Abstract

With the development of the MEMS technology and of wireless networking, the sensor network area has been an active field of research in the last five years. Sensor network propose a new approach to monitoring applications and provide high resolution observations of the environment. The main constraint of the sensor networks is energy as the nodes are powered with non renewable batteries, and numerous techniques and design strategies have been studied to optimize power consumption for these networks. The design space for hardware and software is very large, and the consequences of design decisions may be difficult to foresee in such large and dynamic systems. In order to help the sensor network designer in his task, a simulation framework was developed. It defines a structure representing sensor nodes and their interconnection within the network. The framework models the major aspects of sensor networks such as battery lifetime, node mobility, radio communications and it also represents the behavior of the environment. The implementation of the framework focuses on modularity, allowing to experiment the consequences of using various techniques and architectures. Examples inspired on common sensor network architectures are presented, and the ability of the framework to cover the design space are discussed. Scalability being a main issue on sensor networks, the performance of the simulation is measured.

Preface

The work presented in this Masters thesis has been carried out by Knud Martin Hansen

Date and My signature

Acknowledgements

First of all, I would like to thank my supervisor, Jan Madsen. All along the project, he has supported my work actively and provided constructive guidance. Kashif Virk has also been a great help, and was always available for interesting discussions concerning the project and many other subjects. I would also like to thank Nokia, and more specifically Dan Rebild, for accepting to support my Master by giving me a scholarship that made it possible for me to follow the M.Sc. program at DTU.
I would also like to thank my family for their assistance and help: my parents for their support and their faith in me, my brother and my sister, my grand mother for her steady confidence in me, as well as Inger, Jan, Anna Thora, Johanne, Holger, Katrine, Toni, Sven, Anna, Anja og Troels who have given me the motivation to complete the project. I also thank my friends at DTU who made my stay there a very enjoyable experience.

Contents

Chapter 1

Introduction.

1.1 Introduction

The sensor networks are a new class of wireless networks. They consist of a number of nodes, called sensor nodes. These nodes are loosely coupled in the network and cooperate to provide a monitoring service to users that can access the network through gateways and base-stations. Compared to traditional sensor systems, sensor networks offer a much higher quality of service and provide much more information about the phenomena observed. Traditional sensor systems are wired, and are tightly coupled to a computer that gathers and processes the data received. This limits the number of sensors that can be supported by such a system, and the wires make it difficult to deploy the sensors in the monitored area. Whereas, the sensor nodes are wireless, small and inexpensive devices. It is therefore possible to deploy a large number of nodes in the monitored area. Each node senses the phenomena of interest thus providing a high-resolution picture of the environment. Another key aspect in environmental monitoring is whether it is invasive or not. If the monitoring system affects the environment, the data measured loose their value. The small form-factor of the sensor nodes and their ability to function without human intervention are therefore important factors of the quality of the information collected.
Sensor networks are used in different contexts and may therefore have different characteristics. Some typical applications are habitat and environmental monitoring for surveying animal behaviour and life-cycles. On the Great Duck Island, for example, a sensor network has been deployed to monitor birds called Leach's Storm Petrel [14]. The sensor network is monitoring the meteorological conditions around the nests of the Petrels and the occupancy of the nets. The data is transmitted to a base-station through a gateway. The base-station is connected to the Internet from where any researcher on the world can access the data. Another similar example is the ZebraNet project [10] aimed at monitoring the behavior of Zebras by attaching sensor nodes to them. The ZebraNet node is equipped with a GPS to analyze the movement patterns of the Zebras. Sensor networks are also candidates for tactical applications (for example deploying a network in a enemy area to monitor its activity) or for public applications (for example traffic monitoring on sensible road segments).
The research in the field of sensor networks has increased tremendously during the last

ten years because of the advances in MEMS technologies and in the domain of low-power low-cost short-range radio transceivers. This research aims toward a reduction of the node size (smart dust) and the node cost, with the aim to be able to deploy large number of nodes in an ad-hoc fashion, in order to monitor the environment. Power is the main design problem for these nodes, because the nodes are generally powered with batteries and therefore have limited energy resources. In most sensor networks, the batteries are not re-chargeable, and replacing the batteries is too expensive if at all possible. To maximize the lifetime of the sensor network, it is thus important to manage the power consumption of the nodes very carefully.
As wireless sensor networks is a rather recent research area, the design techniques for sensor nodes are not fixed yet and the design space is very large. Possible designs include solutions from the totally application specific hardware design to general purpose hardware platforms managed by an operating system and running an application. While the first allows very efficient solutions (in terms of power and performance), it is a very expensive solution as the platform can only be used for a given application. In contrast, the second approach allows different applications to use identical platforms often composed of cheap on-the-shelf components and is therefore preferred. This node architecture allows for optimizations at three levels: the hardware platform, the operating system and the application. At the hardware level, the design decisions include the selection of the sensing, controlling and communicating components as well as their interconnection. The selection of operating system properties such as its scheduling algorithm and its power management policy are also important steps of the design. Additionally, the communication protocol is also categorized as part of the operating system of the sensor nodes because communication is one of their fundamental abilities. Decisions on these protocols may also influence greatly the performance and the power consumption of the sensor network. Finally, the application has to be defined. Also at this level decision can be made on the desired functionality of the network and the quality of service that is required compared with the cost of providing it.
In contrast to the traditional systems, the value of the service provided by a sensor network is larger than the sum of the services provided by the individual nodes. Therefore, the focus of the design of a sensor node should be set on the consequences of the design decisions on the sensor network as a whole rather than on the individual sensor node. Thus, the main aim in the design of sensor nodes is to maximize the lifetime of the network and this may not be achieved by maximizing the average lifetime of the individual nodes.
In order to cope with the complexity of the design task, it is critical to provide tools for the designer to evaluate the consequences of the design decisions. One of the tool that has been most used in computer science for these purposes is simulation. While it does not allow the designer to verify the correct behaviour of a system, it allows the designer to get an understanding of the functionality of it and to compare the behaviour of alternate implementations.
A main issue in using simulation to design a system is to select the level of abstraction at which to simulate. Low level simulation has the advantage that it is highly accurate in its representation of the system. However, the lower the simulation abstraction level is, the lower is the simulation performance. The simulation runtime is particularly critical, because designing the system requires simulating a number of alternatives and

comparing them against each other. This is only possible if the simulation runtime is reasonably low. Furthermore, low level simulation requires that the details of the system are known, which is generally not the case in a top-down design process. However, the higher the level of abstraction of the simulator is, the lower its accuracy. It is therefore important to select the right level for the design decision that is considered, trading off accuracy for performance and simplicity.
In system design it is critical to capture the consequences of the high-level design decisions as soon as possible. High level decision consequences propagates in the lower level steps of the design and changing the decision at this level is more expensive and error prone.

1.2 Related Work

There exist numerous network simulators and they can be classified in three groups: network-level simulators represent the nodes as packet generator and do not model the details of the activity of the node, node-level simulators in contrast model the node in more detail and represent the effects of node characteristics on the network, finally application development simulators are based on a defined platform and simulate the execution of actual programs on this platform, they provide a development environment to sensor network programmers allowing them to test and debug the application.
Most of the traditional network simulators are at the network model. The nodes are modeled by the communication protocol they use, but the actual performance of the node and its power-consumption are not the main focus of these simulators. Examples of such tools are OPNET Modeler [15], NS-2 [20], GloMoSim [23]. OPNET Modeler is a commercial C++-based simulation environment that represents the network in three hierarchical levels: the network level defines the interconnection of the nodes of the network, the node level represents the behavior of the node as a set of processes, and the process model defines the processes as state machine. Traditionnaly used for wired networks, OPNET Modeler was extended with a wireless module providing support for modeling mobile wireless networks. NS-2 is an open source simulator based on the REAL simulator and is describing the network using a combination of C++ and on OTcl. These simulators are used to design communication protocols. However, they do not easily allow to evaluate the consequences of node-level design decisions. In [3], the three above-mentioned simulators are compared, and large variations were found in the simulation results and in the conclusions that each simulator would lead to. In order to accurately simulate networks, the model must represent the behavior of the node closely.

With the development of sensor networks node-level simulators for sensor network were developed. In contrast to network-level simulators, they attempt to represent the behavior of the nodes more closely. SensorSim [16] is a sensor network simulator developed at UCLA based on ns-2. It divides the node model in two: the functional and the power model. The functional model represents the software of the node and the application. The power model represents the power consumption of the different hardware components: radio transceiver, processor, sensors. The SensorSim simulator is hard coded to

represent the WINS platform using predefined protocols and is therefore not suited for design space exploration.
TOSSIM [13]is a sensor network simulator used for development of application. It simulates the communication quite accurately with a bit level propagation model with a bit-error representation . It also represent the exact functionality of the nodes as it executes the code of the developed application. However, this simulator does not represent execution times. Moreover, the model of the environment (phenomenon monitored by the sensor) is simplistic: it gives random values to the sensors. A new version of TOSSIM emphasized on power consumption representation PowerTOSSIM is currently being developed. Even though TOSSIM is not very scalable, it provides good facilities for testing applications for networks of MICA nodes running the Tiny OS operating System. However, it is not suited for design space exploration as it does not easily allow to experiment with different platforms or operating systems.

