

Methods and tools for the compilation and software optimization of wireless embedded systems dedicated to applications

Andréea Chis

▶ **To cite this version:**

Andréea Chis. Methods and tools for the compilation and software optimization of wireless embedded systems dedicated to applications. Embedded Systems. Ecole normale supérieure de lyon - ENS LYON, 2012. English. NNT : . tel-00768830

HAL Id: tel-00768830

https://theses.hal.science/tel-00768830

Submitted on 25 Dec 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

PhD THESIS

submitted for the grade of

Doctor of Université de Lyon
ÉCOLE NORMALE SUPÉRIEURE DE LYON

Laboratoire de l'Informatique du Parallélisme
École Doctorale en Informatique et Mathématiques de Lyon

field: Computer Science

by **Andreea CHIS**

Title:

Methods and tools for the compilation and software optimization of wireless embedded systems dedicated to applications

Méthodes et outils pour la compilation et l'optimisation logicielle des systèmes embarqués sans fil dédiés à des applications

Supervisors:

Mr. Eric FLEURY
Mr. Antoine FRABOULET

Reviewers:

Mr. Gerhard FOHLER
Mr. David SIMPLOT

Contents

List of Figures iii

List of Tables v

1 Introduction 1

2 WSN State of the art 7

2.1 Wireless Sensor Applications Scenarios 7

2.2 Wireless Sensor Nodes 10

2.3 MAC Protocols for Wireless Sensor Networks 12

2.4 Device Modelling and Driver Generation 19

3 Hardware Software Interface: Cross Layer Mapping Optimization 23

3.1 Software Protocol Description 23

3.1.1 Definition and Formalism 23

3.1.2 Application on B-MAC protocol 27

3.2 Hardware Protocol Description 29

3.3 Software to Hardware Optimized Mapping 37

3.3.1 Software Automaton Transformation 37

3.3.2 Problem Statement and Complexity Analysis 42

3.3.3 Mapping Heuristic 50

3.4 Code Skeleton Generation 55

3.4.1 Multi-threaded OS Code Skeleton Generation 56

3.4.1.1 Multi-threaded OSs for WSNs 56

3.4.1.2 Multi-threaded OS Code Skeleton Generation Method . 59

3.4.2 Event-driven OS Code Skeleton Generation 61

3.4.2.1 Event-driven OSs for WSNs 61
3.4.2.2 Event-driven OS Code Skeleton Generation Method . . 65

4 Experimental Evaluation **67**
4.1 Stochastic Modeling 69
4.1.1 Energy Consumption Theoretical Analysis 69
4.1.2 Average Energy Consumption 75
4.2 Simulation Results 77
4.2.1 The Worldsens Environment 77
4.2.2 Simulation Set-up 79
4.3 WSN Testbed Experiments using the SensLAB platform 83
4.3.1 SensLAB goals and facilities 83
4.3.2 SensLAB Software and Hardware Infrastructures 85
4.3.2.1 SensLAB Hardware Infrastructure 85
4.3.2.2 SensLAB Software Infrastructure 86
4.3.3 SensLAB Experimental Set-up and Evaluation 88
4.3.3.1 SensLAB Experimental Scenario and Evaluation Metrics 88
4.3.3.2 SensLAB Experimental Set-up 89

5 Conclusion and Perspectives **95**

Bibliography **99**

List of Figures

2.1 T-MAC Schedule 16
2.2 WiseMAC communication 17

3.1 Automaton example. 24
3.2 B-MAC protocol description in the form of a timed automaton. 26
3.3 CC1100 Radio Finite State Machine. 30
3.4 CC1100 Simplified Block Diagram 31
3.5 Synthesis flow for software to hardware mapping. 39
3.6 Software automaton transformation for the case of free states with input transitions not resetting the clock variable defining the state's duration. 39
3.7 Software automaton transformation for free states with output transitions with clock constraints. 40
3.8 Software Automaton A with Free State Sf to be mapped onto hardware automaton. 43
3.9 Transformation of nodes v_i having $\text{val}_i > 1$ 46

4.1 Simulation scenario: n saturated nodes send data to 1 sink 67
4.2 Markov chain model for B-MAC backoffs. 71
4.3 Transmission probabilities for different packet lengths and number of nodes 72
4.4 Busy medium probabilities for different packet lengths and number of nodes 73
4.5 Theoretical gain obtained for the improved backoffs mapping with respect to the unimproved version 76
4.6 Worldsens - distributed simulation environment 77
4.7 Architecture of the WSN430 WSN node 79

LIST OF FIGURES

4.8 WSim flow of events 82
4.9 Energy consumption of the improved protocol versions normalized with respect to the unimproved version 83
4.10 SensLAB platform 84
4.11 SensLAB node architecture 85
4.12 SensLAB software infrastructure 87
4.13 Average number of sent packets per node for the 3 driver implementations 90
4.14 Average number of packets received by the sink for the 3 driver implementations 92

List of Tables

3.1 Description and energy consumption parameters for the states of CC1100 32
3.2 CC1100 Command Strobes Summary 34
3.3 State Transition Timing 35

LIST OF TABLES

1

Introduction

The recent progress achieved in the domain of micro-electro-mechanical systems (MEMS) and micro-electronic devices along with their miniaturisation have lead to the developpment of a new field of applications for wireless networks: the Wireless Sensor Networks (WSN). Sensor networks are composed of a set of small intelligent objects whose role is to measure environmental informations such as the temperature, air/water pollution level, to detect movements, vibrations, intruder presence or the proximity of other sensors etc. Sensor nodes communicate using a radio device in order to manage and organize the network and to gather the produced data on special nodes called sinks.

The nodes in sensor networks are severely constrained in terms of resources: computation, memory, communication as well as energy. In this context and in order to deploy efficient sensor networks, a specific care must be invested in the design of the application, communication protocols and operating systems that are to be executed. The entire hardware as well as software architecture must be carefully chosen with the goal of minimizing the energy consumption.

A sensor node is generally composed of a few simple components: a micro controller(MCU), one or several physical sensors, a radio interface, an external memory and an energy source(typically a battery). Despite the relative simplicity of these basic components when compared to a PC, the great diversity of choices available on the market make the tasks of conception, implementation and deployment of WSNs extremely complex. Diversity in the hardware: MicaZ, Telos, iMote with ARM7, AVR or TI micro-controllers. The diversity concerns the following aspects:

- radio and physical layers: 868MHz, 2.4GHz, modulation, frequency hopping.

- operating systems: TinyOS [26], Contiki[17], FreeRTOS[37], JITS[16].
- constraints: real-time constraints[38], energy constraints, memory or processing constraints.
- applications: from the military to the civil use.
- data traffic patterns, mobility patterns, communication patterns.

Facing this variety of problems, there is no one-fits-all solution but rather one better solution for each specific case.

Choices have to be made at each level of the design process and these choices generally have an impact on the application performances. Moreover, debuging and profiling of micro-controllers and more generally distributed embedded systems is clearly a hard task.

The current approaches for programming these systems remains classical and comparable to the programming models used for systems that do not face the same constraints and requirements for low energy consumption. We find among these approaches the use of operating systems featuring unified hardware abstractions, unified operating system abstraction layers for ensuring portability[33], network protocol stacks and multi-threaded applications. Classical approaches do not take into consideration the possibilities for the energy management available on the nodes used in sensor networks. Technologies like the low-power modes of the devices as well as the use of all hardware resources available remain far from what is attainable by dedicated programming performed by a domain expert. These aspects are highly important since the particular target domain corresponds to deeply embedded applications whose constraints and behavior are known in advance.

In this context, the development of tools and methods aiming at helping the design of these dedicated applications and tackling their complexity and their constraints is crucial.

One of the components that consumes the most energy in a sensor node is its radio device. WSNs and their associated networking stacks are pushing the integration of timeliness and timing constraints, energy consumption and implementation of software stacks. Newer 802.15 communication protocols tend to use precise synchronizations mechanisms such as beacon modes, TDMA communications and frequency hopping

protocols to efficiently use the communication channel. These protocols require precise timings within the software application code in order to drive the radio device including constraints coming from the device itself (calibration, delays to switch between mode, time to wakeup from a particular low power state) and application and protocol requirements.

Writing a portable and/or optimized software stack that can combine precise timing analysis coming from different sets of requirements is a tedious and error prone task. A modification in many of the timing requirements can have an impact on the state machine and on the way the hardware is driven with an energy consumption minimization goal.

Contributions

In this thesis, we propose a methodology that allows a mapping of a software protocol onto a physical device, that guarantees that all the time constraints are met and all feasible transitions in the protocol automaton remain realizable, all while minimizing the energy consumed. We address the problem of mapping a software protocol expressed as a timed automaton to a physical device whose behavior is expressed as a finite state machine with annotations. We classify software states as *fixed*, which correspond to a unique state in the device automaton, and *free*, with their mapping left at the designer's choice. In the case of a physical device with states of fixed or variable duration but with lower limit constraint, we prove that the problem of mapping a free state of fixed duration to a path in the device such that the energy consumption is minimized is NP complete. We propose a heuristic for the mapping of free software states onto physical device paths such that the energy consumption is minimized and the transitions in the software protocol remain realizable. The output of our mapping algorithm is a code skeleton that implements the intended behavior optimized for a given hardware device. This code skeleton can be generated to further provide portability among different platforms and generate low level calls to communication APIs and operating system functionality.

We illustrate the energy gains that can be obtained by our approach with the mapping of the B-MAC WSN protocol onto a radio device. We investigate a network scenario where a set of saturated nodes contend for the channel trying to transmit

data to one sink. We developed a stochastic model for the behavior of a node in this scenario and obtain a theoretical gain of 60%[12] of an optimized mapping with respect to an unoptimized one. The code skeleton obtained for the mapping of the B-MAC protocol to the CC1100 radio device was adapted to MantisOS and simulated under the Worldsens environment and the theoretical gains were confirmed. Finally, real-testbed experiments on the SensLAB platform illustrated that the optimization in terms of energy consumption did not affect the functional parameters of the protocol[13].

Outline

In what follows, in chapter 2 we will present the context of this thesis. We describe some application deployments of WSNs in section 2.1, then some of the existing node platforms with their common architecture and resource constraints in section 2.2. We continue by describing some of the main MAC protocols dedicated to WSNs thus illustrating their characteristics and requirements in section 2.3. Finally, we present some of the approaches to device modelling and device driver generation in section 2.4.

In chapter 3 we present the problem of software to hardware mapping with energy consumption minimization constraint we wish to solve. We describe the model derived from timed automata that we use for the software protocol behavior description in section 3.1. In section 3.2 we describe the finite state machine with annotations model used for describing the hardware device. In section 3.3 we prove the NP completeness of the problem of mapping a free state of the software automaton of fixed duration to a path in the physical device such that the energy consumption should be minimal and we propose a heuristic for the software to hardware mapping. The methodology for code skeleton generation for the software to hardware mapping for two types of operating systems for WSNs is presented in section 3.4.

In chapter 4 we present the evaluation of our methodology in the particular context of mapping the B-MAC protocol onto the CC1100 radio device. We investigate a scenario where saturated nodes contend for the radio channel using B-MAC's CSMA trying to transmit data to one sink. In section 4.1 we present a stochastic model of a sensor node which yields a theoretical gain of 60% between an optimized mapping of the protocol compared to a simple mapping. In section 4.2 we present the results of simulation under the Worldsens platform of the code skeleton adapted for MantisOS.

In section 4.3 we illustrate the results of real-testbed experiments on the SensLAB platform.

The conclusion of our work and perspectives are given in chapter 5.

2

WSN State of the art

A wireless sensor network (WSN) is a distributed system composed of several autonomous sensor nodes whose purpose is to monitor physical or environmental conditions such as temperature, sound, vibration, pressure, motion or pollutants. The nodes in the network communicate and collaborate in order to relay the sensor collected data across the network towards a main location [28],[21],[46].

Typically, a sensor node contains several basic components: a processor, a memory, a power supply, a radio and several sensors. Mechanical, thermal, biological, chemical, optical, and magnetic sensors can be included in the sensor node to measure properties of the environment.

Given their limited memory and deployment in unaccessible locations, the nodes use their radio interface and communicate in order to transfer the collected data to a base station. The nodes are typically battery powered, but they can also use a secondary power source suitable for the deployment environment.

2.1 Wireless Sensor Applications Scenarios

Although initially the development of wireless sensor networks was motivated by military applications nowadays these networks are used in many industrial and consumer applications. There are two main categories of WSN applications [48]: monitoring applications and tracking applications.

Monitoring applications target a variety of domains: military (e.g. intrusion detection), business (e.g inventory), habitat (e.g. animal monitoring), public/industrial

(e.g. structural monitoring, factory monitoring, chemical monitoring), environment (e.g weather, temperature, pressure), health (e.g patient monitoring) - to name a few.

Tracking applications involve objects, animals, humans, and vehicles tracking.

PinPtr [39] is an experimental counter-sniper system developed to detect and locate shooters. The system consists of a large number of inexpensive sensor nodes communicating through an ad-hoc wireless network. After deployment, the sensor nodes synchronize their clocks, perform self-localization and wait for acoustic events. Their task is to detect muzzle blasts and acoustic shock waves and measure their time of arrival. The time-of-arrival measurements are delivered to the base station, where an algorithm calculates the shooter location estimate.

[40] describes an animal monitoring wireless sensor network application. The goal of the deployment was to gather data for studying the distribution and abundance of sea birds on an offshore breading colony on Great Duck Island, Maine. The nodes were supposed to detect the occupancy of the underground nesting burrows and the role of micro-climatic factors (temperature and humidity) in the birds habitat selection. The experiment lasted 4 months producing unique datasets for both systems and biological analysis.

An environmental monitoring case study is presented in [43]. A wireless sensor network was deployed that recorded 44 days in the life of a 70-meter tall redwood tree, at a density of every 5 minutes in time and every 2 meters in space. Each node measured air temperature, relative humidity, and photo-synthetically active solar radiation. The network captured a detailed picture of the complex spatial variation and temporal dynamics of the micro-climate surrounding a coastal redwood tree.

The volcanic monitoring case study presented in [30] is another example of an environmental monitoring application. The network was deployed on the Reventador Volcano in northern Ecuador and it consisted of 16 nodes equipped with seismoacoustic sensors spread over 3 km. The system routed the collected data through a multi hop network and over a long-distance radio link to an observatory, where a laptop logged the collected data. Over three weeks, it captured 230 volcanic events, producing useful data and becoming a proof of the performance of large scale sensor networks for collecting high resolution volcanic data.

PermaSense [41] is a joint computer science and geoscience project aiming in a first place at designing a set of wireless nodes for use in remote areas with harsh environ-

mental conditions as well as an architecture for their network and a self-organizing application capable of operating unattended for years. The project's second goal consists in gathering of environmental data that would help understand the processes that connect climate change and rock fall in permafrost areas. The geoscientists need this data in order to develop models for hazard assessment and the support of infrastructure maintenance. Their network was deployed in the Swiss Alps, each node measuring temperature and conductivity values indicative for rock moisture content and its phase state in the near surface layer.

Two structural health monitoring case studies are presented in [20]. A first WSN composed of 8 nodes was deployed on the railway bridge over the Kersjokk River in Sweden and performed strain measurements with the final goal of assessing whether the axle load of the bridge can be increased. A second deployment of a 6 node WSN was performed on the cable-stayed bridge (Stork Bridge) in Winterthur, with the purpose of measuring the ambient vibrations in the cables as well as the temperature and humidity.

Infections caused by antimicrobial-resistant bacteria (AMRB) account for an increasing proportion of healthcare-associated infections, particularly in high-risk units such as intensive care units and surgery; patients discharged to rehabilitation units often remain carriers of AMRB, contributing to their dissemination into longer-term care areas and within the community. The overall objective of the MOSAR (Mastering hOSpital Antimicrobial Resistance and its spread into the community) european project is to gain breakthrough knowledge in the dynamics of transmission of AMRB. A first step towards this goal is to assess the patient's usage of antibiotics and the contacts between patients and hospital personel. An experiment took place at the Maritime Hospital in Berk. It involved equiping everyone at the hospital with small communicating sensors designed to log all contacts and interractions between people over a six month period. These logs are cross-checked against biological samples and prescriptions for antibiotics.

The TubExpo [23] project focuses especially on the evaluation of contacts intensity and frequency between tuberculosis infected patients and health-care workers inside the Service of Infectious and Tropical Diseases (SMIT) of the Bichat-Claude Bernard hospital in Paris, France. To this end, the time spent by the health-care workers in each patient room of the unit was monitored during a three months period by means of a WSN. In this deployment, each room was equipped with a fixed sensor node plugged

to the power line whose task was to continuously listen to the radio medium. Each health-care worker was given an autonomous sensor node they had to carry during their presence in the unit. These mobile sensor nodes were programmed to periodically transmit a radio packet containing their identity.

2.2 Wireless Sensor Nodes

A sensor node, also known as a mote, is a node in a wireless sensor network that is capable of performing some processing, gathering sensory information and communicating with other connected nodes in the network.

Current research efforts in sensor node design aim at developing smaller and cheaper nodes with less power consumption.

Small sensor nodes are easier to manufacture with lower cost than large scale sensors. Being smaller, they can be deployed very closely to the target phenomena or sensing field and at a high density. This way, the shorter sensing range and lower sensing accuracy of each individual node are compensated for by the shorter sensing distance and large number of sensors around the target objects.

The intelligence of sensor nodes and the availability of multiple on board sensors also enhances the flexibility of the entire system.

Due to their small size and self-contained power supply, sensor nodes can be easily deployed into regions where replenishing energy is not available, including hostile or dangerous environments. The high node density enables system-level fault tolerance through node redundancy.

The miniaturisation of sensor nodes however comes at the price of severe-resource constraints. The nodes are constrained in computation, memory and most importantly-energy.

In what follows we will briefly describe a few of the motes available today.

The **weC** mote released in 1998 was developed within the SmartDust project at UC Berkley. It was built with a small 8-bit 4MHZ Atmel micro-controller with 512 bytes RAM and 8KB flash memory, having an active power consumption of 15mW, sleep state power consumption of 45 μW and a wake up time of 1 ms. The mote also incorporated a RFM TR1000 RF transceiver (36mW transmitting power and 9mW receiving power), an integrated printed circuit board antenna and temperature and

light sensors. All these hardware components were occupying a space approximatively of the diameter of a silver dollar.

The **Rene** motes were produced by Crossbow Technologies. The Rene mote appeared in 1999 had a similar hardware configuration as weC. The Rene 2 mote appeared in 2000 features an ATmega163 CPU with 1KB RAM and 16KB flash memory and reduced wake-up time of 36 μs. The Rene motes feature a modular design. The sensor board and motherboard were connected together via a 51 pin connector. Thus the basic sensor board with temperature and light sensors could be expanded via a 51 pin connector to include other sensor boards. This allowed a great deal of design flexibility and was subsequently used in most of the follow-on motes. The **Dot** mote released in 2000 had a similar hardware configuration as the Rene2 mote minus the 51 pin connector.

The first mote in the Mica family, the **Mica** mote [25] appeared in 2001. Its architecture allows for several different sensor boards, or a data acquisition board, or a network interface board to be stacked on top of the main processor/radio board. Its close resemblance to the layered mineral mica is at the origin of its name. It features an Atmel ATmega103L MCU with 128KB flash memory and 4KB RAM and a radio using RFM TR1000 supporting up to 40kbps for the same power consumption as the radio module on weC. The Mica platform has 3 sleep modes: idle (processor off), power-down (everything is off except the watchdog timer) and power save(similar to power-down but with an asynchronous timer running).

Mica2 and **Mica2Dot** appeared in 2002 both feature an improved microprocessor ATmega128L (with decreased energy consumption, i.e. 33mW active power and 75 μW sleep power) and improved radio modules (the CC1000).

The **MicaZ** appeared in 2003 is the last mote in the Mica family. Its main improvement is the use of the CC2420 radio module that supports IEEE802.15.4 and ZigBee protocols with data rate up to 250kbps.

The **TelosB** mote [35] released in 2004 features an MSP430 micro controller from Texas Instruments (3mW active power and 15 μW sleep power), a CC2420 radio, an on-board USB for easier interface with a PC, integrated humidity, temperature and light sensors and a 64 bit MAC for unique identification.

The **WSN430** platform released in 2005 is another example of sensor node. They are built on top of an MSP430 16bit micro-controller running at 8Mhz. Each platform includes a 6bytes DS2411 electronic registration, an external 1MB flash memory

ST M25P80. There are 2 types of WSN430 platforms based on the radio chip used: WSN430v13b (using a CC1100 radio chip) and the WSN430v14 (using the CC2420 radio chip).

Despite the variety of components, the architecture of the nodes remains the same: one micro-controller, an external memory, a radio device and several sensors. We may safely assume that future research efforts will follow the same type of architecture. Therefore all methodologies and optimizations proposed in the context of today's architectures will remain valid for years to come.

2.3 MAC Protocols for Wireless Sensor Networks

As stated previously, the miniaturisation of sensor nodes comes at the price of severe-resource constraints. The nodes are constrained in computation, memory and most importantly-energy.

The typical behavior of a WSN application involves 4 activities: sensor data acquisition, data storage, data processing and communication. The applications periodically sample the sensors, process, communicate and in the rest of the time the node components are in low-power states. The communication energy cost dominates the data storage and processing cost.

Energy conservation in a WSN maximizes network lifetime. It is addressed through efficient wireless communication, intelligent sensor placement for adequate coverage, security and efficient storage management, and through data aggregation and data compression.

The communication protocol in the case of WSNs consists of five standard protocol layers for packet switching: application layer, transport layer, network layer, data-link layer, and physical layer.

The implementation of the different layers in the protocol stack impacts the energy consumption, the end-to-end delay and the system efficiency. A trade-off is to be made between functional aspects optimization and power consumption. Traditional protocols for wireless networks do not address this tradeoff and are therefore not suitable for WSNs. The energy-efficient protocols for WSNs perform cross-layer optimization by supporting interactions across the protocol layers. Specifically, protocol state informa-

tion at a particular layer is shared across all the layers to meet the specific requirements of the WSN.

The transport layer ensures the reliability and quality of data at the source and the sink. It deals with aspects like variable reliability, packet-loss recovery and congestion control mechanisms. The network layer handles routing of data across the network from the source to the destination. The data-link layer is concerned with the data transfer between two nodes that share the same link and provides medium access control (MAC) and management. Finally, the physical layer provides an interface for transmitting bit streams over the physical-communication medium. It interacts with the MAC layer, performing transmission and reception, and modulation.

In this thesis, we are interested in optimizing the mapping of a MAC protocol over the physical device with the goal of minimizing the energy consumption. In what follows we will identify the main issues in MAC protocol design:

The main causes of energy waste in a MAC layer protocol are:

- **collision**: when 2 or more packets from different senders arrive at the destination node at the same time or just partially intersecting, the data is corrupt, the packets must be discarded; their re-transmission increases the energy consumption;
- **control packet overhead**: control packets transmission/reception incurs energy consumption too, therefore their number should be minimal;
- **idle listening**: occurs when a node is listening to an idle channel waiting for possible traffic;
- **overhearing**: occurs on a node that receives packets without being their destination (i.e. packets destined to other nodes);
- **over-emitting**: occurs on a node that transmits a packet when the destination node is not ready to receive;

There are two main groups of MAC protocols for wireless sensor networks: contention-based and TDMA based.

The contention-based protocols allow nodes to access independently the shared wireless medium. Nodes are not required to form a cluster. They are mainly based on the Carrier Sense Multiple Access (CSMA) or Carrier Sense Multiple Access/Collision

Avoidance (CSMA/CA). The basic idea is that when one node needs to send data it will compete for the wireless channel. Contention-based protocols require no coordination among the nodes accessing the channel. Colliding nodes will back off for a random duration of time before attempting to access the channel. These protocols inherit good scalability and they support node changes and new node inclusions. However, a node is not able to know when to switch its radio to a proper state. Sleeping mechanism becomes rather complex, and to avoid unnecessary energy consumption, while preserving desired latency and throughput, it requires control overhead to keep neighbor nodes synchronized. That is, idle listening, collisions, overhearing, and control packet overheads are the major sources of energy inefficiency.

In the TDMA-based protocols, nodes are often required to form a cluster. System time is divided into time slots and each of the nodes has assigned its own time slot and may access the shared medium only in this specific time slot. It allows avoiding collisions, idle listening, and it schedules sleep of the transceiver without additional overhead. However, such an approach also has certain of drawbacks. The difficulty for the cluster to dynamically change its frame length and time slot assignments, in the event of node changes or node inclusions, determines poor scalability and poor mobility. In addition, effective slot assignment in multi-hop networks is challenging. Moreover, demands of the cluster existence result in a complex inter-cluster communication. Furthermore, the TDMA-based protocol requires high quality time synchronization since the clock drift may lead to disastrous consequences.

We will describe some of the most known contention based protocols first, then some TDMA based protocols and finally some hybrid ones.

Sensor-MAC (S-MAC) [47] is a MAC based on IEEE 802.11 aiming at saving energy. To this end, it uses 3 techniques : sleep schedules, local synchronized virtual clusters and message passing.

The protocol tackles the problem of *idle listening* by defining sleep schedules. It divides the time into frames whose length is determined by applications and within each frame there is a work stage and a sleep stage. Each node goes to sleep for some time, and then wakes up and listens to see if any other node wants to talk to it. During sleep, the node turns off its radio, and sets a timer to awake itself later. Neighboring nodes must be aware of each other's schedules in order to know when to send packets and also re-synchronize periodically to remedy their clock drift.

In order to reduce control overhead, it is preferable that neighboring nodes have the same sleep schedule. Neighboring nodes therefore form virtual clusters to set up a common sleep schedule. If a node resides in two adjacent virtual clusters, it must wake up at listen periods of both clusters.

S-MAC features the same contention mechanism as IEEE 802.11, i.e., using RTS (Request To Send) and CTS (Clear To Send) packets. The node who first sends out the RTS packet wins the medium, and the receiver will reply with a CTS packet.

Neighboring nodes need to periodically update their schedules to prevent long-time clock drift. This is achieved through periodic short SYNC packets containing the address of the sender and the time until it turns to sleep.

Collision avoidance is performed by S-MAC by using virtual and physical carrier sense and RTS/CTS packets. The listen interval is divided into two parts: one for receiving SYNC packets and the other one for receiving RTS packets. Each part is further divided into many time slots for senders to perform physical carrier sense. Under this approach, when a node wants to send a packet (either SYNC or RTS) it chooses a random slot in the corresponding part of the listening interval. If the medium is free until that slot, the medium is considered free. Virtual carrier sense is implemented as follows: every packet contains a duration field, so when a node receives a packet destined for another node, it knows for how long it must be quiet. It implements a timer and knows it cannot send anything until this timer reaches the value 0. A node sends a packet only when both virtual and physical carrier sense indicate a free medium.

