]

3.3 SENSOR INTERFACE

In order to develop a toolchain able to reconfigure the MAC protocol execution process at runtime, some kind of dynamic structure is needed. This structure should store all the instructions that MAC protocol needs to execute, allow any changes that this protocol may need according to the situation and dynamically decides when a function or command must be executed.

Nevertheless, as this toolchain is oriented to resource constrained networks such as Wireless Sensor Networks, the algorithm must be designed according to the features that TinyOS as Operating System and NesC as its own programming language provided. We face to three important disadvantages:

- NesC does not allow dynamic memory allocation which is a problem for queue resizing.Thereby, any kind of structure similar to Linked List provided by other programming languages as C or C++ can not be used in TinyOS.
- In most C-like languages, callbacks have to be registered at run-time with a function pointer. NesC interfaces are wired statically at compile time, for that reason, the callbacks or events in TinyOS are very efficient. The compiler knows exactly what functions are called where and can optimize heavily, so no function pointer is provided.
- NesC is a component-based programming language and everything works through events. It implies that the code is not completely sequential and it is more difficult to control its execution progress. This is a problem because, for instance, if a user programs that *sendPacket()* must be called after doing *startRadio()*, the algorithm has to be able to relocate itself jumping through its code.

The design of our Sensor Interface is shown in FIGURE 3.3 and it has three components:

- *Execution List*: is a list which stores the MetaNodes of the MAC protocol design.
- *Variable List*: is a list to store the variables defined in the MAC protocol design.
- *MetaNode Compiler*: a compiler which will translate the MetaNode to the corresponding action that will be executed.

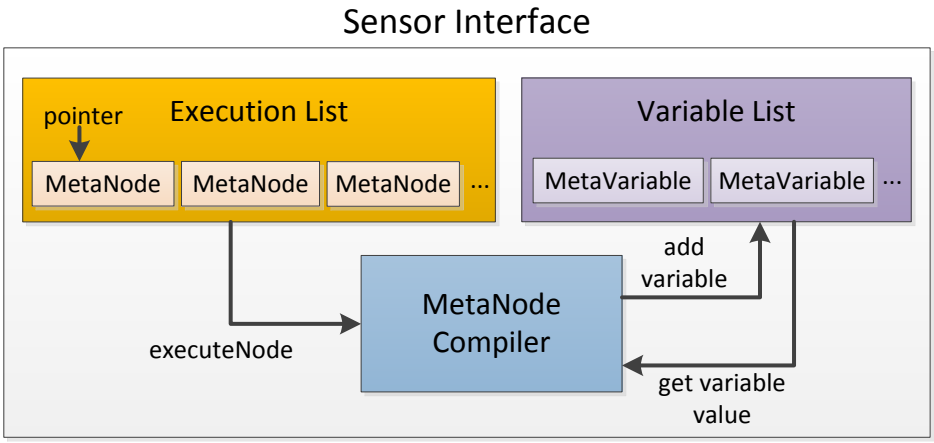

FIGURE 3.3: Sensor Interface design.

3.3.1 Execution List

According to previous related works of reconfigurable toolchains at runtime, many approaches used linked list structures to wire functional blocks or nodes in a logic way to create an execution flow. That is a smart idea because it allows to indicate how the execution will be and which component should be executed after that. One of the most important advantages of this strategy is that it provides support to those logical restrictions, such as a packet can not be send until the radio is started. Thereby, if we need to be sure that the radio is started, the node responsible to send the packet must be wired just after the node responsible of starting the radio.

In order to design the "hand-made linked list", it has been decided that a vector or array structure of MetaNodes will be used as substitute of the Linked List. As resizing is not possible, the size given to this structure will be selected during the implementation after studying the average number of MetaNodes used to implement most common MAC protocols and the RAM limitation of each platform.

Basically, Execution List must provide two basic functionalities:

- Transform an event-based environment to a sequential execution environment.
- Add or remove MetaNodes in order to make the toolchain adaptable at runtime.

Each of these functionalities determine the final design of the Execution List and will be explained in next subsections.

3.3.1.1 Events execution in a sequential way

The design of the MetaNode-language Events implementation has been done thinking how to solve the problem of the sequentiality [1]. The process of defining an event implementation implies using as minimum two MetaNodes: the first used to define which event is and, the second (tag END_EVENT) used to define that the event implementation is finished.

Using this strategy, sensor interface only needs to look for MetaNode event inside the execution list when the event is fired and place the Execution Pointer in its position. After that, it should execute each of the following MetaNodes and moving the pointer until it finds END_EVENT MetaNode. Returning to the previous example, if *sendPacket()* must be executed after *startRadio()*, the sensor interface only needs to find MetaNode with *StartRadioDone* as MetaLabel. As following MetaNodes, it should appear the *SendPacket* one, assuring that radio has been started previously.

Once a new event is fired, Sensor Interface must move the pointer to position where event MetaNode is located inside the list. This process allow the toolchain to jump inside the code freely and control the execution in a sequential way over a event-based environment as it is shown in FIGURE 3.4. This process is also detailed in Algorithm 3.1.

3.3.1.2 Adding and removing MetaNodes

Until the moment the design of the Execution List gives to the toolchain a static environment. However, it is very important to provide some dynamism to the system, allowing to modify the currently running MAC protocol.

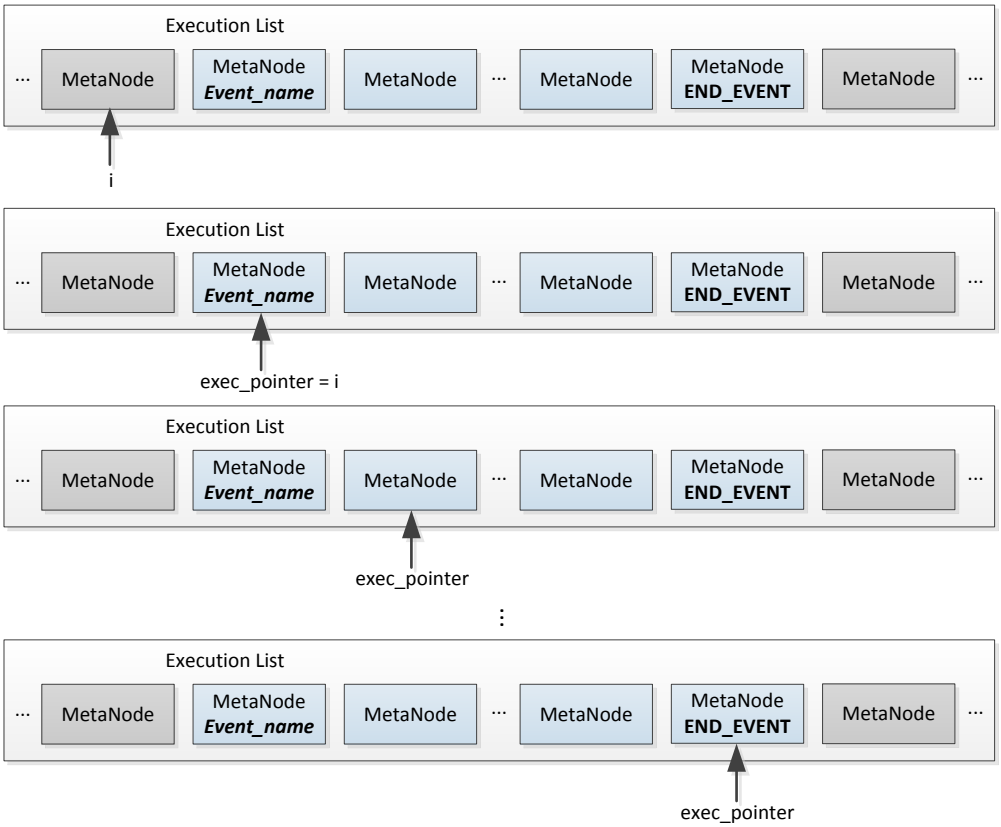

FIGURE 3.4: Procedure to execute sequentially an event implementation.

In order to achieve this, it is important to allow the list may grow or shrink depending on the instructions given by the user. As it has been explained before, dynamic memory allocation is not allowed by TinyOS, therefore, the list will consist of an array of MetaNodes with an static size enough big to implement most common MAC protocols and not collapse hardware of the sensor node.

However, the Execution List should allow to introduce new elements to its structure based on the user requirements. The selected strategy to do it is based on the displacement of MetaNodes inside the list. The position where the new MetaNode is going to be inserted has to be provided by the user. Once the desired location is known, from this position until the last occupied position of the list, all the implied MetaNodes are going to be displace one position to the right as FIGURE 3.5 shows. All the procedure can be simply translated into the Algorithm 3.2.

In a similar way, the list should allow to remove a certain MetaNode selected by the user based on displacement too. Once the desired location to be deleted is known, from the last occupied position of the list to the posterior position of the selected, all the implied MetaNodes are going to be displace one position to the left as FIGURE 3.6 shows. In the same manner as the case of adding a new MetaNode, all the procedure can be described using Algorithm 3.3.

Algorithm 3.1: Sequencial Execution

```
   Data: event_name, list
   Result:
1  i = 0, exec_pointer = 0;
2  while i < list.size do
3  |  if list[i].metaLabel == event_name then
4  |  |  exec_pointer = i;
5  |  |  break;
6  |  else
7  |  |  i++;
8  |  end
9  end
10 if i >= list.size then
11 |  event not found in execution list;
12 else
13 |  exec_pointer++;
14 |  while exec_pointer < list.size do
15 |  |  if list[exec_pointer].metaLabel == END_EVENT then
16 |  |  |  break;
17 |  |  else
18 |  |  |  executeNode(exec_pointer);
19 |  |  |  exec_pointer++;
20 |  |  end
21 |  end
22 end
```

3.3.2 *Variable List*

Implementing a MAC protocol usually involves using a lot of variables. We use an array to store all the variables used by the toolchain due to the lack of dynamic memory allocation.

The toolchain has 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.
- *Event attributes*: the ones that an event return when is triggered. These variables are used by the events to give some information to the user, for example if the medium is busy after execute carrier sensing. As the system variables.
- *Local variables*: the ones defined by the user inside an event implementation. The context of these variables is the event inside they are defined and they can only be used inside it. Two variables can have the same name only if they belong to two different events.

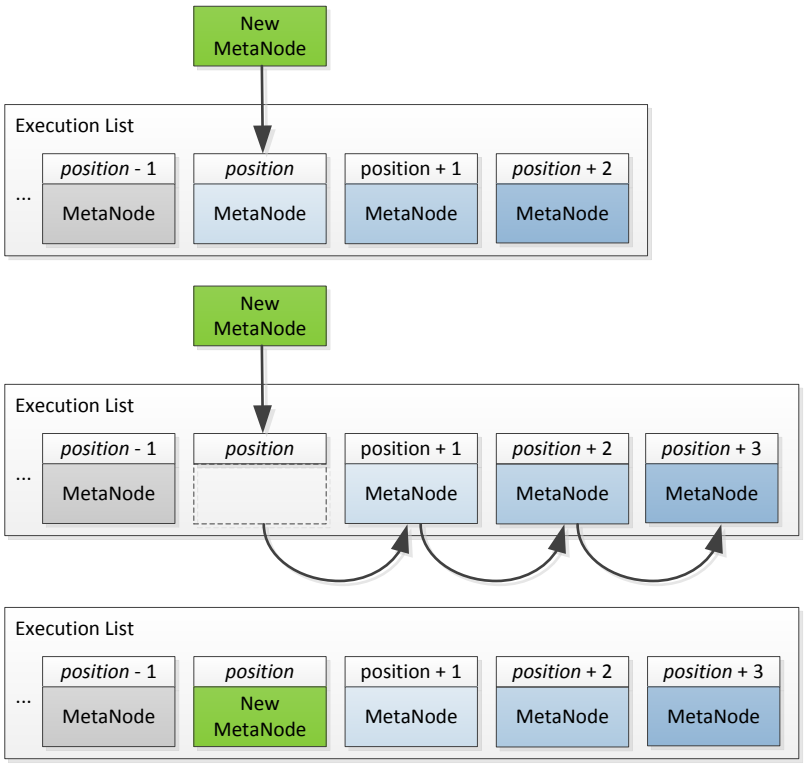

FIGURE 3.5: Procedure to add a new MetaNode to the Execution List.

- *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.

In order to represent a variable and store its value, it has been defined a new structure called *MetaVariable* which consists of:

- *Name*: a name for the specific variable which will allow to identify and make references to it during the execution of the code.
- *Value*: the currently value stored by the variable.
- *Context*: a value which identifies where the variable has been defined. If the context is DEF, it means that the variable has been previously defined by the system and can be used by the user but not modified (system variable); or it is a variable which can be used throughout the code (global variable). In other cases, context will contain the MetaLabel of the event where the variable has been defined and the use of this variable will be only valid inside the event of definition (local variable).