1.3 Node-level Simulation for Design Space Exploration

This thesis proposes a simulation framework for wireless sensor networks using the SystemC simulation engine. It models sensor networks in a modular way reprensenting the application, operating system and hardware of the nodes as well as the networking activities and the environment behavior. The framework is an extension of an earlier System-on-Chip simulation. It defines a model structure of sensor networks putting emphasis on modularity. The purpose of this framework is to gather in a homogeneous model the main issues of sensor network design: communication mechanism and communication protocols, network dynamicity (scalability and mobility), energy management and battery lifetime. The framework consists of a structure for representing sensor networks. The implementations of the actual blocks of this structure depend on the architecture of the node, its operating system and its application. Thus, different design options can be simulated and evaluated by interchanging these implementations, thus allowing design space exploration. This project concentrated on providing the structure and validating it.
The project is connected to the Hogthrob project [7] conducted in cooperation between KVL, DIKU and DTU. This project consist in developing a platform for monitoring of sows. The final goal is to develop nodes to replace the existing RFID tags and to provide a improved monitoring of the sows to detect key aspect of their life cycle, for example, the period when the sows can be bred. The changes in the behavior of the sows in this period (movement, screaming) can be detected automatically by the sensor network rather than by the farmer as it is now. Within this project, a prototype node has been developed that was aimed at testing different implementation. To provide the possibility of evaluating different hardware architectures, an FPGA is included on the node.

The rest of the report is organized in the following way: chapter 2 presents the sensor network niche. It describes the design space and presents approaches proposed in the sensor network research for optimizing the lifetime of the network. The framework presented here is based on a former framework for modeling systems-on-chip implemented on SystemC. The concepts of the framework are presented in chapter 3, and the reasons

why to use it as a starting point are discussed. In chapter 4, the structure of the sensor network model is presented, and the implementation is presented in chapter 5. In this chapter, how the structure of the framework is traduced in the implementation is presented, and some examples of implementations of the components of this structure are presented. In chapterchap:result, the performance of the framework is measured, and different optimization are considered and evaluated. Finally, chapter 7 sums up the work realised in the project and presents potential future developments of the simulation framework.

Chapter 2

Design Space of Sensor Networks.

2.1 Sensor Network Characteristics

Sensor networks may first look similar to the other Mobile Ad-hoc Networks such as the wireless LAN. Indeed while the nodes of some sensor networks are placed at well-defined positions (as is the case for the habitat monitoring application of the Great Duck Island as well as for the traffic monitoring application) most sensor networks are actually ad-hoc networks. One reason for this is that the typical number of node of a sensor network is very high (thousands of nodes is not uncommon): placing all these nodes individually is not a possible solution. Alternatively, the nodes can be deployed randomly (for example dropped by a plane) with the purpose of achieving a uniform coverage of the area to monitor.
For what concerns mobility, whether the nodes are fixed or moving depends on the application supported by the sensor network. For example, the ZebraNet network is obviously mobile, while the monitoring of the nests on the Great Duck Island and the sensor network used to survey the enemy's activity are not. However, even for sensor networks with immobile nodes, the network has to support some degree of dynamicity due to the scalability of he network: the number of nodes in the network changes over time. Nodes progressively disappear from the networks when they run out of battery. Alternatively, sensor nodes can also be added to the network: for example when the density of nodes reaches a critically low level, the users of the network may decide to deploy new nodes in order to maintain the quality of service of the network and to prolong its lifetime. The Ad-hoc character of sensor networks and their dynamicity require that they are able to configure and maintain the network dynamically and autonomously.
Mobile Ad-hoc Networks (MANETs) are not a new area: for example the wireless LAN networks have become common and standards have been established to manage them (e.g. IEEE 802.15). Similar techniques can be used for sensor networks. However, the sensor network domain differs from these traditional networks in the fact that the resources available at the network nodes are very limited: processing, memory, and principally energy resources are main bottlenecks in the development of sensor network applications. Energy is especially precious because for most sensor networks the energy supply is finite and is not re chargeable nor replaceable. Additionally, the lifetime requirements for sensor networks are in the order of months or even of years. For example,

the requirement for the Great Duck Island project is that the node can operate totally autonomously for 9 to 12 months at a time.
These severe requirements in terms of power consumption together with the fact that sensor networks must support a high level of dynamicity make it necessary to manage the network very carefully and to use energy aware design techniques.

2.2 Sensor Node Architecture

A sensor network is a distributed system composed of a large number of sensor nodes that work cooperatively with the purpose of monitoring the physical world as closely as possible. This large number of nodes and the fact that it must be possible to deploy sensor networks in various environments requires that the nodes be wireless and battery powered. The sensor nodes have three main tasks: sensing the environment, communicating with the other nodes of the network and processing sensed or received data.
A sensor node can thus be divided into four subsystems as illustrated in figure 2.1:

- The sensing subsystem is responsible for monitoring physical phenomena of the environment. It is composed of a set of sensors, and may also include analog to digital conversion and digital signal processing facilities.
- The communicating subsystem is responsible for insuring connectivity of the sensor network. The communication ability of sensor nodes has two main purposes: transmitting some information about the information to the rest of the network and routing communication from other nodes to the rest of the network. The communication subsystem is composed of one (or more) transceiver and may also have some dedicated processing facility.
- The processing subsystem is the central part of the sensor node. It controls the communication and sensing subsystems. It can also be used to perform processing on data received either from the sensors or the transceivers. This subsystem is further divided in a three components: the hardware composed of a controller unit which can be composed of one (or more) general purpose processor and may also include some dedicated hardware (accelerators) to assist the processor, the operating system which manages the operation of the controller and provides services to control and communicate with the sensing, communication and battery subsystems, and the user application which defines the task of the node.
- The battery subsystem is responsible for supplying power to the three other subsystems. It not only represents the physical battery, but also DC/DC converters that allow to control the supply voltage of the different components of the node.

2.3 Sensor Node Design Space

Because of the multiple and tight constraints that sensor networks are subject to, the design of the sensor node must be done very carefully. Designing the different components of the sensor node (both hardware components and software components) individually limits the exploration of the design space and may not lead to an optimal solution. Instead, a co-design approach is likely to give better results. For example, choosing a

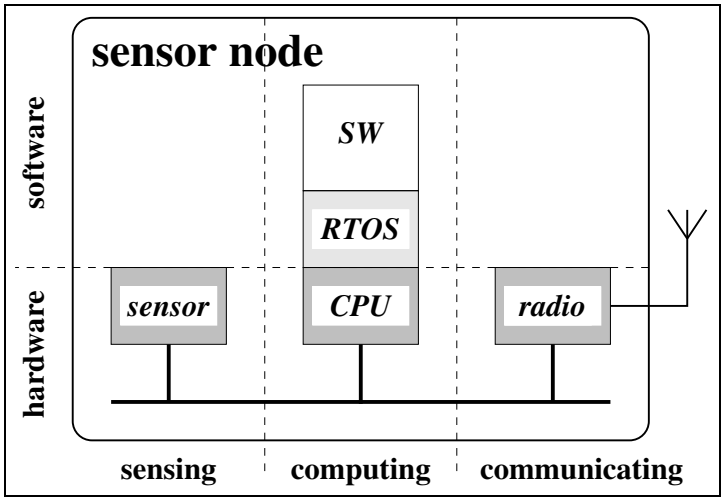

Figure 2.1: Structure of Sensor Nodes

radio transceiver and a communication protocol separately may lead to waste of energy. The co-design is particularly important for sensor networks, because it concerns all the components of the node: the hardware components, the operating system, but even the application are to be designed in parallel to optimize the lifetime of the sensor network. This is indeed a great difference between sensor networks and traditional MANETs: in traditional MANETs, the network provide sets of services that can be used by various applications. In contrast, in sensor networks, the application supported by the network is typically fixed at the time of the design of the network. It is therefore possible to tune the operating system, the communication protocols and the hardware platform for the given application.
In this section, the issues that are encountered in the process of designing a sensor network are presented. Also a number of techniques proposed by the research in the domain of sensor networks are described. The purpose of this section is to give an overview of the techniques used in sensor network design. This is indeed important because the aim of the simulation framework is to make it possible to model and simulate such techniques.