An important feature of S-MAC is the concept of message-passing where long messages are divided into frames and sent in a burst. With this technique, one may achieve energy savings by minimizing communication overhead at the expense of unfairness in medium access.

Timeout-MAC(T-MAC) [45] is an adaptive energy-efficient MAC protocol for WSNs that minimizes idle listening. Similar to S-MAC, there are active periods and sleep periods in a time-frame. T-MAC introduces an adaptive duty cycle. Unlike for S-MAC, a T-MAC active period ends if there is no activity for a time period of Ta. Ta is the minimum listening time in the time-frame. T-MAC reduces the time in active state compared with S-MAC as seen in figure 2.1.

B-MAC [27] is a another MAC protocol for WSNs. Its design goals are: low power operation, effective collision avoidance, simple implementation, efficiency in channel

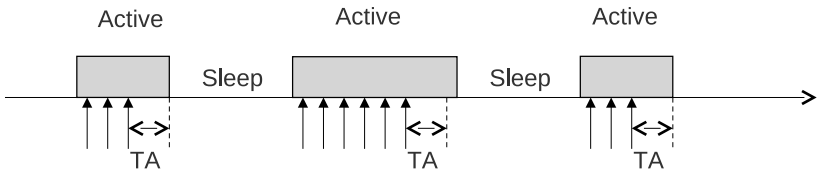


Figure 2.1: T-MAC Schedule

utilization, reconfigurability and scalability. The protocol contains a basic set of media access functionalities that are configurable by higher layers according to the target metric to be optimized (power consumption, latency, throughput, fairness or reliability). In B-MAC, the channel arbitration is achieved through clear channel assessment and packet backoffs. Reliability is insured through the use of link layer acknowledgments whereas the low power goal is achieved through low power listening. The complete description of the protocol's behavior will further be illustrated in section 3.1.

WiseMAC [19] is a protocol that uses non-persistent CSMA and preamble sampling in order to reduce idle listening.

All nodes in a network sample the medium with a common period checking for activity. Their relative schedule offsets are independent. If a node finds the medium busy after it wakes up and samples the medium, it continues to listen until it receives a data packet or the medium becomes idle again. A preamble precedes each packet with the purpose of intersecting the destination node's sampling time interval thus determining it to receive the packet. A simple choice for the preamble length is the length of the sampling period.

In order to reduce the preamble-length, WiseMAC nodes learn the sleep schedule of their neighboring nodes. Thus when a node transmits a packet, it schedules the preamble length so that it the destination's sampling time corresponds to the middle of the sender's preamble. The preamble start also comprises a random factor meant to avoid collisions when two transmitters address the same destination and it also takes into account the possible clock drift between the sender and the destination, as

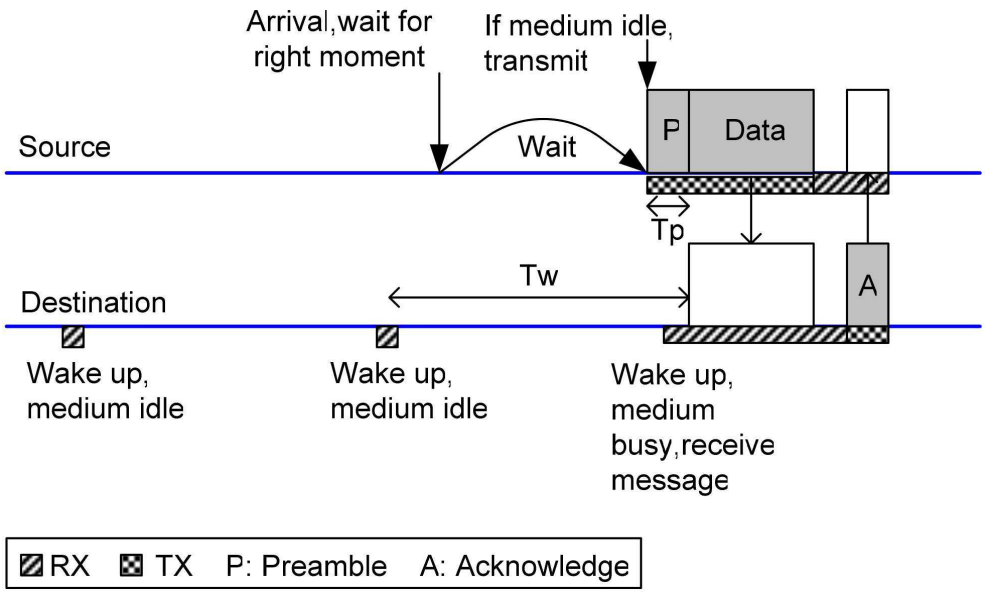


Figure 2.2: WiseMAC communication

seen in figure 2.3. The reliability of the communication is achieved through link layer acknowledgements. The sleep schedule information is piggy-backed on these packets.

The authors in [2] propose a method for improving the preamble-sampling protocols by replacing the continuous preamble by a series of small frames called micro-frames. These micro-frames contain information about the destination node and this makes it possible for nodes that are not the destination to switch their radio off to avoid receiving irrelevant frames and improve energy saving. Another solution proposed in [3] is to have the preamble composed of duplicate copies of the data. The advantage of data-frame preamble (DFP) is that the node that wakes up to check the channel immediately receives the data, so it does not need to wake up again to receive the data.

In contrast to contention-based MAC, TDMA based protocols offer an inherent collision-free scheme by assigning unique time slots for every node to send or receive data. First, interference between adjacent wireless links is guaranteed to be avoided and this reduces the energy waste due to packet collisions. Second, TDMA can solve the hidden terminal (when two nodes not hearing each-other transmit simultaneously to a destination node) problem without the extra message overhead because neighboring nodes transmit at different time slots.

μMAC [4] is a TDMA based protocol aiming at obtaining high sleep ratios while preserving the message latency and reliability at an acceptable level. This is achieved by adoption of a schedule-based approach to access the shared medium. The protocol uses a single time-slotted channel, i.e. the time is divided into time slots. The protocol operations alternate between contention and contention-free periods. They are both composed of time-slots. The contention period is used to organize the network and to initialize transmission sub-channels. This is achieved by packet-exchange in the slots of the contention period, potentially subject to collisions. A sub-channel is formed by a number of approximately equal spaced contention free slots. The contention-free period is used to transfer data between nodes. Since the slots are reserved, no collisions can occur. In this protocol, the contention periods responsible for network organization incur a large overhead, and should not be frequent. But this makes it hard to adapt to frequent network organization changes.

DEE-MAC (dynamic energy efficient TDMA based MAC) [14] proposes a clustering-based TDMA approach for energy consumption reduction. The protocol's operation consists of rounds. Each of the rounds includes a cluster formation phase and a transmission phase. Each cluster is dynamically formed based on the remaining power as all nodes contend (using non-persistent CSMA) to be the cluster head. Based on the power-level advertised, a node with the highest power-level will be elected as a cluster-head. The transmission phase consists of a number of sessions, each of which contains a contention period and a data transmission period. During the time of the contention period, each node keeps their radio on and indicates interest to send a packet to the cluster head. After this period, the cluster head knows which node has data to transmit and it builds a TDMA schedule then broadcasts it to all nodes. Each node is assigned with one data slot in each session. Based on the broadcasted schedule each of the nodes, having data to receive or send, knows when to be awake. Clustering and TDMA based schemes present a rational solution to reduce the cost of idle listening in large-scale wireless sensor networks. However, the power of cluster head is easily depleted so the network partitions happen easily.

Recently hybrid protocols appeared, combining the advantages of both contention-based and TDMA-based protocols. These protocols divide the access channel into two parts. Control packets are sent in the random access channel, and data packets are transmitted in the scheduled channel. The control channel schedules the data access.

Zebra-MAC(Z-MAC)[36] is a hybrid protocol that combines the strengths of TDMA and CSMA. Under low contention conditions, it behaves like CSMA, and under high contention, like TDMA. In Z-MAC, a time slot assignment is performed at the time of deployment. After the slot assignment, each node reuses its assigned slot periodically in every predetermined period, called frame.This means there can be more than one owner per slot. A node assigned to a time slot is the owner of that slot and the others the non-owners of that slot. Unlike TDMA, a node may transmit during any time slot in Z-MAC but performs carrier-sensing and transmits a packet only when the channel is clear. An owner of a slot always has higher priority over its non-owners in accessing the channel. This is implemented by adjusting the initial contention window size in such a way that the owners are always given earlier chances to transmit than non-owners. The goal is that during the slots where owners have data to transmit, Z-MAC reduces the chance of collision since owners are given earlier chances to transmit and their slots are scheduled a priori to avoid collision, but when a slot is not in use by its owners, non-owners can steal the slot.

Conclusion All these protocols have in common the fact that they require precise timings within the software application code in order to drive the radio device including constraints coming from the device itself (calibration, delays to switch between mode, time to wakeup from a particular low power state). Mapping such a software onto a physical radio device is not straightforward.

2.4 Device Modelling and Driver Generation

The desired behavior of embedded systems is usually subject to time constraints and it involves interactions with a set of physical devices or components, each with its own individual and specific constraints. The problem of mapping software specification on top of hardware devices has traditionally been split on one hand on abstraction layers to manage complexity, portability and reusability and on the other hand on tools that could support code generation and application specific embedded software.

The first approach to abstraction layer involves domain specific languages. These languages offer constructs and abstractions specific to a domain so that low level and implementation specific parts of the system can be generated.

Devil [32] is an Interface Definition Language for describing hardware functionalities which allows the high-level definition of the communication with a device. A device can be described by three layers of abstraction: *ports*, *registers* and *device variables*. Ports are abstract physical addresses that hide the type of mapping of the device: port-mapped or memory mapped. Registers define the granularity of interaction with a device (their size in bits must be specified) and they usually have 2 attached ports: one for reading and one for writing. Device variables are independent values grouped within a single register in order to reduce the number of I/O operations. A compiler automatically checks the consistency of a Devil definition and generates efficient low-level code. The IDL proved useful for the description of a wide variety of drivers: mouse, sound, DMA, Ethernet , video etc.

ProGram [34] is a grammar based protocol specification language which is used for modeling the behavior of a software/hardware interface (device driver) independently of the architecture. Specifications in ProGram deal with sequences of allowed events rather than state transitions as in the finite-state machine model. A ProGram description is synthesized into an untimed extended state machine that is served as input for the architecture mapping procedure. The processor specific characteristics and OS kernel functionalities are captured in two separate libraries: ProcLib and OSLib respectively. From the extended finite state machine and the 2 libraries the mapping procedure generates architecture specific code.

NDL [15] is a language for device driver development that provides high-level constructs for device programming, describing the driver in terms of its operational interface. The NDL declarations are designed to resemble the specification document for the device it controls. An NDL driver is typically composed of a set of register definitions, protocols for accessing those registers, and a collection of device functions. The compiler translates register definitions and access protocols into an abstract representation of the device interface. Device functions are then translated into a series of operations on that interface. Device drivers are systems-level code that interact directly with the operating system. The NDL compiler generates C that makes appropriate operating system calls. Platform specific functions are provided through compiler libraries and device templates (platform dependencies are minimal). NDL's main abstraction is a representation of the state of the peripheral device being controlled.

In [6] a solution for global resource control in embedded systems is proposed. Unlike the previously mentioned approaches that deal with devices individually, [6] takes a global view on the platform. This way, problems of concurrency and mutual exclusion for shared resources like buses can easily be solved. A global view of the system also allows the implementation of a global power-consumption policy at the level of the entire platform. In this approach, each device driver is described as a Mealy automaton. These automata are made controllable (i.e. the absence of an additional input may prevent the automaton from changing states). A controller automaton C is built in such a way that global properties are ensured. The controller automaton and the controllable device automata are programmed in some synchronous language. Finally, the control layer is obtained by compiling the parallel composition of these automata into a single piece of sequential C code.

Communication protocols abstractions and other DSL approach often rely on timed automaton and finite state machines. A timed automaton is a finite automaton, *i.e.*, a model of behavior composed of a finite number of states, transitions between these states according to the inputs received and possibly actions to be performed, extended with real-valued variables called clocks. These variables model the logical timers in the system, initialized to zero at system start and increasing synchronously with time. The transitions between states can be conditioned by logical expressions formed with clock variables. The latter can be reset upon state transitions.

Definition 1 *A timed automaton $\mathcal{A}$ is a tuple $\langle N, l_0, \mathcal{C}, \Sigma, E, I\rangle$, where : N is a finite set of locations (nodes); l_0 is the initial location; $\mathcal{C}$ is a finite set of real-valued variables, namely the clocks; Σ is a finite alphabet of symbols standing for actions; $\mathcal{B}(\mathcal{C})$ represents the set of clock constraints; $E \subseteq N \times \mathcal{B}(\mathcal{C}) \times \Sigma \times 2^{\mathcal{C}} \times N$ is the set of edges and $I : N \to \mathcal{B}$ assigns invariants to locations.*

A *clock constraint* is a conjunction of atomic constraints of the form $x \otimes n$ or $x - y \otimes n$ where $\otimes \in \{<, \leq, =, >, \geq\}$ and $n \in \mathbb{N}$. An edge is a tuple of the form $\langle l, g, a, r, l'\rangle$ ($l \xrightarrow{g,a,r} l'$), where l is the start location, l' is the destination location, g is the set of guards, a is the set of actions and r is the set of clocks to be reset upon taking this transition. The location invariants are clock constraints defining the time interval in which the automaton may remain inside the corresponding location. In the sequel,

we will consider timed automata where locations have no invariant and the exit from a state will be specifically expressed through clock constraints of the form $clock = value$.

The priced extension of timed automata extends the classical model $\langle N, l_0, \mathcal{C}, \Sigma, E, I \rangle$ with a cost function $P : (E \cup N) \rightarrow \mathbb{N}$ that assigns fixed costs to transitions and costs per unit time to locations respectively. The general formalism of timed automata introduced in [1] allows expressing the expected software behavior of a device in the form of a finite state automaton extended with real-valued variables called clocks. The particular constraints of physical devices themselves can also be expressed in this form. For the priced extension of timed automata several interesting problems have been proposed and solved: optimal reachability in closed systems [29], optimal infinite schedules for closed systems [9] or the optimal control synthesis for automata with acyclic control graphs [44]. Considering only DAG is a strong limitation since several cases like MAC protocols present cycles in their control graph.

3

Hardware Software Interface: Cross Layer Mapping Optimization

In this chapter we introduce a timed automata based model for describing the behavior of a software protocol in section 3.1. We introduce a finite state machine model for describing the behavior of a hardware device in section 3.2. We classify software states as either *fixed*, which correspond to a unique state of the hardware automaton, or *free*, for which the mapping is left at the programmer's choice. We investigate the problem of mapping the free states of such a software protocol onto a hardware device. We prove the problem to be NP-complete in subsection 3.3.2 and we propose a heuristic in subsection 3.3.3. From the software description and the hardware paths determined by the heuristic, we derive code skeletons suitable for two types of wireless sensor networks operating systems in section 3.4.

3.1 Software Protocol Description

3.1.1 Definition and Formalism

This section presents the derived model from priced timed automata that we use for modeling the behavior of a software protocol.

Figure 3.1 illustrates an automaton. It is a strictly sequential model of computation composed of states ($S1, S2, S3$ in figure 3.1) and transition between them, having ex-

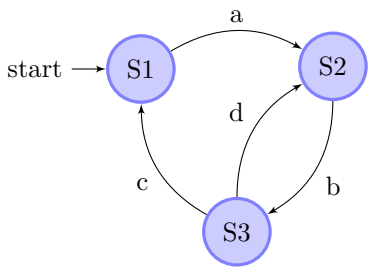


Figure 3.1: Automaton example.

actly one state active at a given moment. The transitions are ordered 3-tuples composed of a start state, a label and a destination state ($S1 \xrightarrow{a} S2$, $S2 \xrightarrow{b} S3$, $S3 \xrightarrow{c} S1$, $S3 \xrightarrow{d} S2$ in figure 3.1). The labels have different meaning according to the adopted formalism: they can correspond to an action, to an instruction, a communication. There can be one or more initial states ($S1$ in the example). The execution of this model of computation corresponds to a sequence of states starting with one of the initial states and such that 2 consecutive states are linked through a transition. $S1 \xrightarrow{a} S2 \xrightarrow{b} S3 \xrightarrow{c} S1$ is an example of such an execution for figure 3.1.

The general model of automata can be extended in order to correctly reflect the behavior of the system to be modeled.

The transitions can be labeled with input/output signals modeling the interaction between the automaton and its environment. The significance of a transition of the form $S1 \xrightarrow{a/b} S2$ is that the occurrence of the input signal a while the automaton is in the state $S1$ triggers the transition of the automaton to the state $S2$ with the emission of the signal b.

Another possible extension of the automata is to label its transitions with conditions/actions. The significance of a transition of the form $S1 \xrightarrow{condition/action} S2$ is that the condition must be true in order for the transition to be taken and the corresponding action is executed along with the transition.

The transitions of a software protocol can be triggered by hardware or software signals. A particular type of software signal is represented by timer events. These signals occur when a time interval elapses.

The general formalism of timed automata introduced in [1] allows expressing the expected software behavior of a physical device in the form of a finite state automaton

extended with real-valued variables called clocks. These variables model the logical timers in the system, initialized to zero at system start and increasing synchronously with time. The transitions between states can be conditioned by logical expressions formed with clock variables. The latter can be reset upon state transitions.

The definition of Timed Automata as presented in [5] is as follows:

Definition 2 *A timed automaton $\mathcal{A}$ is a tuple $\langle N, l_0, \mathcal{C}, \Sigma, E, I\rangle$, where : N is a finite set of locations (nodes); l_0 is the initial location; $\mathcal{C}$ is a finite set of real-valued variables, namely the clocks; Σ is a finite alphabet of symbols standing for actions; $\mathcal{B}(\mathcal{C})$ represents the set of clock constraints; $E \subseteq N \times \mathcal{B}(\mathcal{C}) \times \Sigma \times 2^{\mathcal{C}} \times N$ is the set of edges and $I : N \to \mathcal{B}(\mathcal{C})$ assigns invariants to locations.*

A *clock constraint* is a conjunction of atomic constraints of the form $x \otimes n$ or $x - y \otimes n$ where $\otimes \in \{<, \leq, =, >, \geq\}$ and $n \in \mathbb{N}$. An edge is a tuple of the form $\langle l, g, a, r, l'\rangle$ ($l \xrightarrow{g,a,r} l'$), where l is the start location, l' is the destination location, g is the set of guards, a is the set of actions and r is the set of clocks to be reset upon taking this transition. The location invariants are clock constraints defining the time interval in which the automaton may remain inside the corresponding location.

In order to model the behavior of a software automaton, we consider a timed software automaton extended with integer variables and boolean variables, where the transitions between its states occur on events/signals from other layers in the network stack or on timer events and they can be conditioned by guards formed with clock variables (the clock constraints), boolean variables and integer variables. Clock variables may be reset on transitions, while integer variables may be incremented/decremented/reset upon transitions.

More formally, the definition of a Software Timed Automata is as follows:

Definition 3 *The software timed automaton $\mathcal{A}$ is a tuple $\langle N, l_0, \mathcal{C}, \beta, \vartheta, \Sigma, E, I\rangle$, where N is a finite set of locations (nodes); l_0 is the initial location; $\mathcal{C}$ is a finite set of real-valued variables, namely the clocks; Σ is a finite alphabet of symbols standing for signals; ϑ is a finite set of integer-valued variables; β is a finite set of boolean variables; $\mathcal{B}(\mathcal{C})$ represents the set of clock constraints; $\mathcal{B}(\vartheta)$ represents the set of integer variables constraints; $\mathcal{B}(\beta)$ represents the set of boolean constraints and $E \subseteq N \times \mathcal{B}(\beta) \times \mathcal{B}(\mathcal{C}) \times \mathcal{B}(\vartheta) \times \Sigma \times 2^{\mathcal{C}} \times 2^{\vartheta} \times 2^{\vartheta} \times 2^{\vartheta} \times N$ is the set of transitions.*

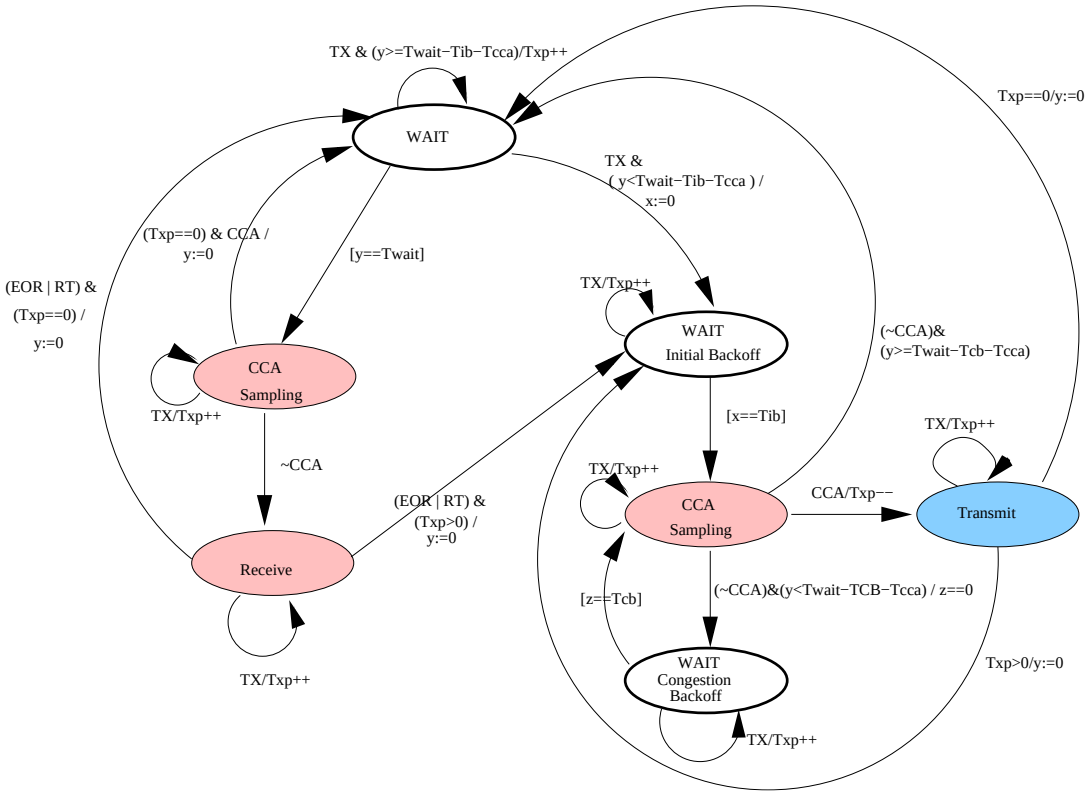


Figure 3.2: B-MAC protocol description in the form of a timed automaton.

Thus, an edge is a tuple of the form $\langle l, g, s, r, \text{inc}, \text{dec}, \text{res}, l' \rangle$ where l is the start location, l' is the destination location, g is the set of guards, s is a signal, r is the set of clocks to be reset, *inc* is the set of integer variables to be incremented, *dec* is the set of integer variables to be decremented and *res* is the set of integer variables to be reset upon taking this transition.

Two types of events can trigger a transition in the software automaton: either a signal or a timer event. A signal can originate from the hardware or other applications. A timer event has the form $\text{clock}_{\text{var}} = \text{value}$, where $clock_{var}$ is a clock variable and value $\in \mathbb{N}$. The two types of events are mutually exclusive. If two such triggers arrive at the same time, they will be sequentialized by the micro-controller. The transitions that they trigger can be conditioned by guards, which are conjunctions of boolean constraints, integer variables constraints and clock constraints.

The set of possible boolean constraints $\mathcal{B}(\beta)$ comprises boolean expressions formed with boolean variables. The set of possible relational constraints involving integer

valued variables are of the form var $\otimes n$, where $\otimes \in \{<, \leq, =, \geq, >\}$ and $n \in \mathbb{N}$. The set of clock constraints comprises conjunctions of relational expressions of the form $x \otimes n$ or $x - y \otimes n$ where $\otimes \in \{<, \leq, >, \geq\}$, $x, y \in \mathcal{C}$ and $n \in \mathbb{N}$.
To summarize: A transition has the form $\langle l, t, g, a, l' \rangle$ ($l \xrightarrow{t,g,a} l'$), where t stands for the triggering condition of the transition (either $\text{clock}_{\text{var}}$ = value or signal), g stands for the guards while a stands for the actions to be performed (clock resets and integer variables increment/decrement/reset).

3.1.2 Application on B-MAC protocol

Figure 3.2 illustrates the timed automaton representation of the B-MAC protocol [27], a well known MAC protocol for WSNs.

Although we focus here on the description of a MAC protocol, the timed automata approach is general. It can be applied to any software protocol intended to control a physical device whose behavior can be expressed as a finite state machine such as sensing devices, flash memory, RAM, SPI bus, DSPs, FPGAs, etc.

B-MAC's design goals are: low power operation, effective collision avoidance, simple implementation, efficiency in channel utilization, reconfigurability and scalability. The protocol contains a basic set of media access functionalities that are configurable by higher layers according to the target metric to be optimized (power consumption, latency, throughput, fairness or reliability).

In B-MAC, the channel arbitration is achieved through clear channel assessment and packet backoffs. Reliability is insured through the use of link layer acknowledgments whereas the low power goal is achieved through low power listening.

The finite state machine in figure 3.2 captures a possible configuration of the protocol which uses the CSMA and low power listening features of B-MAC.

The Low Power Listening (LPL) feature of the protocol consists in periodically driving the radio into the receive physical device state in order to sample the medium, checking for activity, thus performing the Clear Channel Assessment (CCA). If the medium is found busy, the radio is kept in receive state until a packet is received or a timer expires, after which it is turned back off. If on the contrary, no activity is detected on the channel, the radio is turned back off until the next medium checking event. This sequence of states is illustrated on the left part of figure 3.2. From the default WAIT state, the MAC protocol switches to the CCA Sampling state when the

clock variable y reaches the T_{wait} value. The CCA Sampling state corresponds to the RX-receive state of the device. The transitions from the CCA Sampling state according to the check result are: in case of a busy channel (¬CCA) to the Receive state while in case of a free channel (CCA), either back to the default WAIT state if there are no packets pending transmission, or to the WAIT Initial Backoff state if there exist packets waiting for transmission. The transmission pending integer variable (T_{xp}) records the number of queued packets. In case of a free medium, both possible transitions reset the y clock variable. The transition from the Receive state back to the WAIT state occurs after a successful packet reception signaled by the End Of Receive signal (EOR) from the radio device, or upon expiration of a second timer RT – Receive Timer – in case no packet is received (a false positive) as long as there are no packets pending ($T_{\text{xp}} = 0$). Otherwise, a transition to WAIT Initial Backoff is performed, initiating the Carrier Sense Multiple Access (CSMA) process for packet transmission.