Therefore, the Variable List will consist of an array of MetaVariables with an specific size selected according to hardware limitations. Once a variable is defined by the

Algorithm 3.2: Add new MetaNode to Execution List

```
   Data: position, node, list
 1 i = occupancy − 1;
 2 if position + 1 > occupancy then
 3 |   trying to fill one position isolated from the rest of the list;
 4 |   break;
 5 else
 6 |   while i >= position do
 7 |   |   list[i + 1] = list[i];
 8 |   |   i − −;
 9 |   end
10 |   list[position] = node;
11 |   occupancy + +;
12 end
```

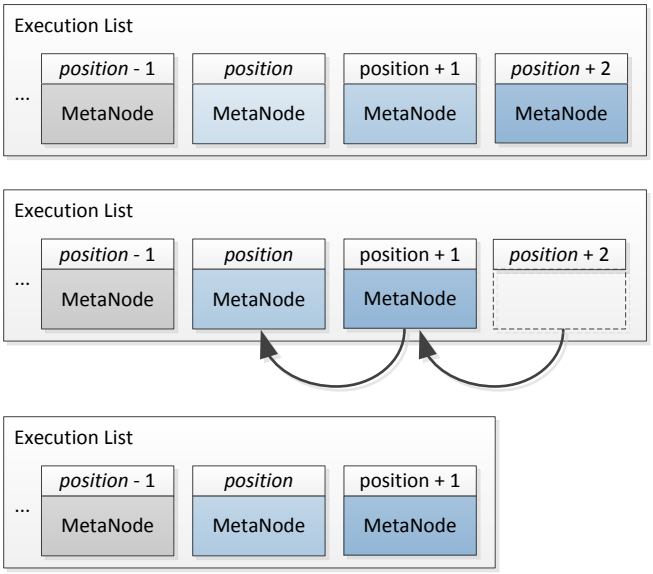

FIGURE 3.6: Procedure to remove a MetaNode from the Execution List.

user, a MetaVariable is created and inserted as the last element of the array. Every time a MetaNode which makes reference to a certain variable is read, it is only needed to look for the variable name inside the Execution List and verify that the context coincides. Verification of the context is very important because it is possible to have different variables with the same name but having different contexts.

3.3.3 MetaNode Compiler

MetaNode Compiler is the main block from the Sensor Interface design. Its primary purpose is to get MetaNodes from the Execution List, analyse them and execute the corresponding action.

Algorithm 3.3: Remove a MetaNode from Execution List

```
   Data: position, list
1  i = position;
2  while i < (occupancy − 1) do
3  |  list[i] = list[i + 1];
4  |  i + +;
5  end
6  occupancy − −;
```

As many MetaNodes exist, many actions can be taken by the MetaNode Compiler as listed below:

- *Variable definition*: it must create a new MetaVariable, taking the name from the MetaNode and value 0 due to it is a definition. As context, it will take the name of the event where it is defined or DEF if it is a general variable. After that, it will insert the MetaVariable inside the Variable List.
- *Expression*: it will get the variables implied from the Variable List and will modify their value according to the expression.
- *Function call*: it will translate the IdLabel from the MetaNode into a certain function of MAC Basic Components designed in 3.1 and execute it.
- *Event implementation*: it will move the Execution Pointer to the position where the event is located inside the Execution List using MetaLabel as identifier. After that, it will begin to execute one by one each of the elements contained by that event, until it finds a notification of event ending.
- *If-else structure*: it will separate the condition elements from the *if statement* and identify if they are variables or simple values. After that, it will check if the condition is true. In that case, it will execute all the elements following the *if* MetaNode until it finds an *else* or *endif* MetaNode. In case that the condition is false, it will move the pointer to next *else* MetaNode to execute the appropiate MetaNodes.
- *Label-goto structure*: in case of a *label definition* it will not do anything. However, in case of a *goto statement*, it will move backwards the Execution Pointer to the position where the label has been defined.

3.4 USER INTERFACE

One of the most important features of this thesis is the reconfigurability of the MAC protocol at runtime. A user interface is designed to read the user commands, interpret and send them to the sensor node through the serial port.

Mainly, the user interface will consist of three components which can be seen in FIGURE 3.7:

- *Command Shell*: a command-line shell through which the user sends commands to the sensor node and receives some feedback. In addition, it translates user commands into real actions, such as load and parse a new MAC protocol design, add or remove an instruction, etc.
- *Meta-language Parser* [1]: it translates MAC protocol instructions from Meta-language to MetaNodes. As this is a hard process and sensor nodes are resources constrained platforms with a very limited ROM and RAM, the user interface will include it.
- *Serial Controller*: responsible of the communication between the user/computer and the sensor node.

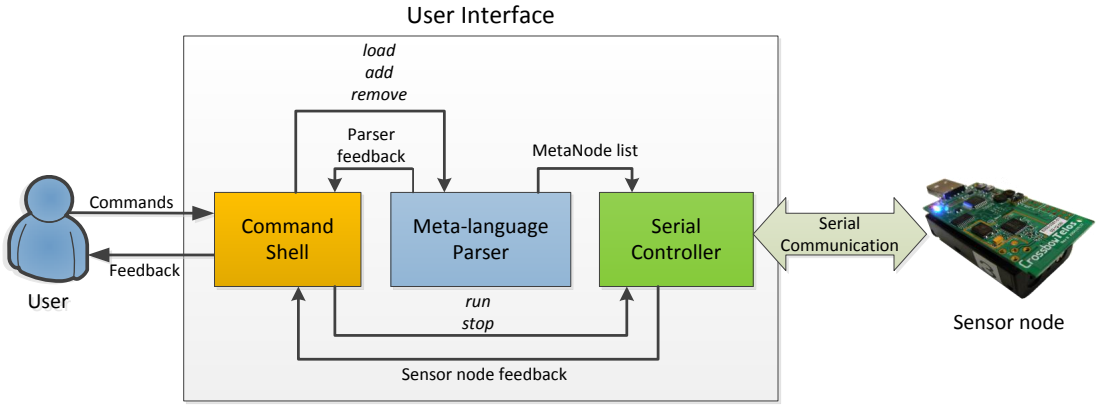

FIGURE 3.7: User Interface design.

3.4.1 *Command Shell*

Main functionality of Command Shell is to translate user commands into real actions. Therefore, it has been designed to understand functionalities according to the actions that can be performed by the user which are listed below:

- *Load new MAC protocol*: The user can load a text file which contains the instructions of the specific MAC protocol. If there are some errors, the user will be notified about that. Otherwise, MAC protocol will be loaded into the sensor node which will be ready to execute it.
- *Run a loaded MAC protocol*: After loading a certain MAC protocol, the user should be able to execute it.
- *Stop a running MAC protocol*: In the same way that a user can execute a MAC protocol, the user should be able to stop it.
- *Add or remove instructions to a loaded MAC protocol*: In some cases, the user will need to modify the currently running MAC protocol. For instance, the user

will need to modify the sleep interval from the *Low Power Listening* functionality without recompiling again the MAC protocol code. In order to be able to do that, *add* and *remove* functionalities must be provided by Command Shell. They allow to introduce or eliminate instruction lines from the preloaded MAC protocol text file.

- *Help*: In addition to the previous functionalities, a *Help* functionality is necessary to let the user know how Command Shell works and how the Meta-language syntax is.

3.4.2 Serial Controller

As a communication between the computer and the sensor node is needed, a component to control that communication should exist. This component will be the responsible of sending all the commands (load, run, stop, etc.) from the computer to the sensor node and transferring the MAC protocols instructions. In a similar manner, it will notify when a feedback from the sensor node is received.

4

IMPLEMENTATION

This chapter describes the implementation details of the reconfigurable toolchain. It can be divided in three parts: MAC Functional Blocks, Sensor Interface and User Interface. In section 4.1, implemented MAC Components and their relationships are described. Finally, the Sensor Interface implementation and the User Interface implementation are described in section 4.2 and 4.3 respectively.

4.1 MAC FUNCTIONAL BLOCKS

Our toolchain follows a component structure, starting with very simple and independent blocks which can be used to build more complex blocks quickly [1]. In this section, the MAC components implemented following the basic and complex functionalities of section 3.1 are presented.

In TABLE 4.1 a list of basic and complex components and their composition is defined. The basic components are the simplest ones built using only TinyOS components. The complex components are built combining existing components of the framework and TinyOS components.

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. [1]

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.

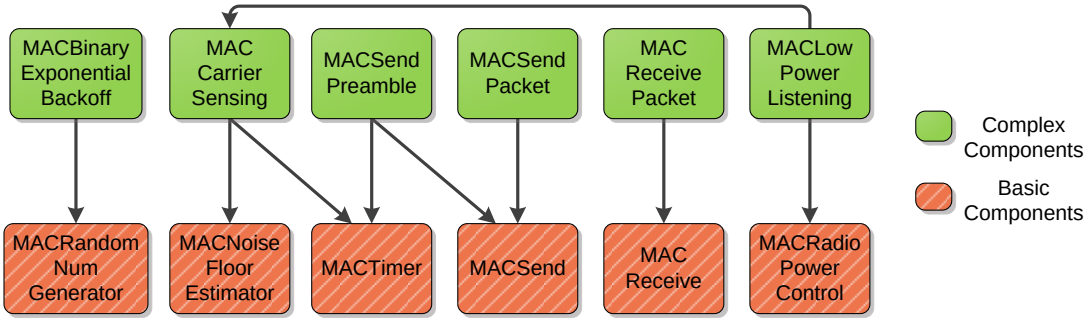

FIGURE 4.1: Diagram of reusable components. [1]

4.2 SENSOR INTERFACE

In this section we explain how the Sensor Interface has been implemented. It is written in NesC. As it is explained in section 3.3, the Sensor Interface consists of three components: the execution list, the variable list and the MetaNode compiler. This is the part of the project which introduces dynamism and allows reconfigurability at runtime by adding and removing nodes from the execution list. In the implementation, as it is shown in FIGURE 4.2, two more components are included:

- *ExecutionScheduler Component*: the execution scheduler will be implemented as a TinyOS independent component which will include and manage the other three components. Moreover, it will be built using all the MAC components making possible to call their commands and listen to their events. Finally, the component will have a TinyOS interface which will provide commands to execute the principal functionalities of the execution scheduler.
- *Sensor Node Serial Controller*: it is responsible for the communication between the computer and the sensor node. It reads the messages sent by the computer through the serial port and sends messages to the computer. This component will be the main one and will call the Execution Scheduler interface to execute the required functionalities.

4.2.1 Execution List

As it is explained in section 3.3, in NesC it is not possible to use dynamic memory allocation. For that reason, the execution list which allocates the MetaNodes of the MAC protocol is an array of MetaNode structures. Moreover, as resizing is not possible, the array has a predefined size. The size depends on the RAM of the platform; due to it is the most restrictive feature. In addition, we decide to do not select the size of the array depending on the MAC protocols sizes because a user can implement a new protocol which needs more instructions than an existing protocol.

For TelosB platforms which have 10 kB of RAM the size of the execution list will be of 300 MetaNodes. In case of Mica2 platform, the size is only 130 due to they have only 4 kB of RAM. In order to make our toolchain platform independent, we use the syntax

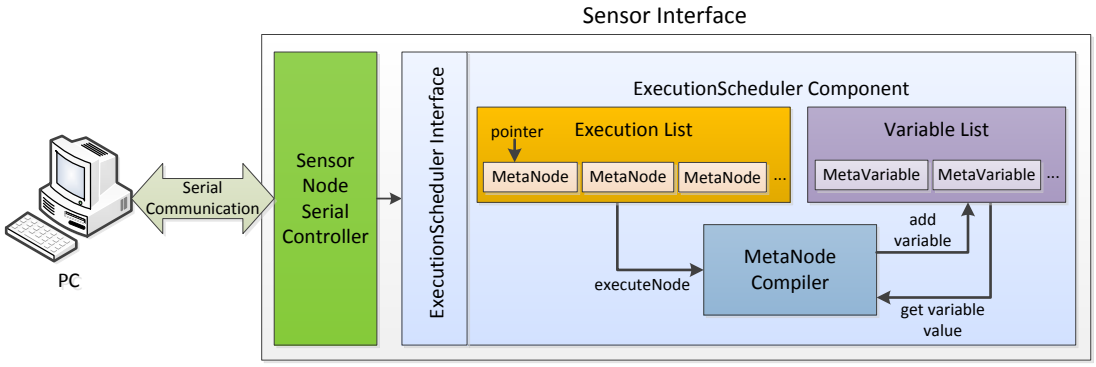

FIGURE 4.2: Sensor Interface implementation.

shown in FIGURE 4.3 which can be used in TinyOS. Thereby, we define the maximum size in a header list with the name *MAX_EXECBUFFERSIZE* and depending on the platform the proper value will be selected.

```
#if defined(PLATFORM_MICA2) || defined(PLATFORM_MICA2DOT)
  #define MAX_EXECBUFFERSIZE 130
#else
  #define MAX_EXECBUFFERSIZE 300
#endif
```

FIGURE 4.3: Definition of the maximum size of Execution list.

Therefore, Execution List consists of an array of MetaNodes called *execBuffer* which its size is specified by the constant *MAX_EXECBUFFERSIZE*. It is also an integer variable of type *uint16_t* called *execBufferSize* which will contain the actual occupation of the list. When a new node is added to the list, the variable is incremented by one unit and, in the same manner, when a node is removed, the variable will be decremented by one. Thereby, the number of nodes occupying the list is always known.

4.2.1.1 *Events execution in a sequential way*

In NesC language everything works through events which implies that the code is not completely sequential. The only sequential part of the code is the one that is executed inside an event. However, when an event is triggered, the execution jump from on event execution to another. As it is explained in sections above, our MetaNode-language has been designed to follow the same idea: in the execution list, there are events definitions delimited by two MetaNodes, the first indicates which event will be defined and the other indicates that the event implementation is finished. Therefore, when an event is triggered, the Sensor Interface has to look for the MetaNode of this event inside the execution list and execute the following MetaNodes until the end of the event implementation is found.

In order to do that, each time that an event is triggered the MetaNode which indicates the beginning of the event implementation is searched in the execution list.