2.3.1 Hardware Architecture of Sensor Nodes

The most common approach to design sensor nodes is to combine off-the-shelf components to provide the sensing, processing and controlling, and communication functionalities. The Mica node [5] is built in this way. It was originally developed at UCLA Berkeley by Jason Hill, but since the original Mica node, a number of nodes derived from it were developed. The original Mica node was built around an AVR ATmega103 microcontroller. This processor controls the battery, communication and sensing subsystems of the node. The TR1000 [18] simply converts one of the output signals from the processor to a radio signal, and the node offers an interface to connect some sensors.
In nodes like the Mica node, all the processing is done by the microcontroller. Even the handling of the communication is entirely done by it: the TR1000 is a bit transceiver that

converts its input directly to a radio signal at rates up to 115 kbps. The construction and the encoding of the packets, the control of the access to the medium, and the detection of incoming packets are entirely handled by the microcontroller. entirely managed by the processor: encoding of the data, construction of the packets, medium access control. Alternatively, packet level transceiver can be used: these transceiver have processing abilities and handle the communication according to a protocol to send the packets given to them from the microcontroller and to detect and decode messages, presenting the data packets to the microcontroller. In the hogthrob platform, the transceiver used is the nRF2401 from Nordic VLSI. It is a packet-level transceiver, that is given packets serially and that transmits them in short bursts at a high bit rate (1 Mbit per second): Nordic VLSI calls this mechanism the ShockBurst. In order to select a radio, it is not sufficient to compare their power consumption. The transceiver also has an effect on the microcontroller workload (TR1000 requires more processing from the CPU than nRF2401). The mechanism of sending packets in short burst may also reduce the probability of collision and thereby reduce the number of retransmission of a packet.
The example above can be generalized: an alternative to having a single general purpose microcontroller doing all the processing (as in the Mica node) is to add some accelerators that are customized to perform a specific operation efficiently. The nRF2401 radio is not only a transceiver, but it is also an accelerator controlling radio transmission. Intuitively, it seems that adding components increases the power consumption of the node. However, if the power management is done carefully, it may be that the lifetime of the node can be increased. The effect of changing the architecture of the node by adding accelerators on the lifetime depends strongly on the power management policy, but also on the application and the workload profiles that it generates. The hogthrob platform was designed with an FPGA in order to make it possible to experiment the effect of adding accelerators on the behavior of the node.

other approaches to the design of sensor nodes focus on developing more customized and efficient platforms. Ultimative example of such approaches are illustrated by the WINS [1] and SmartDust [11] projects, in which nodes that are totally integrated on silicon dies are developed. For these projects, the functionality of the nodes is directly implemented in hardware.

2.3.2 Operating Systems

It is a general trend in embedded system to use operating system to manage the execution of the application. In the field of sensor networks, a number of approaches to providing intermediate layers between the hardware and the application also exist. In some projects (Rockwell WINS, Sensoria WINS), existing embedded systems such as μC/OSII or linux are used. The platforms supporting these projects are large and allow to have these large operating systems. However, in most cases, the memory of the node is very limited (4 kBytes of RAM for Mica nodes), and these systems are not a option. Instead, light weight operating systems were developed. At UCLA Berkeley, Tiny OS [6] was developed. Tiny OS is an event-based operating system with two level of scheduling: tasks that run to completion and are executed in FIFO sequence, and hardware interrupts that preempt the currently running task or the handled interrupt.

Due to this scheduling policy, Tiny OS only requires one stack but can still handle the concurrency inherent to sensor networks. In addition to the scheduler, Tiny OS defines a component based representation of the application where the components are stacked and communicate with each other through events and commands. In contrast to traditional operating system, the scheduler is compiled together with the application code before it is downloaded on the node. TinyOS suits well to sensor networks because of its small footprint.
Depending on the application and on the platform used, decisions on features of the operating system must be made. Tiny OS is widely used in the sensor network area. However, for application requiring the satisfaction of real-time constraints, no support (priority based scheduling) is given by Tiny OS. Furthermore, techniques as the Dynamic Voltage Scaling scheduling techniques could be candidates to reduce the power consumption of the processing unit.

2.3.3 Communication Protocols

Communication protocols in sensor networks are generally only composed of the data link and the routing layers. As communication is the main power consumption source in sensor networks, research has proposed numerous protocols to reduce power consumption in this area.
At the MAC layer, two major protocols can be distinguished: Carrier Sense Multiple Access (CSMA) and Time Division Multiple Access (TDMA). The advantage of CSMA is that nodes do not have to wait when they want to transmit unless the channel is busy. However, carrier sensing does not completely avoid collisions. TDMA in contrast guarantees that only one node can transmit data in one cycle and avoid collisions. Furthermore, with TDMA, it is possible for odes to know when their neighbour may send packets, and in the rest of the frame (period at which the slot pattern repeats), they can sleep. The drawbacks of TDMA is that the node must be synchronized. Furthermore, slot allocation is a complicated task and does not adapt easily to scalability of the network (nodes appearing and dying). An example of MAC protocol that specifically addresses sensor networks is the S-MAC protocol [22]. It combines the advantages of TDMA by only having nodes to be listening periodically while not requiring centralized slot allocation. Instead, each node acquires knowledge of the listening periods of its neighbors and can thereby communicate with them using an RTS/CTS mechanism.
The routing is also a main concern in sensor networks. The radio of sensor networks are generally short range radios, and do not cover the integrity of the deployment area. Therefore, multi-hop protocols are used, where the sensor nodes between the source and the destination act as routers. The simplest form of routing is flooding. It is widely used in sensor networks for messages that need to propagate to the whole network. Other protocols offer the service of sending messages to defined nodes. Ad-hoc on-demand routing consists in constructing the route when it is needed. In contrast, other techniques maintain information about the neighboring nodes to decide what the next hop will be. GPRS [12] is an example of such where each node maintains the position of its neighbors and forward the messages to the neighbor which is closest to the destination. Another approach was proposed for sensor network with directed diffusion []. This protocol is data-centric: the nodes that are interested in receiving a given type of messages

subscribe for these packets. When packets of this type are generated, they are passed to the node which subscribed following paths set up a subscription.
There are numerous approaches to communication protocols, and especially for routing, which protocol is best suited strongly depends on the application.

2.3.4 Power Management

Power management is a major topic of sensor networks, and much research is done in this field. The power management techniques can be divided into three classes: node-level power management and network-level power management. Node-level power management are techniques that consist of shutting down the unused components of the platform (e.g. sensors, transceiver). The components used on the sensor network platforms generally have a number of power states. They constitute a number of trade-offs between the power consumed and the responsivity of the component when it is in this sleep mode. In [19], policies to manage such components are presented. The decision of changing the state is based on some knowledge about the inter-event period and the latencies between the different power states. Prediction techniques allow the sensor node to learn the behavior of the environment and thereby decide whether to allow components to switch down.
In network-level power management techniques, the nodes cooperate to optimize the lifetime of the network. For example, some techniques attempt to even the power consumption among the nodes of the network to avoid having the key nodes die very fast and compromize the ability of the network to deliver the service it was designed for. Another approach was proposed in [21], a coverage maintenance protocol is presented. The principle of the protocol is to allow nodes to sleep as long as the application constraints are satisfied. For sensor networks, the application constraints are the sensing constraints and communication constraints. These constraints define a minimum density of awake nodes. Thus, the nodes of the sensor network can agree on which nodes have to stay awake to maintain the constraints, and the other nodes can go to sleep. This type of protocols is particularly useful in dense network where they reduce the redundancy. The constraints may vary depending on the quality of service that the application has to provide.

The design space of sensor networks is very large, and in contrast to traditional MANET which provide a set of services for supporting various applications, the application that a sensor network has to run is known at design time, and allows to make application-specific optimizations. To explore the design space and enable such optimization, the simulation framework enables to simulate and compare the different options.

Chapter 3

The SoC/NoC Model.

The simulation framework for sensor networks presented in this report is based on a previous simulation framework developed at the Institute of Informatics and Mathematical Modeling (IMM) at DTU [8] This framework was developed with the purpose of providing a high level simulation tool for Systems on Chip (SoC) and Network on Chip (NoC) architectures. The SystemC simulation platform was chosen to implement this framework because of its modularity and expressivity. In this chapter, we present SystemC and the SoC/NoC simulation framework.

3.1 SystemC: a Modular and Expressive Simulation Library

SystemC is a C++ library implementing a simulation engine. The simulation handles systems that are described with the modularity of the hardware description languages while keeping all the expressivity of C++. The fact that the simulator is based on C++ allows to define the behavior of the modules at a very high level of abstraction. Furthermore, All the C++ constructs and object libraries (e.g. STL templates, open source libraries, custom libraries previously implemented) are supported; This greatly reduces the time needed to implement the modules of the simulation and makes this process less error prone.
The basic component in SystemC is the *SC_MODULE* (simply named module in the rest of the report). The module has an interface that is composed of the ports that can be used to connect it to other modules, and a behavioral description that consists of a number of SystemC methods (*SC_METHODS*). At declaration time, the methods are affected a C++ function and a sensitivity list. When the value of a signal from this list changes, the function gets executed. This signal activation mechanism is used in the framework only for the synchronization of the modules. For inter-module communication, the framework uses another mechanism offered by the Master-Slave SystemC add-on library.