The protocol behavior on packet transmission is illustrated in the right side of figure 3.2. The channel arbitration is achieved through CSMA, that starts with an initial backoff followed by successive congestion backoffs in case of unsuccessful clear channel assessment. Upon a transmission request from an upper layer (TX signal) occurring in the WAIT state, if there is sufficient time to perform the CCA without missing the LPL periodic CCA Sampling ($y < T_{\text{wait}} - T_{\text{ib}} - T_{\text{cca}}$), the protocol switches to an initial wait state (WAIT Initial Backoff). The transition is conditioned by a guard which is a logical conjunction of the signal TX triggered by the application and the clock constraint specified before. After the elapse of T_{ib} (time initial backoff-random value chosen in a bounded interval) time units, a transition to CCA Sampling state is enabled. The result of the channel activity check determines the next transition: either to the TX state if the medium was found idle - the CCA boolean variable in the figure which encodes the state of the Clear Channel Assessment procedure - or to a next wait state - WAIT Congestion Backoff- in case of a busy medium (¬CCA) and if sufficient time remains for the congestion backoff time to elapse and a new CCA to be issued without missing the LPL CCA periodic sampling. From the Transmit state, the protocol switches after a successful transmission modeled by EOT- end of transmission signal- either back to the default WAIT state if there are no packets pending or to WAIT Initial Backoff otherwise, both transitions resetting the clock variable y. From the WAIT Congestion Backoff state, the protocol switches back to the CCA state when the clock variable z

reaches the T_{cb} value also randomly chosen in a bounded interval. The occurrence of a TX - transmit signal from the upper layer in any of the automaton's states keeps the current state but increments the T_{xp} variable, except for the WAIT state where the self-loop or the transition to the WAIT Initial Backoff are conditioned by disjoint clock constraints ($y >= T_{\text{wait}} - T_{\text{ib}} - T_{\text{cca}}$ and $y < T_{\text{wait}} - T_{\text{ib}} - T_{\text{cca}}$ respectively).

The states appearing in the MAC protocol automaton can be divided into 2 categories: *fixed states* (Receive, CCA Sampling, Transmit) that corresponds to a unique state in the physical device automaton, and *free states* (WAIT, WAIT Initial Backoff, WAIT Congestion Backoff) whose mapping is left at the designer's choice.

3.2 Hardware Protocol Description

The behavior of most embedded components can usually be expressed in the form of a finite state machine: Flash memory, RAM memories, SPI bus, DSPs, radio devices and so on.

We will consider here a type of physical device whose behavior can be expressed as a finite state machine in which states can have either a fixed duration (*transitional* states), or a variable duration but with a lower bound constraint (*non-transitional* states). A cost per unit time (*e.g.*, energy) will also be associated to each state.

Definition 4 *The device finite state machine $\mathcal{F}$ is a tuple $\langle N, l_0, E, \Sigma, type, t_{\mathsf{min}}, cost\rangle$, where: N is a set of finite location (nodes); l_0 is the initial location; Σ is a finite alphabet of symbols standing for input commands; $E \subseteq N \times \Sigma \times N$ is the set of edges; $type : N \rightarrow \{\mathsf{transitional}, \mathsf{nontransitional}\}$ is a function assigning types to nodes; $t_{\mathsf{min}} : N \rightarrow \mathbb{N}$ is a function assigning minimum time duration to states and $cost : N \rightarrow \mathbb{N}$ is a function assigning cost per unit time to states.*

In the sequel, we will illustrate this definition by taking the CC1100 RF transceiver[42] as an example. The CC1100 is a low-cost sub- 1 GHz transceiver designed for very low-power wireless applications. CC1100 provides extensive hardware support for packet handling, data buffering, burst transmissions, clear channel assessment, link quality indication, and wake-on-radio. The main operating parameters and the 64- byte transmit/receive FIFOs of CC1100 can be controlled via an SPI interface. In a typical system, the CC1100 will be used together with a micro-controller and a few additional passive components.

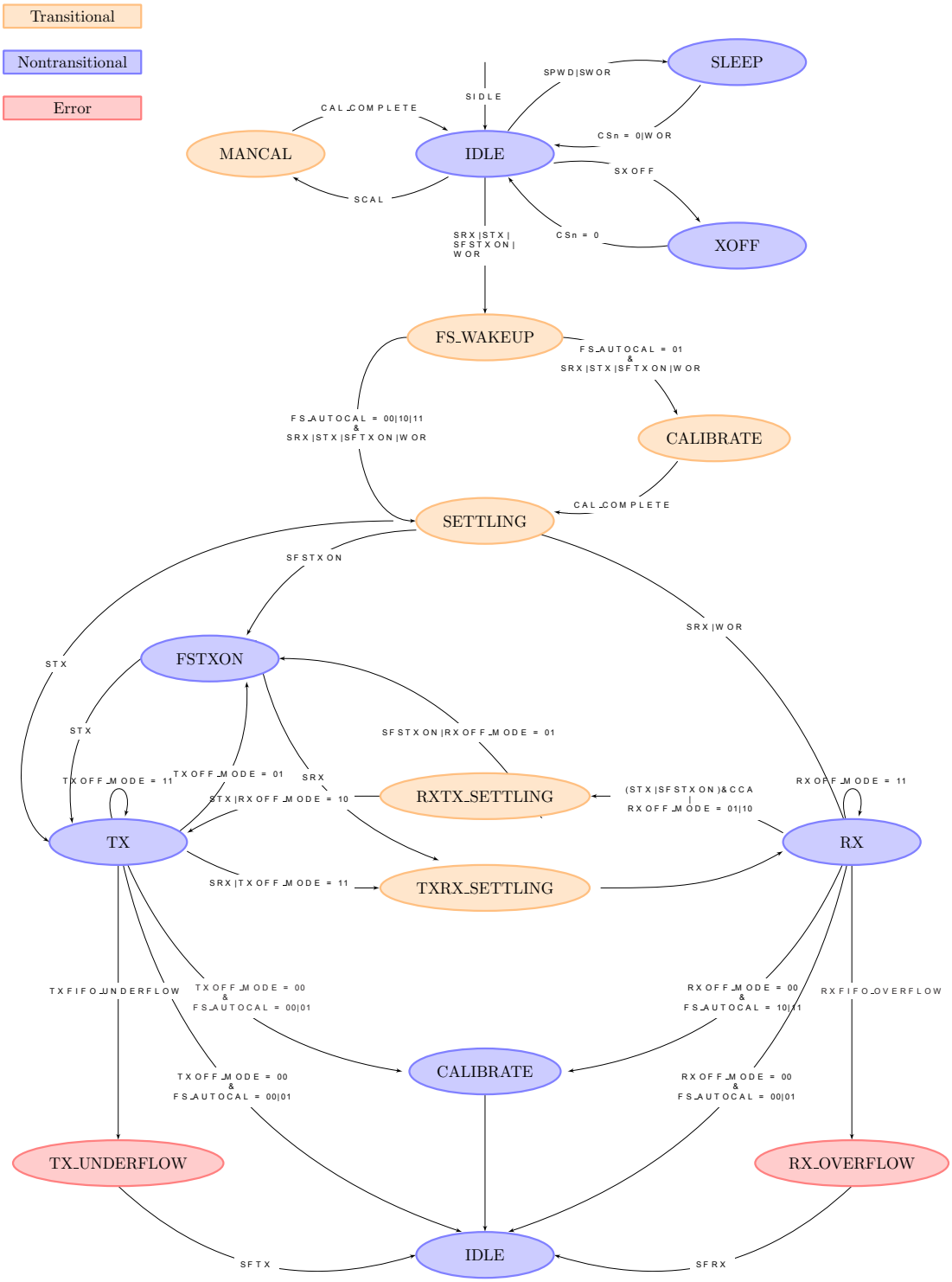


Figure 3.3: CC1100 Radio Finite State Machine.

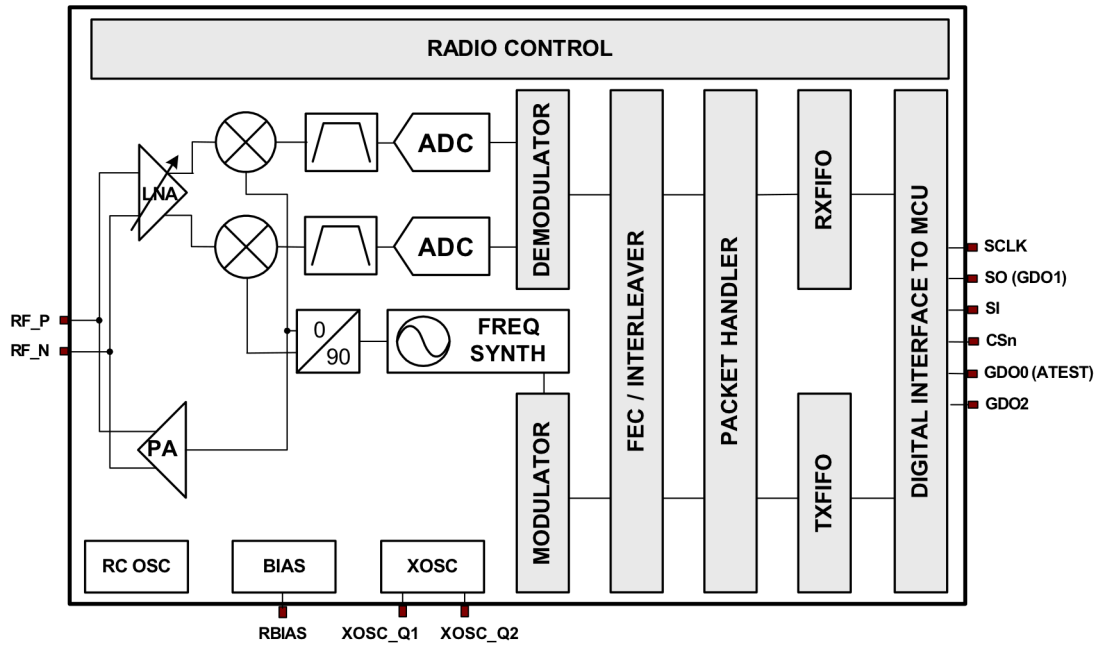


Figure 3.4: CC1100 Simplified Block Diagram (taken from CC1100 datasheet)

The simplified block diagram of CC1100 is illustrated in figure 3.4. The radio features a low-IF receiver. The received RF signal is amplified by the low noise amplifier (LNA) and down-converted in quadrature (I and Q) to the intermediate frequency (IF). The I/Q signals are digitized by the ADCs. Automatic gain control (AGC), fine channel filtering and demodulation bit/packet synchronization are performed digitally. The transmitter part of CC1100 is based on direct synthesis of the RF frequency. The frequency synthesizer includes a completely on-chip inductance-capacitance voltage controlled oscillator (LC VCO) and a 90 degree phase shifter for generating the I and Q LO signals to the down-conversion mixers in receive mode. A crystal is to be connected to XOSC_Q1 and XOSC_Q2. The crystal oscillator generates the reference frequency for the synthesizer, as well as clocks for the ADC and the digital part. A 4-wire SPI serial interface is used for configuration and data buffer access. The digital baseband includes support for channel configuration, packet handling, and data buffering.

The CC1100 has two dedicated configurable pins (GDO0 and GDO2) and one shared pin (GDO1) that can output internal status information useful for control software. These pins can be used to generate interrupts on the MCU. They can be

configured to raise an interrupt for various types of events: when sync word has been received/transmitted, when a complete packet has been received/transmitted, when RX FIFO is filled at or above the RX FIFO threshold, when the TX FIFO is filled at or above the TX FIFO threshold, etc.

CC1100 has a built-in state machine that is used to switch between different operational states (modes). Figure 3.3 shows the state diagram of the CC1100 device while table 3.1 summarizes the description and energy consumption parameters of its states.

State	Current Consumption	Characteristics
SLEEP	400nA	Lowest power mode, voltage regulator to digital part off, most register values retained.
XOFF	$160\mu A$	Crystal oscillator(XOSC) off, voltage regulator to digital part on, all other modules in power down.
IDLE	1.6 mA	Only voltage regulator to digital part and crystal oscillator running
FS_WAKEUP	8.2 mA	Frequency synthesizer is turned on.
MANCAL	8.2 mA	Manual calibration(through command strobe) of the frequency synthesizer.
CALIBRATE	8.2 mA	Auto calibration of the frequency synthesizer according to the configuration settings.
SETTLING	8.2 mA	Frequency synthesizer settles to the correct frequency.
RXTX_SETTLING	8.2 mA	
TXRX_SETTLING	8.2 mA	
FS_TXON	8.2 mA	Frequency synthesizer is on ready to start transmitting.
TX	13.5mA at -6dBm output 16.9mA at 0dBm output 30.7mA at +10dBm output	Transmit state
RX	from 14.4mA to 15.4mA	Receive state

Table 3.1: Description and energy consumption parameters for the states of CC1100

The communication between the micro-controller(MCU) and the radio is achieved over the 4-wire SPI interface (SI,SO,SCLK and CSn) with the CC1100 as the slave. All transactions start with a header byte containing a read/write $R/\bar{W}$ bit, a burst access bit and a 6-bit address ($A_5 - A_0$).

The communication is initiated by the micro-controller by pulling the chip select CSn pin low (CSn must be kept low during SPI transfers). The micro-controller must wait until the SO pin goes low (CHIP_RDYn), indicating that the radio crystal is running. Whenever the micro-controller transmits a byte over the SI pin, the radio responds by sending the chip status byte over the SO pin. Bit 7 is the CHIP_RDYn signal, bits 6:4 code the radio device state, while bits 3:0 contain the FIFO bytes available (in the RX or TX FIFO respectively according to the type of operation $R/\bar{W}$ initiated by the MCU).

The CC1100 contains several 8 bit registers with different functionalities.

Registers located on the SPI addresses from 0x00 to 0x2E are **configuration registers**. The CC1100 has its own configuration software, the SmartRF Studio software but it can also be configured "manually" by writing the desired configuration values in the configuration registers using the SPI interface. These registers can be both written and read. An example of such a register is MCSM1 at address 0x17. Bits 7:6 are reserved, bits 5:4 configure the CCA_MODE (behavior for the automatic clear channel assessment with the following possibilities: 00-always, 01-if RSSI below threshold, 10-unless currently receiving a packet, 11-if RSSI below threshold unless currently receiving a packet), bits 3:2 configure the RXOFF_MODE (the next state of the radio device upon packet reception with the following possibilities: 00-IDLE, 01-FSTXON, 10-TX, 11-Stay in RX) while bits 1:0 configure the TXOFF_MODE (the next state of the radio device upon packet transmission with the following possibilities: 00-IDLE, 01-FSTXON, 10-Stay in TX, 11-RX).

Registers located located in the address range 0x30-0x3D are multiplexed using the burst bit in the header byte of the SPI transfer: burst bit 0 selects **command strobes** while burst bit 1 selects **status registers**.

Address	Strobe Name	Description
0x30	SRES	Reset chip.
0x31	SFSTXON	Enable and calibrate frequency synthesizer (if MCSM0.FS_AUTOCAL=1). If in RX (with CCA): Go to a wait state where only the synthesizer is running (for quick RX / TX turnaround).
0x32	SXOFF	Turn off crystal oscillator.
0x33	SCAL	Calibrate frequency synthesizer and turn it off. SCAL can be strobed from IDLE mode without setting manual calibration mode (MCSM0.FS_AUTOCAL=0)
0x34	SRX	Enable RX. Perform calibration first if coming from IDLE and MCSM0.FS_AUTOCAL=1.
0x35	STX	In IDLE state: Enable TX. Perform calibration first if MCSM0.FS_AUTOCAL=1. If in RX state and CCA is enabled: Only go to TX if channel is clear
0x36	SIDLE	Exit RX / TX, turn off frequency synthesizer and exit Wake-On-Radio mode if applicable.
0x38	SWOR	Start automatic RX polling sequence (Wake-on-Radio)
0x39	SPWD	Enter power down mode when CSn goes high.
0x3A	SFRX	Flush the RX FIFO buffer. Only issue SFRX in IDLE or, RXFIFO_OVERFLOW states.
0x3B	SFTX	Flush the TX FIFO buffer. Only issue SFTX in IDLE or, TXFIFO_UNDERFLOW states.
0x3C	SWORRST	Reset real time clock to Event1 value.
0x3D	SNOP	No operation. May be used to get access to the chip status byte.

Table 3.2: CC1100 Command Strobes Summary

Description	XOSC Periods	26MHz Crystal
IDLE to RX/TX/FSTXON, no calibration	2298	88.4 μs
IDLE to RX/TX/FSTXON, with calibration	~21037	809 μs
TX to RX switch	560	21.5 μs
RX to TX switch	250	9.6 μs
RX/TX to IDLE, no calibration	2	0.1 μs
RX/TX to IDLE, with calibration	~187392	721 μs
Manual calibration	~187392	721 μs

Table 3.3: State Transition Timing

Command strobes may be viewed as single byte instructions to CC1100. By addressing a command strobe register, internal sequences will be started determining state transitions and state change in the radio state machine. These registers are accessed by transferring a single header byte. Table 3.2 provides a detailed description of the 13 command strobe registers. These commands are used to disable the crystal oscillator, enable receive mode etc. Their use in initiating radio state machine changes is illustrated by annotating the state transitions in figure 3.3 with the corresponding command strobes. The command strobes are executed immediately, except for SPWD and SXOFF which are executed when CSn goes high.

Status registers are read-only and they contain information about the status of CC1100: the received signal strength indication (RSSI), the control state machine state (MARCSTATE), underflow and number of bytes in the TX FIFO (TXBYTES) and so on.

CC1100 contains two 64 byte FIFOs: one for received data (RX FIFO) and one for data to be transmitter (TX FIFO). They are accessed via SPI interface through the 0x3F addresses, with the $R/\bar{W}$ bit in the header selecting which of the 2 FIFOs will be accessed. The access can be single byte or burst access. If the TX FIFO runs empty before the complete packet has been transmitted, the radio will enter TXFIFO_UNDERFLOW error state. Similarly in reception when the RX FIFO becomes full before the content has been read, the radio will enter RXFIFO_OVERFLOW error state. The only way to exit these error states is by issuing an SFTX or an SFRX strobe respectively.

As we previously mentioned, figure 3.3 shows the CC1100 device internal finite state machine, while table 3.1 describes the states' characteristics. RX and TX are the active modes/states of the CC1100, while SLEEP, XOFF and IDLE are low-power modes. For the CC1100 to receive/transmit, the frequency synthesizer must be on and it must be calibrated regularly. One possibility is to configure automatic calibration through a configuration register (MCSM0). There are 3 calibration options: calibrate when going from IDLE to RX/TX/FSTXON; calibrate when going from RX/TX to IDLE; calibrate every fourth time when going from RX/TX to IDLE. Another possibility is to explicitly ask for a calibration through the SCAL command strobe when in IDLE state. In both cases (automatic or manual) the calibration takes a fixed amount of time. Similarly, the frequency synthesiser start-up (FS_WAKEUP device state), the settling (SETTLING device state), the RX to TX (RXTX_SETTLING) and TX to RX (TXRX_SETTLING) transitions have a fixed duration. Table 3.3 summarizes the fixed transition times in the device state machine.

The states of CC1100 can be classified into 3 categories: nontransitional, transitional and error. State transitions occur upon issuing command strobes that explicitly request a specific transition, or on internal events (*i.e.*, FIFO overflow or underflow, end of packet transmission, end of packet reception) with the next state indicated in configuration registers(i.e the next state after a successful transmission/reception).

The transitional states like CALIBRATION, FS_WAKEUP, SETTLING, RXTX_SETTLING and RXTX_SETTLING have fixed duration, i.e. the time the radio remains in such a state is device dependent, thus beyond user/programmer control. The transitions into transitional states occur on command strobes. The transition out of such a state is determined by the signal issued for the transition into the state and both the time and energy consumed are fixed for these states.

The nontransitional states like RX, TX, IDLE, SLEEP, XOFF, FSTXON have a configurable duration with a lower bound constraint. The duration of these states has a lower bound which means that each such state has a t_{min} minimum duration constraint but no upper bound since the device may remain in these states for an unlimited amount of time. The transitions into and out of nontransitional states occur upon issuing command strobes or on internal events according to configuration registers.

The transition into error states like RX_OVERFLOW or TX_UNDERFLOW occurs on external error conditions beyond user/programmer control. The device exits these

states upon issuing a command strobe after acknowledging the error.

3.3 Software to Hardware Optimized Mapping

After having introduced the timed automaton model for describing the software application's behavior in section 3.1 and the finite state machine model used for describing the physical device in section 3.2, we will focus now on the problem of mapping such a software protocol onto a physical device such that the energy consumption to be minimal.

3.3.1 Software Automaton Transformation

The first step in the process of software to hardware state mapping is the identification of states in the software automaton that map directly to physical states of the hardware device: the *fixed states*. For the case of B-MAC, these states are: CCA Sampling and Receive which map to the **RX** radio state and Transmit which maps to **TX** radio state. The remaining states are *free states*: WAIT, WAIT Initial Backoff and WAIT Congestion Backoff. These states need to be mapped onto physical device paths, i.e. non empty lists of successive physical states, that guarantee their time duration constraint. As noticed above, the transition into and out of a *free state* may occur upon a signal from an upper layer or from the physical layer or upon a timer event ($\text{clock}_{\text{var}} = \text{value}$), having as predecessor or successor respectively either another free state or a fixed state.

Timer enabled transitions impose a maximum time constraint on free states. Signal enabled transitions that are not coupled with time guards on the other hand are not time correlated since such an external event can occur at any time while inside such a free state and any physical mapping for the free state should guarantee that any path from the free state to a fixed state using that transition remains realizable. The case where the signal enabled transition is coupled with time guards will be transformed in order to eliminate the time guard. Timer signals can also occur at any moment of time according to the way they were programmed and it is the protocol's designer choice whether to take them into consideration or not with regard to the current state of the protocol automaton.

In this context, our goal is to define for every time constrained free state S of the software protocol, a mapping to a unique physical state or to a path of physical states

from the physical device automaton such that all paths towards fixed states starting with the signal enabled transitions out of S remain realizable, while minimizing the energy consumption of the state until the end of its duration.

We will consider the following restrictions with respect to the general formalism of timed automata as defined in [5]: *(i)* for every free state S in the software automaton, there should exist at most one output transition of the form $\text{clock}_{\text{var}} = \text{value}$ defining its duration; in the case such a transition exists, the clock variable $\text{clock}_{\text{var}}$ is associated to the free state S and no other timer enabled output transition from a free state can contain the same variable; *(ii)* for every free state S in the software automaton, transitions into the state that do not reset the clock variable defining its duration should have as source a fixed state; *(iii)* for every free state S in the software automaton, transitions out of the state guarded by clock constraints should only involve the clock variable defining the state's duration; *(iv)* for every free state S in the software automaton, transitions into the state guarded by clock constraints should only involve the clock variable defining the parent state's duration unless the start state of the transition is a fixed state; *(v)* transitions conditioned by logical expressions formed with integer and boolean variables should have as source a fixed state, *(vi)* fixed states should have a duration superior or equal to the minimum time constraint of the corresponding physical state.

Figure 3.5 presents the flow of activities in the software to hardware mapping. The desired behavior of the software application/protocol (here the B-MAC protocol) is expressed as a timed automaton respecting the constraints introduced in the previous paragraph. The behavior of the hardware device is expressed in the form of an automaton with states of fixed or variable duration but having lower limit constraint. The restricted timed automaton describing the functionality of the software protocol needs several adjustments ("Software Automaton Transformation" in figure 3.5) which will be described further. From the transformed software automaton description and the physical automaton description, an optimized mapping of the software protocol to the radio device is obtained by minimizing the global energy consumed. The resulted mapping can be obtained in several formats: graph, code skeleton for event-driven Operating System and multi-threaded Operating System respectively.

We recall that the software timed automaton $\mathcal{A}$ is a tuple $\langle N, l_0, \mathcal{C}, \beta, \vartheta, \Sigma, E, I \rangle$.

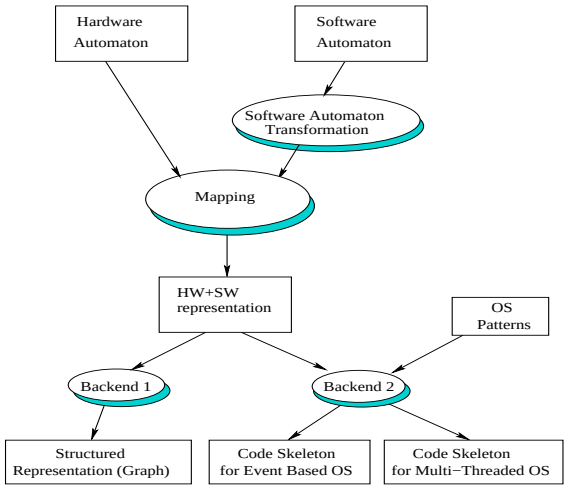


Figure 3.5: Synthesis flow for software to hardware mapping.

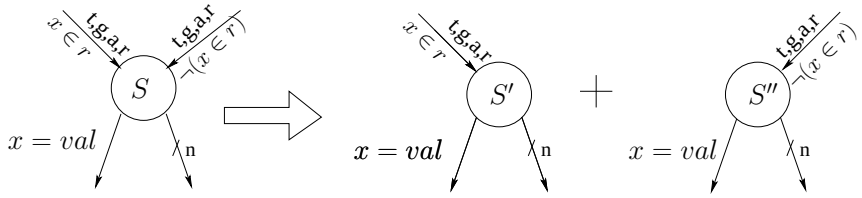


Figure 3.6: Software automaton transformation for the case of free states with input transitions not resetting the clock variable defining the state's duration.

As mentioned before in section 3.1, a transition has the form $\langle l, t, g, a, l' \rangle$ ($l \xrightarrow{t,g,a} l'$), where t stands for the triggering condition of the transition (either $\text{clock}_{\text{var}} = \text{value}$ or signal), g stands for the guards while a stands for the actions to be performed (clock resets and integer variables increment/decrement/reset).

In what follows we isolate in the notation the set of clock variables to be reset in order to clarify the following explanations. Thus a transition has the form $\langle l, t, g, a, r, l' \rangle$ ($l \xrightarrow{t,g,a,r} l'$), with the same meaning as in the previous definition except for r-standing for the set of clocks to be reset upon taking this transition.