Once the position is found, it is stored in a variable called *pointer* and incremented in one unit, then, a loop starts. In this loop, *pointer* is incremented each time and the MetaNode placed in this new position is executed. When the MetaNode with MetaLabel *END_EVENT* is found the loop is stopped. In section 4.2.3, it will be explained how the *pointer* is incremented, due to the units to increment it depends on the type of MetaNode executed. Each time that *pointer* is incremented, it is checked whether its value is lower than the actual list occupation. This process can be seen in FIGURE 4.4.

The variable *context* indicates the event that is running and *attributesList* is an array of MetaVariables. In this array are stored attributes that the event may return and the user is able to check. In order to use the attributes, they are included in the array when the event is triggered. The MetaVariable structure and its characteristics will be explained in the next section. FIGURE 4.5 shows an example of how the attributes are included in the array. The identifier of each variable is described in [1].

Boot event is a special case, because the existence of global variables definition must be checked. As it explained before, these variables are defined in the first lines of the MAC protocol, thus the corresponding MetaNodes are placed in the first positions of the execution list outside any event implementation. For that reason, before executing the Boot event we include the process in FIGURE 4.6 in order to execute these MetaNodes and define the variables before executing the rest of the code. In the loop, we execute all the MetaNodes placed in the first positions until a MetaNode with a MetaLabel different of DEF is found, because it means that the following MetaNodes is an event implementation.

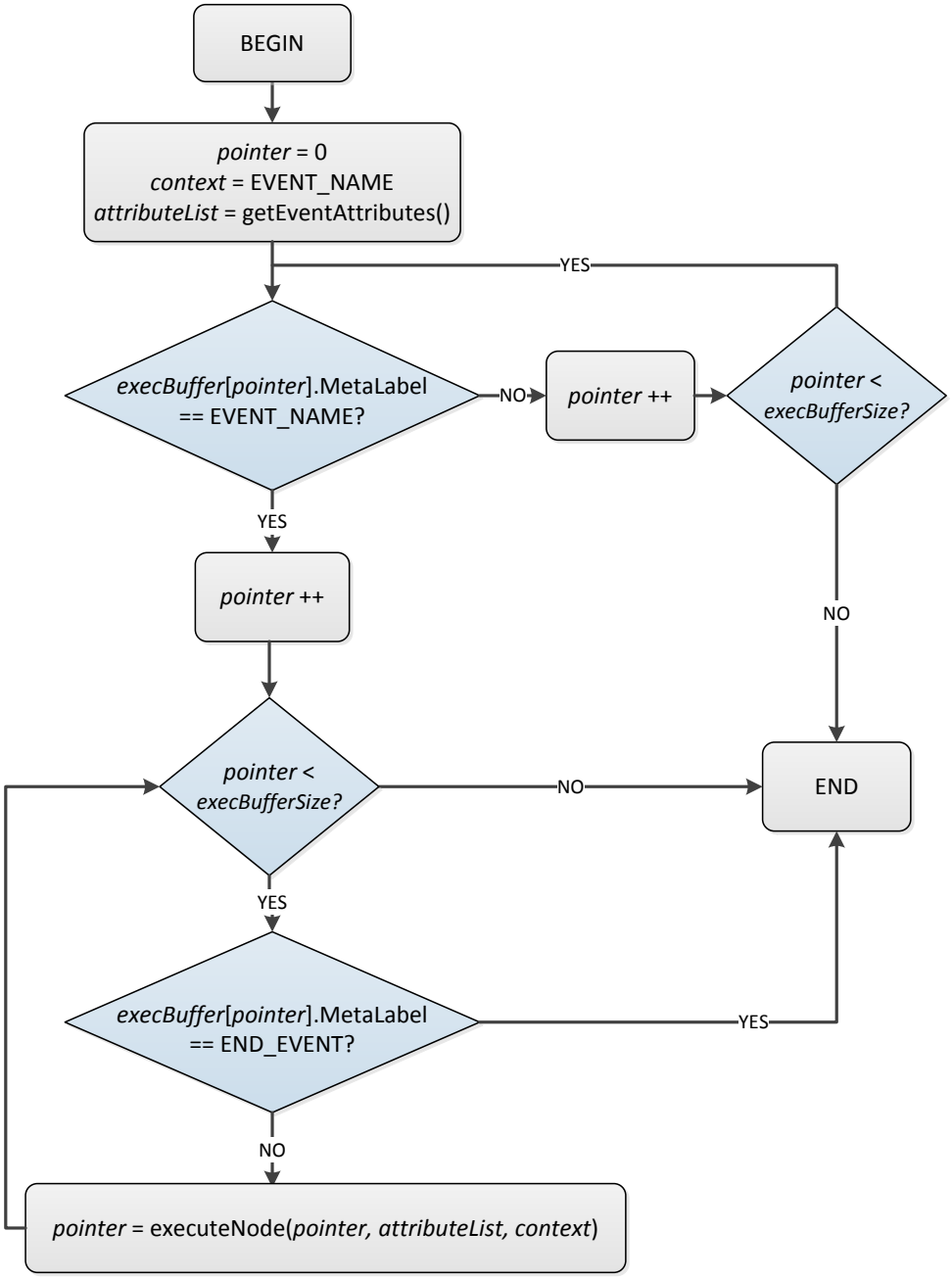

FIGURE 4.4: Execution of an event.

4.2.1.2 *Adding and removing MetaNodes*

As it is commented before, in order to provide some dynamism to the system, there is an array of MetaNodes with a static size enough big to implement most common MAC protocols and to not collapse hardware of the sensor node. This array allows introducing new elements in the first empty position or between already occupied positions.

```
event void MACCarrierSensing.firedCS(bool mediumBusy){
    MetaVariable att[3];
    MetaVariable mediumBusyVar;
    mediumBusyVar.name = 7;
    mediumBusyVar.value = mediumBusy;
    att[0] = mediumBusyVar;

    ...
}
```

FIGURE 4.5: Example of storage of the attributes of an event.

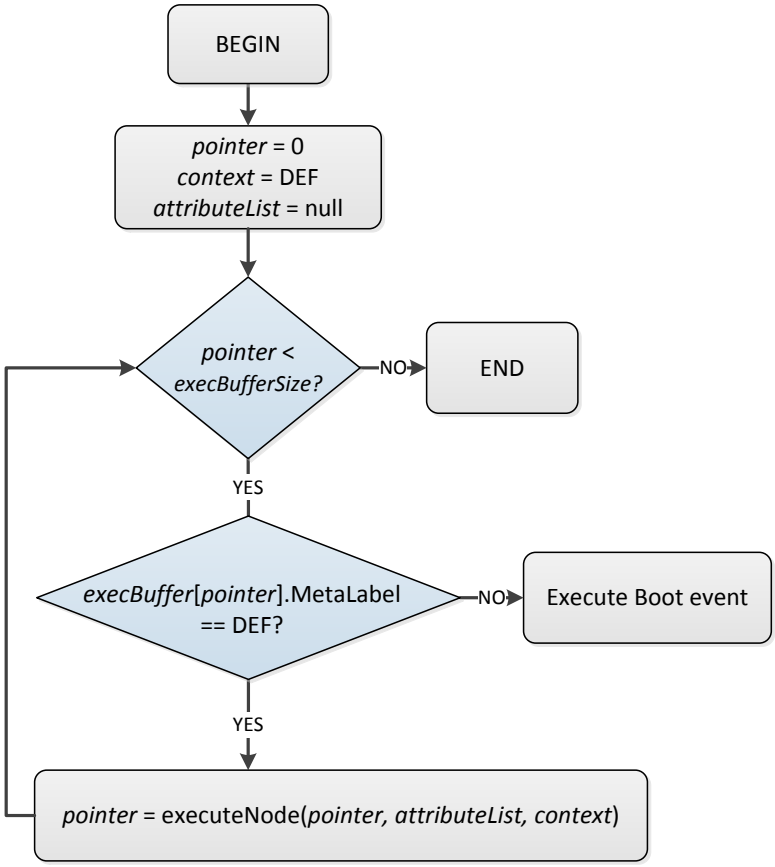

FIGURE 4.6: Execution of global variable definitions.

Add new elements in the first empty position is easy to implement. As it is shown in FIGURE 4.7, the new MetaNode will be placed next to the last occupied position of the list. This position is stored in the variable *execBufferSize*. Nevertheless, no more new nodes cannot be placed in case the list is already full.

FIGURE 4.8 shows how to add a new element in a specified position between already occupied ones. In this case, the element in the required position and the following ones until the last occupied node will be displaced one position in order to

```
execBuffer[execBufferSize] = new_node;
execBufferSize ++;
```

FIGURE 4.7: Addition of a node next to last occupied position of the list.

leave this position empty and be able to place the new MetaNode. As in the previous case, we can not place new nodes if the list is already full.

```
if(position+1 <= execBufferSize){
  uint16_t i;
  for (i = realExecBufferSize-1; i >= position; i--) {
    execBuffer[i+1] = execBuffer[i];
  }
  execBuffer[position] = new_node;
  execBufferSize ++;
}
```

FIGURE 4.8: Addition of a node in a specified position of the list.

Moreover, the array allows removing elements from the list. In order to do that, the position of the element which have to be removed is specified. Then, as it is shown in FIGURE 4.9, from the node placed next to the position until the last occupied position of the list, all the nodes are moved one position to the left. Thereby, the node in the position specified is overwritten and the occupation of the list is decremented by one unit.

```
uint16_t i;
for (i = position; i < execBufferSize-1; i++) {
  execBuffer[i] = execBuffer[i+1];
}
execBufferSize--;
```

FIGURE 4.9: Removal of a node in a specified position of the list.

4.2.2 Variable List

In the MAC protocol implementation, it is possible to define general and context variables and change their values. For that reason, the Sensor Interface needs some structure to store them and access to them when it needs. As with the execution list, an array with a predefined size will be used. In this case, an array of MetaVariable structures and a size of 50. FIGURE 4.10 shows MetaVariable structure which has three fields: name, value and context.

A variable of type *uint8_t* called *variablesBufferSize* will contain the actual occupation of the list of variables. When a variable is defined, it is inserted next to the last occupied position of the array and the occupation is incremented by one unit. In the same manner, when a variable is removed, the following positions are displaced one position to the left and the occupation is decreased one unit.

```
typedef struct MetaVariable{
  int16_t name;
  int16_t value;
  uint8_t context;
}MetaVariable;
```

FIGURE 4.10: MetaVariable structure.

Every time a MetaNode which makes reference to a certain variable is executed, MetaVariable position inside the variable list is searched. In order to do that, its name and its context are verified as it is shown in FIGURE 4.11. Its name should coincide with the name given by the MetaNode and its context should coincide with the actual context (the event that is being executed) or it should be DEF, which means that the variable is general and can be used in all the events. The context verification is very important, due to it is possible to have different variables with the same name but in different contexts.

```
uint8_t getVariablePosition(int16_t name, uint8_t context){
  uint8_t pos;
  bool found = FALSE;
  for (pos = 0; pos < variablesBufferSize; pos++) {
    if(variablesBuffer[pos].context == DEF ||
       variablesBuffer[pos].context == actual_context){
      if(variablesBuffer[pos].name == name){
        found = TRUE;
        break;
      }
    }
  }
  return pos;
}
```

FIGURE 4.11: Search for a variable position inside Variable list.

This list will include three types of variables:

- *System variables*: they are those defined previously by the toolchain. Their context is DEF, they can be used in all the code and they are included in the list when the sensor node starts to run the MAC protocol. These variables are used to help the user and provide some information, i.e. the *node_id* of the sensor node.
- *Local variables*: they are those defined inside an event implementation. The context of these variables is the event inside they are defined and they can only be used inside it. In order to not waste space in the list, these variables are removed when the event execution is finished. In order to do that, the code in FIGURE 4.12 is used to remove only the variables with the context specified.
- *Global variables*: they are those defined by the user at the top of the code, outside the events. The context value of these variables is DEF. They cannot be removed from the list, because they can be used in all the events.

```
void deleteLocalVariables(uint8_t context){
  uint8_t pos;
  uint8_t i;
  for (pos = 0; pos < variablesBufferSize; pos++) {
    if(variablesBuffer[pos].context == context){
      for (i = pos; i < variablesBufferSize-1; i++) {
        variablesBuffer[pos] = variablesBuffer[pos+1];
      }
      variablesBufferSize--;
      pos--;
    }
  }
}
```

FIGURE 4.12: Removal of local variables when an event execution is finished.

4.2.3 MetaNode Compiler

As it is mentioned above, NesC language does not provide function pointers. For that reason, in the Sensor Interface a compiler is needed to translate the MetaNodes into real functions calls. This compiler will also define new variables, change their values and interpret if-else and label-goto structures.

When an event is triggered, the pointer is placed to the position where the event is located, the actual context is changed and its attributes are stored in an attribute list. After that, the pointer is incremented by one unit and all the following elements are executed until the MetaNode with MetaLabel END_EVENT is found. For each MetaNode that has to be executed, first of all, the compiler chooses which action it has to be performed depending on the MetaLabel using a switch condition. Below it is explained the actions carried out for each one of them, except for events and end of event implementation:

- *DEF*: It is a variable definition. A new variable with specified name, value 0 and the actual context is added to the variable list. Finally, the pointer is incremented by one unit and returned.