3.2 The SystemC Master-Slave Communication Mechanism

The master-slave communication mechanism is similar to Remote Procedure Call: a module with a slave port and a slave function attached to this port runs the slave function whenever another module with a master port connected to the slave port writes

the master port. This remote procedure call is blocking the master until the slave returns. The link connecting the master port to the slave port not only triggers the slave function, it can also be carrying some data. This can be used to parameterize the call of the slave function. This mechanism is a simple and efficient way to pass messages between modules for behavioral simulation.

3.3 The SoC/NoC Model

The SoC/NoC model [8] is a model for representing real-time application running on a multiprocessor system-on-chip. Each processor of the system is operated by its Real Time Operating System (RTOS). The application running on top of it is represented at an abstract level by tasks which are mapped to the processors of the SoC platform used. In this section, the task and the RTOS models introduced by the SoC/NoC framework are presented.

3.3.1 The Task Model

In the SoC/NoC model, tasks represent the execution of parts of the application. Tasks of the SoC model are periodic tasks. Tasks are represented as SystemC modules and are specified by parameters defining their mapping, their time constraints and their resource constraints. These parameters are:

- the execution time of the task is given as a worst case and a best case execution time. The actual execution time is determined randomly at simulation time,
- the real-time characteristics of the task specify the period, the offset and the deadline of the task,
- the priority of the task,
- the resource constraints define the resources required by the task and the time of the execution at which they are required, and
- the assignment of the task to a processing element.

3.3.2 The RTOS Model

The operating system models are controlling the execution of the tasks that they are assigned. The services of the operating system are divided into three activities each represented by a module:

- the synchronizer is responsible for the sequencing of the task according to the dependencies described by a global static task graph. This task graph represents both dependences internal to processing elements and between tasks mapped to different processing elements. The synchronization is done at the system level and there is therefore a single synchronizer for the system.
- the allocator is responsible for allocating the resources needed by the tasks.
- the scheduler is responsible for allocating CPU time to the tasks. As illustrated by figure 3.2, these three components are layered: the synchronizer receives requests from the tasks. Requests that have been resolved are passed to the allocator, and similarly, the requests resolved by the allocator are passed to the scheduler. The scheduler then sends commands back to the tasks

3.3.3 Communication between the tasks and the operating system

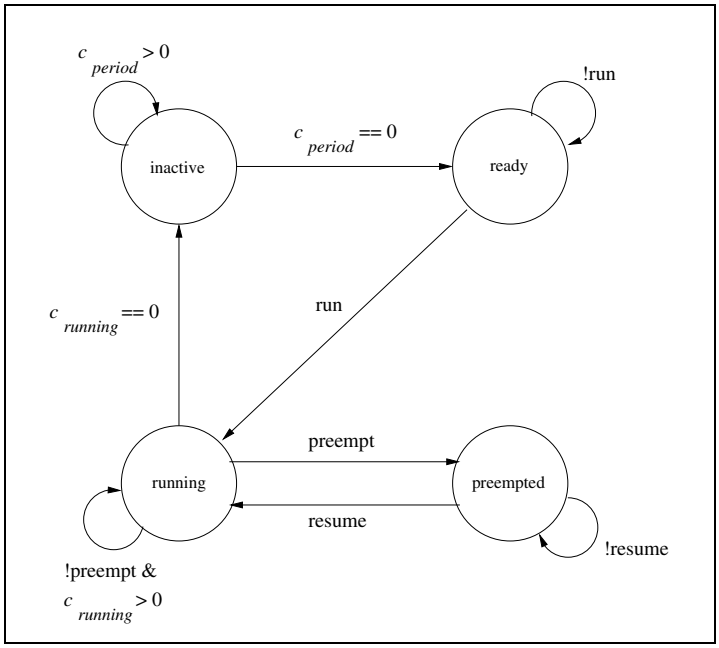

Figure 3.1: Task State Machine

The execution of the application is simulated using a message passing mechanism between the tasks and the RTOS model. These messages are exchanged on the master-slave links of SystemC. Three different requests can be sent from the task to the RTOS. The *ready* request informs the scheduler that the task has been released and requests for execution. The *finished* message informs the scheduler that the task has completed execution. The *resource request* message requests a resource from the RTOS.
The RTOS controls the tasks using three commands. The *run* command informs a task that it is allocated the cpu and can start execution. The *preempt* command is used for preemption of the task, and the *resume* command informs the task that it returns execution after it has been preempted. In the model, the state of the execution of the application is defined as the collection of the state of the tasks. Each task module maintain its state according to the state machine of figure 3.1). The state machine is controlled by the cooperation of the task module and the RTOS modules. Transitions from idle to ready and from running to idle are controlled by the task according to its execution time characteristics, whereas, the transition from the ready to the running state and between the running and preempted states are controlled by the commands received by the task from the RTOS model.

3.3.4 The NoC Model

Communication between processing elements of a system-on-chip may have a large influence on the execution of the application. To model the communication overhead, a model for the network-on-chip (NoC) was proposed [9]. This model represents the NoC as a processing element. The communication of information is represented as a task,

and the network arbitration of the NoC as an NoC-allocator and a NoC-scheduler. The model obtained for the SoC/NoC is presented in figure 3.2

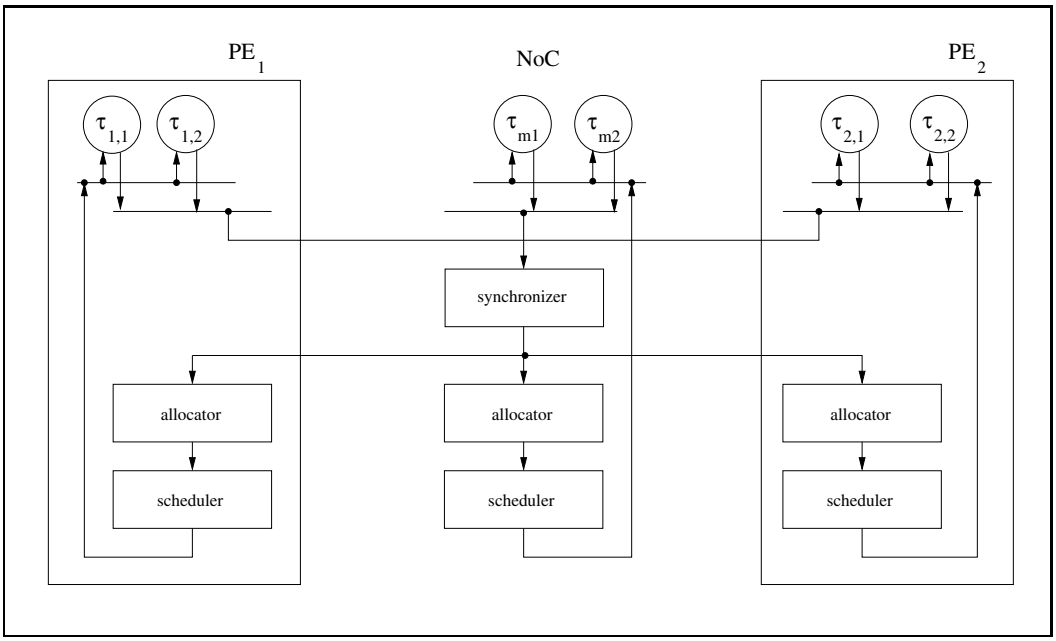

Figure 3.2: Model of a Network-on-Chip

3.4 Motivation for Sensor Network Modeling

The goal of the SoC/NoC model is to model the execution of real-time applications on multiprocessor platforms at the system level. As the model is simple, it is fast to simulate and it allows to explore different design options such as how to map the tasks to the different processors, which scheduling algorithm or which network topology and arbitration policy to use. Not only the goal of the SoC/NoC framework is similar to the goal of the sensor network framework, but also, the structures of NoC systems and sensor networks are similar. In both cases, the system is composed of a number of processor elements (processors of the SoC and nodes of the sensor network) that are interconnected to some network. It is therefore natural to extend the SoC/NoC model to sensor networks. The remaining of this report presents how I have extended this framework to support sensor network modeling and simulation.

Chapter 4

The Wireless Sensor Network Model.

4.1 The SoC/NoC Framework and Sensor Networks Requirements

As Mentioned previously, there are important similarities between wireless networks and SoC/NoC designs. However, sensor networks modeling requires specific characteristics that are not present in the SoC/NoC framework:

- Energy consumption is a main concern.
- Sensor networks are dynamic and reactive systems.
- Sensor nodes take part in network management.
- Sensor networks are composed of a large number of nodes.
- Input/Output tasks represent an important part of sensor network applications.

These characteristics are explained below, and the modification that they require on the simulation framework are presented.