Two types of transformations should be performed on the restricted automaton describing the behavior of a software protocol.

First, a time constrained free state either has a fixed duration (all input transitions into such a state reset the clock variable defining its duration or have a triggering condition of the form $\text{clock}_{\text{var}} = \text{value}$, where $\text{clock}_{\text{var}}$ is the clock variable defining

its duration) or has a fixed or variable duration depending on the input transition that was taken for reaching the state (whether that transition did or did not reset its corresponding clock variable). In the first transformation we separate input transitions that reset the clock variable defining its duration from the rest of the transitions. This implies duplicating the state (as well as the output transitions) and separating its input entries (those resetting the clock and those not resetting it), as seen in Figure 3.6. Thus for a free state S with corresponding clock variable x (we have imposed the constraint that $\forall s \in V$ s.t. type(s)=free, $(\exists)!\ e = s \xrightarrow{t,g,a,r} s' \in E$ s.t. $t = (x = Max_{value})$) we create two states S' and S'' and distribute adjacent edges as follows:

- $\forall e \in E, e = S \xrightarrow{t,g,a,r} X$ remove e from E and add $e' = S' \xrightarrow{t,g,a,r} X$ and $e'' = S'' \xrightarrow{t,g,a,r} X$ to E;
- $\forall e \in E, e = X \xrightarrow{t,g,a,r} S$ replace e with $e = X \xrightarrow{t,g,a,r} S'$ if $x \in r$;
- $\forall e \in E, e = X \xrightarrow{t,g,a,r} S$ replace e with $e = X \xrightarrow{t,g,a,r} S''$ if $\neg(x \in r)$;

Second, for a free state with signal enabled out transitions that contain guards (clock constraints involving the clock variable defining the state's duration) or input transitions triggered by its own clock variable, the constants appearing in the clock constrains and input transitions define several disjoint intervals into which the state should be split as seen in Figure 3.7. Thus, for a free state S with corresponding clock variable x and increasingly ordered set of constants appearing in transitions' guards $C = \{c_1, c_2, ..., c_n\}$ we create n states $S_1, S_2, ...S_n$ with corresponding clock intervals

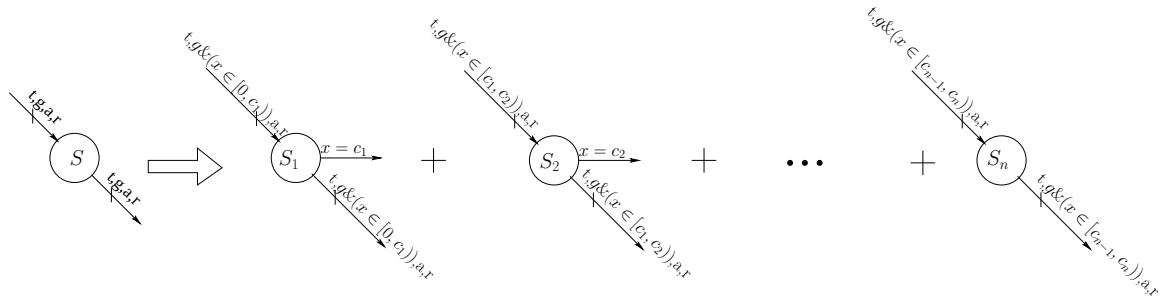


Figure 3.7: Software automaton transformation for free states with output transitions with clock constraints.

$[0, c_1], [c_1, c_2]...[c_{n-1}, c_n]$ respectively and create/distribute/duplicate transitions as follows:

- $\forall S_i$ a state with corresponding clock interval $[c_{i-1}, c_i], i \in \{1, 2, ..., n-1\}$, add edge $e = S_i \xrightarrow{x=c_i} S_{i+1}$ to E;
- $\forall e \in E, e = X \xrightarrow{t,g,a,r} S$ create transitions $X \xrightarrow{t,g\&(x\in[c_{i-1},c_i)),a,r} S_i$ for all $i \in \{1, 2, ..., n\}$;
- $\forall e \in E, e = S \xrightarrow{t,g,a,r} X$ create transitions $S_i \xrightarrow{t,g\&(x\in[c_{i-1},c_i)),a,r} X$ for all $i \in \{1, 2, ..., n\}$;

We need to make some remarks regarding this transformation. If an input transition $e \in E, e = X \xrightarrow{t,g,a,r} S$ contains guards formed with x, according to restrictions *(i)* and *(iv)*, X has to be a fixed state. If the triggering condition of the transition t has the form $t = (x == cst)$, then the constant cst should be taken into account among the constants used for splitting the state S.

If in an input transitions $e \in E, e = X \xrightarrow{t,g,a,r} S$, X is a free state, by restriction *(ii)*, it is necessary that $x \in r$, i.e. this transition must reset the clock variable x. Thus x will have the value 0 and only the edge $X \xrightarrow{t,g\&(x\in[0,c_1)),a,r} S_1$ will have a valid guard. The rest of edges of the form $X \xrightarrow{t,g\&(x\in[c_{i-1},c_i)),a,r} S_i$ with $i \in \{2, ..n\}$ will be discarded.

If in an input transition $e \in E, e = X \xrightarrow{t,g,a,r} S$, X is a fixed state, then only the added transitions $X \xrightarrow{t,g\&(x\in[c_i,c_{i+1})),a,r} S_1$ for which $g\&(x \in [c_i, c_{i+1}))$ is a valid guard will be kept. The validity of the guard depends on two aspects. If the initial guard g contains x (allowed since X is fixed), the validity of the guard depends on the result of the evaluation $g\&(x \in [c_i, c_{i+1}))$. The guard will be kept or discarded accordingly. If the initial guard g does not contain x, then the transition is kept and its guard $g\&(x \in [c_i, c_{i+1}))$ will be tested at run-time.

Similarly, for added output transitions $S_i \xrightarrow{t,g\&(x\in[c_{i-1},c_i)),a,r} X$, only those with valid guards $g\&(x \in [c_{i-1}, c_i))$ will be kept. For example for the state S_3 with the clock x in the corresponding interval $[c_2, c3)$ if the guard g is $x \geq c_4$ then $g\&(x \in [c_2, c3))$ will not be a valid guard and the transition can be discarded.

3.3.2 Problem Statement and Complexity Analysis

The transformations described in section 3.3.1 have as a result a transformed software automaton composed of three types of free states:

- states of fixed duration
- states of an unknown duration ranging from 0 to an upper bound, with the exact duration known at run-time;
- states with unknown unbounded duration;

Each such free state might have other constraints besides its duration due to the signal enabled out transitions. These transitions might restrict even further the physical states and paths onto which the software state could be mapped.

The general problem we wish to solve is the mapping of a free state of a software automaton onto a path in the physical device automaton such that transitions out of the state remain realizable and the energy consumed is minimized.

Consider a software automaton A with a free state Sf as seen in figure 3.8. Consider Sf is having only 1 input transition originating in a fixed state S and only 1 output transition towards a fixed state T. Consider S_{phys} and T_{phys} to be the physical states corresponding to the fixed software states S and T respectively. Consider the state Sf has a fixed duration L. We are fixing ourselves as a goal to map this free state Sf onto a path in the physical device starting from the physical state corresponding to S and ending in the physical state corresponding to T, having time duration L and minimizing the energy consumed.

Note that this problem - mapping a fixed duration free state lacking signal enabled output transitions- is a sub-problem of the general problem we wish to solve. In what follows, we will prove its complexity.

In order to investigate the complexity of the defined problem, we choose to modelize it by means of a path search problem in a graph whose vertices are annotated by means of some functions. Let $G = \{V, E\}$ be the graph. This graph will correspond to the physical device automaton. Each vertex $v \in V$ will correspond to a unique state in the physical device, while each oriented edge $e \in E$ will correspond to a transition in the physical device. Every vertex $v \in V$ will have a unique type - either $transitional$ or $non - transitional$, according to the corresponding node in the physical device. Also,

we consider each vertex to be annotated with 2 values: the minimum time duration and the per unit energy consumption of the corresponding physical state.

From the software automaton in figure 3.8, we retain only the information of start state S, destination state T and duration L of the free state Sf. We therefore search for a general path (passing several times through the same node or through the same oriented edge is allowed) in the graph we just defined from s to t, where s is the vertex in G corresponding to S_{phy} and t is the vertex G corresponding to T_{phy}. We require that this path should have duration exactly L and that its corresponding energy consumption should be minimal.

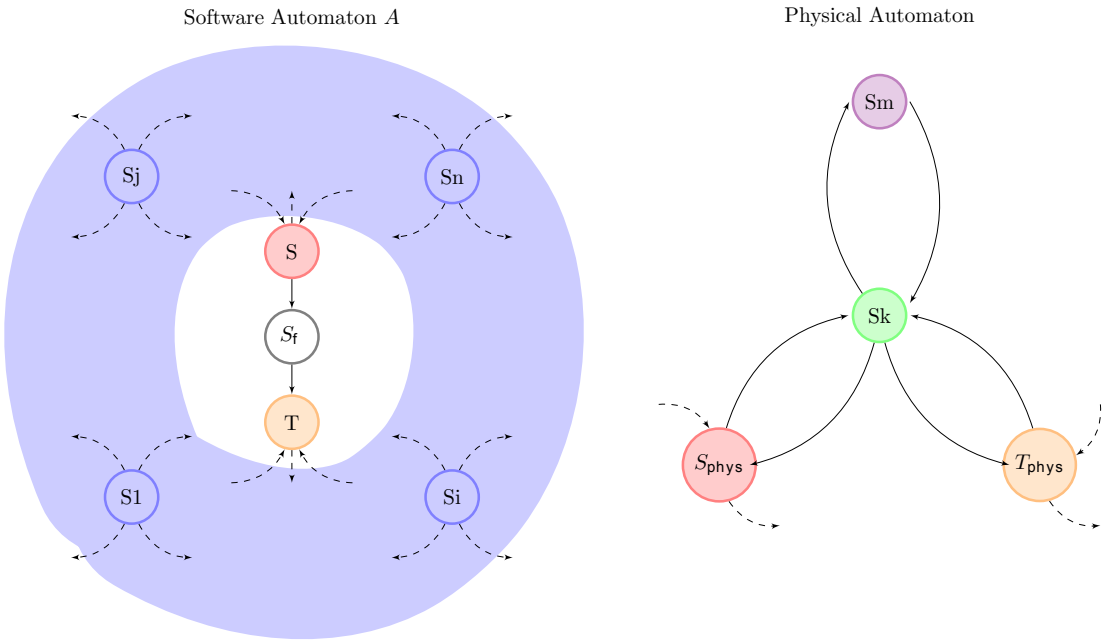


Figure 3.8: Software Automaton A with Free State Sf to be mapped onto hardware automaton.

With this introduction we now formally define the problem of **Minimum Consumption Free State Mapping - *MCFSM*** :
Given:

- a graph $G = \{V, E\}$
- a function $\mathsf{type} : V \rightarrow \{\mathsf{trans}, \mathsf{nontrans}\}$ assigning types to states;

- a function $t_{\mathsf{min}} : V \rightarrow \mathbb{N}^*$ assigning minimum time durations to states;
- a function $\mathsf{cons} : V \rightarrow \mathbb{N}^*$ assigning costs per unit time to states.
- $s, t \in V$ two vertices
- an integer L

we aim at finding:

- a path $P = \{s, v_1, v_2, ..., v_n, t\}$ from s to t in G
- a duration assignment set $T = \{t_i \mid i \in \{1, ..n\} \wedge (t_i \geq t_{\mathsf{min}}(v_i) \text{ if } \mathsf{type}(v_i) = \mathsf{nontrans}) \wedge (t_i = t_{\mathsf{min}}(v_i) \text{ if } \mathsf{type}(v_i) = \mathsf{trans})\}$, assigning durations t_i to the vertices v_i in the solution path P

such that

- the path energy consumption $\sum_{i=1}^{n} t_i \times \mathsf{cons}(v_i)$ is minimal
- the path duration satisfies $\sum_{i=1}^{n} t_i = L$.

The decision version of the problem is stated as follows:
Instance: A graph $G = \{V, E\}$, a type function $(\forall v \in V, \mathsf{type}(v) \in \{\mathsf{trans}, \mathsf{nontrans}\})$, a minimum time duration function t_{min} $(\forall v \in V, t_{\mathsf{min}}(v) \in \mathbb{N}^*)$ and per unit time energy consumption function cons $(\forall v \in V, \mathsf{cons}(v) \in \mathbb{N}^*)$, two vertices $s, t \in V$ and two integers L and W.
Question: Is there a path $P = \{s, v_1, v_2, ..., v_n, t\}$ in G from s to t and a duration assignment set $T = \{t_i \mid i \in \{1, ..n\} \wedge (t_i \geq t_{\mathsf{min}}(v_i) \text{ if } \mathsf{type}(v_i) = \mathsf{nontrans}) \wedge (t_i = t_{\mathsf{min}}(v_i)$ if $\mathsf{type}(v_i) = \mathsf{trans})\}$ such that $\sum_{i=1}^{n} t_i = L$ and $\sum_{i=1}^{n} t_i \times \mathsf{cons}(v_i) \leq W$?

Theorem 1 *The problem of Minimum Consumption Free State Mapping (MCFSM) is NP complete.*

It is easy to see that $MCFSM \in NP$, since a nondeterministic algorithm needs only to guess a path and associated timing set and check in polynomial time if the path duration and the path consumption satisfy the required conditions.

The proof is achieved by transforming the weighted version of the *Change Making Problem* (**CMP**) into **MCFSM**.

Given a finite set of coin denominations along with an unlimited supply of coins in each denomination, **CMP** is the problem of paying a sum C with the fewest coins possible. The decision version of the problem is as follows:

Instance: There are given : n coin types, val_i is the value of coin type i, $i \in \{1, 2, ..n\}$, 2 constants $M, C \in \mathbb{N}^*$.

Question: Is there a set $X = \{x_1, x_2, ...x_n\}$, $x_i \in \mathbb{N}$ such that $\sum_{i=1}^{n} \text{val}_i \times x_i = C$ and $\sum_{i=1}^{n} x_i \leq M$?

In this definition, C is the sum to be payed, val_i and are the coin denominations. **CMP** was proven to be NP hard[31].

The weighted version of the change making problem (**CMPW**) associates weight to coin denominations and it requires the payment of a sum C with the smallest cumulated weight possible. The decision version of the problem is as follows:

Instance: There are given : n coin types, val_i is the value of coin type i, w_i is the weight of coin type i, $i \in \{1, 2, ..n\}$, two constants $M, C \in \mathbb{N}^*$.

Question: Is there a set $X = \{x_1, x_2, ...x_n\}$, $x_i \in \mathbb{N}$ such that $\sum_{i=1}^{n} \text{val}_i \times x_i = C$ and $\sum_{i=1}^{n} x_i \times w_i \leq M$?

Note that every instance of the classical **CMP** can be viewed as an instance of **CMPW**, where the weight of each type of coin is constant and equal to 1. **CMP** is a sub-problem of **CMPW** and therefore **CMPW** is also NP-hard.

In what follows, we will transform an instance of CMPW into an instance of MCFSM.

In a first step, we create a bidirectional clique $G = \{V, E\}$ where $V = \{v_1, v_2, ...v_n\}, E = V \times V = \{(v_i, v_j) \mid i, j \in \{1, ..n\}, i \neq j\}$, having a vertex corresponding to each coin type and an edge for each ordered pair (v_i, v_j).

For each $i \in \{1, ..n\}$ such that $\text{val}_i > 1$, split v_i in two nodes $v_{i,1}, v_{i,2}$ and transform the edges involving v_i as illustrated in figure 3.9. For all in-edges, $e = (u, v_i)$, modify it as $e = (u, v_{i,1})$. For all out-edges, $e = (v_i, u)$, modify it as $e = (v_{i,2}, u)$. Add edges $(v_{i,1}, v_{i,2})$ and $(v_{i,2}, v_{i,1})$ to E and set:

$$\begin{cases} \mathsf{type}(v_{i,1}) = \mathsf{type}(v_{i,2}) = \mathsf{trans} \\ t_{\mathsf{min}}(v_{i,1}) = 1 \\ t_{\mathsf{min}}(v_{i,2}) = \text{val}_i - 1 \\ \mathsf{cons}(v_{i,1}) = \mathsf{cons}(v_{i,2}) = \frac{w_i}{\text{val}_i} \in \mathbb{Q} \end{cases}$$

For each $i \in \{1, ..n\}$ such that $\text{val}_i = 1$, we set:

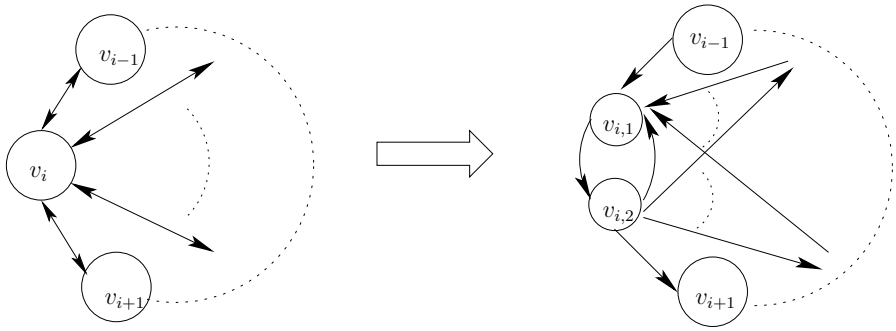


Figure 3.9: Transformation of nodes v_i having $\text{val}_i > 1$

$$\begin{cases} \mathsf{type}(v_i) = \mathsf{nontrans} \\ t_{\mathsf{min}}(v_i) = 1 \\ \mathsf{cons}(v_i) = w_i \end{cases}$$

We add two new vertices s and t to V, and edges $e = (s, v_{i,1})$, $e = (v_{i,2}, t)$, $e = (s, v_i)$, $e = (v_i, t)$ to E and define:

$$\begin{cases} \mathsf{type}(s) = \mathsf{type}(t) = \mathsf{nontrans} \\ t_{\mathsf{min}}(s) = t_{\mathsf{min}}(t) = 1 \\ \mathsf{cons}(s) = \mathsf{cons}(t) = M + 1 \end{cases}$$

We set $L = C$ and $W = M$.

We have thus defined an instance of MCFSM that contains $2 + 2 * j + i$ nodes, where i is the number of coins of denomination 1, $j = n - i$ is the number of coins of denomination different from 1 and the 2 extra nodes correspond to the source and target nodes. MCFSM contains a number of edges equal to $2 * n + n \times (n - 1) + 2 * j$ with the explanation that $n \times (n-1)$ are the directed edges in the initial clique, $2*j$ are the extra added edges in a dipole $v_{i,1}, v_{i,2}$, while the $2*n$ extra edges correspond to the links from the source s to every node of the form v_i or $v_{i,1}$ and from every node of the form v_i or $v_{i,2}$ to the target t. Therefore we have operated a polynomial transformation of **CMPW** into **MCFSM**.

We must now prove that MCFSM has a solution ***iff*** CMPW has a solution.

We discuss first the case $\Rightarrow$.

Let $P = \{s, u_1, u_2, ..., u_m, t\}$ a path in G from s to t and duration assignment set $T = \{t_i \mid i \in \{1, ..m\} \wedge (t_i \geq t_{\mathsf{min}}(u_i) \text{ if } \mathsf{type}(u_i) = \mathsf{nontrans}) \wedge (t_i = t_{\mathsf{min}}(u_i) \text{ if}$

$\mathsf{type}(u_i) = \mathsf{trans})\}$ s.t. $\sum_{i=1}^{m} t_i = L$ and $\sum_{i=1}^{m} t_i \times \mathsf{cons}(u_i) \leq W$ be a solution to MCFSM.

From the definition of G, the graph in MCFSM, if a node $v_{i,1}$ appears in a path from s to t, then the node $v_{i,2}$ also appears in the path (because the node $v_{i,1}$ has only one output transition towards the nodes $v_{i,2}$). Similarly, if the node $v_{i,2}$ appears in a path from s to t, then the node $v_{i,1}$ also appears in the path (because the node $v_{i,2}$ has only one input transition coming from the node $v_{i,1}$). Therefore if $v_{i,2}$ appears in the path, $v_{i,1}$ appears also and for the same number of times, corresponding to the number of times the loop ($v_{i,1}$,$v_{i,2}$) was traversed. Since $v_{i,1}$ and $v_{i,2}$ are transitional nodes, their corresponding duration in the set T will be their corresponding t_{min} value. We denote n_{v_i} the number of appearances of the tuple $(v_{i,1}, v_{i,2})$ in the path P.

For the nontransitional nodes v_i appearing in the path P, we define $n_{v_i} = \sum_{j=1,u_j=v_i}^{m} t_j$- the total time in the path P that is spent in the nontransitional node v_i.

We define the solution for **CMPW**:

$$\begin{cases} X = \{x_i \mid i \in \{1, ..n\} x_i = n_{v_i}\} \\ C = L \\ W = M \end{cases}$$

From the definition of MCFSM, we see that s and t cannot be part of the solution, having consumption M+1.

We note that $C = L = \sum_{i=1}^{m} t_i = \sum_{i=1,type(u_i)=trans}^{m} t_i + \sum_{i=1,type(u_i)=nontrans}^{m} t_i$.

The nodes of type $v_{j,1}$ and $v_{j,2}$ are the only transitional nodes that appear in the solution. As stated before, if a node $v_{j,1}$ appears in the solution to MCFSM, the node $v_{j,2}$ appears and for the same number of times n_{v_j}. If the nodes do not appear in the solution, then $n_{v_j} = 0$. Also, the duration assignment for a transitional node is its t_{min} value. Therefore

$$\begin{aligned} \sum_{i=1,type(u_i)=trans}^{m} t_i &= \sum_{j=1,v_{j,1}\in V}^{|V|} n_{v_j} \times t_{\mathsf{min}}(v_{j,1}) + \sum_{j=1,v_{j,2}\in V}^{|V|} n_{v_j} \times t_{\mathsf{min}}(v_{j,2}) \\ &= \sum_{j=1,v_{j,1}\in V}^{|V|} n_{v_j} \times 1 + \sum_{j=1,v_{j,2}\in V}^{|V|} n_{v_j} \times (val_j - 1) \\ &= \sum_{j=1,v_{j,1},v_{j,2}\in V}^{|V|} x_j \times val_j \end{aligned}$$

For the nontransitional nodes,

$$\begin{aligned}\sum_{i=1,type(u_i)=nontrans}^{m} t_i &= \sum_{j=1,type(v_j)=nontrans}^{|V|} n_{v_j} \times 1 \\ &= \sum_{j=1,type(v_j)=nontrans}^{|V|} n_{v_j} \times val_j \\ &= \sum_{j=1,type(v_j)=nontrans}^{|V|} x_j \times val_j\end{aligned}$$

Therefore $C = L = \sum_{i=1}^{n} x_i \times \text{val}_i$.

Similarly, $W = M \geq \sum_{i=1}^{m} t_i \times cons(u_i) = \sum_{i=1,type(u_i)=trans}^{m} t_i \times cons(u_i) + \sum_{i=1,type(u_i)=nontrans}^{m} t_i \times cons(u_i)$.

For the transitional nodes,

$$\begin{aligned}\sum_{i=1,type(u_i)=trans}^{m} t_i \times cons(u_i) &= \sum_{j=1,v_{j,1}\in V}^{|V|} n_{v_j} \times 1 \times cons(v_{j,1}) \\ &+ \sum_{j=1,v_{j,2}\in V}^{|V|} n_{v_j} \times (val_j - 1) \times cons(v_{j,2}) \\ &= \sum_{j=1,v_{j,1}\in V}^{|V|} n_{v_j} \times 1 \times w_j/val_j \\ &+ \sum_{j=1,v_{j,2}\in V}^{|V|} n_{v_j} \times (val_j - 1) \times w_j/val_j \\ &= \sum_{j=1,v_{j,1},v_{j,2}\in V}^{|V|} n_{v_j} \times (1 + val_j - 1) \times w_j/val_j \\ &= \sum_{j=1,v_{j,1},v_{j,2}\in V}^{|V|} x_j \times w_j\end{aligned}$$

For the nontransitional nodes,

$$\begin{aligned}\sum_{i=1,type(u_i)=nontrans}^{m} t_i \times cons(u_i) &= \sum_{j=1,type(v_j)=nontrans}^{|V|} n_{v_j} \times 1 \times cons(v_j) \\ &= \sum_{j=1,type(v_j)=nontrans}^{|V|} n_{v_j} \times w_j \\ &= \sum_{j=1,type(v_j)=nontrans}^{|V|} x_j \times w_j\end{aligned}$$

Therefore , $W = M \geq \sum_{i=1}^{m} t_i \times cons(u_i) = \sum_{j=1}^{|V|} x_j \times w_j$

Let us now discuss the case $\Leftarrow$. Let $X = \{x_i \mid i \in \{1, ...n\}\}$ be a solution to CMPW. Consider $\cup$ to be a concatenation operator on paths.

In the case of nontransitional nodes($val_i = 1$), we define the path P_i as:

$$P_i = \begin{cases} \{v_i\} & \mathrm{x}_i > 0 \\ \{\} & \text{otherwise} \end{cases}$$

For transitional nodes ($val_i > 1$) we define P_i as:

$$P_i = \cup_{j=1}^{x_i} P'_j, \text{ where } P'_j = \{v_{j,1}, v_{j,2}\}$$

We define $P = \cup_{i=1}^{n} P_i$. Now that we defined the path P we still have to define the duration assignment set T of the solution. $\forall u_j \in P, j \in \{1, 2, ..|P|\}$ we define:

$$t_j = \begin{cases} x_i \text{ s.t. } u_j = v_i \in V, & \mathsf{type}(u_j) = \mathsf{nontrans}, \\ t_{\mathsf{min}}(v_i) \text{ s.t. } u_j = v_i \in V, & \mathsf{type}(u_j) = \mathsf{trans} \end{cases}$$

Note that the j index denotes the position of a vertex in the solution path P. The explanation is that if u_j is a vertex in the path on position j and it corresponds to the vertex v_i in G of type nontransitional, this vertex appears at most once in the path (by construction of the path, if $x_i > 0$) and should be assigned as duration in the duration set the x_i value in the solution to CMPW. If on the contrary, v_i is a transitional node, then it might appear several times in the path and should have as duration its t_{min} value.