- *EXP*: It is an expression. First of all, the position of the variable which its value has to be changed is searched. Then, the second parameter is evaluated following the scheme shown in FIGURE 4.13: it can be a numerical value, another variable or a function call. In the first case, the value is taken directly from the MetaNode. In the second case, the variable is searched in the attributes list of the event. In case of existing as attribute, its value is taken. Otherwise, the position of this variable is searched in the variable list and its value is taken from there. In the last case, the function is executed like a MetaNode FUNC with the parameters indicated and the returned value is taken. Once variable position is known as well as the value of the second parameter of the expression, the arithmetical operation is performed over the mentioned variable depending of IdLabel field. Finally, the pointer is increment one unit and returned.

- *FUNC*: It is a function call. Depending on the IdLabel, the corresponding com-

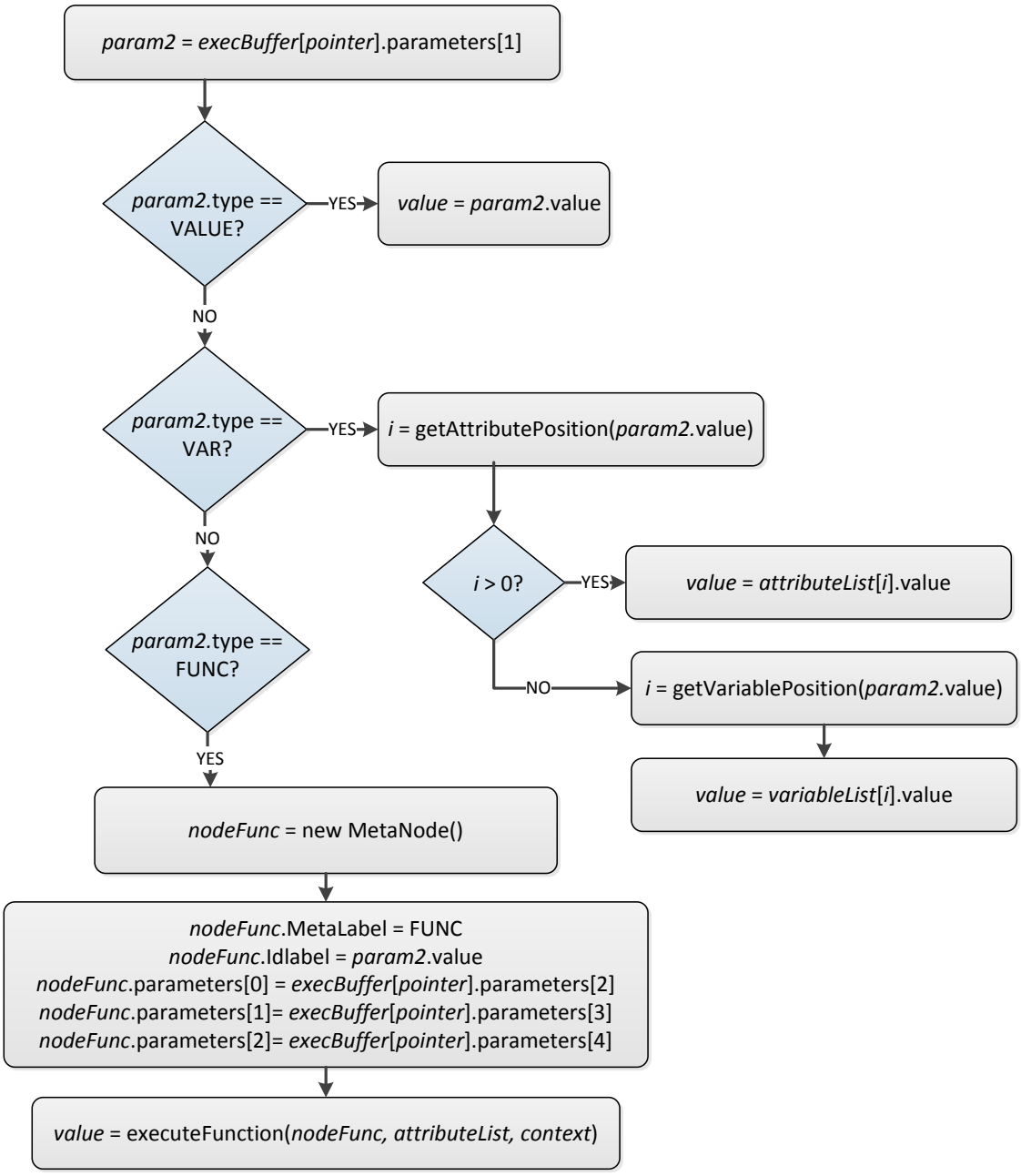

FIGURE 4.13: Get the value of a parameter flowchart.

mand call is selected using a switch condition. Before executing the function, if it needs parameters, they are taken from the MetaNode. They can be numerical values or variables. As in the expression, if the parameter is a number, its value is taken directly from the MetaNode. Otherwise, the position of the variable is searched in the attributes list of the event and in the variable list and its value is taken from where it is found. Once all the parameters are found and recognized, it is executed. In case of having a value to be returned, it is returned to the

compiler. If the function is called in an expression, this value is the one that used inside the expression. Otherwise, the value returned is ignored, the pointer is incremented by one unit and returned while the compiler continues executing the rest of the nodes.

- *IF*: It indicates the start of an if-else structure. First of all the two values to be compared are taken. As in function calls, they can be numerical values or variables. Once the compiler has the two values, the condition is taken from the MetaNode and it is checked.
  If the condition is true, the pointer is incremented by one unit and a loop starts. Inside this loop, if the node placed in the pointer position has the MetaLabel ELSE or END_IF and the same IdLabel, the loop is stopped. Otherwise, the node is executed, the pointer gets the returned value and the loop continues. In case END_IF is found, the loop stops and the pointer is incremented by one unit and returned. Otherwise, if ELSE is found, the loops stops and starts another one which increments the pointer by one unit until END_IF with the same IdLabel is found. As in the other case, when END_IF is found the pointer is incremented by one unit and returned.
  If the condition is false, another loop starts. In this case, the pointer is incremented by one unit until ELSE of END_IF with the same IdLabel is found. If the END_IF is found the pointer is incremented by one unit and returned. Otherwise, the pointer is directly returned in order to the compiler reads the ELSE node and executes it.

- *ELSE*: If the compiler found a MetaNode ELSE, it means that the condition checked in the MetaNode IF has been false. The pointer is incremented by one unit and a loop starts. Inside the loop, the node located in the position which the pointer indicates is executed. As in MetaNode IF, the pointer is incremented depending on the MetaNode executed and the node located in this new position is executed, and so on, until a MetaNode with MetaLabel END_IF and the same IdLabel is found. Then, the loop ends and the pointer is incremented by one unit and returned.

- *END_IF*: As it is explained in the previous cases, this MetaNode is looked for in the loop of the MetaNode IF or ELSE. It marks the final of an if-else structure but cannot be executed in the compiler due to the pointer is increment in one unit after it is found.

- *LABEL*: It is a label definition. The Sensor Interface does not have to do any action. In this case, only the pointer is incremented by one unit and returned.

- *GOTO*: It is a goto statement. A MetaNode with LABEL tag as MetaLabel and with the same IdLabel is searched in the same context. If it is not found, the pointer is incremented by one unit. Otherwise, the pointer is moved to the next position after the MetaNode LABEL and returned.

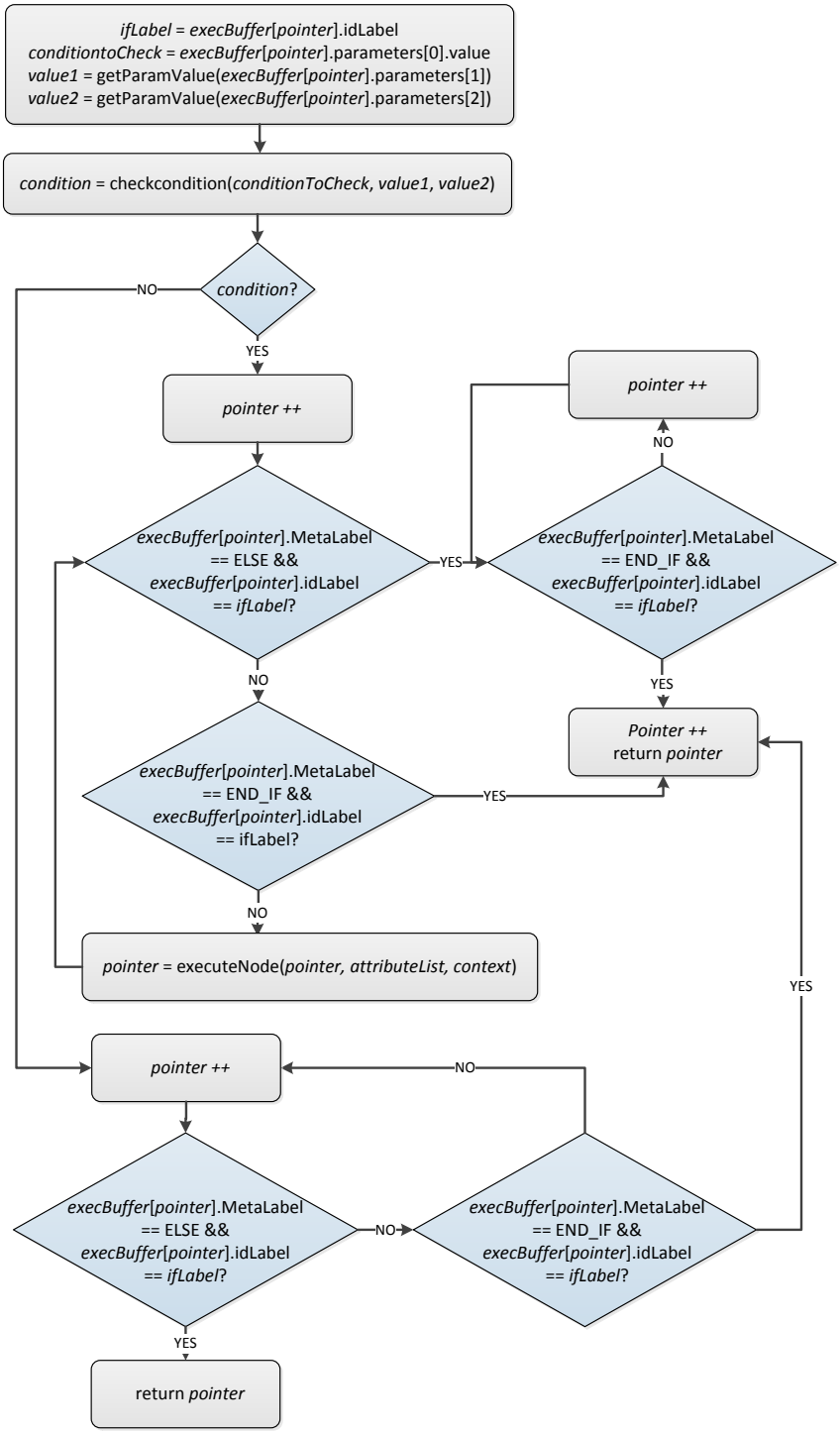

FIGURE 4.14: Execution of an *if* MetaNode.

4.2.4 *ExecutionScheduler Component*

The execution list, the variable list and the MetaNode compiler are implemented inside a TinyOS component called ExecutionScheduler. Thereby, this component will allow to add or remove a node, run or stop the actual execution list. An interface for this component must be created in order to make public these functionalities.

The implemented interface is shown in FIGURE 4.15 and it provides seven commands. The command *executeList()* puts the execution pointer in the first position of the execution list and starts the execution of MetaNodes included in the list. The next command is *stopExecution()*, it stops the execution of the MAC protocol. Regarding to adding a node, two commands are provided. The first one is *loadNode(node)* is used when a new MetaNode list is loaded, thus each new MetaNode is added in the last free position of the execution list. The other one is *addNode(node, position)*, as it is can be observed, when this command is used the position should be provided and this MetaNode is added in this certain position to the list. The command *removeNode(position)* allows to remove the MetaNode which corresponds to the position of the list. The command *getListSize()* returns the actual occupation of the execution list. Finally, the command *initListSize()* changes the value of the occupation of the list to 0, thus it provokes a behaviour very similar as when the list is empty.

```
interface ExecutionScheduler{
    command void executeList();
    command void stopExecution();
    command void loadNode(MetaNode* node);
    command void addNode(MetaNode* node, uint16_t position);
    command void removeNode(uint16_t position);
    command uint16_t getListSize();
    command void initListSize();
}
```

FIGURE 4.15: Interface of ExecutionScheduler component.

This component is built using all the MAC Components implemented and explained in section 4.1, as well as it is included the TinyOS component LedsC to manage the leds behaviour. This will allow the MetaNode compiler to listen to their events and call their commands. There is a special case; in the implementation are instantiated 9 MACTimer components in order to give the user the possibility to create and use many timers. Thereby, user will not be limited to use only one timer in the MAC protocol design.

4.2.5 *Sensor Node Serial Controller*

As it is explained in section 3.4.2, the user will need a communication with the sensor node in order to interact with it. This communication will be performed through a computer connected to the sensor node via USB or serial cable. In the sensor node side, a TinyOS component called SerialActiveMessageC is used to manage this communication. It provides the interfaces Receive and AMSend in order to read and write data in the serial port.

The main component of the toolchain in the sensor node side is the one built using SerialActiveMessageC and it is the responsible to receive user commands through the computer. In order to be able to execute the desired task, i.e. load a new MAC protocol or run it, this component should use the ExecutionScheduler interface. Thereby, the main component is built using the SerialActiveMessageC and the ExecutionScheduler components.