4.1.1 Energy Consumption

The aim of the SoC/NoC framework is to simulate the performance of multi-processor systems and to verify their compliance to some real-time constraints. The energy consumption is not addressed by the framework. In sensor networks, the lifetime of the sensor nodes is more important than their performance. The ability to estimate energy consumption as well as the battery lifetime must be supported by the sensor network framework.

4.1.2 Reactive Applications

While Systems on Chip are static (the number of components is constant and the way they are interconnected remains fixed), the behavior of sensor networks is highly dynamic.
The first cause of this dynamicity is the fact that the applications of sensor networks are strongly connected to a dynamic environment which they have the task to monitor. Therefore, rather than being defined as periodic processing tasks, the applications of

sensor networks are in most cases reactive and are defined by the actions that should be taken in response of environment events.
The mobility of the nodes and the scalability of the network are other sources of the dynamicity of the network. In contrast to SoC/NoC systems where the routing of messages from a source processor to a destination processor can be done statically, most sensor networks use multi-hop routing. For such routing protocols, the changes in the topology of the network (movement of the nodes, addition/removal of nodes) must be handled by dynamically updating the routing paths. It is therefore impossible to represent the communication as static task graphs before the start of the simulation. This is illustrated in figure 4.1. Figure 4.1.a presents a task graph for sending a message from one component to another in SoC's. As the NoC is static, it is possible to characterize the message task with a behavior (occupation of NoC links, communication latency, etc.) that models the transmission of the message quite accurately. Figure 4.1.b illustrates that the multi-hop communication cannot be represented similarly unless the positions of the network topology is constant and known by the user before deployment. Both the dynamic and the ad-hoc nature of the sensor networks prohibits such a representation.

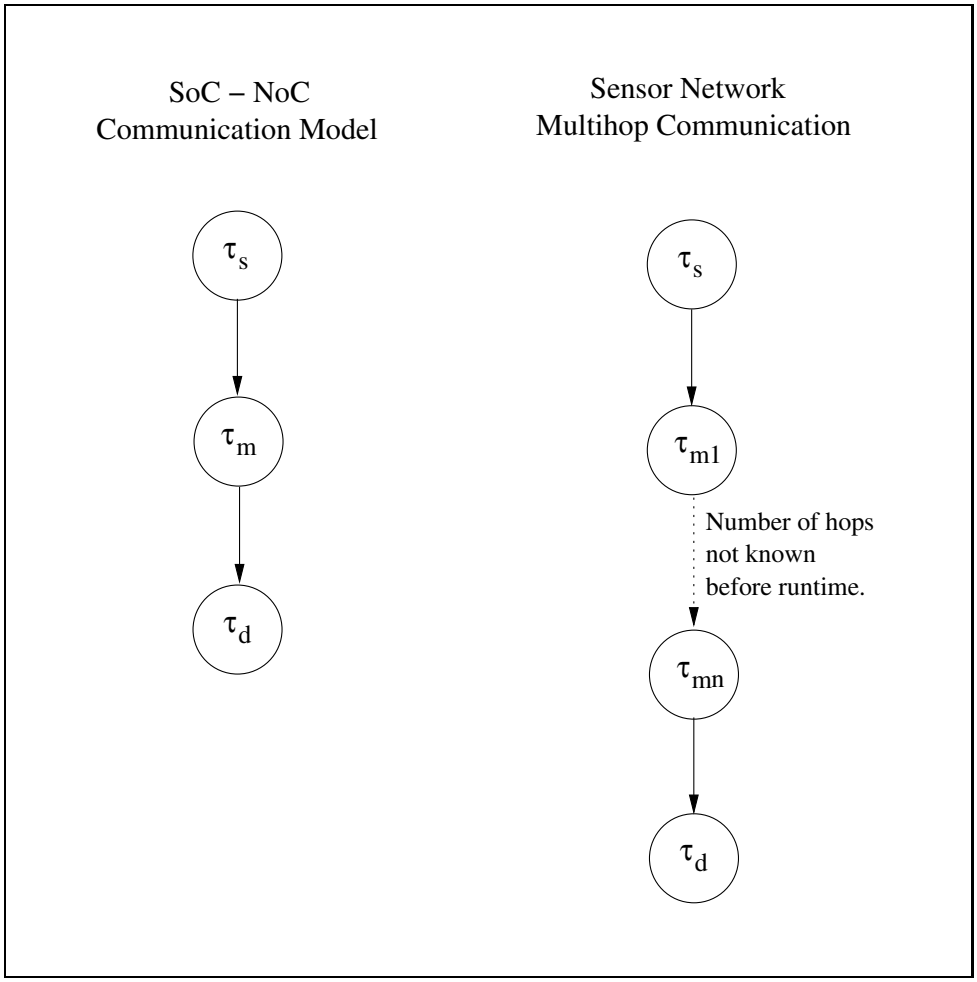

Figure 4.1: Task Graph Representation of Communication

To model sensor networks, the sources of their dynamicity must be modelled and the representation of application proposed by the SoC/NoC framework must be extended to support reactivity.
Additionally, in order to represent real applications and protocols, the set of tasks that have to be executed may depend on for example the value sensed, or the type of packet received, etc. For example, when receiving a packet, a node may take different actions: it may ignore the packet if the packet was not addressed to it, forward the packet if the node is on the multi-hop path between the source and the destination of the packet, or update its routing table if the packet is a beacon. Conditional executions like these cannot be represented by the static task graph of the SoC/NoC framework but are required for simulating communication protocols.

4.1.3 Communication Model

Another important difference between SoC/NoC architectures and sensor networks is how messages are exchanged. The NoC is a complex component composed not only of connections between processors In SoC, but also of network interfaces that manages the connections to ensure the end-to-end delivery of the messages. Therefore, the communication of messages can be modeled by single tasks that are handled by the NoC module. The NoC model represents the characteristics of the protocol at a high level and models the latency of the task. The energy consumption of the communication could also easily be added to the message task because the whole SoC shares a common power supply. In sensor networks, It is not possible to represent communication in the same way. Indeed, the protocol not handled by a common network component but is distributed over the nodes. Transmitting or receiving a message are activities that require energy and processing time both at the sender node and at the receiver node and as each node has its own energy resource, it is important to represent the impact of the communication separately for the two nodes. In the case of multi-hop routing, the communication not only involves the two end-nodes, but also all the intermediate nodes of the route of the message. The energy and processing time costs of the communication must be modeled at each of these nodes too.
Communication protocols have a major impact on the energy consumption of the nodes and the global lifetime of the network, and the choice of a protocol is an important decision in the sensor network design. Therefore, it is crucial that the sensor network simulation framework is able to represent consequences of choosing one or another protocol. For this purpose, the high-level model of communication used in the SoC/NoC framework cannot be used, and a lower-level model must be proposed.

4.1.4 Size of Sensor Networks

Sensor networks are typically composed of hundreds or thousands of nodes (against processor counts in the order of tens for SoC's). Defining the components (task models, RTOS models, etc.) of all the node at one level as it is done in the SoC framework is not possible: the sensor network specification file for simulation would become too large and very difficult to maintain. Instead, it is possible to define a sensor networks in two steps: first define the nodes (RTOS model, application model, etc.) and then define the

sensor network as a set of nodes. This definition in two steps is done even simpler by the fact that the sensor nodes often are identical. Therefore, it is possible to define a class of nodes and to instantiate as many nodes as desired in the sensor network.

4.1.5 Input/Output Tasks

The interconnection between a sensor node and the physical world is done using sensors (and actuators). These components are controlled in software by drivers that can be represented by tasks in the simulation framework. The communication between these drivers running on the CPU and the sensors is specified by the sensor protocol which varies very much from sensor to sensor. However a common property is that most Input/Output (IO) devices use interrupts to communicate with the driver. The interrupts are used in mainly two contexts:

- interrupts can indicate that an event has happened. Some sensors are equipped with intelligence and can detect events autonomously. On an event (e.g. sensed value is larger than a threshold) the sensed value is converted and an interrupt is generated to indicate that the sensor should be read.
- interrupts can indicate that a request has been completed. Some sensors are passive and only measure the environment when it is requested by the driver. After requesting the sensor, the driver suspends execution and waits that the sensor has completed sensing and that the sensed value has been converted to a digital value. During the sensing and conversion latency, the operating system can switch to another task. When the digital value is ready, an interrupt is generated and the driver is returned.

For SoC's used to perform intensive processing (for example multimedia application like MPEG encoding/decoding), the weight of the IO operation in the application performance may not be important. Therefore, a system-level model of such systems may not require detailed modeling of IO. Whereas, the situation in sensor network is opposite: the processing activity of the nodes is generally small while IO operation with the environment and with the wireless channel represent a large fraction of the application. To model the behavior of the application correctly, modeling the IO interrupts is important.