With these notations, the path P and the duration assignment set $T = \{t_1, t_2, ...t_{|P|}\}$ represent a solution to MCFSM since

$$\begin{cases} C = L = \sum_{i=1}^{n} x_i \times \mathrm{val}_i = \sum_{i=1}^{|P|} t_i \\ W = M \geq \sum_{i=1}^{n} x_i \times w_i = \sum_{i=1}^{|P|} t_i \times \mathsf{cons}(v_i) \end{cases}$$

3.3.3 Mapping Heuristic

We have proved in subsection 3.3.2 that the problem of mapping a free software state of fixed duration onto a path in the physical device between 2 given states while minimizing the energy consumption is NP complete. We therefore propose a heuristic for mapping such free states, guaranteeing that all the transitions in the software automaton remain realizable, all while trying to consume the least possible energy. We proceed by investigating the temporal relations in the software automaton and determining temporal distances between free states and the fixed states accessible. Next, we propose to investigate certain paths in the hardware device and we give the function computing their energy consumption. We conclude the section by giving the algorithm allowing the mapping of the free software states onto physical device paths.

Software Automaton Analysis One of our restrictions with respect to the original model of timed automata was that there should be at most a transition of the form $clock_{var} = constant$ out of a free state. As we stated before, the free states to be mapped can be categorized into 3 types:

- states of fixed duration;
- states of an unknown duration ranging from 0 to an upper bound, with the exact duration known at run-time;
- states with unknown unbounded duration;

The first type of free state has its duration defined by a timer variable that has a fixed known value when entering the state. There are 2 possibilities: either it was reset on every input transition or it had a fixed unique value on its input transitions i.e. the input transitions were of the form $clock_{var} = constant$. The latter type of transitions appear from the second type of transformation illustrated in section 3.3.1 and figure 3.7. There is a unique exit from this type of free state of the form $clock_{var} = constant$ - this was one of the constraints imposed on the model. Apart form the output transition enabled by the timer $clock_{var} = constant$, there can be other output transitions triggered by signals.

The second type of state has a unique output transition triggered by a timer of the form $clock_{var} = constant$, but none of its input transitions resets the clock variable

defining its duration. These states are the result of the first transformation in section 3.3.1. We have imposed the constraint *(ii)* on the software model which states that the start states of these type of transitions should be fixed. If this particular transition was taken for entering the state, its duration is known only at run-time and can vary between 0 and the *constant* in $clock_{var} = constant$. If the $clock_{var}$ is higher than the *constant*, this state transforms into the third type of state.

The third type of state has no output transition of the form $clock_{var} = constant$.

All these free states have output transitions triggered by signals. Such a signal can occur at any time when the software automaton is in the free state and in particular, the worst case scenario is the exact beginning of the state.

Consider $G_{\mathsf{HW}} = \{V_{\mathsf{HW}}, E_{\mathsf{HW}}, t_{min}, cons\}$ to be the graph corresponding to the hardware automaton and its associated functions. Consider $G_{\mathsf{SW}} = \{V_{\mathsf{SW}}, E_{\mathsf{SW}}, duration, type\}$ to be the graph associated to the modified software automaton. *Duration* is the function assigning durations to states: fixed, interval or infinite, according to the 3 types of states identified previously. Consider S_{SW} to be a free state of the software automaton and S_{HW} to be a fixed state appearing in the software automaton that is reachable from S_{SW}, i.e. there exists a path in A_{SW} from S_{SW} to S_{HW}.

In order to determine the temporal distance between S_{SW} and S_{HW}, we must assign weights equivalent to time durations to the transitions of the software automaton and we must perform some transformations and state duplications.

We define the signal bottom level of a state S_{SW} of the software automaton with respect to a state S_{HW} of the hardware automaton $\mathsf{BL}_s(S_{\mathsf{SW}}, S_{\mathsf{HW}})$ as the minimum length of all the elementary paths in G_{SW} from the state S_{SW} to the state S_{HW} corresponding to a fixed state in the software automaton, having as first edge a signal enabled edge. We define the timer bottom level of a time constrained free state S_{SW} of the software automaton with respect to a state S_{HW} of the hardware automaton – $\mathsf{BL}_t(S_{\mathsf{SW}}, S_{\mathsf{HW}})$ – as the minimum length of all the elementary paths from the state S_{SW} to the state S_{HW} corresponding to a fixed state in the software automaton, having as first edge a timer enabled edge ($\text{clock}_{\text{var}} = \text{value}$). The computation of these metrics implies some duplications of the states of the software automaton and assignment of weights to the edges.

For time constrained states with input transitions without clock reset, all output edges are assigned 0 weight. Similarly for time unconstrained states, all output edges in G_{SW} are assigned 0 weight. Time constrained states with input transitions with clock reset must be duplicated ($S \Rightarrow S_s + S_t$) in order to separate timer enabled output transitions (used for computing the state's BL_t) and signal enabled output transitions (used for computing the state's BL_s), while input transitions remain common to the 2 new states. Timer enabled output transitions (corresponding to state S_t) are assigned weight equal to the duration of the state and all signal enabled output transitions (corresponding to state S_s) are assigned 0 weight.

With these transformations, the computation of BL_s and BL_t resides in a classical all-pairs shortest paths in G_{SW} restricted to paths containing only free states as intermediate nodes and taking the minimum value when there exist several instances of the same physical state as a fixed state of the software automaton.

Physical Automaton Analysis For a path $P = \{s_1, s_2, ..., s_n\}$ in the physical device automaton, a measure of the energy $\mathcal{E}[P]$ consumed by traversing the states in the path can be obtained by integrating the current/power consumption I_{s_i} corresponding to each state s_i with the time spent in it t_i, thus yielding: $\mathcal{E}[P] = \sum_{i=1}^{n} I_{s_i} \times t_i$.

For any path $P = \{s_1, s_2, ..., s_n\}$ in the physical device automaton, there exists a minimum transition time $t_{\mathsf{min}}(P)$ ($t_{\mathsf{min}}(P) = \sum_{i=1}^{n} t_{\mathsf{min}}(s_i)$) and a corresponding minimum current consumption $I_{\mathsf{min}}(P)$ ($I_{\mathsf{min}}(P) = \sum_{i=1}^{n} t_{\mathsf{min}}(s_i) \times I_{s_i}$), given by the minimum transition times $t_{\mathsf{min}}(s_i)$ of the states in the path and their respective current consumption. For a timed path in the radio device automaton, if the transition time of the path t_{path} allows extra time with respect to the minimum transition time of the path t_{min}, this extra time should be spent in the non-transitional state of the path with the minimum energy consumption s_{min}. Hence, the measure of the energy consumed during a timed path is:

$$\mathcal{E}[P_{\mathsf{timed}}] = \begin{cases} I_{\mathsf{min}} + (t_{\mathsf{path}} - t_{\mathsf{min}}) \times I_{s_{\mathsf{min}}} & t_{\mathsf{path}} \geq t_{\mathsf{min}} \\ \infty & \mathsf{otherwise} \end{cases} \tag{3.1}$$

We define the distance $\mathsf{dist}(S_1, S_2)$ from state S_1 to state S_2 in the physical device automaton as the minimum of all the lengths of the elementary paths leading from S_1 to S_2 in the weighted device automaton, where each transition/edge is assigned as weight the minimum duration constraint value of the source state of the edge. The

computation of the time distances between states in the physical automaton can be obtained as an all-pairs shortest paths.

Mapping algorithm The computed signal bottom levels of a state S_{SW} with respect to all fixed states accessible (through paths containing only free states as intermediate states) define the set of physical states admissible for S_{SW}, *i.e.*, the physical states from which all paths towards fixed states of the software automaton remain realizable in the case of the occurrence of the signals enabling the output transitions corresponding to those paths. For a given $\mathsf{BL}_s(S_{\mathsf{SW}}, S_{\mathsf{HW}})$, the set of admissible states with respect to S_{HW}, is defined by: $\mathsf{Admissible}(S_{\mathsf{SW}}, S_{\mathsf{HW}}) = \{s \in V_{\mathsf{HW}} \mid \mathsf{dist}(s, S_{\mathsf{HW}}) \leq \mathsf{BL}_s(S_{\mathsf{SW}}, S_{\mathsf{HW}})\}$. For a state S_{SW}, its set of admissible states is the intersection of the admissible states with respect to all fixed states reachable through signal enabled transitions, $S_{\mathsf{HW}} \in R_{\mathsf{Sig}}$.

$$\mathsf{Admissible}(S_{\mathsf{SW}}) = \bigcap_{S_{\mathsf{HW}} \in R_{\mathsf{Sig}}(S_{\mathsf{SW}})} \mathsf{Admissible}(S_{\mathsf{SW}}, S_{\mathsf{HW}}).$$

Both timer and signal bottom levels of a state S_{SW} (further simply denoted by BL) with respect to all fixed states reachable from S_{HW} in G_{HW}, $S_{\mathsf{HW}} \in R(S_{\mathsf{SW}})$, define the set of physical states admissible as input states for S_{SW}, *i.e.*, in which physical state can be the automaton at the beginning of the state: $\mathsf{Input_Admissible}(S_{\mathsf{SW}}, S_{\mathsf{HW}}) = \{s \in V_{\mathsf{HW}} \mid \mathsf{dist}(s, S_{\mathsf{HW}}) - t_{\mathsf{min}}(s) \leq \mathsf{BL}(S_{\mathsf{SW}}, S_{\mathsf{HW}})\}$. And thus we have:

$$\mathsf{Input_Admissible}(S_{\mathsf{SW}}) = \bigcap_{S_{\mathsf{HW}} \in R(S_{\mathsf{HW}})} \mathsf{Input_Admissible}(S_{\mathsf{SW}}, S_{\mathsf{HW}}) \tag{3.2}$$

The transition into a state $s \in \mathsf{Input_Admissible}(S_{\mathsf{SW}})$ might have to be performed before the beginning of S_{SW} in order for all the transitions out of it to remain realizable. The offset of a state $s \in \mathsf{Input_Admissible}(S_{\mathsf{SW}})$ with respect to S_{SW} is given by: $\mathsf{offset}(s, S_{\mathsf{SW}}) = \max\{0, \max_{S_{\mathsf{HW}} \in R(S_{\mathsf{SW}})}\{\mathsf{dist}(s, S_{\mathsf{HW}}) - \mathsf{BL}(S_{\mathsf{SW}}, S_{\mathsf{HW}})\}\}$

For a free state S_{SW} of fixed duration, the mapping considers for a given initial physical state S' from $\mathsf{Input_Admissible}(S_{\mathsf{SW}})$ with remaining time t_{rem} until the completion of its minimum time duration, the realizable path of minimum energy consumption containing only states from $\mathsf{Admissible}(S_{\mathsf{SW}})$ and ending in a state s from $\{s \in \mathsf{Input_Admissible}(S) \mid \mathsf{offset}(s, S) = 0\} \bigcup \{s \in \mathsf{Input_Admissible}(S) \bigwedge s \in \mathsf{Admissible}(S_{\mathsf{SW}}) \mid \mathsf{offset}(s, S) \neq 0\}$ and of path duration $S_{\mathsf{SW}\,\mathsf{duration}} - t_{\mathsf{rem}} - \mathsf{offset}(s, S)$,

where S is the successor of S_{SW} on the timer enabled transition. Since t_{rem} is variable in the interval $[0..t_{\mathsf{min}}(S')]$, the minimal energy consumption path depends on its current value at the moment when the transition towards S_{SW} was taken. Equation 3.1 defines the energy consumption function for such a path, with $t_{\mathsf{path}} \in [S_{\mathsf{SWduration}} - t_{\mathsf{min}}(S') - \mathsf{offset}(s,S) \ \ldots S_{\mathsf{SWduration}} - \mathsf{offset}(s,S)]$. The intersection of the graphs of the potential paths divides the interval $[0 \ldots t_{\mathsf{min}}(S')]$ into several disjoint intervals, each with its corresponding optimal path.

The free states S_{SW} with variable duration have input transitions that do not reset their corresponding clock variable and all these transitions have as start state fixed states. For these states, the mapping searches for paths between the fixed input states containing only states from $\mathsf{Admissible}(S_{\mathsf{SW}})$ and ending in a state s from $\{s \in \mathsf{Input_Admissible}(S) \mid \mathsf{offset}(s,S) = 0\} \bigcup \{s \in \mathsf{Input_Admissible}(S) \bigwedge s \in \mathsf{Admissible}(S_{\mathsf{SW}}) \mid \mathsf{offset}(s,S) \neq 0\}$ and of path duration $S_{\mathsf{SWduration}} - \mathsf{offset}(s,S)$, where S is the succesor of S_{SW} on the timer enabled transition. Equation 3.1 defines the energy consumption function for such a path, with $t_{\mathsf{path}} \in [0 \ \ldots S_{\mathsf{SW}max} - \mathsf{offset}(s,S)]$, where $S_{\mathsf{SW}max}$ is the maximum potential duration of S_{SW}. The intersection of the graphs of the potential paths divides the interval $[0 \ldots S_{\mathsf{SW}max}]$ into several disjoint intervals, each with its corresponding optimal path.

For a state S_{SW} with unbounded duration, the algorithm searches for paths from all $s \in \mathsf{Input_Admissible}(S_{SW})$ containing only states from $\mathsf{Admissibles}(S_{SW})$ towards the lowest energy consuming state from $\mathsf{Admissibles}(S_{SW})$.

The mapping is outlined in algorithm 1.

By restricting our search to paths with distinct edges, we are allowing the possibility for a path to pass several times through the same node, hence allowing the path to contain cycles, but we omit the possibility of passing several times through a cycle. Even so, the mapping phase implies an exhaustive search of all existing paths in the physical finite state machine between 2 given states, pruning the paths with minimum duration exceeding the duration of the software state.

Scalability/Complexity: The worst case scenario for such a search consists of a finite state machine whose graph is a clique. In such a graph, a path between a source node and a destination node containing a cycle can always be transformed in a path without cycles by replacing the edge entering the cycle with an edge to the first node in the cycle. Considering a clique of size n, we denote by N_i the number of paths between

the source and the destination containing i edges. N_i can be thought as the way of choosing i nodes among the $n-2$ remaining since the source and the destination of the path have been fixed. Thus, $N_i = C_{n-2}^i$ and the total number of possible paths is $\sum_{i=0}^{n-2} C_{n-2}^i = 2^{n-2}$, thus a worst case exponential computation time. Even so, we argue that the automata corresponding to physical devices usually have a small number of states and a small number of edges. Moreover, we improve the search by using upper and lower estimated bounds of the path cost which allows to discard large subsets of fruitless candidates. Also, given that low-power consumption states have a consumption that is usually several orders of magnitude lower than that of active states, we might improve our search by specifically targeting these states.

The complexity of the mapping is bounded by $m \times n^2 \times 2^{n-2}$, where m is the number of free states in the transformed software automaton and n is the number of states in the physical finite state machine. The explanation is that for every free state, the physical states admissible in input are paired with all the physical states admissible as input for its successor software state and the optimal path between these 2 is searched, according to the duration of the software state. The upper bound on the complexity is exponential in the number of states n in the physical automaton. Hopefully, as previously stated, the physical state machine has a small number of states and edges. The exponential particular clique worst case is thus very unlikely.

3.4 Code Skeleton Generation

WSN applications developed without an underlying operating system support represent a substantial part of the applications deployed today. There exist however operating systems developed with the purpose of responding to the needs of the resource constrained sensor networks and their applications. From the complexity point of view they are closer to OSs dedicated to embedded systems than to general purpose OSs. Just like the former, they must respond to a particular application's needs rather than to offer extensive functionalities for all possible applications. Also, some general functionalities such as virtual memory are either unnecessary or too expensive to implement.

There are two main categories of such operating systems with respect to the programming model used: the event-driven OSs (Tiny OS) and the multi-threaded OSs (Contiki, FreeRTOS).

Algorithm 1: Software Automaton to Physical Device Mapping Algorithm

input : $\{G_{\mathsf{HW}} = \{V_{\mathsf{HW}}, E_{\mathsf{HW}}\}, t_{\mathsf{min}}, \mathsf{cons}\}$ $\{G_{\mathsf{SW}} = \{V_{\mathsf{SW}}, E_{\mathsf{SW}}\}, \mathsf{duration}, \mathsf{type}\}$
output: Mapping_Set M

```
1  dist ← compute_time_distances(G_HW, t_min);
2  BL_s ← compute_signal_bottom_levels(G_SW, duration);
3  BL_t ← compute_timer_bottom_levels(G_SW, duration);
4  IA ← compute_Input_Admissible(G_SW, G_HW, BL_t, BL_s, dist);
5  A ← compute_Admissible(G_SW, G_HW, BL_s, dist);
6  offsets ← compute_offsets(G_SW, G_HW, BL_t, BL_s, dist);
7  foreach S_SW ∈ V_SW do
8      if type(S_SW) == free then
9          S ← timer_successor(S_SW);
10         foreach s ∈ Input_Admissible(S_SW) do
11             p ← compute_min_paths(s, S);
12             M ← M ∪ {p};
```

In what follows, we will illustrate the characteristics of these two types of OSs through examples. Then we present our methodology of deriving code skeletons for the two types of WSN OSs, starting from the paths identified through our mapping heuristic.

3.4.1 Multi-threaded OS Code Skeleton Generation

3.4.1.1 Multi-threaded OSs for WSNs

The multi-threaded programming model for WSNs is close to the classical programming model, where several threads reside in the same memory space and context switches occur for changing the current active thread.

Mantis OS (MultimodAL NeTworks of In-situ Sensors) [7] is a multi-threaded operating system designed for WSNs aiming for a low memory footprint. Its design resembles classical UNIX-style schedulers and it implements a subset of POSIX threads [11].

Mantis OS consists of a lightweight kernel (less than 500bytes) with integrated scheduler, a low level communication stack for serial or radio communication interfaces

and a device abstraction layer that provides uniform access to devices of all sorts. On top of these, a platform-independent System API is layered.

Its scheduler implements the preemptive time-sliced priority based scheduling with round-robin semantics within a priority level. The CPU time is divided into time slices and every time slice the scheduler is run to choose which process among the ones with highest priority should be granted the CPU for the next time slice. The 5 priority levels in descending priority order are: kernel, sleep, high, normal and idle. The thread synchronization is achieved through binary (mutex) and counting semaphores. The kernel creates at startup a low priority *idle thread* which runs when all the other threads are blocked and which is used to implement power management by deciding which power saving mode has to be activated.

The RAM space has 2 sections: the one for statically allocated structures and the rest of the RAM, managed as a heap. At thread creation, stack space is allocated for the thread out of the heap and this space is recovered when the thread exits. Dynamic allocation is possible but not encouraged.

MantisOS statically allocates a thread table with a fixed number of thread entries defined at compile time and defaulting to 12. Each entry contains a stack pointer, a pointer to the thread's starting function, the thread's priority level and a next thread pointer. A thread's context (including saved registers) is saved on its stack when the thread blocks. The kernel also maintains ready-list head and tail pointers for each priority level.

Networking communication is typically realized as a layered network stack. In Mantis, the different layers of the stack can be implemented as one or more user-level threads with the remark that this thread implementation regards layer 3 and above, as the MAC layer is performed by the communication layer, located in a separate layer of the OS, below the user-level networking stack. Aiming at minimizing memory buffer allocation through layers, the data body for a packet is common through all layers within a thread. This way, data copies are avoided in a "zero copy" approach.

The communication ("comm") layer provides a uniform unified interface for communications device drivers (serial, radio). It manages packet buffering and synchronization functions. The application threads interact with communication devices through 4 function calls: *com_send*,*com_recv*,*com_mode* and *com_ioctl*. The first 3 deal with sending, receiving and changing power modes of a device respectively. The fourth has

a particular significance according to the device. The comm layer manages a queue of buffers (comBufs) with the goal of enabling the "zero copy" approach. The sending of packets can be synchronous while the receiving is always asynchronous. The comm layer is fully interrupt driven, thus achieving zero-polling.

Another example of multi-threaded kernel is FreeRTOS [37]. It is a mini real time kernel, not a fully fledged operating system. Such a kernel is responsible for managing system resources (processor, memory, I/O peripherals). The reduced functionalities offered by a kernel make it more suitable for the constrained WSN nodes than an operating system because extensive operating system support (file systems, virtual memory) is not only not required but even cumbersome.

Just like Mantis OS, FreeRTOS is mostly written in standard C. It features a round-robin priority based scheduler. The scheduler can be configured as either preemptive or collaborative. The preemptive scheduler can preempt a running task to give CPU resources to another task that is ready to run. This feature is used for CPU time sharing between ready tasks with the same priority. It is also used in case of an interrupt which may wake up a task waiting for a signal or for some data. The woken task should have a higher priority than the current running task to be allocated CPU time directly. In cooperative scheduling, context switches only occur if a task blocks or voluntarily yields.

FreeRTOS offers message queues for inter-task communication and binary semaphores for synchronization. The queue mechanism can be used to exchange data either between tasks or between tasks and Interrupt Service Routines(ISR). Tasks can block on a queue when reading to wait for data to become available; similarly it can block when writing if the queue is full and it has to wait for space to be freed on the queue. The access between blocked tasks on the same queue follows priority rules based on tasks individual priorities.

FreeRTOS offers 3 memory management schemes. The simplest one allocates a huge table in the memory called the heap. A memory allocation system call increments the upkeep pointer by the size of the zone requested, while the memory freeing behavior is not implemented in order to avoid the overhead of a garbage collection algorithm. A second scheme uses the best-fit algorithm to re-allocate previously freed memory blocks. The third scheme uses the standard malloc() and free() functions. The latter 2 schemes are not deterministic and can affect the real-time behavior of the system.

3.4.1.2 Multi-threaded OS Code Skeleton Generation Method

In what follows, we will consider a multi-threaded OS that offers the possibility of using message queues for inter-process and inter-thread communication. In this context, the events that trigger transitions in the software automaton (hardware or software events) should place corresponding messages on the queue associated to the software protocol. Thus, Interrupt Service Routines (ISRs) corresponding to hardware events as well as the other software applications initiating requests to the software protocol should place a message on its message queue. These messages are further retrieved and treated appropriately by the software protocol handler.

With these hypotheses in mind, the mapping of the protocol onto a device is realized in its protocol handler and can be achieved through several nested **switch** statements enclosed in an infinite loop, as seen in algorithm 2.

The outer level **switch** statement branches through all the software protocol states $\mathsf{SW}_i \in V_{\mathsf{SW}}, i \in \{1, ..., |V_{\mathsf{SW}}|\}$ (lines 3 – 48). The **case** branch corresponding to a fixed state contains only a call to $\mathsf{wait_event}()$ on the message queue, followed by the evaluation of the type of event retrieved by this call as seen in lines 37 – 46.

Thus, for a fixed state SW_i, that has m out-transitions of the form $\mathsf{SW}_i \xrightarrow{\mathsf{trigger}_j/\mathsf{guard}_j/\mathsf{actions}_j} \mathsf{next}(\mathsf{SW}_i, j)$, where the $\mathsf{trigger}_j$ is either a signal or a timer event, guard_j is the guard of the transition and $\mathsf{actions}_j$ stands for the actions (clock resets, variables modifications) to be performed upon this transition, the corresponding code skeleton branches through the m possible triggers of a software transition, tests the corresponding guard, makes the software state change, performs the actions and finally breaks out of the **switch** statements only to re-enter the appropriate branch of the outer **switch**, corresponding to the new software state, as seen in lines 41–45.

For each free state SW_i, there exist several physical states into which the device automaton might be found at the moment of entering the free state, states that have previously been identified in the $\mathsf{Input_Admissible}(\mathsf{SW}_i)$ set. The second level of **switch** statements branches through the candidates states in the $\mathsf{Input_Admissible}(\mathsf{SW}_i)$ set.

For every candidate physical start state $\mathsf{SS}_k \in \mathsf{Input_admissible}(\mathsf{SW}_i)$ there exists a path $P_k = \{v_{k,1}, v_{k,2}, ...v_{k,n_k}\}$ and associated duration assignment set $T_k = \{t_{k,1}, t_{k,2}, ...t_{k,n_k}\}$ to which there corresponds a **switch** (the third level) statement in the pseudocode, branching through states $v_{k,1}$ to v_{k,n_k-1} and the start state

Algorithm 2: Generated Skeleton for Multi-threaded OS

```
1  initialization;
2  while true do
3      switch mac_state do
4          ...;
5          case SW_i
6              if free(SW_i) then /* free state                                  */
7                  switch radio_start_state do
8                      ...;
9                      case SS_k
10                         switch radio_state do
11                             case SS_k
12                                 radio_state ←v_{k,1};
13                                 start_timer(phys_timer, t_{k,1}) e = wait_event();
14                                 break;
15                             ...;
16                             case v_{k,p}
17                                 radio_state ←v_{k,p+1};
18                                 start_timer(phys_timer,t_{k,p+1}) e = wait_event();
19                                 break;
20                             ...;
21                             case v_{k,n_k-1}
22                                 radio_state ←v_{k,n_k};
23                                 start_timer(phys_timer,t_{k,n_k}) e = wait_event();
24                                 break;
25                         switch e do
26                             ...;
27                             case Sig_l
28                                 if guard_l then
29                                     mac_state ← next(SW_i, Sig_l);
30                                     radio_start_state ← radio_state;
31                                     actions_l;
32                                     break;
33                             ...;
34                             case phys_timer
35                                 break;
36                     break;
37             else /* fixed state                                              */
38                 e = wait_event();
39                 switch e do
40                     ...;
41                     case trigger_j
42                         if guard_j then
43                             mac_state ← next(SW_i, j);
44                             actions_j;
45                         break;
46                     ...;
47             break;
48         ...;
```

for this path, SS_k. Each branch $v_{k,p}$ of the statement ($p \in 1, .., n_k - 1$) starts by making a physical state transition to state $v_{k,p+1}$ then activates the phys_timer timer for the duration $t_{k,p+1}$ and then yields through a call to wait_event(), waiting for the posting of an event on the message queue, as seen in lines 16-19.

The events that could cause a potential software or hardware state change are dealt with in a following **switch** statement in lines 25 to 35. Since SW_i is a free state, the only triggers that could cause a software state change besides the timer event defining its duration, are signals. The only event that could cause a hardware state change is a timer event from phys_timer.

Thus, for every out transitions of the form $\mathsf{SW}_i \xrightarrow{\mathsf{Sig}_l/\mathsf{guard}_l/\mathsf{actions}_l} \mathsf{next}(\mathsf{SW}_i, \mathsf{Sig}_l)$, where the Sig_l is a signal, guard_l is the guard of the transition and $\mathsf{actions}_l$ stands for the actions (clock resets, variables modifications) to be performed upon this transition, the corresponding code skeleton tests the corresponding guard, makes the software state change, performs the actions and finally breaks out of the **switch** statement only to re-enter the appropriate branch of the outer **switch**, corresponding to the new software state, as seen in lines 27–32.

If the event posted on the message queue corresponds to a timer event from phys_timer, the only action to perform is to break out the **switch** statements and re-enter the corresponding **cases** in order to perform the hardware state change, as seen in lines 34-35.