As it will be explained in section 4.3.3, the sensor node will receive specific commands for each action that user wants to do. In that moment, Receive interface will trigger an event with the message. The node will check for errors, send a feedback and execute the required task. There are five cases and they are explained below:

- *Load*: The sensor node checks that the number of lines to load is lower than the maximum number of lines allowed. As it explained before, for TelosB platforms the execution list size is 300, but for Mica2 it is only 130. If the number of lines is greater, the sensor node will send a feedback error to the computer. Otherwise, it will initialize the lists of the Execution Scheduler component using the command *initListSize()* and it will listen for the next messages which contain the MetaNode list of the MAC protocol. For each MetaNode it will call the command *loadNode(node)* to add it to the execution list.

- *Add new node*: In this case, first of all, the sensor node will check if the execution list is already full. If it is full, it will return an error. Otherwise, it will check the line when the new node should be placed. If this number is greater than the actual occupation of the list which can be consulted using the command *getListSize()*, it will return an error. This means that no empty positions can exist between two consecutive MetaNodes in the execution list. Otherwise, the sensor node will place the new MetaNode in the specified position using the command *addNode(node, position)*.

- *Remove node*: The sensor node checks if the specified position to remove is greater or lower than the actual occupation of the list. If the number is greater, it means that this node is already removed and the sensor node will return an error. Otherwise, this node will be removed using the command *removeNode(position)*.

- *Run*: The sensor node checks if the list is empty, in affirmative case, it will return an error. Otherwise, it starts to execute the MAC protocol loaded in the execution list using the command *executeList()*.

- *Stop*: The sensor node stops the execution of the whole execution list using the command *stopExecution()*. In this case, if the execution is already stopped, it does not return an error. The sensor node always sends a feedback message to report that the execution is stopped.

4.3 USER INTERFACE

One of the most important features of this thesis is that user is able to interact with the sensor node and change its behaviour at runtime. In order to achieve it, a user interface is necessary to communicate each other. This section will describe the implementation of the design described in section 3.4.

All the implementation of the user interface has been programmed in Java for many reasons. First, TinyOS provides native libraries for different programming languages like Java, C or Python; giving access to TinyOS serial stack in order to be able to interact with the different platforms. Moreover, Java is multiplatform which means that any Java application can run on any computer architecture or operative system if Java Virtual Machine (JVM) is installed. Therefore, the user interface which is going to be implemented will work in Windows, Linux or MacOS.

4.3.1 Command Shell

As it has been previously mentioned in 3.4.1, Command Shell is designed to be able to provide 6 basic functionalities: load a new MAC protocol, run a loaded MAC protocol, stop the execution, add new instructions to the running MAC protocol, remove instructions from the running MAC protocol and provide some help to the user.

First of all, the user will input commands in text format, thus these commands should be parsed, identify properly and execute its action. In order to manage it, a simple enumeration of constants has been created as FIGURE 4.16 shows. Each of these constants makes reference to a certain command, except for the last three which are arguments of the HELP command.

```
public final static int LOAD = 0,
                        ADD = 1,
                        REMOVE = 2,
                        RUN = 3,
                        STOP = 4,
                        HELP = 5,
                        EXIT = 6,
                        SYNTAX = 7,
                        FUNCTION = 8,
                        CONSTANT = 9;
```

FIGURE 4.16: Definition of Commands enumeration.

In order to recognize a command properly independently if user inputs the commands in lowercase or uppercase, the code in FIGURE 4.17 shows how a command can be recognized and translated to a numeric value. First, the command is capitalized and compared one by one with the name of the different elements of the commands enumeration defined in FIGURE 4.16. If it matches, the value of this field is stored in *commandType* which will be used to decided which action should be taken. In addition, a boolean variable called *commandRecognized* is set to true to indicate that the command has been finally recognized. If this variable is false, an error is indicated to the user through the shell.

Once the command is translated from a string to a numeric value, through a switch condition it will be examined. Depending on the case, different actions can be carried out as FIGURE 4.18 shows.

- *LOAD*: This command is used to load a file containing a MAC protocol description. It must follow the next structure:

```
for(java.lang.reflect.Field f : fields){
  if(f.getName().equals(command.toUpperCase())){
    commandType = f.getInt(Shell.class);
    commandRecognized = true;
  }
}
```

FIGURE 4.17: Recognition of a command.

```
>> LOAD [filename]
```

After entering LOAD, a valid file path must appear as parameter of the command in order to open and read it. Once the number of parameters and its validity have been checked, the Parser is called with file path as parameter and it will return a list of MetaNodes after processing the file. After that, the list of MetaNodes are sent to the sensor node using the SerialController.

- *ADD*: This command allows to add a new line or instruction to the previously loaded MAC protocol. It must follow the next structure:

  ```
  >> ADD [numLine] [func/exp/def/if-else/label&goto]
  ```

  As first parameter must appear the line number where the instruction is going to be added. The second parameter must be the instruction to insert into the MAC protocol following Meta-language syntax explained in [1]. It can be a function call, a expression, a definition of a variable or an if-else or label-goto statement. The instruction is processed by the Parser and it returns a certain MetaNode which is stored in a general list of MetaNodes called *nodeList* and in a list of changes called *modifiedNodeList*. *NodeList* is the list where all MetaNodes are stored after parsing the text file. Thereby, the MetaNode is inserted in the position indicated as *numLine* parameter. On the other hand, it is inserted at the end of the *modifiedNodeList* indicating that the change to be made is ADD.

  If the MetaNode returned after parsing the instruction of ADD command is an IF, ELSE, ENDIF, EVENT or END_EVENT; a boolean variable called *reload* is set to true. This step is very important as it determines when a big change is introduced into the MAC protocol. The main difference between a big and a little change is the instruction inserted. In case of instructions like mentioned before includes other instructions inside them. For that reason, all the code should be checked after introducing all the changes to check if there is no new errors, e.g. if each curly bracket is closed. It implies to resend all the MetaNodes again i.e., it is necessary to make a LOAD of the *nodeList*. On the other hand, the other instructions must be checked individually and there is no need to check the rest of the code. In that case, it is not necessary to send *nodeList* which is quite big, but *modifiedNodeList* which its size is smaller must be sent .

  In any case, all the instructions used in ADD command are inserted in the text file previously loaded.

- *REMOVE*: This command allows to remove a line or instruction from the previously loaded MAC protocol. It must follow the next structure:

```
>> REMOVE [numLine]
```

  As first parameter must appear the line number of the instruction is going to be removed. In the same manner as ADD command, it is important to distinguish between big or little changes introduced. If the MetaNode to be removed is an IF, ELSE, ENDIF, EVENT, END_EVENT, LABEL or DEF; a boolean variable called *reload* is set to true. As a difference with ADD command, LABEL and DEF are taken into account when distinguishing a big change, because removing one of these MetaNodes generates broken references, e.g. expressions using variables which are not defined anymore. Therefore, all the code must be checked before running again. Regarding to the lists of MetaNodes, the MetaNode in the position determined by the *numLine* parameter is removed from *nodeList*. Moreover, the MetaNode will be inserted in *modifiedNodeList* but, in this case, indicating that the action to be taken is REMOVE.

  In any case, all the instructions used in REMOVE command are removed from the text file previously loaded.

- *RUN*: This command is used to initiate the execution of the previously loaded MAC protocol. It must follow the next structure:

```
>> RUN
```

  As a difference to other commands, it does not need any parameter. Before sending the order of running the loaded code, *nodeList* is checked to find some error. After that *reload* variable is checked. In case of *reload* value is true, all the *nodeList* will be sent again. Otherwise, *modifiedNodeList* will be sent instead. Using this approach, it allows us to improve the efficiency of modifying a MAC protocol due to when just an little change is introduced *modifiedNodeList* which is smaller than *nodeList* is sent. Finally, the order of running the code is sent to the sensor node, *reload* variable is set to false again and *modifiedNodeList* is emptied.

- STOP: This command is used to stop the execution of the running MAC protocol. It must follow the next structure:

```
>> STOP
```

  In the same manner as RUN command, it does not need any parameter. After recognizing the command, it is sent directly to the sensor node.

- HELP: This command is used to visualize some helpful information that user may need , e.g. information about the previously explained commands or about Meta-language syntax. It must follow the next structure:

```
>> HELP [load/add/remove/run/stop/syntax/function/variables]
```

As first parameter must appear one of the options shown above. First five options are used to show information about the previously explained commands. Last three commands five information about: syntax of Meta-language, functions available to be used in Meta-language and default variables or attributes provided by the toolchain. This command does not interact with the sensor node.

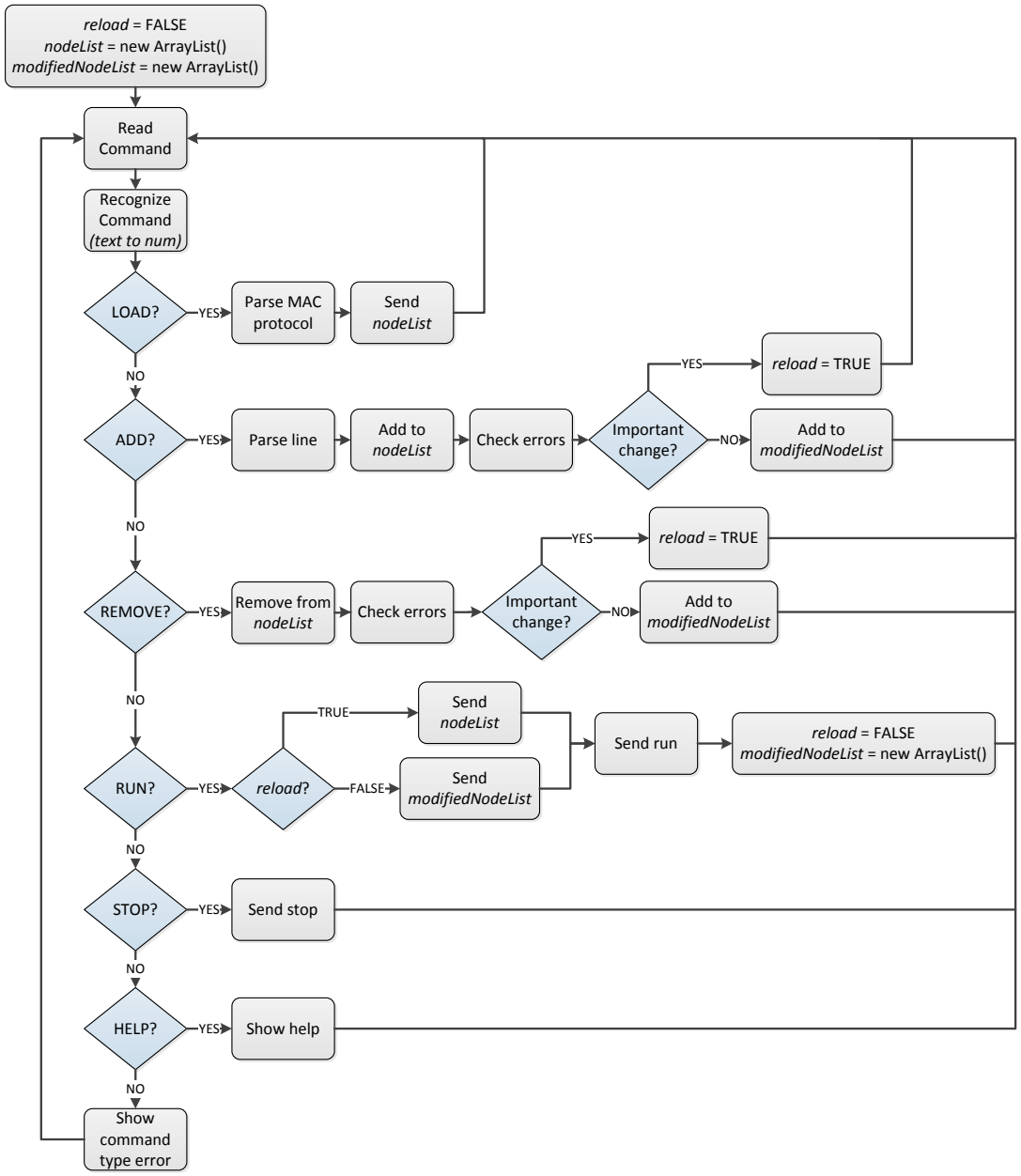

FIGURE 4.18: Command Shell flowchart.

4.3.2 Parser

One of the most important components of the User Interface is the Parser which is the responsible of reading text files written using Meta-language and translate them into MetaNode-language. Mainly, the process of parsing consists of four basic steps: init, read line, identify node and parse node. This process is explained with more details in [1].

4.3.3 Serial Controller

In order to establish a communication between the user and the sensor node, it is necessary to implement the module designed in 3.4.2.

According to TinyOS 2.1 documentation [28], the basic abstraction for sensor node-PC communication is a packet source. A packet source is a communication medium over which an application can receive packets from and send packets to a sensor node. Examples of packet sources include serial ports, TCP sockets, etc. Therefore, the communication is based on packets using basic structure *message_t* and it can be performed in a similar way as radio communication works.

Rather than directly writing and reading the payload area of the *message_t* with the data to be sent, we will use a structure to hold them and then use structure assignment to copy the data into the message payload area. Using a structure allows reading and writing the message payload more conveniently when the message has multiple fields or multi-byte field, like *uint16_t* or *uint32_t*, because it is possible to avoid reading and writing bytes from/to the payload using indices and then shifting and adding (e.g. uint16_t x = data[0] « 8 + data[1]).

As it has been explained in previous section 4.3.1, a list of MetaNodes must be sent to the sensor node due to it is the way the sensor node loads a MAC protocol. Thereby, we have decided to use it as data structure to fill the message payload of the communication between PC and sensor node. It will be used for sending the MAC protocol as well as the user commands.