4.2 Sensor Network Model

The sensor network model is composed of two types of components: the sensor node components and the environment components. The environment components represent abstractions of the different phenomena of the environment monitored by the sensors on the sensor nodes. The environment component(s) describes the evolution of the different phenomena of interest for the sensor networks. The node components model the behavior of the sensor node and their interaction with the environment. In addition to these two types of components, the monitoring components are connected to the nodes in order to monitor the activity of the network. Thus, the proposed framework models sensor networks as is illustrated by figure 4.2.
The emphasis of the sensor network framework design was put on its modularity. This allows to interchange different implementations of the different components of the model.

Different node designs and different communication protocols are represented by different implementations of the node components. Replacing a design with another can simply be done by changing the node component used while the structure of the network model remains the same. The design space exploration can easily be done in this way. In the design process, the different decisions to be made may not all require the same level of detail in the environment modeling. Similarly to the nodes, it is also possible to have a number of implementations of environment models and to select which one to use for a given simulation. This representation of networks makes it also very easy to add or remove nodes or environment models.

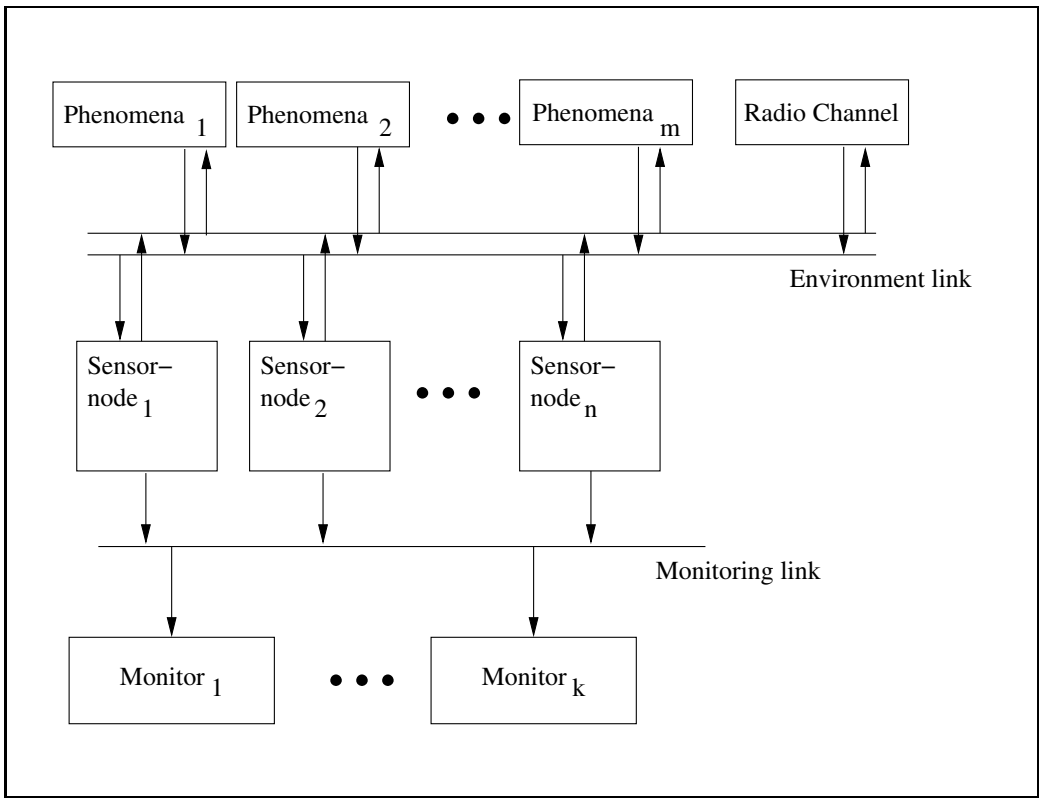

Figure 4.2: Sensor Network Model

4.2.1 Network Monitoring

The monitoring components are not a physical part of the network but provide the possibility to the user to monitor specific aspects of the network behavior. Indeed, most of the metrics of interest for measuring the performance of a design are not observed at the node level but rather at the network level. Some of the common metrics are end-to-end packet latency, packet delivery ratio, etc. The monitoring components fulfil this purpose. A link (called monitoring link) from the nodes to the monitoring components makes it possible for the nodes to communicate the information that is needed for computing the metrics of interest to the monitoring components. Such components may also be used for debugging a model.

4.2.2 Environment Model

The environment can be described by one or more environment components. One approach consists of representing each of the monitored phenomena (temperature, light, etc.) with separate environment components. This approach is modular and allows to change the individual models very easily. However, this is not suitable for modeling environments where the phenomena considered are dependent; in this situation, it might be easier to represent several phenomena within a single component. The environment model All the node and environment components are connected to the environment link. This link allows interaction between a node and a phenomenon of the environment. Three different classes of interaction can be distinguished:

1. event-detecting sensors. The environment can actively notify nodes of the network to model the phenomena that are monitored using event-detecting sensors.
2. polling sensors. For sensors that do not have the event-detection facility, the environment can be polled. This is done in two steps: the node sends a request to the environment and the environment sends the computed value back to the node.
3. actuators. It is also possible to represent actuators by sending some actuation message to a phenomenon component. This actuation will affect the model of the connected phenomenon.

4.2.3 Radio Channel Model

In the SensorSim framework [16], The idea of using the radio propagation model of the ns-2 network simulator to model the physical environment was introduced with the concept of sensor channels. The framework proposed in this report is based on a general simulation environment (SystemC) rather than on a network simulator. However, the idea of representing the radio channel similarly to the environment model can be used. Indeed, the radio channel is nothing but a physical phenomenon, namely electro-magnetic waves. The sensor node have the ability to sense (listening to the radio channel) and to actuate (transmitting on the radio channel) this phenomenon. As discussed in the previous section, both of these actions can easily be modeled.
This representation of the radio channel naturally decouples the transmission and the reception of packets and thereby make it possible to represent the costs of communication, in terms of processing time and energy, at the individual nodes.

4.2.4 Sensor Node Model

The sensor node model is composed of components that can be classified into five groups:

- The Real Time Operating System Model
- The Application Model
- The Hardware Components Model
- The Battery Model
- The Mobility Model

The sensor node model is illustrated by figure 4.3 and the five groups of components are presented in more detail in the sections below.