3.4.2 Event-driven OS Code Skeleton Generation

3.4.2.1 Event-driven OSs for WSNs

Three of the main drawbacks of a classical multi-threaded operating system with respect to memory constrained WSNs nodes are: the memory requirements for stack space (since the exact needs are not known in advance, the stack is usually over-provisioned) of multiple threads, the computational overhead of managing several threads or processes and finally the need for locking mechanism to prevent concurrent threads from modifying shared resources. The event-driven run-to-completion single-threaded execution paradigm attempts to tackle these drawbacks.

The basic principle behind the event-driven programming model is that the flow of the program is determined by events initiated by hardware interrupts. Once such an event occurs, the corresponding handler function is called and runs to completion. Thus processes in this paradigm are represented by event handlers. New events can be put in

the event list while a function is running. Following the run-to-completion semantics, an event handler cannot be blocked or interrupted, therefore all processes can share the same stack. This is a significant improvement in memory usage and computational overhead when compared to multi-threaded OSs. Also, locking mechanisms aren't usually needed since two handlers never run concurrently.

A common particularity of these OSs is the non-blocking operations implemented using split-phase form. Under this approach, a command to start an operation returns immediately, while the completion of the operation is signaled through a callback. For example, to acquire a sensor reading with an analog-to-digital converter (ADC), software writes to a few configuration registers to start a sample. When the ADC sample completes, the hardware issues an interrupt, and the software reads the value out of a data register.

The main loop of such an application consists of an infinite loop that runs the handler task corresponding to the first event in the scheduler list as seen in Algo 3.

Algorithm 3: Event-driven application main loop

```
1 initialization;
2 while true do
3     e = get_event();
4     e();
```

TinyOS [26] is among the first operating systems specifically targeted for the WSNs limitations and their applications. It is a highly popular OS in the WSNs community and it offers a rich library of networking and application components available for re-use.

TinyOS implements the event-driven paradigm where every execution is triggered by some external event representing hardware interrupts coming from the radio, timers or sensor interfaces.

TinyOS defines a simple **component model**. In order to do so, it uses a special description language NesC [24] - an extension of the C language. The NesC compiler consists of a preprocessor translating a NesC application into a C module, where only the necessary parts of the operating system are compiled with the application and unused parts are omitted.

In some ways, nesC components are similar to objects. For example, they encapsulate state and couple state with functionality. Each TinyOS component has a specification, a code block that declares the functions it provides (implements) and the functions that it uses (calls). In order to simplify the specification of components, nesC provides the concept of interfaces which are collections of related functions. An interface declares a set of functions called **commands** that the interface provider must implement and another set of functions called **events** that the interface user must implement. For example, a Send interface meant to send packets declares a send command and a sendDone event. If a component "provides" the Send interface, it must define the send function and it can signal a "sendDone" event. If a component "uses" the Send interface, it must define the sendDone event and can call the send command. Connecting components that are interface providers to components that are interface users together is called wiring. Therefore, a TinyOS application consists of a scheduler and a graph of components "wired" together.

There are two sources of concurrency in TinyOS: tasks and events. Tasks are a deferred computation mechanism running to completion without preempting each other. A component can post a task through an operation that immediately returns, while the task will be dispatched later by the scheduler in first come first served (FCFS) order. Events also run to completion, but may preempt the execution of a task or another event. Events signify that a hardware interrupt has been received or that a split-phase operation has been concluded. Under the split-phase approach an operation request is performed by a command while the operation completion is performed by the execution of an event.

Although the event-driven approach seem to be suitable for the resource constrained WSN nodes, it is not flawless. First, in a purely event-driven operating system, a lengthy computation running to completion (for example in the case of cryptographic operations) completely monopolizes the processor, penalizing the responsiveness to other events. This would not be the case in a multi-threaded OS. Also, the split-phase concept makes it hard for a programmer to follow the control flow of its program and requires skills that make programming prohibitive for non-experts.

Contiki [17] is an hybrid operating system that tries to combine the benefits of both worlds. The system is based on an event-driven kernel where preemptive multi-tasking

is implemented as an application library that is optionally linked with programs that explicitly require it.

A running Contiki system consists of the kernel, libraries, a program loader and a set of processes. A process is defined by an event handler function and optionally a poll handler function. All processes share the same address space and the interprocess communication is done by posting events. The kernel's event scheduler dispatches events from the system queue to running processes and periodically calls the processes' polling handlers. Event handlers run to completion once they have been scheduled.

There are two types of events: *asynchronous* - that behave like a deferred procedure call, being enqueued by the kernel and dispatched later- and *synchronous*- that are immediately scheduled by the kernel and the control returns to the posting process only after the target finished processing it.

The polling mechanism is used by the processes that operate near the hardware to check for status updates of hardware devices. When a poll is scheduled, all processes that implement a poll handler are called in priority order.

Because of the run-to-completion semantics, an event-driven programming model does not support a blocking wait abstraction. Thus when an operation cannot complete immediately, it must be split across multiple invocations of the event handler in a state-machine style. To tackle this problem, Contiki provides protothreads [18]. It is a programming abstraction that provides a conditional blocking wait operation on top of an event-driven system. Protothreads are very lightweight when compared to threads in the sense that all protothreads run on the same stack and context switching is done by means of stack rewinding. Unlike a thread, a protothread runs only within a single C function and cannot span over other functions. It can call functions, but cannot block inside of a called function. Another limitation when compared to threads is that function local variables allocated on the stack must be explicitly saved before a blocking wait since otherwise they will be destroyed by the stack rewind. When a blocking wait is reached, the state of the function (CPU registers and program counter) excluding the stack are captured. When the waiting is resumed, the function is reset to what it was before the waiting call was encountered.

3.4.2.2 Event-driven OS Code Skeleton Generation Method

In this context, the code skeleton corresponding to this type of OS consists in a main software protocol handler function having as argument the event which triggered its execution. Every event triggering transitions in the software automaton (signal or timer event) posts in the event queue a call to the protocol handler function corresponding to that event.

With the split-phase approach, the invocation of an operation and its completion are 2 separate phases of execution. Thus, every software automaton state SW_i is duplicated: SW_i_init and SW_i_term. The software protocol handler consists of 2 levels of imbricated **switch** statements. The outermost **switch** statement branches through all the duplicated states of the software automaton SW_i_init, SW_i_term, where $\mathsf{SW}_i \in V_{\mathsf{SW}}, i \in \{1, ..., |V_{\mathsf{SW}}|\}$.

In the state SW_i_init, the timers triggering transitions out of it in the software automaton are set to the corresponding values (algorithm 4 line 4), after which the change of software state to SW_i_term is performed.

If SW_i is a free state, the current physical state is identified among the potential candidates previously determined in Input_Admissible(SW_i) - algorithm 4 lines 6-15. To each candidate state $\mathsf{SS}_k \in$ Input_Admissible(SW_i) there correspond a path $P_k = \{v_{k,1}, v_{k,2}, ...v_{k,n_k}\}$ and associated duration assignment set $T_k = \{t_{k,1}, t_{k,2}, ...t_{k,n_k}\}$. Once the physical start state is identified, the corresponding path is initiated by performing the first transition of the path - SS_k to $v_{k,1}$ (lines 9-13). Further transitions are dealt with in the **case** branch corresponding to SW_i_term.

In the state SW_i_term, the argument of the software handler is tested across the possible events that could trigger either software state changes (signals or the software timer) or the physical state change, according to the physical state timer (lines 18-30).

Thus, if the state SW_i has m out transitions of the form $\mathsf{SW}_i \xrightarrow{\mathsf{trigger}_j/\mathsf{guard}_j/\mathsf{actions}_j} \mathsf{next}(\mathsf{SW}_i, j)$, where the $\mathsf{trigger}_j$ is either a signal or a timer event, guard_j is the guard of the transition and $\mathsf{actions}_j$ stands for the actions (clock resets, variables modifications) to be performed upon this transition, the corresponding code skeleton branches through the m possible triggers of a software transition, tests the corresponding guard, makes the software state change, performs the actions and finally breaks out of the **switch** statements (lines 20-24).

In the case of a phys_timer event, the transition to the next state in the current path is performed- lines 26 to 30.

Algorithm 4: Generated Skeleton for Event-driven OS

```
   input : arg
1  switch mac_state do
2      ...;
3      case SW_i_init
4          set_timers(SW_i_timers);
5          mac_state ⟵ SW_i_term;
6          if free(SW_i) then /* free state                                  */
7              switch phys_state do
8                  ...;
9                  case SS_k
10                     phys_state ←v_{k,1};
11                     start_timer(phys_timer,t_{k,1});
12                     p ← 1;
13                     break;
14                 ...;
15             break;
16     ...;
17     case SW_i_term
18         switch arg do
19             ... ;
20             case trigger_l
21                 if guard_l then
22                     mac_state ← SW_next(SW_i,trigger_l)_init;
23                     actions;
24                     Jump to SW_next(SW_i,trigger_l)_init case branch by posting an event
25             ...;
26             case phys_timer
27                 p++;
28                 phys_state ⟵v_{k,p};
29                 start_timer(phys_timer,t_{k,p});
30                 break;
```

4

Experimental Evaluation

In order to assess the benefits of the presented methodology, we investigate the mapping of a wireless sensor network MAC layer protocol (i.e. B-MAC) onto a physical device (the CC1100 RF transceiver).

We choose a scenario where n nodes transmit data to one sink as seen in figure 4.1. We consider the nodes to be saturated i.e. having the transmission queue always nonempty and that the nodes use B-MAC's Carrier Sense Multiple Access (CSMA) in order to contend for the channel. We consider that the nodes only transmit packets, thus the Low Power Listening feature of B-MAC is disabled. Each attempt of packet transmission starts with an initial backoff followed by successive congestion backoffs in case of unsuccessful clear channel assessment.

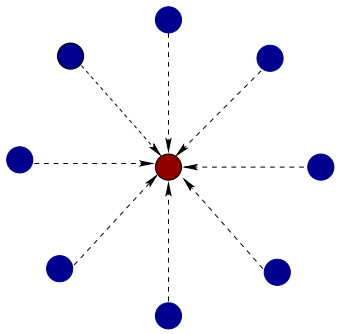


Figure 4.1: Simulation scenario: n saturated nodes send data to 1 sink

This behavior is illustrated on the right side of figure 3.2. After each successful transmission, the protocol switches to an initial wait state namely the WAIT Initial Backoff state. After the elapse of the initial backoff time randomly chosen in a fixed

interval, a transition to CCA Sampling state is enabled. The result of the channel activity check performed in the CCA Sampling state determines the next transition: either to the Transmit state if the medium was found idle or to a next wait state (WAIT Congestion Backoff) in case of a busy medium. The WAIT Initial Backoff and WAIT Congestion Backoff are free states, whose mapping is left at the designer's choice.

The original implementation for B-MAC in the MantisOS networking stack for the CC1000 radio was keeping device in the RX radio state. This simple approach does not take benefit of low-power consumption states in the physical device automaton but it was used due to its simplicity.

We will study several mappings of the initial and congestion backoff wait states:

- map to the RX device state - this simple approach does not take benefit of low-power consumption states in the physical device automaton but its simplicity was used in the B-MAC original implementation in MantisOS for the CC1000 radio device;
- map to the IDLE state taking into account the transitional times between IDLE and RX and their corresponding energy consumption;
- optimized according to the backoff value and making use of all lower power consumption states (IDLE, SLEEP and XOFF).

We chose this particular simulation scenario (n saturated nodes only transmitting and never receiving) because it is the one where the most gain can be illustrated for our optimized mapping of the free wait states. It's clear that the busier the medium, the higher the energy gain if the optimization is performed.

We proceed our investigation on the attainable energy gain at the radio level by deriving a stochastic model in section 4.1. Next, in section 4.2 the theoretical results are confirmed through simulations of the code obtained by adapting the generated code skeleton for the protocol to a real operating system for WSNs - Mantis OS. Finally in section 4.3 the impact of the optimization in terms of energy consumption with respect to the functional parameters of the protocol is evaluated through real-testbed experiments.

4.1 Stochastic Modeling

As mentioned in the chapter introduction, we will derive in this section a stochastic model with the goal of investigating the potential energy gain attainable through our methodology for the particular scenario chosen. First we will derive a model for the behavior of a node and then we will investigate the average energy consumption of such a node.

4.1.1 Energy Consumption Theoretical Analysis

In this section, an energy consumption model for the radio chip for the scenario described in introduction is derived, using as a starting point the throughput analysis obtained in [8].

The analysis is carried out under the assumption of ideal channel conditions (no hidden terminals or capture). The hidden terminal problem occurs when a node is visible from a destination node, but not from other nodes communicating with the said destination node. This can lead to collisions between the packets of nodes that do not "hear" each other. We consider that all nodes are within range and can listen to each other's communications. The channel capture effect is a phenomenon where one user of a shared medium "captures" the medium for a significant time, while the other users are denied use of the medium. We consider that in our scenario, the nodes have equal chances of sending a packetof constant size. Since our protocol has no acknowledgements and no retransmission, collisions will not be taken into account.

Note that these hypothesis do not influence the performance evaluation in terms of energy since we consider a fixed number of saturated stations (the transmission queue of each station is always nonempty) contending for the channel.

Our analysis proceeds by investigating the behavior of a single station which will be modelized by a two-stage Markov chain with the goal of deriving the stationary probability τ that the station transmits in a randomly chosen slot time. In a second step, the average energy consumption is evaluated by observing the events that can occur in a randomly chosen slot time.

Let n be the number of saturated stations contending for the channel and let $b(t)$ be the stochastic process representing the backoff time counter for a given station at a given time. We adopt a discrete and integer time scale, where t and $t+1$ correspond

to the beginning of two consecutive slot times, spaced by exactly the length of the slot time σ. The backoff counter is decremented at the beginning of each slot time.

In the case of B-MAC, there are two backoff stages: the first initial backoff stage traversed by every packed and the congestion backoff stage reached either in case of a busy medium found after the initial backoff time is elapsed or recursively after consecutive congestion backoffs followed by the occurrence of a busy medium. We denote by W_0 and W_1 the length of the backoff interval for the initial and congestion backoffs respectively. Taking into consideration that the value of the backoff counter for each station depends also on its transmission history (whether it was in initial or congestion backoff before), the stochastic process $b(t)$ is non-Markovian.

We denote by $s(t)$ the stochastic process representing the backoff stage of the station at time t. It is equal to 0 for the initial backoff and 1 for the congestion backoff.

We denote by p_b the probability that when performing the clear channel assessment when the backoff counter reaches the value 0, the medium is found busy and we make the approximation that this probability is constant for a given number of contending stations and for a certain packet length l.

We make the notation $P\{(i_1,k_1)|(i_0,k_0)\} = P\{s(t+1)=i_1, b(t+1)=k_1 \mid s(t)=i_0, b(t)=k_0\}$, i.e. the probability for a station to be in the (i_1,k_1) state of the Markov chain at time $t+1$ given that it was in the state (i_0,k_0) state of the Markov chain at time t.

With all these hypotheses in mind, the bi-dimensional process $\{(s(t),b(t))\}$ can be modeled as the discrete-time Markov chain illustrated in figure 4.2. The upper level of this Markov chain corresponds to the initial backoff, while the lower level corresponds to the congestion backoff. A state in the model has 2 indices (i,j), where i denotes the backoff stage – 0 for initial backoff, 1 for congestion backoff – and j denotes the current value of the backoff counter.

For this Markov chain, the non-null transition probabilities are:

$$P\{(i,k)|(i,k+1)\} = 1 \qquad k \in (0, W_i - 2), i \in (0,1) \tag{4.1}$$

$$P\{(0,k)|(i,0)\} = \frac{(1-p_b)}{W_0}, \qquad k \in (0, W_0 - 1), i \in (0,1) \tag{4.2}$$

$$P\{(1,k)|(0,0)\} = p_b/W_1, \qquad k \in (0, W_1 - 1) \tag{4.3}$$

$$P\{(1,k)|(1,0)\} = p_b/W_1, \qquad k \in (0, W_1 - 1) \tag{4.4}$$

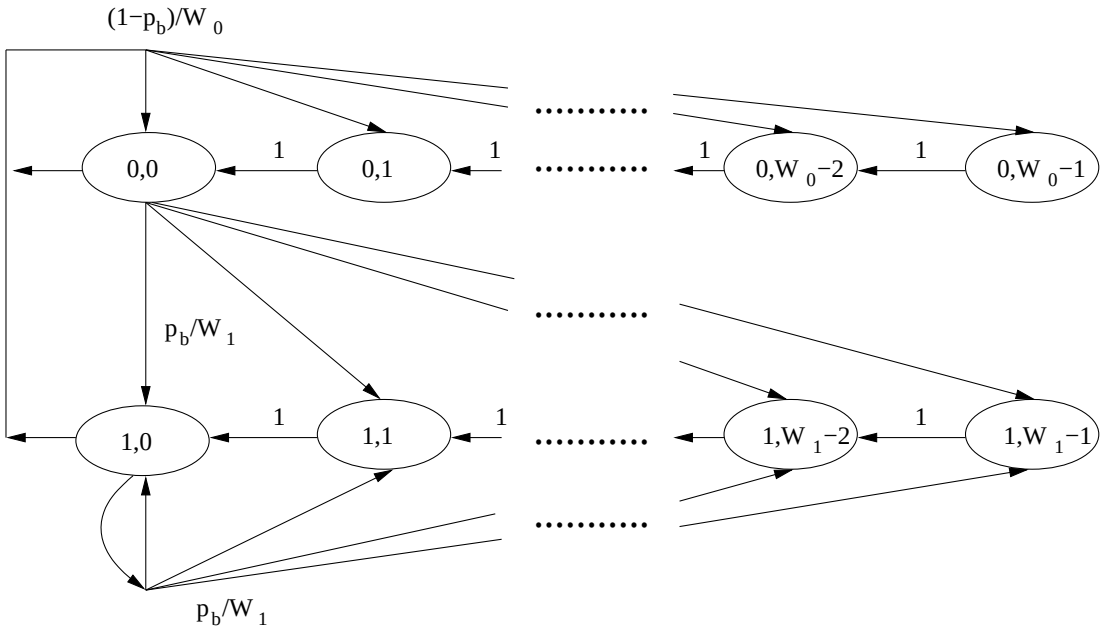


Figure 4.2: Markov chain model for B-MAC backoffs.

Equation 4.1 stands for the fact that the backoff time counter is decremented at the beginning of each slot time. For example, when in state $(1, 1)$ with certainty (probability 1) the next state will be $(1, 0)$.

Equation 4.2 stands for the fact that a new packet after a transmission (i.e. medium is found not busy when performing clear channel assessment) starts with initial backoff stage 0 and the backoff value chosen randomly in the interval $(0, W_0 - 1)$. More precisely, if the station is in the state $(i, 0)$ in the Markov chain, i.e. the backoff counter has reached the value 0 and it is either in initial or congestion backoff state (the variable counter i) it finds the medium to be free with probability $(1 - p_b)$, and after the packet transmission its next state will be one of the possible initial backoff states with probability $1/W_0$.

Equation 4.3 accounts for the situation where after the initial backoff time is elapsed (situation corresponding to the state $(0, 0)$ in figure 4.2), the medium is found busy while performing clear channel assessment (with probability p_b) and afterwards the station passes into congestion backoff stage 1 (a state of the form $(1, k)$ with k between 0 and $W_1 - 1$), with the backoff time counter value chosen randomly in the interval $(0, W_1 - 1)$.

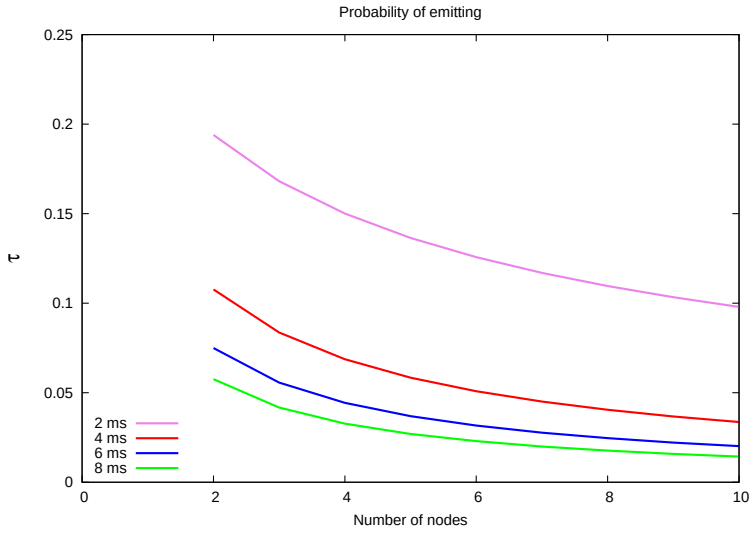


Figure 4.3: Transmission probabilities for different packet lengths and number of nodes

Similarly, equation 4.4 accounts for the situation where after a congestion backoff time is elapsed (situation corresponding to the state $(1,0)$ in figure 4.2) and the medium is found busy while performing clear channel assessment(with probability p_b), the station remains into congestion backoff stage 1 with the backoff counter value chosen randomly in the interval $(0, W_1 - 1)$.

Let $b_{i,k} = \lim_{t\to\infty} P\{s(t) = i, b(t) = k\}$, $i \in (0,1)$, $k \in (0, W_i - 1)$ be the stationary distribution of the chain. In the following, we will determine a closed form solution for this Markov chain. First we shall note that:

$$b_{1,0} = \frac{p_b}{W_1} b_{0,0} + b_{1,1} + \frac{p_b}{W_1} b_{1,0} \tag{4.5}$$

$$b_{1,k} = \frac{p_b}{W_1} b_{0,0} + b_{1,k+1} + \frac{p_b}{W_1} b_{1,0}, k \in (0, W_1 - 2) \tag{4.6}$$

By recursively substituting 4.6 into 4.5, we obtain:

$$(1 - p_b) \times b_{1,0} = b_{0,0} \times p_b \tag{4.7}$$

Let us note that: $b_{0,k} = \frac{(1-p_b)}{W_0} b_{0,0} + \frac{(1-p_b)}{W_0} b_{1,0} + b_{0,k+1}$, $k \in (0, W_0 - 2)$ which yields after recursive substitutions:

$$b_{0,k} = \frac{(W_0 - k) \times (1 - p_b)}{W_0} \times (b_{0,0} + b_{1,0}), k \in (0, W_0 - 2) \tag{4.8}$$

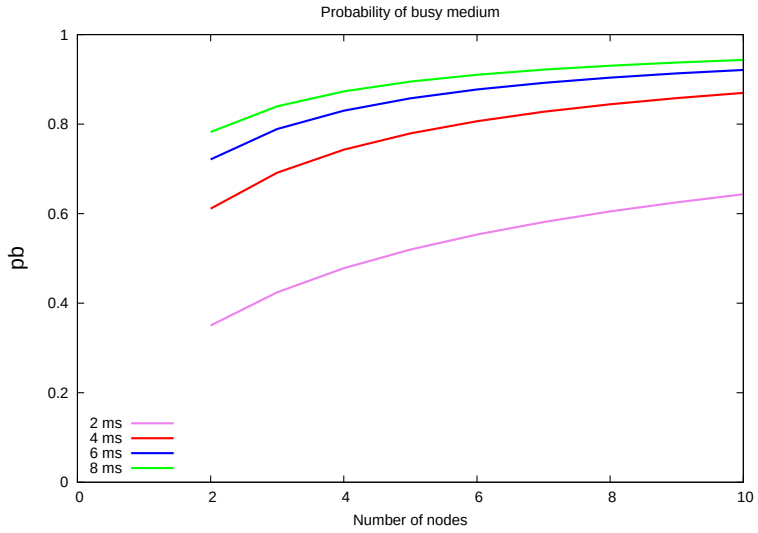


Figure 4.4: Busy medium probabilities for different packet lengths and number of nodes

Similarly, let us note that: $b_{1,k} = \frac{p_b}{W_1} b_{0,0} + \frac{p_b}{W_1} b_{1,0} + b_{1,k+1}$, $k \in (0, W_1 - 2)$ which after recursive substitutions yields:

$$b_{1,k} = \frac{(W_1 - k) \times p_b}{W_1} \times (b_{0,0} + b_{1,0}), k \in (0, W_1 - 2) \tag{4.9}$$

Using 4.7, 4.8 and 4.9 become:

$$b_{0,k} = \frac{(W_0 - k)}{W_0} \times b_{0,0}, k \in (0, W_0 - 2) \tag{4.10}$$

$$b_{1,k} = \frac{(W_1 - k)}{W_1} \times \frac{p_b}{(1 - p_b)} \times b_{0,0}, k \in (0, W_1 - 2) \tag{4.11}$$

All the values $b_{i,k}$ are thus expressed as a function of $b_{0,0}$ and the probability of finding the medium busy p_b. By imposing the normalization condition, we obtain:

$$1 = \sum_{i=0}^{1} \sum_{k=0}^{W_i - 1} b_{i,k} = b_{0,0} \times \left[\frac{W_0 + 1}{2} + \frac{p_b}{(1 - p_b)} \times \frac{W_1 + 1}{2} \right] \tag{4.12}$$

By means of 4.7 and 4.12, we obtain:

$$b_{0,0} = \frac{1 - p_b}{(1 - p_b) \times \frac{W_0 + 1}{2} + p_b \times \frac{W_1 + 1}{2}} \tag{4.13}$$

$$b_{1,0} = \frac{p_b}{(1 - p_b) \times \frac{W_0 + 1}{2} + p_b \times \frac{W_1 + 1}{2}} \tag{4.14}$$

At this point, we can express the probability τ that a station transmits in a randomly chosen slot time. Since in our model any transmission occurs when the backoff time counter equals 0 and the medium is found not busy while performing clear channel assessment, we have:

$$\tau = (1 - p_b) \times \sum_{i=0}^{1} b_{i,0} = b_{0,0} = \frac{1 - p_b}{(1 - p_b) \times \frac{W_0+1}{2} + p_b \times \frac{W_1+1}{2}} \tag{4.15}$$

Equation 4.15 expresses the probability that a station transmits in a randomly chosen slot time τ as a function of the probability p_b of finding the medium busy when performing clear channel assessment, which is still unknown. The medium is busy during a randomly chosen slot time if at least one station has started transmitting during one the previous $l - 1$ slots, where l is the packet length expressed in number of slots.Since we consider the modulation speed to be constant, the packet length is directly linked and related to the TX duration.