We have programmed a class called *Serial* that allows us to connect to the sensor node, send and receive messages and close the connection. In order to implement that, we used *MoteIF*, an object from TinyOS Java Library. Taking the URL of the serial device as a parameter in the constructor of the class, it establishes a connection between them. Every time a message is desired to be sent, it is only needed to call the *send* method which takes as parameter a MetaNode. In addition, it is necessary that *Serial* implements the interface *MessageListener* and register it as listener from the *MoteIF* to be able to listen the packets sent by the sensor node. Once a message is received, the method *messageReceived* is called and the message is returned to the Shell class which will interpret it.

Toolchain Serial Protocol has been designed and implemented to establish some rules in the communication process an allow us to transfer commands as well as MAC protocol description.

4.3.3.1 Toolchain Serial Protocol

The protocol has been designed according to the functionalities that the user may request to the sensor node. Thereby, we will describe the protocol distinguishing the

three functionalities which requires some message exchange:

- *LOAD*: Before initiating the transfer of MetaNodes generated after parsing the text file, it is necessary to create a new MetaNode with LOAD tag as MetaLabel value. Besides, it needs one parameter indicating that the type field is a value (tag VALUE) and in the value field the number of MetaNodes that the sensor node should expect before concluding the MAC protocol description transfer.

  After receiving this MetaNode, the sensor node may answer with two different messages. In case there is any error, sensor node answers with an specific MetaNode with tag ERROR as MetaLabel and IdLabel LOAD. Usually, this MetaNode is sent when the size of *nodeList* is too big for the buffer permitted by the sensor node. On the other hand, if there is no error, sensor node will answer with a MetaNode whose MetaLabel will contain the tag LOAD and 0 as IdLabel. It will notify the Shell that it can initiate the transfer *nodeList*.

  After transferring one by one all MetaNodes from *nodeList*, the sensor node will answer with a MetaNode with LOAD as MetaLabel and 1 as IdLabel, to indicate that everything went fine. All this procedure is shown in FIGURE 4.19.

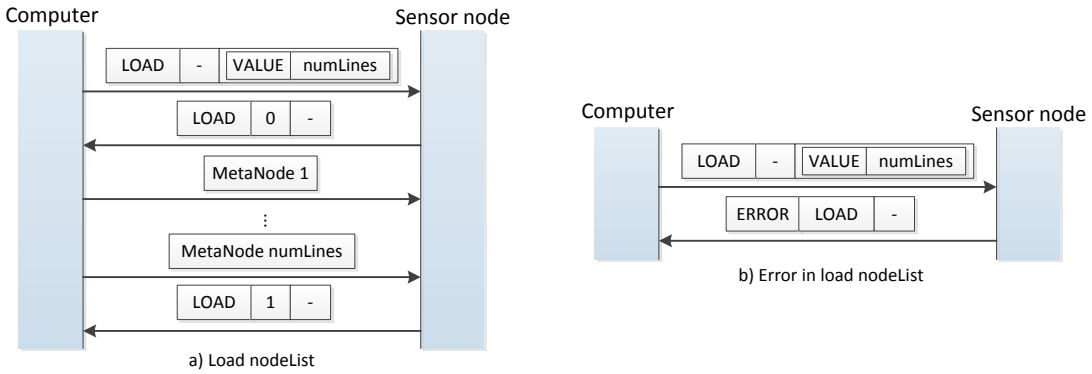

FIGURE 4.19: Messages exchange during LOAD process.

- *RUN*: As it has been described in section 4.3.1, before sending a RUN command, it is necessary to check if its necessary to make a reload of *nodeList* or only send the changes made by the user which are stored in *modifiedNodeList*.

  In case it is necessary to make a reload of *nodeList*, the protocol acts in the same manner as in the previous case.

  In case of sending the changes stored in *modifiedNodeList*, it is a bit more complicated. All the MetaNodes inside *modifiedNodeList* are extracted and sent one by one. As it has been mentioned previously, user can make two types of changes: add or remove instructions. Regarding to adding instructions, the way to send these MetaNodes consists in sending two MetaNodes at the same time. First MetaNode will have ADD_NODE tag as MetaLabel and one parameter indicating that the type field is a value (tag VALUE) and in the value field the position where it is desired to be inserted. Second MetaNode will be the MetaNode to be

inserted. If there is an error regarding to an invalid position to insert the MetaNode, the sensor node will answer with another MetaNode containing tag ERROR as MetaLabel and IdLabel ADD_NODE. In case of removing instructions, it is only needed to send one MetaNode with REMOVE_NODE tag as MetaLabel and one parameter indicating that the type field is a value (tag VALUE) and in the value field the position of the MetaNode to be removed. In case of error with the position indicated, the sensor node will answer in the same manner as adding case but using REMOVE_NODE as IdLabel.

After checking that reload or changes load process is finished, it is time to send the order of starting the execution. It is only needed to send one MetaNode with RUN tag as MetaLabel. None parameter is needed as difference with the other messages. In case there is no code loaded in the sensor node, it will send a MetaNode with tag ERROR as MetaLabel and IdLabel RUN. All this procedure is shown in FIGURE 4.20.

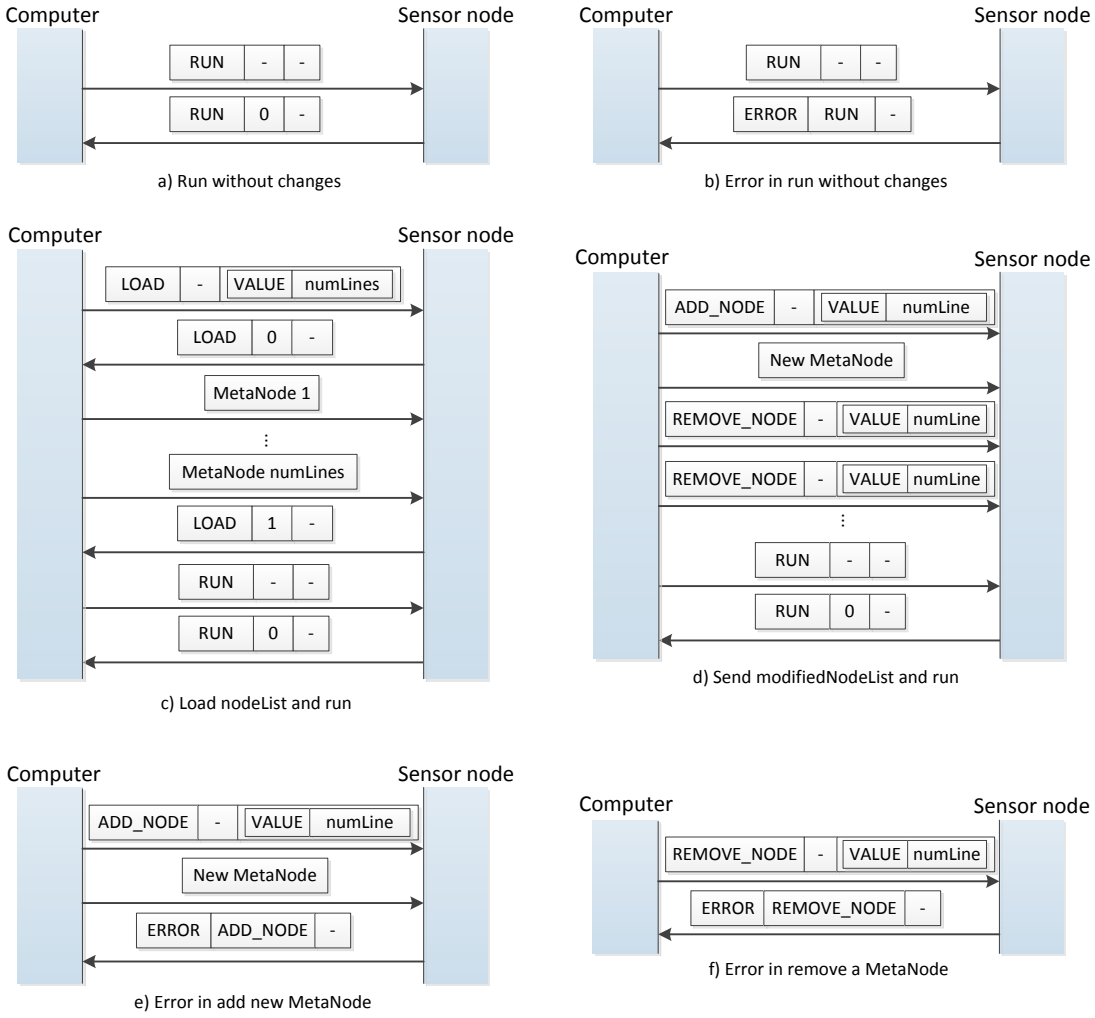

FIGURE 4.20: Messages exchange during RUN process.

- *STOP*: In order to stop the execution of a running MAC protocol, it is only needed to send one MetaNode with STOP tag as MetaLabel and none Parameter. Once the sensor node stops the execution, it answers by sending back the same MetaNode. All this procedure is shown in FIGURE 4.21.

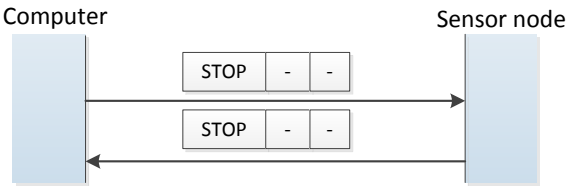

FIGURE 4.21: Messages exchange during STOP process.

5

EXPERIMENTAL RESULTS AND EVALUATION

The main goal of the toolchain is to provide reconfigurability at runtime of MAC protocols. Moreover, it is necessary to assure that the behaviour of the MAC protocols executed in the sensor nodes following this approach is comparable to the behavior of monolithic implementation.

In this chapter, the experimental results and evaluation in terms of execution time overhead and reconfiguration costs are presented. These metrics will allow us to determine the disadvantages of the toolchain. As it is mentioned, the two metrics used to perform the evaluation are:

- *Execution time overhead*: it refers to the time that the toolchain needs to execute a function or a MAC protocol.
- *Reconfiguration costs*: it evaluates the cost of the reconfigurations operations for monolithic implementations and for the toolchain.

All the experiments were carried out on TelosB [4] and Mica2 [5] platforms. TelosB sensors use the radio chip Chipcon CC2420 [29] and the chip Chipcon CC1000 [30] is used by Mica2 sensors. The basic functionalities, the monolithic MAC protocol implementations and our toolchain which have been used to make the evaluation were implemented in TinyOS 2.x.

5.1 EXECUTION OVERHEAD

In order to evaluate the time that the toolchain needs to execute a task, we measured the execution time for a monolithic approach and for our toolchain. The two results have been compared in terms of absolute time and the overhead added by the toolchain. There are two type of measurements: first, we have taken measurements for basic functionalities like the time needed to start the radio or send a packet; after that, the total time needed to send a packet using a defined MAC protocol have been measured. Before presenting the experimental results, the setups which have been used will be explained.

First of all, execution time for the monolithic approach have been measured using the experimental setup shown in FIGURE 5.1. This setup consists of four components. The resistor R is formed by three 10 Ω resistor connected in parallel to make its resistance very small and not interfere in the measurements. The resistor value is 3.33 Ω. The resistor is connected in series with the sensor node which is powered up by a power supply set to 3.0 V, due to it is the voltage provided by the batteries used by the sensor node. Finally, an oscilloscope is used to measure the voltage across the resistor.

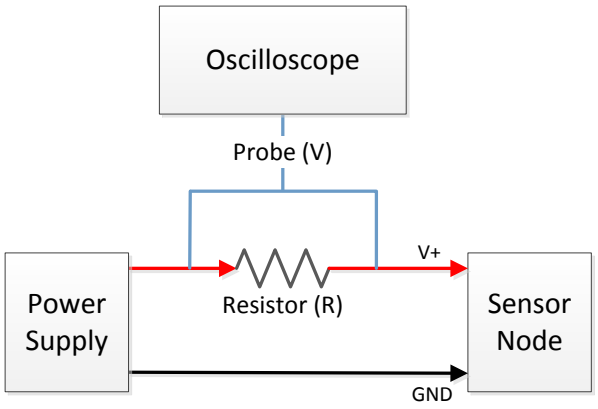

FIGURE 5.1: Block diagram of the experimental setup for monolithic execution time measurements.

The execution time measurements for the toolchain cannot be performed using the experimental setup explained above, because the sensor should be connected through a USB cable to a PC, in order to receive user commands and load the MAC protocol design. For that reason, another experimental setup which is shown in FIGURE 5.2, have been used to measure the toolchain execution time. There are also four components, the resistor R is the same used in the first setup and is connected in series with the sensor node. However, in this case, the sensor node is powered up by the PC through the USB connection and can receive data from or send data to the PC. Finally, two probes with their grounds connected together are used, one to measure the voltage before the resistor and the other to measure the voltage after the resistor. Thereby, an oscilloscope is used to measure the voltage across the resistor by subtracting the voltage of the one probe to the other. In this case, the main difference is the USB port provides 5 V as a difference with the 3 V provided by the power supply and the batteries of the sensor node. Thereby, all the measurements will be amplified in terms of voltage. However, since we are only interested in the time domain, no modification is necessary for our results.

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

5.1.1 *Basic Functionalities*

Five basic functionalities measurements have been selected due to they are commonly used in all the MAC protocols: start the radio, stop the radio, read the actual RSSI value, perform carrier sensing and send a packet.