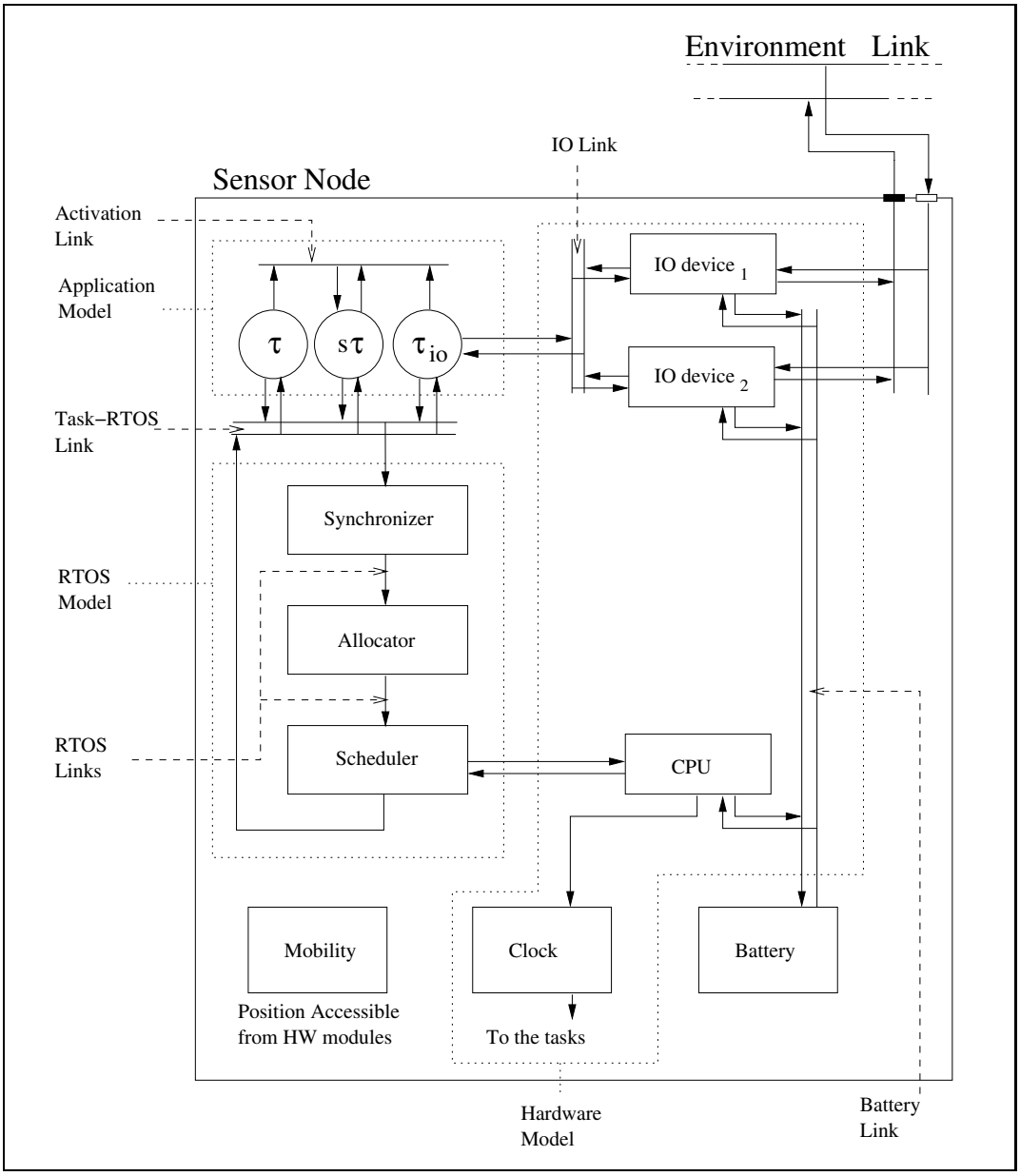

Figure 4.3: Sensor Node Model

Real Time Operating System Model

The RTOS model is maintained from the SoC/NoC framework and is composed of a synchronizer, an allocator and a scheduler that have the same functionalities as in the SoC model. However, the sensor node RTOS model differs by the fact that the synchronizer is local to a node. As mentioned in the previous section, the abstraction of having a global synchronizer representing the dependencies of tasks at the network level would not be able to represent the actual behavior and the energy consumption of the nodes accurately enough. Instead, each node has its own synchronizer that manages the dependencies

among its local tasks. The abstraction of dependencies between nodes is thereby replaced by explicit communication through the radio channel.

Application Model

The application is represented by tasks similarly to the SoC/NoC framework: they represent all activities that are requiring CPU time and the CPU allocation to the task is managed by the RTOS model. As discussed before, the periodic task model is not sufficient to model sensor network because of the reactivity of sensor network application. Therefore, new tasks classes are defined for the sensor network framework.

Sporadic tasks are tasks that are externally activated by activation messages coming from other tasks or from events of the environment thus allowing to represent reactivity of the environment. These activation messages are passed through the activation link of the node. Sporadic tasks also offer the possibility to be associated a functional description. Activation messages not only activate the sporadic task, but also data. This way, the activating task has the ability to pass its computation results to the activated task. The activating task can be of any task class: it can be another sporadic task, but it can also be a periodic task, an input/output task or any other user defined task.
In addition to the external activation mechanism, sporadic task are also able to support a *functionality*. The functionality defines data dependent behavior and is defined in two points: task activation control and task processing. Task activation control consists in enabling or not the activation of the task depending on the value of the data carried by the activating message. This ability is used to represent conditional execution. The task processing part of the functionality generates an output value based on the value it received from the activating task, and thus allows to represent the functional behavior of the task.
The state machine of sporadic tasks is very similar to the state machine of the periodic task (c.f. figure 3.1) with the difference that the transition from the inactive state to the ready state is controlled by the activation message and the task activation control as a guard. The ability of emitting an activation message on the transition from the running back to the inactive state is added to all the task classes of the sensor network framework.

Input/Output tasks are task that represent the interconnection of the sensor node application with the IO devices of the node. They are used to model the drivers of the sensors and the radio transceiver(s). Because these tasks are connected to the IO devices of the nodes, they are affected by the latency characteristics of these devices due to for example ADC conversion or radio signal demodulation. During these latency periods, the task does not use the processing resources of the CPU and is normally suspended while waiting for the IO operation to complete. To represent this behavior of IO tasks, an additional state is added to the task state machine: the self-preempted state. Transition from and to this state are entirely controlled by the IO task: when requesting the IO device, the task self-preempts indicating the RTOS that it thereby releases the processor; when the IO operation completes, the task self-resumes indicating the RTOS that it is now ready for execution. The self-resumption represents interrupts from the IO devices. The IO tasks state machine is presented in figure 4.4. The transitions to and from the

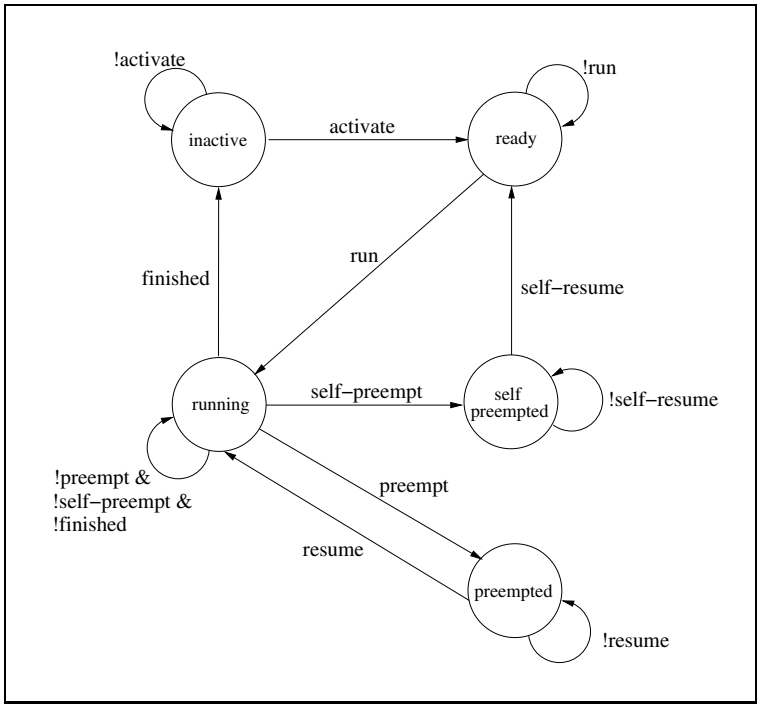

Figure 4.4: Task State Machine with Self-Preemption

self-preempted state are controlled by the task itself and represent the protocol used for communicating with the corresponding IO device. Because of the variety of protocols, it is not convenient to specify them using parameters. Instead, specific implementation corresponding to the different IO devices used have to be implemented.

Sporadic and IO tasks are the three task classes proposed for this sensor network framework. These two classes do not exclude each other and a task can belong both of them: this is for example the case with a task that sends packets on the radio when defined events occur.

Communication Protocol

The network protocols are functions that execute on the CPU and control the transmission of messages on the radio channel. Therefore, they are represented as part of the application using task modules. The lower layers (data link and physical) are represented by IO tasks as they are directly linked with the manipulation of the transceiver. In contrast, the network link does not manipulate the transceiver directly and is therefore simply represented by sporadic tasks. The activation of the sporadic task includes the packet that was received or that must be sent, and the functionality member of the task (mainly the `execute` function determines what action should be taken and sends the corresponding message to an IO task responsible of taking this action. Examples of how this is done are given in section 5.2.2.

Hardware Component Model

The hardware component models represent the latency and the power consumption of the processors , sensors and radio transceiver(s) of the sensor node. The sensors and the radio transceiver components are IO devices and are therefore connected to IO tasks of the application model. They handle the request of the IO tasks by requesting the environment model they are connected to, and they pass the event of the environment to the IO tasks.
The hardware component models also model the different power states of the components as state machines. The states of these state machines represent the power state of the components and are specified by a power consumption. The transitions between the power states are characterized by the average power consumption during the transition and the transition delay. The transition between different power states are controlled by the IO task for the IO devices and by the scheduler for the processor(s).
In addition to the power consumption, a state of an IO device is also characterized by the request that can be handled in this state (e.g. a transceiver in receive mode is not able to transmit). Also, depending on the power state, events from the environment may or may not be passed to the connected IO task.
The power state of a processor not only affects its power consumption, but also the frequency of its clock and thus the time of execution of the task run by the processor. This requires to also model the clocks of the individual nodes. Originally, the tasks are clock that represent time. To model the fact that the CPU can run at different frequencies, a clock generator component is added that can divide this global clock. The task are connected to this clock and their execution time is thereby determined by the rate at which this clock runs. The clock generator component can also be used to model clock drifting.
This representation of the hardware components makes it possible to model the consequences of using dynamic power management techniques and DVS scheduling.

Battery Model

The battery model models the energy resources of the sensor nodes. It is connected to each of the hardware components of the node and can thereby decrease its energy resource depending on the current power draw. Every clock cycle, the battery model updates it resource according to a function that is defined by the user depending on which battery model he chooses to use. Thereby, simplistic linear models as well as more advanced models taking the hysteresis phenomena into account can be represented. The link between the hardware components is bidirectional: this allows to model the death of a node when the battery is empty. The battery model can also be implemented to inform the hardware components when its energy resources go below predefined thresholds.