Denoting by p_i the probability that at least one station starts transmitting in the i^{th} slot before the current slot, we have: $p_i = 1 - (1 - \tau)^n$. The events "at least one station transmits during the i^{th} slot before the current slot" are mutually exclusive for i taking values from 1 to $l - 1$. Thus,

$$p_b = \sum_{i=1}^{l-1} p_i = (l - 1) \times (1 - (1 - \tau)^n) \tag{4.16}$$

Equations 4.15 and 4.16 form a non-linear system of equations in the two unknowns τ and p_b that can be solved using numerical techniques.

Figure 4.3 illustrates the transmission probability for different packet sizes and different number of nodes. As it can be seen, for the same number of sending nodes, the probability of sending a packet is smaller for longer packets. The explanation is that longer packets occupy the medium longer, therefore reducing the probability of sending new packets. As for the number of nodes, the higher the number of senders, the smaller the probability of transmitting.

Figure 4.4 illustrates the probability of finding the medium busy for different packet sizes and different number of nodes. As it can be seen, for the same number of sending nodes, the probability of finding the medium busy is higher for longer packets. The explanation is that longer packets occupy the medium longer, therefore increasing the probability of finding the medium busy. As for the number of nodes, the higher the number of senders, the higher the probability of finding the medium busy.

4.1.2 Average Energy Consumption

The average energy consumption of a station can be readily obtained considering that in a randomly chosen slot time, with probability τ the station transmits a packet. We consider all stations transmit packets of the same "temporal" length. They consume current I_{TX} (the current consumption per unit time for the TX state in the CC1100; see table 3.1) for a duration of l. With probability $(1-\tau)$ the station is in a backoff stage with an average energy consumption $E[B]$:

$$E = \tau \cdot I_{TX} \cdot l + (1-\tau) \cdot E[B]; \tag{4.17}$$

The average backoff energy consumption $E[B]$ can be derived considering that it always includes an initial average backoff energy $E[IB]$ and several successive congestion backoffs of average energy value $E[CB]$, according to the probability of finding the medium free or busy respectively when performing the clear channel assessment.

The average backoff energy (whether it be for an initial backoff or for a congestion backoff) is the sum of products $p(k) \times \mathcal{E}(k)$, where $p(k)$ if the probability of choosing a backoff value of k ($1/W_0$ for initial backoff and $1/W_1$ for congestion backoff respectively, regardless of the particular value of k) and $\mathcal{E}(k)$ is the value of the energy consumed if a backoff value of k is chosen (it takes into account the time spent during transitional states between the optimal state and RX).

$$E[IB] = \sum_{k=0}^{W_0-1} \frac{1}{W_0} \times \mathcal{E}(k); \tag{4.18}$$

$$E[CB] = \sum_{k=0}^{W_1-1} \frac{1}{W_1} \times \mathcal{E}(k); \tag{4.19}$$

The probability that the packet would be successfully sent after the first clear channel assessment is $(1-p_b)$ and the corresponding congestion backoff time is 0 and it also includes 1 CCA time. The probability that the packet would be successfully transmitted on the second try is the probability of finding the medium busy during the first clear channel assessment and free at the second attempt and it includes one congestion backoff of average value $E[CB]$ and 2 CCA times. The probability that the packet would be successfully sent on the i^{th} attempt is the probability of having found the medium busy during the first $(i-1)$ attempts and clear at the i^{th} attempt

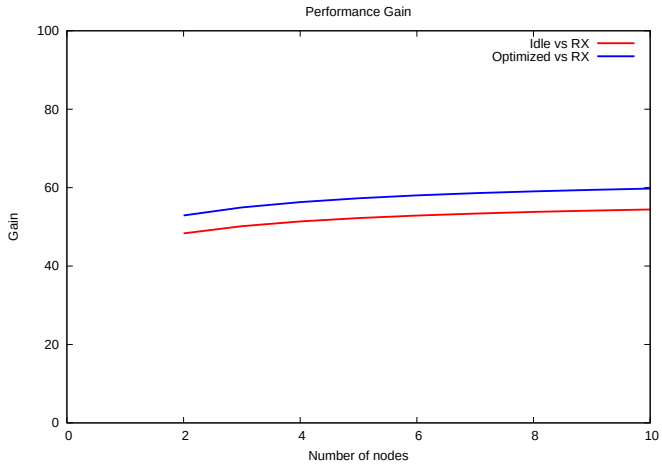


Figure 4.5: Theoretical gain obtained for the improved backoffs mapping with respect to the unimproved version

and it includes $(i-1)$ congestion backoffs and i CCA times, thus yielding an energy consumption of $p_b^{(i-1)}(1-p_b)*[(i-1)*E[CB]+i*\mathcal{E}(CCA)]$ (where $\mathcal{E}(CCA)$ is the energy spent performing clear channel assessment). The average backoff energy consumption is thus:

$$\begin{aligned} E[B] =& E[IB] + \sum_{i=1}^{\infty} {p_b}^{(i-1)} \times (1-p_b) \times [(i-1) \times E[CB] + i \times \mathcal{E}(CCA)]\,; \\ =& E[IB] + \frac{1}{1-p_b} \times \mathcal{E}(CCA) + \frac{pb}{1-pb} \times E[CB]; \end{aligned} \tag{4.20}$$

Figure 4.5 illustrates the gains obtained for two improved implementations of the B-MAC protocol (one which switches the radio chip into IDLE state taking into account the transitional times between IDLE and RX and their corresponding energy consumption and one optimized according to the backoff value and making use of all lower power consumption states) with respect to the unimproved version of B-MAC (which keeps the radio chip into RX state while performing backoffs) for different number of stations contending for the channel (from 2 to 10 stations). As it can be seen, the gain increases with the number of stations.

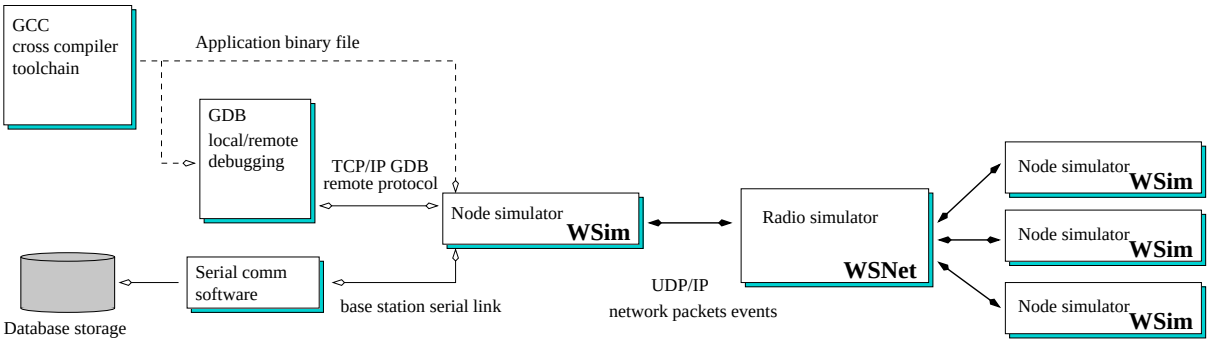


Figure 4.6: Worldsens - distributed simulation environment

4.2 Simulation Results

In order to further validate the theoretical energy gains of the method, we conducted some simulations for the WSN430 platform under the Worldsens [22] simulation environment, an integrated environment for development and rapid prototyping of wireless sensor network applications. In section 4.2.1 we describe the Worldsens framework while in 4.2.2 we describe the simulation set-up and results.

4.2.1 The Worldsens Environment

Worldsens [22] is an integrated environment for development and rapid prototyping of wireless sensor network applications. It consists of two simulators - WSim and WSNet - that can be used either independently or in conjunction.

- **WSim** is a platform simulator that performs *cycle accurate full platform simulation* using microprocessor instruction driven timings.
- **WSNet** is a *modular event-driven* wireless network simulator.

Figure 4.6 presents the global architecture of the simulation platform. The *node simulator* takes as input the binary file of the application. WSim runs the native code as deployed in the sensor hardware without any change. In order to do so, it emulates all components/peripherals embedded in the hardware sensor nodes: flash memory, radio device and so on. All instructions are simulated and in particular those sending commands to the radio device on the mote. Each node simulator can be connected to a central *radio simulator* which deals with aspects such as radio propagation, interference

- basically it simulates the communication between the nodes. As seen in the figure, a remote gdb instance can connect to each node simulator, allowing a step by step execution at instruction level (C or assembly language). The serial port output of the mote can be obtained.

WSim is composed of hardware block descriptions that match the chip level description of the system. Each such block description is available as a software library within the simulation framework. A sensor node platform can be built by selecting the components description and by writing a single file that describes the physical interconnection between these blocks. The hardware simulator uses timed finite state machines to describe the behavior of hardware blocks with a well defined API to connect the block to the different communications interfaces (GPIO ports, USART, SPI, . . .).

WSim is able to perform a full simulation of hardware events that occur in the platform and to give back to the developer a precise timing and performance analysis of the simulated software.

Worldsens includes a *logger* library used to record selected system events (e.g. missed interrupts occurring while interrupts are disabled) or to catch a set of programming errors chosen by the block developer (e.g. sending a byte to an UART without proper initialization).

WSim supports an *event tracer mechanism* that records in a file the activity of selected signals. Several types of events can be recorded such as interrupt arrival, low power modes changes, peripherals activity. The events along with their arrival times logged in special format files can then be used for offline performance estimations and timing validations among the platform peripherals.

WSNet is a modular event-driven wireless network simulator. When used alone, it helps to evaluate, refine and validate the application high level design choices of protocols, traffic pattern, application dimensioning, protocol parameters tuning and physical layer choices. When used in conjunction with WSim, it simulates a whole sensor network with a high accuracy.

WSNet is parameterized with several configuration files describing the characteristics of the physical medium and each network node. The nodes are defined by specifying 3 parameters: the antenna model (i.e. omnidirectional, directiona), the mobility model (i.e. random, static group mobility) and the radio model (i.e. FDMA, CDMA). The wireless medium is defined by specifying 2 parameters: the propagation model

and the interference model. Based on the specified parameters, WSNet computes the SNR(Signal over Noise Ratio) and BER(Bit Error Rate) for eah radio symbol sent over the medium and for each receiving node.

4.2.2 Simulation Set-up

We conducted several simulations for the WSN430 platform under the Worldsens [22] environment for the same simulation scenario described in the chapter introduction: n saturated nodes contending on the channel, trying to send data to 1 sink.

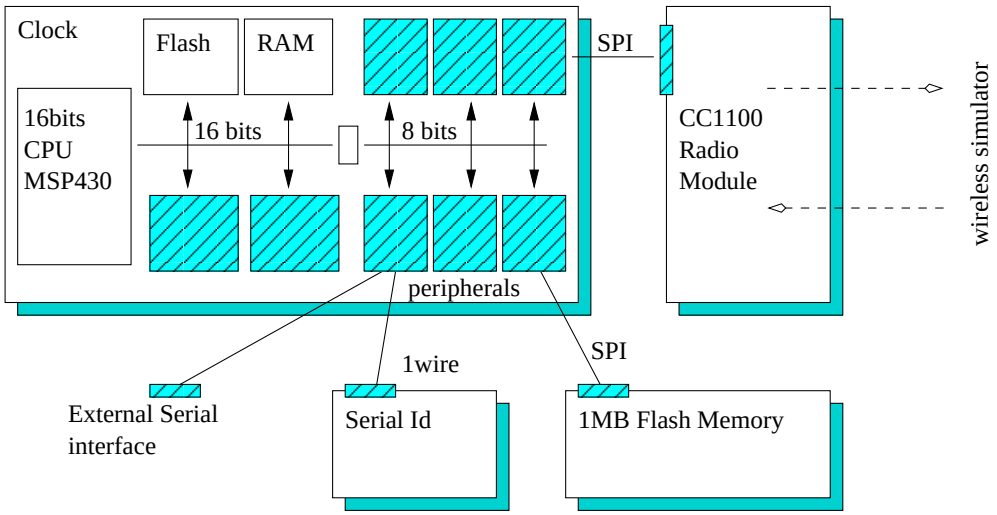


Figure 4.7: Architecture of the WSN430 WSN node

Figure 4.7 presents an example of node that can be fully simulated with WSim - the WSN430. It is the node used in our simulations. The WSN430 platform includes a full Texas Instrument MSP430f1611 micro-controller with its complete instruction set and all digital blocks (timers, basic clock module, serial ports with UART and SPI modes, etc). The microcontroller is connected to external system peripherals: 1MB flash memory module (ST M25P80), serial id (Maxim DS2411), a Chipcon CC1100 packet radio interface.

Figure 4.8 illustrates the steps to follow in performing a WSim simulation as well as the general methodology of the framework. The application code is compiled using cross-compiler tools in order to generate an executable (.elf file) that is given as input to the simulator. Since WSim simulates native code, it is independent of any programming

language and any OS. The feedback from the simulator is used in order to improve the application until the desired performances are attained and the executable can be deployed on the real hardware.

In our case, we adapted the code skeleton obtained for mapping the backoff free states for the contending nodes for Mantis OS[7] - a multi-threaded operating system for WSNs. The design goals and tradeoffs of this WSN dedicated OS were presented in section 3.4.1. We will insist now on some particularities related to its communication stack.

The user-level Mantis OS network stack supports levels three and above, i.e. routing, transport and application layers. The MAC layer support is offered by the communication layer, also named "comm" layer. The latter is located in a separate lower layer of the OS. The MOS comm layer provides a unified interface for communication device drivers (i.e. radio, USB, serial interfaces). It exposes functionality to network or application threads through 4 functions: *com_send*,*com_recv*, *com_mode* and *com_ioctl*.

The *com_mode* function call powers up or powers down the device when needed. It must specify as a parameter on which of the communication devices it acts and what "mode" it selects for the device. In the case of the radio device on the WSN430 node - the CC1100 RF Transceiver, the *com_mode* triggers certain state changes in the radio finite state machine. Its first parameter must be IFACE_RADIO, while the second can be any of the following: IF_IDLE (enable IDLE state of the CC1100), IF_OFF (enable XOFF state of the CC1100), IF_STANDBY (enable SLEEP state of the CC1100), IF_LISTEN (enable the RX state of the CC1100). Hence this call can put the CC1100 in one of the low power modes or the RX state. The call to *com_mode* blocks until the target CC1100 state is reached. The manual calibration and transmission primitives have separate functions.

A particularity of MOS is the zero-copy buffer management policy. Reception happens in the background. The memory for the received packets is managed by the comm layer itself, which owns a number of comBufs. Device drivers may request comBufs, which are allocated to devices. After the device receives a packet it must exchange it for an empty one from the comm layer. The latter buffers the full packets in order and dispatches them to threads when these call *com_recv*. Also, if a full packet for the specified device is not available, the thread calling *com_recv* blocks until one becomes available, i.e. the device receives a packet. The calling thread must explicitly "liberate"

the comBuf at the end, thus informing the comm layer that the buffer may be reused. When a call to *com_send* is made, the sending thread passes a pointer to a packet buffer, called comBuf. The comm layer blocks the sending thread and passes the pointer to the specified device driver. The calls to *com_send*,*com_recv* must therefore specify a comBuf pointer and the device within the comm layer to which they are addressed.

The MAC layer protocol is located within the device driver for the radio which in turn is situated inside the comm layer. We encapsulated the adapted code skeleton within the *com_send* function body. Since the value of timers in the code skeleton is not explicitly required for clock constraints, we used a simple delay function instead-*mos_udelay(delay)* that performs a number of *nop()*-no operation- resulting in the requested delay.

The code thus obtained for the optimized version of the mapping of the BMAC backoffs (using all low power modes according to the backoff value) was compiled using the GCC toolchain for the Texas Instruments MSP430 family. The latter includes the GNU C compiler (msp430-gcc), the assembler and linker (binutils), the debugger (GDB), and some other tools needed to make a complete development environment for the MSP430. The resulting binary was emulated by WSim in the network scenario simulated by WSNet.

We need to make a remark regarding the received signal strength indicator (RSSI) in WSNet. The network simulator was used in a mode fow which no signal attenuation was computed. A packet ocuppying the medium "produces" a constant RSSI of 0 dBm, while a free medium is equivalent to an RSSI of -110dBm. Hence testing for Clear Chanel Assessment is equivalent to testing whether the CC1100 computed value for RSSI is equal to 0 or not.

We chose the autocalibration option of the CC1100 performing the calibration every time when going from IDLE to RX/TX.

The traces produced by WSim allow a very accurate evaluation of the energy consumption of each peripheral device as well as the entire platform. Several types of events are reported(interrupts arrival, peripherals activity). Recording all such events with their corresponding arrival time is important for performance validation. WSim traces are output in binary format that can be transformed in the Gnuplot, VCD or linear formats. The most commonly used output format is VCD, which stands for Value Change Dump.

In our particular simulation case, for each node a *.vcd* file is created recording events for signals and states of the microcontroller, the ds2411, the cc1100 radio. For the radio, it logs events on the internal state, the strobes, the CSn, SO, GDO0 and GDO2 pins. At the top of the file, variable names are defined for the signals to be recorded. The records in the *.vcd* file start with a line denoting a time instant. The value of the time instant is preceded by a # character. On consecutive lines follow $(b < value >, < variable >)$ pairs denoting the signals which have changed their value at the time instant of the record and the corresponding value. The value is coded in binary with fixed number of bits and is preceded by a b character.

From the *.vcd* trace files obtained for the radio signals for the optimized version of mapping of the backoffs we generated the other two mappings-the one turning the radio state machine to IDLE when the backoff value was sufficiently large and the one keeping the radio in RX during backoffs. After eliminating the initialization part of the WSN430 from the traces, we computed the average energy spent for sending 1 packet for the three versions of BMAC.

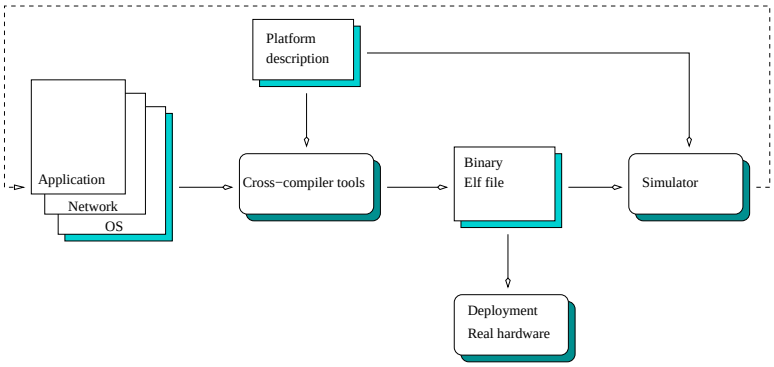


Figure 4.8: WSim flow of events

Figure 4.9 illustrates the normalized energy consumed by the radio device for the two improved implementations of the B-MAC protocol (one switching the radio into IDLE and one optimized according to the backoff value using all lower power consumption states, both taking into account the transition times of the intermediate states) with respect to the energy consumed by the unimproved version of B-MAC (keeping radio into RX state while performing backoffs). The observed energy gain at the radio level is

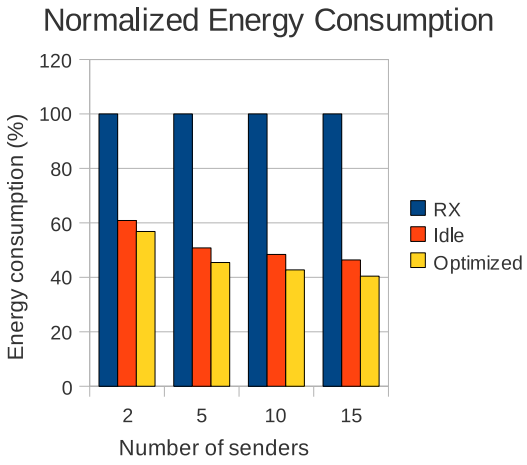


Figure 4.9: Energy consumption of the improved protocol versions normalized with respect to the unimproved version

significant and it confirms the theoretical gain values obtained by the stochastic model described in [12].

4.3 WSN Testbed Experiments using the SensLAB platform

4.3.1 SensLAB goals and facilities

The process of design, implementation, deployment and evaluation of software and protocols for WSN appears to be tedious and error prone, mostly due to the distributed nature of this type of systems. Simulators play an important role in the evaluation of a WSN application, but they are not sufficiently accurate as they always rely on simplifying assumptions.

Real experiments are needed in order to correctly assess the performance and parameters of the designed software. However, such experiments are facing serious challenges as soon as the number of nodes increases: the limited capacities of the nodes in terms of debugging and programming, the process of deployment involving manipulation of each individual node, the limited lifetime of nodes' batteries, etc.

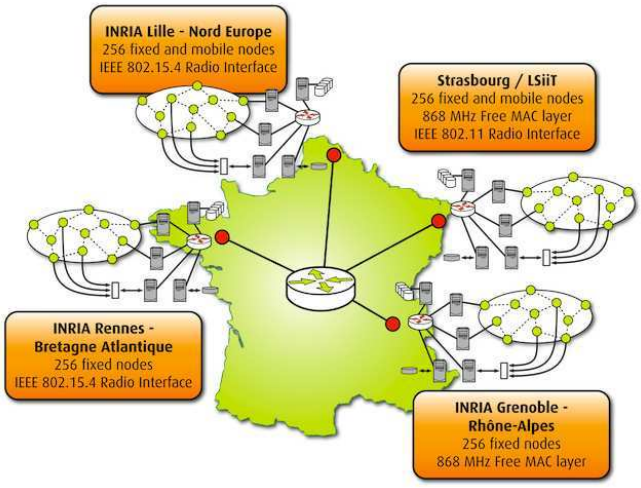


Figure 4.10: SensLAB platform

SensLAB [10] is a very large scale open wireless sensor network testbed that has been developed and deployed in order to allow the evaluation of scalable wireless sensor network protocols and applications. It is composed of 1024 WSN430 nodes evenly distributed across 4 sites: INRIA Grenoble, INRIA Lille, INRIA Rennes and University of Strasbourg/LSIIT, as seen in figure 4.10. In order to cover a wide range of experimental scenarios, the sites have different nodes emplacements: the Grenoble and Rennes sites have only indoor fixed nodes, Lille and Strasbourg sites have mobile nodes in addition to the fixed ones, while Strasbourg also has outdoor fixed nodes.

The SensLAB platform offers several facilities which we will briefly cover in what follows.

It is generic, open and flexible in the sense that it makes no restrictions on the programming model, language or Operating System of the applications to be deployed on the nodes. The platform user is offered an easy way to set-up an experiment through a webportal. He can either choose the individual nodes from a map of the site or allow the platform's scheduler to choose the nodes based on his sensor and radio characteristics specifications and the required experimental time.

The platform offers reliable remote access to each individual node in an experiment allowing the reset, stop, start or code re-flashing at any time during the experiment.

Another important facility offered by this testbed is the non intrusive application

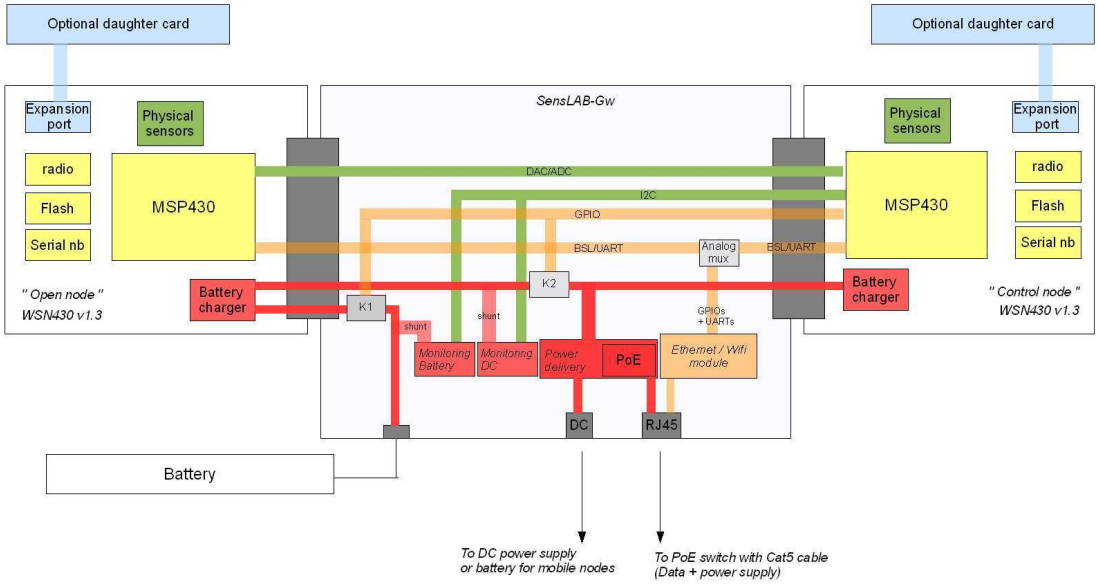


Figure 4.11: SensLAB node architecture

transparent real time monitoring of the sensor nodes (energy consumption and radio activity on each individual node). The monitored data is securely logged for each individual experiment.

4.3.2 SensLAB Software and Hardware Infrastructures

4.3.2.1 SensLAB Hardware Infrastructure

Each one of the 4 sites is composed of SensLAB nodes relied by networking backbone. The latter provides power and connectivity to the individual nodes as well as communication with the exterior for command and monitoring purposes.

The SensLAB node is composed of 3 hardware elements, as seen in figure 4.11:

- the WSN430 open wireless sensor node : it is the node accessible to the user during his experiment for deploying his application

- the control WSN430 wireless sensor node : its role is to control the open node allowing to power up/power down, reset, monitor the open node activity (power consumption, radio activity) and even send artificial stimuli to the open node

- the node gateway: it connects the 2 WSN430 nodes between themselves as well as with the SensLAB server, allowing the user to command and to retrieve the data on the open node's serial link

The WSN430 nodes are built on top of a low-power MSP430 16bit micro-controller running at 8Mhz. Each WSN430 platform includes a 6bytes DS2411 electronic registration number that provides a unique identity to the node, an external 1MB flash memory ST M25P80. There are 2 types of WSN430 platforms based on the radio chip used: WSN430v13b (using a CC1100 radio chip) and the WSN430v14 (using the CC2420 radio chip).

4.3.2.2 SensLAB Software Infrastructure

Figure 4.12 illustrates the software architecture replicated over the 4 sites. We will cover the software components while following the experimental set-up that a user must perform.

The first step that must be followed when conducting an experiment is the configuration through the **webportal**. The user must specify a name for the experiment, a duration and possibly a start date. There are 2 possibilities for choosing the nodes: either the user chooses them from a map, thus having a direct choice on the topology, or he specifies their type and characteristics and allows the platform to make the choice for him. Each node in the software must be a associated with a given firmware. The user must also specify a profile associated to his experiment, i.e. an ensemble of values for the radio (CC1100 or CC2420), power mode (line or battery), polling measurements (type and frequency).