For the monolithic approach, the time since the specific command is called until the corresponding event is fired has been measured. However, for the toolchain, the time since the Execution Scheduler finds a MetaNode with *FUNC* as MetaLabel and the specific function as IdLabel, until the event is fired has been measured. For instance, to measure the time overhead when a packet is sent for the monolithic approach, the time since the command *sendData()* of *MACSendPacket* component is called until *SENDDATA_EVENT* is fired has been measured. For the toolchain, the time since the

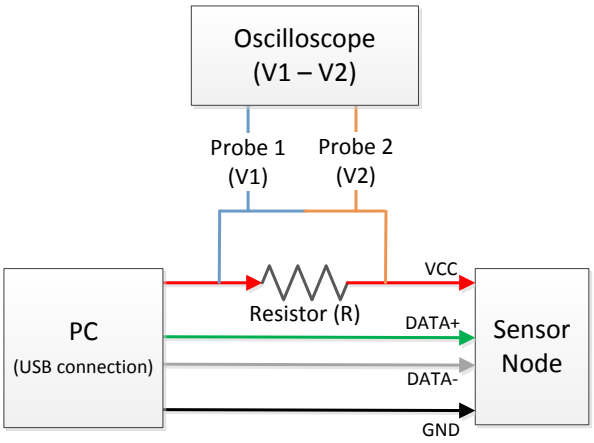

FIGURE 5.2: Block diagram of the experimental setup for toolchain execution time measurements.

Execution Scheduler finds a MetaNode with *FUNC* as MetaLabel and *SENDDATA* as IdLabel until *SENDDATA_EVENT* is fired has been measured. Therefore, theoretically, the time added in our toolchain is the one it needs to compile the MetaNode into the corresponding command call.

TABLE 5.1 and TABLE 5.2 shows the results for start the radio, stop the radio, read the actual RSSI value and perform carrier sensing in TelosB and Mica2, respectively. In TelosB, the worst time overhead in percentage is 37 % when the radio is stopped. However, the time added in the worst case is 180 µs which is an acceptable time. The same happens in Mica2, the worst time overhead in percentage is 39 % when the radio is stopped too. However, the time added in the worst case is only 120 µs. The time fluctuations depend on the position of the function calls and their events inside the execution list and the number of variables which those functions take as parameters. Due to the execution list and variable list search algorithms need a certain time to located them.

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 %

TABLE 5.1: Time results for basic functionalities executed in a TelosB sensor node.

TABLE 5.3 shows the results obtained in a TelosB sensor node after measuring the time needed to send a packet varying the packet payload. As it can be seen, the time added in the worst case is 250 µs which it is an acceptable overhead time.

In FIGURE 5.3 is shown a graphic of the time needed to send a packet depending on the size of the packet payload. The trend of the two approaches is to rise in a linear way and almost parallel between them which means that the time difference between

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 %

TABLE 5.2: Time results for basic functionalities executed in a Mica2 sensor node.

Payload	*Monolithic Time*	*Toolchain Time*	*Overhead Time*	*Overhead*
11 bytes	8.74 ms	8.99 ms	250 µs	2.86 %
25 bytes	10.15 ms	10.26 ms	110 µs	1.08 %
50 bytes	12.41 ms	12.63 ms	220 µs	1.77 %
75 bytes	14.82 ms	14.99 ms	170 µs	1.15 %
100 bytes	16.78 ms	17.03 ms	250 µs	1.49 %
127 bytes	19.20 ms	19.42 ms	220 µs	1.15 %

TABLE 5.3: Time results when sending a packet using different payloads in a TelosB sensor node.

them in all the cases is approximately constant. Therefore, we conclude that in TelosB sensor nodes our toolchain performance do not depends on the payload size.

Moreover, in case the time added by our toolchain was always the same, in the worst case it would be of 250 µs for a TelosB sensor node. In that case, as bigger size of the payload was, lower would be the overhead since more time would be needed to send the packet and the time added would be less significant.

The same evaluation has been performed to measure the overhead time when a packet is sent in a Mica2 sensor node. TABLE 5.4 shows the final results in this case. The most remarkable fact is that the time added in the worst case is 90 µs, which is an acceptable overhead time.

Payload	*Monolithic Time*	*Toolchain Time*	*Overhead Time*	*Overhead*
11 bytes	22.83 ms	22.90 ms	70 µs	0.31 %
25 bytes	28.65 ms	28.74 ms	90 µs	0.31 %
50 bytes	39.17 ms	39.21 ms	40 µs	0.10 %
75 bytes	49.47 ms	49.54 ms	70 µs	0.14 %
100 bytes	59.88 ms	59.93 ms	50 µs	0.08 %
127 bytes	71.17 ms	71.23 ms	60 µs	0.08 %

TABLE 5.4: Time results when sending a packet using different payloads in a Mica2 sensor node.

In the same way as with the TelosB sensor node, in FIGURE 5.4 is shown a graphic of the time needed to send a packet depending on the size of the packet payload. The trend of the two approaches is to rise in a linear way and parallel between them too; both are almost identical in this case. Therefore, as it happened in TelosB, our

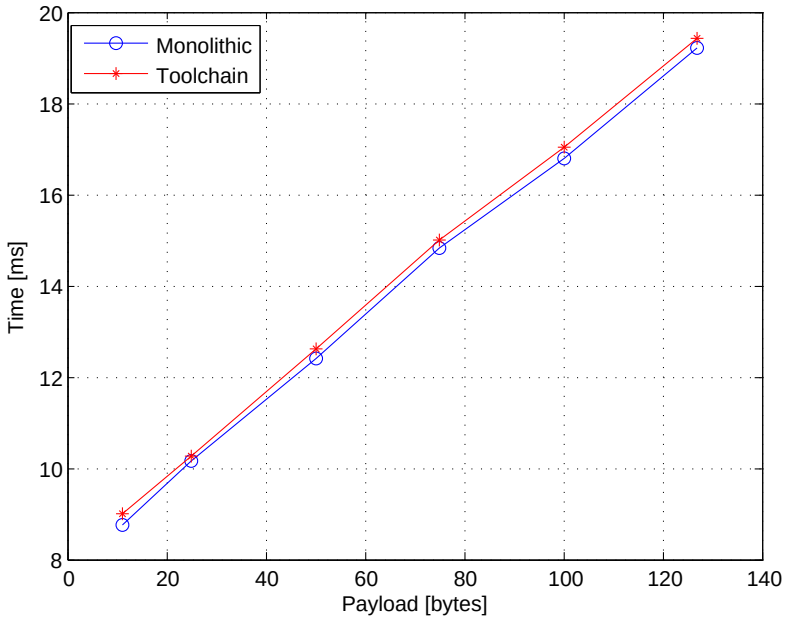

FIGURE 5.3: Time needed to send a packet using different payloads in a TelosB sensor node.

toolchain performance do not depends on the payload size in Mica2 sensor nodes.

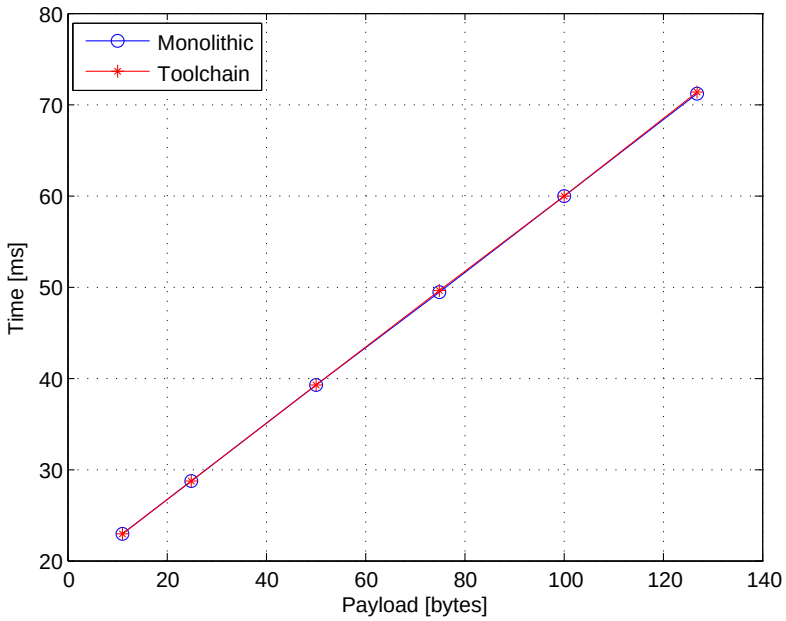

FIGURE 5.4: Time needed to send a packet using different payloads in a Mica2 sensor node.

5.1.2 MAC protocols

After measuring the overhead time added by the toolchain for basic functionalities, it is important to measure which is the time added when executing a complete MAC protocol or a set of functionalities.

The tests were carried out using B-MAC as preamble-sampling protocols example and S-MAC as common period protocols example. Besides, another test was carried out not using any protocol in order to measure the time added by the toolchain in a very simple program.

In case of not using any protocol, the test consists in measuring the time needed to start the radio and send a packet. In the other cases, the tests are very similar. In B-MAC, the sensor node starts the radio, performs carrier sensing during 50 ms. If the medium is free, it sends a preamble of 100 ms duration followed by a data packet. Finally, using S-MAC, the sensor node starts the radio, starts two timers, one for synchronization and another to go to sleep; and performs carrier sensing during 50 ms. Then, if the medium is free, it sends a packet of synchronization. When the first timer fires, it makes again carrier sensing during 50 ms and if the medium is free, the RTS-CTS-DATA-ACK exchange is carried out. All the messages used in the communication during all the tests are 11 bytes of payload size.

As it can be observed in TABLE 5.5, the overhead percentage added by the toolchain in TelosB is less than 5 % in all the cases. Taking into account that the number of operations executed by each test: 4 for No-Protocol, 10 for B-MAC and 23 for S-MAC; the time added by the toolchain for each operation oscillates between 100 and 160 µs, which matches with the results obtained in section 5.1.1.

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.

For Mica2 platform, the same experiments have been carried out. However, in this case, S-MAC has not been included in the test due to the hardware limitations does not allow S-MAC implementation using the toolchain as explained in section 4.2.1. The results are shown in TABLE 5.6. In this case, the time added by the toolchain for each operation executed oscillates between 50 and 70 µs which is also very similar to the results presented in section 5.1.1.

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.

5.2 RECONFIGURATION COSTS

Nowadays, a rapid reconfiguration of MAC schemes is required in order to achieve Quality of Service (QoS) demands depending on the application. For instance, in WSNs, applications of data weather collection might prioritize the energy efficiency over other aspects, in order to extend sensors battery life; while for video streaming applications, data rate and jitter might be the most important parameters. Another possible situation where the reconfiguration can be very useful is in those environments where a WSN should coexist with other networks working at the same frequency.

Therefore, the application should be able to specify the its preferences in order to achieve the desired QoS. In the same manner, toolchain will enable MAC protocol to reconfigure at runtime to accomplish that.

In the monolithic approach, in case some reconfiguration may be needed, the only way to perform it is to modify the source code of the application, recompile and deploy it again in the sensor node. In order to measure the time needed, a tool provided in Linux operative systems has been used. This tool is called *time* and it measures the total execution time of an application or command. It usually shows the results distinguishing three different times:

- *Real*: it is wall clock time - time from start to finish of the call. This is all elapsed time including time slices used by other processes and time the process spends blocked, i.e. if it is waiting for I/O to complete.
- *User*: it is the amount of CPU time spent in user-mode code (outside the kernel) within the process. This is only actual CPU time used in executing the process. Other processes and time the process spends blocked do not count towards this figure.
- *Sys*: it is the amount of CPU time spent in the kernel within the process. This means executing CPU time spent in system calls within the kernel, as opposed to library code, which is still running in user-space. Like 'user', this is only CPU time used by the process. See below for a brief description of kernel mode (also known as 'supervisor' mode) and the system call mechanism.

User+Sys will indicate how much actual CPU time the process or application has used and, ideally, it would be equal to *Real* output. Unfortunately, this equality is never fulfilled due to normal processor performances.

TABLE 5.7 shows the time needed by the two evaluated platforms, TelosB and Mica2, to reconfigure a MAC protocol. Taking *Real* value as the worst result, it can be observed that the time needed oscillates between 10.3 and 13.2 seconds depending on the protocol in case of TelosB, and between 8.9 and 12.7 seconds in case of Mica2. We consider that it would be better to compare our toolchain with the monolithic approach best case which is the *User+Sys*. Therefore, the time needed to reconfigure oscillates between 1.9 and 2.4 seconds in case of TelosB, and between 2.5 and 3.2 seconds in case of Mica2.

In order to evaluate the time needed to reconfigure a MAC protocol in case of using the toolchain, we distinguish three different cases of reconfiguration:

Protocols	***TelosB Times***			***Mica2 Times***		
	Real	*User*	*Sys*	*Real*	*User*	*Sys*
B-MAC	13215 ms	2224 ms	180 ms	12715 ms	2816 ms	168 ms
MFP-MAC	13115 ms	2048 ms	124 ms	12881 ms	2908 ms	160 ms
X-MAC	13213 ms	2084 ms	124 ms	12944 ms	3060 ms	204 ms
S-MAC	13044 ms	2028 ms	148 ms	-	-	-
T-MAC	11811 ms	1916 ms	152 ms	-	-	-
No-Protocol	10273 ms	1772 ms	128 ms	8919 ms	2360 ms	152 ms

TABLE 5.7: Time needed to reconfigure a MAC protocol in TelosB and Mica2 using monolithic approach.