Based on the user's choices and node availability, the platform's **batch scheduler software** determines a start time for the experiment. This server side module is based on OAR [1] batch scheduler for large clusters and it performs optimal scheduling and resource allocation for experiments.

[1] http://oar.imag.fr

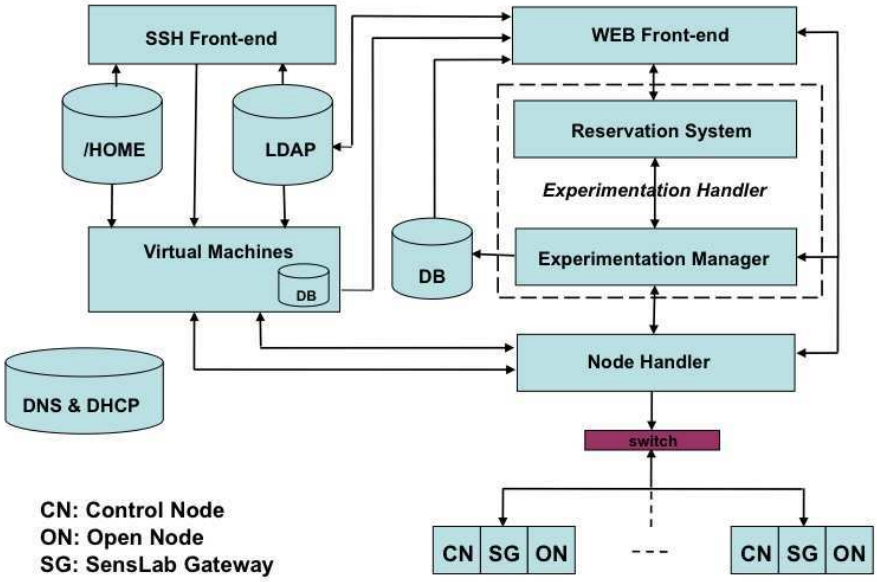


Figure 4.12: SensLAB software infrastructure

At the start time determined by the batch scheduler, the configuration of control nodes, firmware deployment and node resets are performed automatically by the SensLAB platform.

The **experiment handler software** is the server-side module that handles the interaction with the 256 nodes of its corresponding site. It handles firmware updates, energy consumption monitoring, polling, receives the data coming from the serial links of open nodes.

The **user virtual machine** helps the user by offering him a complete Linux environment as well as a set of development tools, cross compilation chains, OSs, drivers, communication libraries and simulators. From this environment, the user can launch a dedicated software (*senslab-cli*) and interact directly with the individual nodes(start, stop, reset, re-flash) of the current experiment.*Senslab-cli* achieves this functionalities by connecting itself to the **experiment handler software** which further treats the user's requests. Another dedicated application allows the user to retrieve the output of the nodes' serial link.

4.3.3 SensLAB Experimental Set-up and Evaluation

As mentioned previously, WSN simulators play an important role in the early stages of software design, prototyping and evaluation. However, this validation is not sufficient due to the simplifying assumptions made in the design of the simulators. Therefore real-platform experiments need to be carried out.

With this goal in mind, we have used the testbed located in Grenoble composed of 256 WSN430v13b (using a CC1100 radio chip) fixed interior sensor nodes.

4.3.3.1 SensLAB Experimental Scenario and Evaluation Metrics

We consider the same simulation scenario as before: n saturated nodes contending for the channel using B-MAC's CSMA (no LPL) and we are interested in the impact of the optimization in terms of energy with regard to the functional properties of the protocol. To this end we chose 2 metrics:

- the average number of packets sent by a node in a fixed time interval
- the average number of packets received by the sink per node in fixed time interval and the associated standard deviation

For an experiment with n nodes, if we denote by Ns_i the number of packets sent by the i^{th} sender node during the experimental time interval, the first metric Avg_{sent} will be equal to the ratio of the total number of sent packets divided by the number of sender nodes:

$$Avg_{sent} = \sum_{i=1}^{n} Ns_i/n \tag{4.21}$$

For an experiment with n nodes, if we denote by Nr_i the number of packets received by the sink from the i^{th} sender node during the experimental time interval, the second metric Avg_{rec} will be equal to the ratio of the total number of packets received by the sink (from all the sender nodes) divided by the number of sender nodes:

$$Avg_{rec} = \sum_{i=1}^{n} Nr_i/n \tag{4.22}$$

The standard deviation will be computed as follows:

$$S_{rec} = \sqrt{\frac{1}{N}\sum_{i=1}^{n}(Nr_i - Avg_{rec})^2} \tag{4.23}$$

Energy consumption The SensLAB platform allows the periodic sampling of the current, voltage, power, rssi, luminosity and temperature values for some predefined sampling period values. The physical quantities for which a recording is needed must be specified in the experimental profile. For the current, voltage and power, these predefined sampling periods are: 70ms, 100ms, 500ms, 1s and 5s.

The platform creates one file per physical quantity per experiment and records the values of the selected physical quantities in files containing successive lines of text of the form <node_number, time_instant, value>. The values of the physical measured quantity for all the nodes in the platform will therefore appear in one file. Computing the energy consumed during the experiment time interval for a node implies extracting the values corresponding to that node from the file and integrating either the power over the discrete sampling time moments or the product of the sampled current intensity and voltage over the discrete sampling time moments.

The evaluation based on the samplings is not sufficiently accurate for our particular simulation scenario given the sampling frequencies currently supported by the SensLAB platform. The events in the studied protocol have millisecond precision while the smallest possible sampling period for the platform is 70ms. After discussions with the SensLab developers, we agreed they would provide us with a special profile with higher sampling frequency for our particular experimentation scenario. We have made some trials for a sampling period of 2ms but the platform could not keep up with such a high sampling frequency and the resulting measurement files contained data gaps.

The values sampled allow the energy consumption evaluation at the level of the entire WSN430 platform node while the evaluation at the level of individual peripherals is not possible for the moment.

Due to these 2 reasons, we were unable to consider the energy consumption as a metric in our evaluation. We focused however on evaluating the energy consumption optimization impact on the functional aspects of the protocol under real testbed conditions.

4.3.3.2 SensLAB Experimental Set-up

To help the user in developing his/her application, a virtual machine is setup with all the development tools preconfigured (cross compilation chains, OS, drivers,communication libraries).

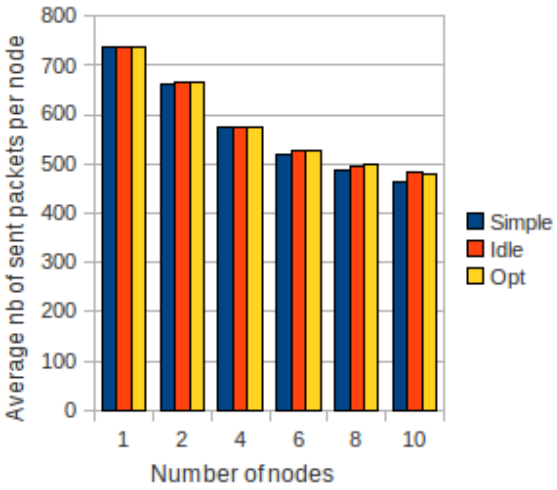


Figure 4.13: Average number of sent packets per node for the 3 driver implementations

In order to quantify the number of packets sent by each node that are successfully received by the sink, each packet contains the node ID of the sender as the first byte. We chose as node ID the CRC field of the DS2411 component of the node, paying attention not to have two nodes in the experimental set with the same CRC field. The receiver node sends on its serial link two characters corresponding to the ID of the sender node for each received packet in hexadecimal format.

A sender node outputs one character on its serial link for each successfully sent packet.

The outputs on the serial link of both the sender nodes and the receivers will be logged in individual log files during the experiment and processed later.

Another issue we had to address was the proper node synchronization, i.e. having all the senders start at the same time instant. To this end, we chose to deploy an extra node that will be the synchronizer. Thus the sender nodes code starts with a wait for a packet from the synchronizer. Once this packet is received, the nodes can pass to the packet sending phase.

With the executables on the senslab server, the experiment launching script creates an *nc* – netcat – process attached to each node retrieving the serial link output of the node. The logging is realized by several *tee* processes, one for each node, retrieving the

output of the corresponding *nc* process. Once these are setup the code is flashed by the script trough *senslab-cli update* calls. The last call is for the synchronizer that starts the experiment. At the end of the experiment duration, the netcat processes are killed and the resulting logs are retrieved and processed.

During our experiments we experienced several software bugs, both in the senslab platform infrastructure and in the MantisOS code. The experimental setup of several nodes saturating the medium with packets transmission is a very high network load for sensor network applications. From our experiments it seems clear that the MantisOS developers did not test their operating systems in these conditions and we experienced several bugs due to race conditions between threads or lack of precision in driver implementations. Many of the bugs we found were related to timing issues concerning the time spent in transitional phases from sleep to wakeup or radio calibration phase. We particularly spent a lot of time and effort on the packet reception part of MantisOS as the original application clearly could not keep up with the packet arrival rate. After three months of debugging and testing the code without success we decided to use the SensLAB pre-defined driver primitives which were known to work. We rebuilt the application on top of these primitives using the code skeleton generated by our method. We encountered problems with this approach as well: blocking reception application under high network traffic, different outcomes according to the choices for API calls made for adapting the code skeleton. Another three months of coding, debugging and testing followed until we fixed the bugs and reached the correct configuration of API calls for the final code.

The SensLAB pre-defined drivers offer primitives for writing the configuration registers and primitives for command strobes. The command strobes can be initiated by calling the *cc1100_strobe_cmd* function having as argument the name of the command strobe register to be accessed. The call is non-blocking, i.e. it returns immediately even if the target state of the CC1100 radio device hasn't been reached yet.

A first aspect to be settled was the clear channel assessment procedure. The RSSI value is an estimate of the signal power level in the chosen channel. For a CC1100 in RX mode, the RSSI value can be read continuously from the RSSI status register until the demodulator detects a sync word (when sync word detection is enabled). At that point the RSSI readout value is frozen until the next time the chip enters the RX state. We made a test application in order to determine the time required for the RSSI

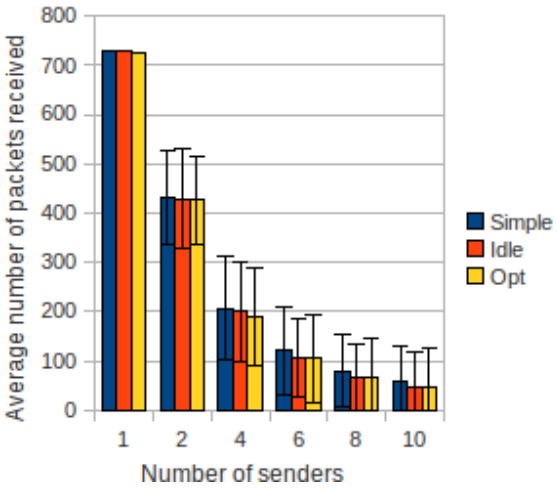


Figure 4.14: Average number of packets received by the sink for the 3 driver implementations

value to settle when the medium is occupied as well as the threshold indicating a busy medium. We determined experimentally that 400 ns are sufficient for the RSSI register to hold an accurate value and that -80dBm is the threshold value indicating a busy medium.

Another issue to be settled was the avoidance of the overflow of the RX FIFO, especially in the case of the simple protocol keeping the radio in RX device state during backoffs. One solution would be to flush the FIFO regularly and sufficiently often through explicit calls of *cc1100_fifo_flush* and it would require the processor to be awake for every such call. Another solution is to define an interrupt such that every time the radio starts receiving a packet and a FIFO threshold is reached, the FIFO is to be flushed. We chose the latter option and we configured an interrupt for the GDO0 CC1100 pin whose interrupt service routine flushes the FIFO. Although the CC1100 datasheet specifies that the FIFO should only be flushed when in IDLE or RXFIFO_OVERFLOW states, our experiments proved that flushing the FIFO when in RX state is safe.

Another issue to be settled was the way of implementing the time delays in the protocol. A first option was to use a simple delay function performing no operations

like in the simulations performed under the Worldsens environment. A second option was to use the MSP430 timers.

The first option proved to be unsuitable in the real-life experiments. Due to the fact that in the case of the simple protocol (keeping the CC1100 in RX during backoffs) the interrupt for flushing the RX FIFO was called much more often and for a number of times impossible to estimate and that the time required for the RX FIFO flushing was added to the delay time, the simple protocol did not behave similarly to the optimized ones.

We therefore chose the second option, i.e. we used the timerB on the MSP430. Thus the flushing of the radio was performed while the timers were counting and the flushing time was not added to the total delays.

Figure 4.13 illustrates the number of packets sent by each node for simulation scenarios using up to 10 senders for each of the 3 driver implementations, i.e. the Avg_{sent} parameter previously defined. Figure 4.14 illustrated the average number of packets (total number of packets received by the sink divided by the number of senders) received by the sink, i.e. Avg_{rec}, as well as the standard deviation S_{rec} computed according to the number of packets successfully received by the sink for each individual sender.

As it can be seen, the three driver implementations send approximately the same number of packets while the sink receives the same average number of packets. Although the evaluation of the energy consumption on real platform was not possible, the experiments still proved that optimizing the protocol from the energy consumption point of view doesn't change its functional parameters.

If the energy evaluation would be possible in the future, it will report the consumption at the level of the entire platform. The stochastic model and the simulation results reflect the energy gain only at the radio level. Evaluating the energy consumption at the level of the entire platform would surely result in lower overall gains as the base power consumption of the platform will remain unchanged.

Conclusions Although the Senslab experiments could not confirm the power consumption gains due to the previously mentioned problems to be fixed on the platform, the experimentations made allowed us to draw several conclusions:

- Experimentations are tedious and error prone

- Testbeds are the only way to go to test software in high load conditions
- Even if MantisOS has been used in many projects it still contained severe bugs when used in corner cases communications
- Our method allows to generate replacement code that does not alter the system behavior (proved in real-testbed experimentation) while optimizing the power consumption (proved by simulation)
- Writing low level code like the one considered in this section should be automated or made using assistants and design tools as the code is very error prone
- Generating code skeletons instead of real code is useful since it offers a base which can later be completed with the API calls which are most suitable for the particular application/protocol that needs to be mapped (e.g. timer calls proved to be more suitable to implement delays in the B-MAC example than busy waiting which was interfering with the radio RXFIFO flush from the CPU point of view).

5

Conclusion and Perspectives

Recent achievements in the domain of micro-electro-mechanical systems (MEMS) and micro-electronic devices have lead to the developpment of a new field of applications for wireless networks: the Wireless Sensor Networks (WSN). The constant search towards the miniaturisation of sensor nodes comes at the price of increased resource constraints: computation, memory and most importantly-energy.

The desired behavior of embedded systems is usually subject to time constraints and it involves interactions with a set of physical devices or components, each with its own individual and specific constraints.

In this thesis, we addressed the problem of mapping a software protocol to a physical device such that software time constraints are guaranteed and the energy consumption is minimized.

We chose a model derived from timed automata for describing the software protocol behavior. We classified the software states as *fixed* – corresponding to a unique state in the device automaton – and *free* – whose mapping is left at the designer's choice. We described the physical device through a finite state machine with annotations.

In the case of a physical device with states of fixed or variable duration but with lower limit constraint, we prove that the problem of mapping a free software state of fixed duration to a path in the physical device such that the energy consumption of the path is minimal is NP complete.

We proposed a generic methodology that allows a mapping of a software protocol to a physical device that guarantees that all the time constraints are met and all feasible transitions in the protocol automaton remain realizable, all while minimizing the energy consumed.

The output of this mapping algorithm is a code skeleton that implements the intended behavior optimized for a given hardware. This code skeleton can be generated to further provide portability among different platforms and generate low level calls to communication APIs and operating system functionality.

We illustrated the energy gains that can be obtained by our approach with the mapping of the well-known B-MAC WSN protocol onto a common radio device namely the CC1100 RF transceiver. We investigated a network scenario where a set of saturated nodes contend for the channel trying to transmit data to one sink.

We developed a stochastic model for modeling the behavior of a node in this network scenario and obtained a theoretical gain of 60%. The code skeleton obtained for the MAC to radio mapping adapted to MantisOS was simulated under the Worldsens environment and the theoretical gains at the radio level were confirmed. Finally, real-testbed experiments on the SensLAB platform illustrated that the optimization in terms of energy consumption did not affect the functional parameters of the protocol.

Perspectives

In this thesis we investigated the problem of mapping one software protocol on one physical device while minimizing the energy consumption. Basically we focused on solving a one-to-one mapping problem with the additional goal of minimizing the energy consumed.

We propose three directions for the extension of our work.

A straight-forward extension is the mapping of a software application that interacts with several devices: a one-to-many mapping. A typical example of such an application is a machine-to-machine bio-logging application, where a sensor node attached to the subject of observation periodically takes samples of a physical characteristic (e.g. cardiac rhythm, temperature) and then sends these values to the network. If the application is simple and the two (or more) devices do not interfere in using other shared resources like buses, there exists an immediate solution: map each automaton independently to the corresponding device. The communication between the two can be achieved through signals. In this particular example, when the sampling automaton has obtained a sampled value, it can output a signal indicating so. The latter would be an input signal for the radio protocol that would trigger its execution.

A second extension would be the mapping of several applications interacting concurrently with the same device: a many-to-one mapping. A typical example would be

the case of two application protocols driving the same physical device. An immediate solution to this problem would be to generate the cartesian product of the two software automata of the interacting applications. A first problem that is raised by this attempt of solution is the increased number of states in the product. Secondly, the generated product might not respect the constraints we imposed to the general model of timed automata in order to allow the mapping. The mapping for this case of combined automata remains an open problem. Another solution would be to use global control as in [6] to ensure the mutual exclusion over the device for the interacting applications.

Finally, achieving local optimum does not necessarily imply global optimum. Thus, fine energy optimization at the level of a particular device might imply frequent transitions in the device finite state machine. These transitions might require the microcontroller to remain in a higher consumption state or at least to be woken up frequently in order to command the transitions in the device state machine. It might be possible to obtain lower energy consumption by keeping the device in a higher consumption state with the advantage of keeping the MCU into a low-power mode. Solving this problem requires to extend the model in order to take into consideration the MCU as well. A possible attempt of a solution is to inject the MCU external constraints into the software protocol models.

Bibliography

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994. 22, 24

[2] A. Bachir, D. Barthel, M. Heusse, and A. Duda. Microframe preamble mac for multihop wireless sensor networks. In *Communications, 2006. ICC '06. IEEE International Conference on*, volume 7, pages 3365 –3370, june 2006. 17

[3] A. Bachir, L. Samper, D. Barthel, M. Heusse, and A. Duda. Link cost and reliability of frame preamble mac protocols. In *Sensor and Ad Hoc Communications and Networks, 2006. SECON '06. 2006 3rd Annual IEEE Communications Society on*, volume 2, pages 632 –638, sept. 2006. 17

[4] A. Barroso, U. Roedig, and C. Sreenan. mu;-mac: an energy-efficient medium access control for wireless sensor networks. In *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, pages 70 – 80, jan.-2 feb. 2005. 18

[5] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, LNCS. 2004. 25, 38

[6] N. Berthier, F. Maraninchi, and L. Mounier. Synchronous programming of device drivers for global resource control in embedded operating systems. *SIGPLAN Not.*, 46:81–90. 21, 97

[7] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms. In *ACM/Kluwer Mobile Networks and Applications (MONET), Special Issue on Wireless Sensor Networks*, page 2005, 2005. 56, 80

[8] G. Bianchi. Performance analysis of the ieee 802.11 distributed coordination function. *IEEE J-SAC*, 18(3):535–547, March 2000. 69

[9] P. Bouyer, E. Brinksma, and K. G. Larsen. Optimal infinite scheduling for multi-priced timed automata. *Form. Methods Syst. Des.*, 32(1):3–23, 2008. 22

[10] C. Burin Des Rosiers, G. Chelius, E. Fleury, A. Fraboulet, A. Gallais, N. Mitton, and T. Noël. SensLAB Very Large Scale Open Wireless Sensor Network Testbed. In *Proc. 7th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCOM)*, Shanghai, China, Apr. 2011. 84

[11] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. 56

[12] A. Chis, E. Fleury, and A. Fraboulet. An optimized mac layer to physical device mapping methodology. In *Mobility '09: Proceedings of the 6th International Conference on Mobile Technology, Application and Systems*, pages 1–8, New York, NY, USA, 2009. ACM. 4, 83

[13] A. Chis, E. Fleury, and A. Fraboulet. Cross-layer optimization for mac layer to physical device communication protocol mapping. In *Proceedings of the 19th International Conference on Real-Time and Network Systems (RTNS'11)*, Proceedings of 19th International Conference on Real-Time and Network Systems (RTNS11), pages 169–178, September 2011. 4

[14] S. Cho, K. Kanuri, J.-W. Cho, J.-Y. Lee, and S.-D. June. Dynamic energy efficient tdma-based mac protocol forwireless sensor networks. In *Autonomic and Autonomous Systems and International Conference on Networking and Services, 2005. ICAS-ICNS 2005. Joint International Conference on*, page 48, oct. 2005. 18

[15] C. L. Conway. Ndl: a domain-specific language for device drivers. In *In Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES*, pages 30–36. ACM Press, 2004. 20

[16] A. Courbot, G. Grimaud, J.-J. Vandewalle, and D. Simplot-Ryl. Application-driven customization of an embedded java virtual machine. In *EUC Workshops*, pages 81–90, 2005. 2

[17] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, LCN '04, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society. 2, 63

[18] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, SenSys '06, pages 29–42, New York, NY, USA, 2006. ACM. 64

[19] C. Enz, A. El-Hoiydi, J.-D. Decotignie, and V. Peiris. Wisenet: an ultralow-power wireless sensor network solution. *Computer*, 37(8):62 – 70, aug. 2004. 16

[20] G. Feltrin, O. Saukh, R. Bischoff, J. Meyer, and M. Motavalli. Structural monitoring with wireless sensor networks: Experiences from field deployments. In *Proceedings of First Middle East Conference on Smart Monitoring, Assessment and Rehabilitation of Civil Structures SMAR*, 2011. 9

[21] E. Fleury and D. Simplot-Ryl. *Réseaux de capteurs : théorie et modélisation*. Architecture, Applications, Service. Hermes, 2009. 7

[22] A. Fraboulet, G. Chelius, and E. Fleury. Worldsens: Development and prototyping tools for application specific wireless sensors networks. In *IPSN'07 (SPOTS)*. ACM, 2007. 77, 79

[23] A. Friggeri, G. Chelius, E. Fleury, A. Fraboulet, F. Mentré, and J. C. Lucet. Reconstructing social interactions using an unreliable wireless sensor network. *Comput. Commun.*, 34:609–618, April 2011. 9

[24] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 1–11, New York, NY, USA, 2003. ACM. 62

[25] J. Hill and D. Culler. Mica: a wireless platform for deeply embedded networks. *Micro, IEEE*, 22(6):12 – 24, nov/dec 2002. 11

[26] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *SIGPLAN Not.*, 35:93–104, November 2000. 2, 62

BIBLIOGRAPHY

[27] D. C. Joseph Polastre, Jason Hill. Versatile low power media access for wireless sensor networks. In *SenSys*, 2004. 15, 27

[28] H. Karl and A. Willig. *Protocols and Architectures for Wireless Sensor Networks*. John Wiley & Sons, 2005. 7

[29] K. G. Larsen, G. Behrmann, E. Brinksma, A. Fehnker, T. Hune, P. Pettersson, and J. Romijn. As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In *CAV*, 2001. 22

[30] K. Lorincz, M. Welsh, O. Marcillo, J. Johnson, M. Ruiz, and J. Lees. Deploying a wireless sensor network on an active volcano. In *IEEE Internet Computing*, pages 18–25, 2006. 8

[31] G. S. Lueker. Two NP-complete problems in nonnegative integer programming. Technical Report 178, CS, Princeton University, 1975. 45

[32] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: an IDL for hardware programming. In *OSDI*. USENIX, 2000. 20

[33] R. S. Oliver, I. Shcherbakov, and G. Fohler. An operating system abstraction layer for portable applications in wireless sensor networks. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 742–748, New York, NY, USA, 2010. ACM. 2

[34] M. O'Nils and A. Jantsch. Operating system sensitive device driver synthesis from implementation independent protocol specification. In *DATE*. ACM, 1999. 20

[35] J. Polastre, R. Szewczyk, and D. Culler. Telos: enabling ultra-low power wireless research. In *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, pages 364 – 369, april 2005. 11

[36] I. Rhee, A. Warrier, M. Aia, J. Min, and M. Sichitiu. Z-mac: A hybrid mac for wireless sensor networks. *Networking, IEEE/ACM Transactions on*, 16(3):511 –524, june 2008. 19

[37] Richard Barry. The FreeRTOS Project. online http://www.freertos.org/, 2011. 2, 58

[38] R. Serna Oliver and G. Fohler. Timeliness in wireless sensor networks: Common misconceptions. In *Proceedings of the 9th International Workshop on Real-Time Networks RTN'2010*, Brussels, Belgium, July 2010. 2

[39] G. Simon, M. Marti, kos Ldeczi, G. Balogh, B. Kusy, A. Ndas, G. Pap, J. Sallai, and K. Frampton. Sensor network-based countersniper system. pages 1–12. ACM Press, 2004. 8

[40] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler. An analysis of a large scale habitat monitoring application. In *In Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys*, pages 214–226, 2004. 8

[41] I. Talzi, A. Hasler, S. Gruber, and C. Tschudin. Permasense: investigating permafrost with a wsn in the swiss alps. In *Proceedings of the 4th workshop on Embedded networked sensors*, EmNets '07, pages 8–12, New York, NY, USA, 2007. ACM. 8

[42] Texas Instruments. CC1100 single chip low cost low power rf-transceiver, 2006. 29

[43] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong. A macroscope in the redwoods. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, SenSys '05, pages 51–63, New York, NY, USA, 2005. ACM. 8

[44] S. L. Torre, S. Mukhopadhyay, and A. Murano. Optimal-reachability and control for acyclic weighted timed automata. In *TCS*. IFIP, 2002. 22

[45] T. van Dam and K. Langendoen. An adaptive energy-efficient mac protocol for wireless sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, SenSys '03, pages 171–180, New York, NY, USA, 2003. ACM. 15

[46] D. Wagner and R. Wattenhofer. *Algorithms for Sensor and Ad Hoc Networks: Advanced Lectures (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. 7

[47] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient mac protocol for wireless sensor networks. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1567 – 1576 vol.3, 2002. 14

[48] J. Yick, B. Mukherjee, and D. Ghosal. Wireless sensor network survey. *Comput. Netw.*, 52:2292–2330, August 2008. 7