- *LOAD*: which means that a new protocol replaces the running protocol. This process includes reading and parsing the text file as well as transferring all MetaNodes and sending the command needed to start the execution to the sensor node.
- *CHANGES + RELOAD*: which implies to insert some changes into the protocol which provokes a reload of all MetaNodes as explained in section 4.3.3.1. Moreover, it includes sending the command needed to start the execution. This test will consist in adding an if statement with a function call inside.

```
>> add 5 if(tos_node_id == 1){
>> add 6 led0On();
>> add 7 }endif;
```

- *CHANGES + LOAD CHANGES LIST*: which implies to introduce some changes and send them to the sensor node as well as sending the command needed to start the execution. This test will consist in modifying some function parameters or changing an expression of the running protocol. For instance, in preamble-sampling protocols the sleep interval is set to 200 ms, in common period protocols the time of wake up timer is set to 2 seconds and for the simple application the data of the packet is changed.

```
>> remove 5
>> add 5 setSleepInterval(200);

>> remove 24
>> add 24 wakeupTime = 2000;

>> remove 5
>> add 5 sendData(2,30);
```

All the results of this experiment are shown in TABLE 5.8 and TABLE 5.9, for TelosB and Mica2 platform respectively. As it can be observed, in all the possible cases the toolchain offers a high performance in comparison with monolithic approach.

For instance, in case of introducing some little modifications into the B-MAC protocol design using TelosB, the toolchain only needs 54 ms to carry it out, while monolithic approach needs 2.4 s due to it needs to recompile again the whole project. Therefore, the reconfigurable toolchain saves more than 97 % of the time needed by the monolithic approach, providing more flexibility which is ideal for dynamic environments such as those where WSNs are deployed. Another example can be the time needed to perform a total change of MAC protocol, from S-MAC to B-MAC. For TelosB, the toolchain needs 947 ms, while the monolithic approach needs the same time as mentioned before, 2.4 s. In this case, the percentage of time saved is slightly more than 60 %.

The same results are obtained for Mica2 platforms. In case of introducing some little modifications into the B-MAC protocol, the toolchain needs 45 ms and the monolithic approach 3.2 s, thereby it saves more than 98 % of time needed. In the other case, to change from S-MAC to B-MAC the toolchain needs 1077 ms and the monolithic approach 3.2 s, the time saved is approximately 66 %.

Protocols	***No. of Nodes***	***TelosB Times***		
		LOAD	CHANGES + RELOAD	CHANGES + LOAD CHANGES LIST
B-MAC	112	947 ms	898 ms	54 ms
MFP-MAC	122	988ms	955 ms	48 ms
X-MAC	125	1049 ms	1015 ms	41 ms
S-MAC	218	1603 ms	1562 ms	50 ms
T-MAC	194	1483 ms	1401 ms	54 ms
No-Protocol	12	225 ms	237 ms	31 ms

TABLE 5.8: Time needed to reconfigure a MAC protocol in TelosB using reconfigurable toolchain.

Protocols	***No. of Nodes***	***Mica2 Times***		
		LOAD	CHANGES + RELOAD	CHANGES + LOAD CHANGES LIST
B-MAC	112	1077 ms	1100 ms	45 ms
MFP-MAC	122	1174 ms	1168 ms	44 ms
X-MAC	125	1273 ms	1222 ms	47 ms
No-Protocol	12	277 ms	297 ms	47 ms

TABLE 5.9: Time needed to reconfigure a MAC protocol in Mica2 using reconfigurable toolchain.

In FIGURE 5.5 and FIGURE 5.6, a graphical comparison of the time needed to change completely the running MAC protocol by a new one between the monolithic approach and the toolchain is shown in TelosB and Mica2 respectively. As it can be observed the difference is quite significant between the two approaches and the minor time saved in TelosB is more than 25 % while in Mica2 is more than 60 %.

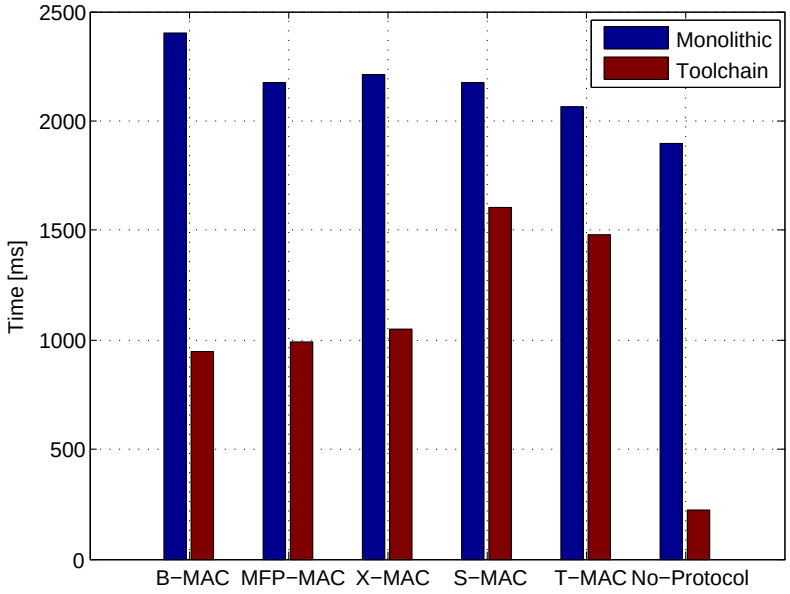

FIGURE 5.5: Comparison of the reconfiguration time in TelosB node.

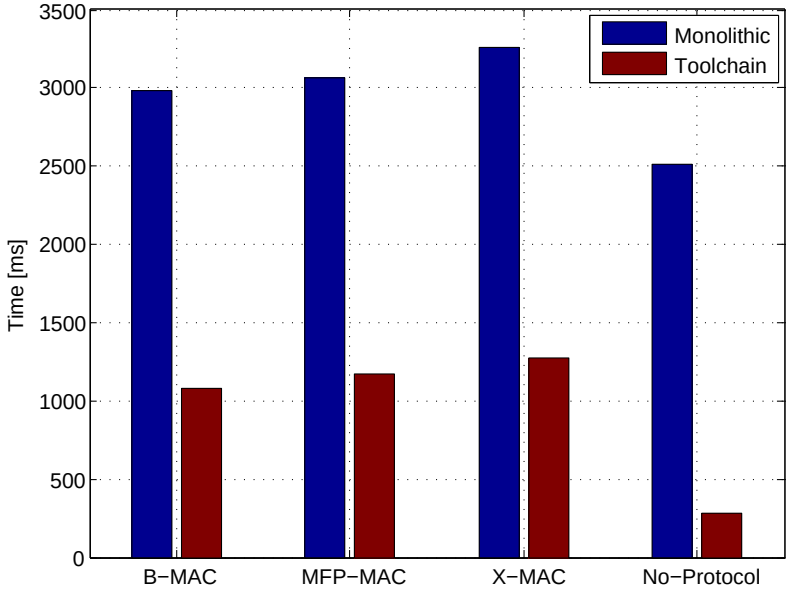

FIGURE 5.6: Comparison of the reconfiguration time in Mica2 node.

6

CONCLUSIONS

In this thesis, we have designed and implemented in collaboration with [1] an efficient toolchain to reconfigure MAC protocols at runtime and provide flexibility in dynamic environments. In order to enable simple MAC protocol design, MAC components which provide basic MAC protocol functionalities have been implemented following a hardware independent approach. Therefore, a user can design a MAC protocol without having knowledge of the platform for deployment. On sensor nodes, a sensor interface has been implemented to execute and reconfigure the MAC protocol design. In order to provide a simple way for the user to reconfigure the sensor node, a user interface has been implemented. By sending commands the user can interact with the sensor node: load a new protocol, reconfigure, run and stop it.

Our toolchain has been compared in terms of execution time and reconfiguration costs with a monolithic approach. The experiments have been carried out in TelosB and Mica2 which are two platforms with different radio chips. In both cases, the results obtained show that the toolchain enables fast runtime reconfiguration of MAC protocols with an acceptable execution time overhead. In the worst case, our toolchain saves between 26 % and 88 % of the MAC protocol reconfiguration time; while in the best case, the time saved is around 98 %.

In conclusion, our toolchain provides rapid runtime reconfiguration of MAC protocols for sensor nodes when applications requirements or networks conditions change. Moreover, as MAC components and sensor interface have been designed and implemented following the hardware independent approach, the toolchain can be executed in any platform. Thereby, any MAC protocol designed by the user can be executed and reconfigured in many platforms, such as TelosB or Mica2.

In order to achieve a better performance of the toolchain, the execution efficiency can be optimized. One solution could be replacing all static arrays by hash-tables implemented in TinyOS. If this is achieved, the time added by the toolchain would be reduced due to the searches in hashtables are more efficient.

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

SNR Signal to Noise Ratio

SP Sensor-net Protocol

ULLA Unified Link-Layer API

UPMA Unified Power Management Architecture

WSN Wireless Sensor Network

List of Tables

3.1 Summary of the basic functionalities of the most common MAC protocols. [1] . . . 8
3.2 Summary of the complex functionalities of the most common MAC protocols. [1] . . . 8

4.1 MAC components library and their composition. [1] . . . 19

5.1 Time results for basic functionalities executed in a TelosB sensor node. . . . 43
5.2 Time results for basic functionalities executed in a Mica2 sensor node. . . . 44
5.3 Time results when sending a packet using different payloads in a TelosB sensor node. . . . 44
5.4 Time results when sending a packet using different payloads in a Mica2 sensor node. . . . 44
5.5 Time results for different protocols executed in a TelosB sensor node. . . . 46
5.6 Time results for different protocols executed in a Mica2 sensor node. . . . 46
5.7 Time needed to reconfigure a MAC protocol in TelosB and Mica2 using monolithic approach. . . . 48
5.8 Time needed to reconfigure a MAC protocol in TelosB using reconfigurable toolchain. . . . 49
5.9 Time needed to reconfigure a MAC protocol in Mica2 using reconfigurable toolchain. . . . 49

LIST OF FIGURES

2.1 Different applications realized by a component-based design approach. [1] 3
2.2 MAC runtime reconfiguration toolchain structure. [17] 5

3.1 Struct definition for *MetaNode*. [1] 9
3.2 Struct definition for *MetaParam*. [1] 9
3.3 Sensor Interface design. 10
3.4 Procedure to execute sequentially an event implementation. 12
3.5 Procedure to add a new MetaNode to the Execution List. 14
3.6 Procedure to remove a MetaNode from the Execution List. 15
3.7 User Interface design. 17

4.1 Diagram of reusable components. [1] 20
4.2 Sensor Interface implementation. 21
4.3 Definition of the maximum size of Execution list. 21
4.4 Execution of an event. 23
4.5 Example of storage of the attributes of an event. 24
4.6 Execution of global variable definitions. 24
4.7 Addition of a node next to last occupied position of the list. 25
4.8 Addition of a node in a specified position of the list. 25
4.9 Removal of a node in a specified position of the list. 25
4.10 MetaVariable structure. 26
4.11 Search for a variable position inside Variable list. 26
4.12 Removal of local variables when an event execution is finished. 27
4.13 Get the value of a parameter flowchart. 28
4.14 Execution of an *if* MetaNode. 30
4.15 Interface of ExecutionScheduler component. 31
4.16 Definition of Commands enumeration. 33
4.17 Recognition of a command. 34
4.18 Command Shell flowchart. 36
4.19 Messages exchange during LOAD process. 38
4.20 Messages exchange during RUN process. 39
4.21 Messages exchange during STOP process. 40

5.1 Block diagram of the experimental setup for monolithic execution time measurements. 42
5.2 Block diagram of the experimental setup for toolchain execution time measurements. 43
5.3 Time needed to send a packet using different payloads in a TelosB sensor node. 45

5.4 Time needed to send a packet using different payloads in a Mica2 sensor node. 45
5.5 Comparison of the reconfiguration time in TelosB node. 50
5.6 Comparison of the reconfiguration time in Mica2 node. 50

BIBLIOGRAPHY

[1] L. M. Amorós, "A Tool for Rapid MAC Protocol Prototyping and Design for Wireless Sensor Networks," M.S. thesis, Institute for Networked Systems, RWTH Aachen University, September 2012.

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

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

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

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

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

[7] 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.

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

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

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

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

[12] 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.

[13] G. Holland, N. Vaidya, and P. Bahl, "A Rate-Adaptive MAC Protocol for Multi-Hop Wireless Networks," *ACM/IEEE Int. Conf on Mobile Computing and Networking*, pp. 929–933, 2001.

[14] Z. Wang, U. Mani, M. Ju, and H. Che, "A Rate Adaptive Hybrid MAC Protocol for Wireless Cellular Networks," *International Conference of Wireless and Mobile Communications (ICWMC)*, 2006.

[15] A. Faragó, A. D. Myers, V. R. Syrotiuk, and G. V. Záruba, "Meta-MAC Protocols: Automatic Combination of MAC Protocols to Optimize Performance for Unknown Conditions," *IEEE Journal on Selected Areas in Communications*, vol. 18, no. 9, September 2000.

[16] H. S. Lichte and S. Valentin, "Implementing MAC Protocols for Cooperative Relaying: A Compiler-Assisted Approach," *Proceedings of the International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTools)*, March 2008.

[17] 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.

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

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] B. Greenstein and P. Levis, "TinyOS Enhancement Proposals: Serial Communication (TEP 113)," Tech. Rep., Computer Science Laboratory, Stanford University, 2007.

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

[30] 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
Noemí Arbós