Mobility Model

The mobility model represents the movement of the node and keeps track of its position. This position is accessible to the IO devices that include it in all the network requests. The environment requested can then compute the value of the requested phenomena at the position of the node. Similarly, when events are sent to the nodes, each node must

compare its position against the position of the event to determine whether the event is visible or not.

Chapter 5

Implementation of the Model.

The model described in the previous chapter is implemented in SystemC extended with the master-slave add-on library. In this chapter, The implementation of the structure of the model is documented, and examples of implementation of the different components of the model are implemented.
The examples implemented are mainly based on the MICA node [5] running the Tiny OS operating system [6]. This platform and operating system were chosen because they are well-known in the sensor network area, and because it was possible to find detailed description of their characteristics. Depending on the platform and the application modeled, new components may be needed. In section 5.2, the methodology for adding new components is presented, and the ability of the framework to cover the sensor node design space is discussed in section 5.3.

5.1 Organisation of the Model Implementation

Based on the structure presented in the previous chapter (figure 4.3), the components of the model are implemented as SystemC modules. The class diagram in figure 5.1 presents the implementation organisation of the framework. All the classes presented here (boxes with plain borders) represent parent classes for the different types of modules that are used. They define the common features of these module types such as their interface and some functions that all the modules of the given type utilise. The SystemC methods describing the behaviour of the different modules are attached abstract methods and it is therefore not possible to instantiate any of these abstract modules in a simulation. Instead, implementation modules (boxes with dotted borders) must be defined. These modules inherit from the abstract module of the module type they belong to and implement the abstract functions of it. This implementation approach using abstract classes ensures the uniformity and compatibility of the implementation modules developed and facilitates the task of adding new modules. It also reduces the redundancy of code in the modules of same type and thus eases the maintenance of the framework. In this section, we present the different classes of modules of the framework. For each module class, we present its important parameters (mainly the parameters needed by the constructor), its interface and its SystemC methods and the functions that are attached to them. Many module classes also provide function to generate messages to communicate

on the master-slave links. These functions are referred to as communication functions. At the end of the section, we also present the module interconnection technique used.

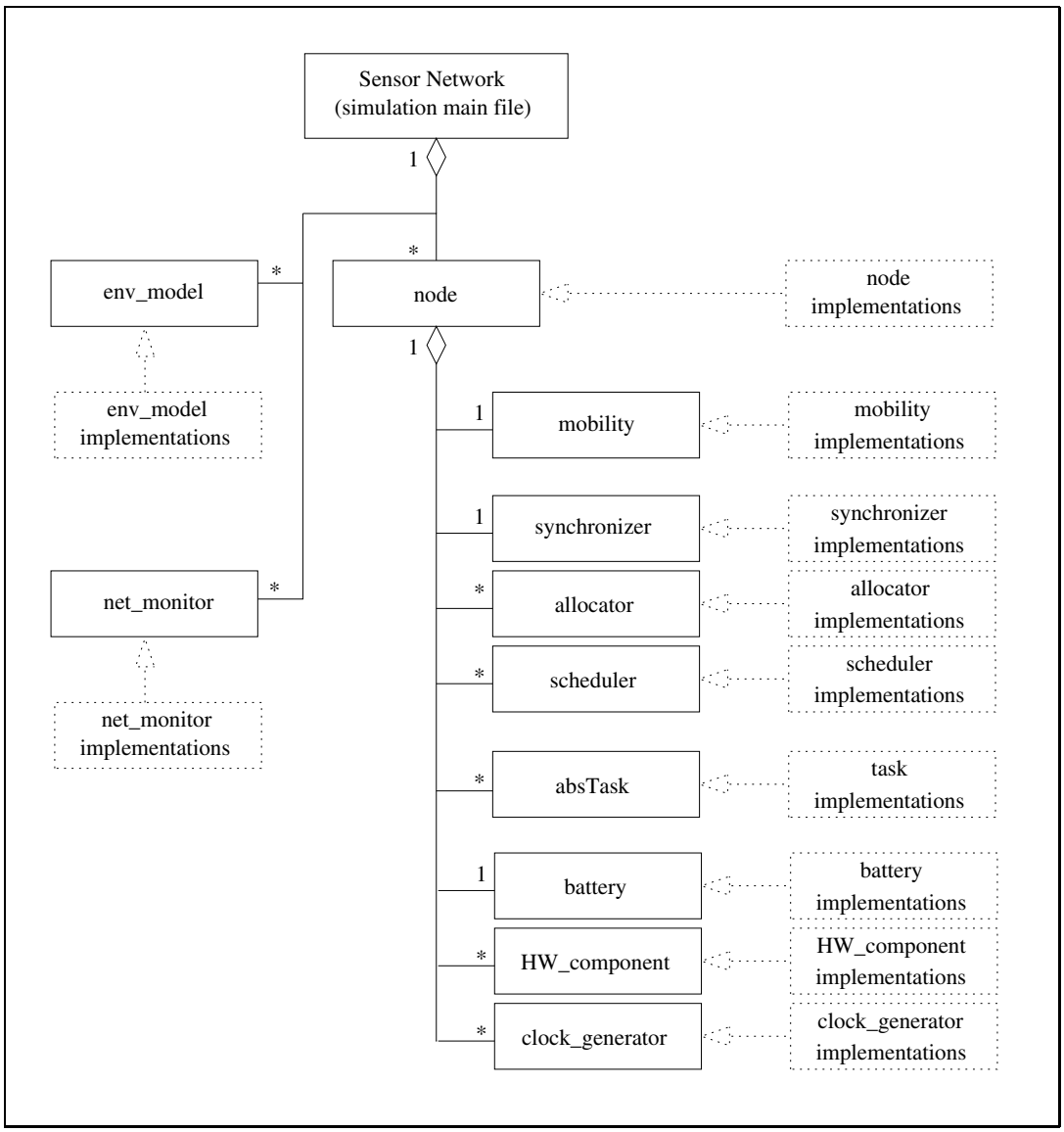

Figure 5.1: Class Diagram of the Framework

5.1.1 Environment Model Modules

The *env_ model* module class defines the standard structure of the environment models.

- Parameters.
  Each environment model is uniquely specified by its `phenomenonID` member. The nodes insert this number in their request to indicate which environment model the request is addressed to.

- Interface.
  The basic interface of environment models is simply composed of a pair of master-slave ports that connects them to all the nodes of the network. `in_request` is a slave port for receiving the node requests and `out_status` is a master port to communicate information about the modeled phenomenon to the nodes.
- SystemC Methods.
  The *env_ model* module class also declares the `env_request` function attached to the slave of the `in_request` port. This function is called each time a node sends a request to the environment.

5.1.2 Node Modules

The *node* module class defines the standard structure of the node models.

- Parameters.
  Each node model is uniquely specified by its `nodenum` member. This value can be used by the environmental models when they need to send a message to a particular node.
- Interface.
  The interface of the node module is complementary to the interface of the environment models. it is composed of the `out_envRequest` master port used to send requests to the environment models and the `in_envReply` slave port receiving the resource responses and events. It is to be noted that there is no SystemC method with the slave port. This is due to the fact that the external slave port of the node is connected internally to the slave ports of the IO devices. The nodes also have a clock port from which they receive the global synchronization signal for the simulation.
- Node Components.
  The *node* modules also declares the node components as described by the class diagram of figure 5.1. Each node also have a *position* object that defines its position and is accessible to the IO devices.

5.1.3 RTOS Modules

The *synchronizer*, *absAllocator* and *scheduler* module classes define the structure of the three components of the RTOS model. They have a similar structure:

- Parameters.
  While there is only one synchronizer per node, there might be more than one allocator and scheduler (multiprocessor SoC). Therefore, each allocator-scheduler pair has a `schid` that is unique within the node. Tasks must specify the id of the allocator-scheduler pair they are mapped to within the messages they send to the RTOS.
- Interface.
  The RTOS modules have a pair of master-slave port. The slave port receives requests and handles them and/or passes them on using the master port. They are interconnected as in the SoC/NoC model.

- SystemC Methods.
  Each of the class module has a slave SystemC method connected to their respective slave ports. The functions attached to these methods (`synchronize` for the synchronizer, `updateResourceDatabase` for the allocator and `do_schedule` for the scheduler) are declared as abstract in the respective module classes.
- Communication Functions.
  These three module classes also define some functions generating messages for the RTOS internal communication and the communication with the task modules.

5.1.4 Mobility Modules

The *mobility* module class defines the structure of the mobility model.

- Interface.
  The mobility model has only one port for the clock. However, it is also given a pointer to the position object of the node that.
- SystemC Methods.
  the *mobility* module class has an clock-sensitive SystemC method. The abstract function `movement` is attached to this method and is responsible for changing the position of the node according to the chosen mobility model.

5.1.5 Task Modules

The tasks are modules that represent the execution of instruction on the CPU. They are managed by the Operating System (OS). The task modules of the sensor network model are based on the concept introduced by the SoC/NoC model. In addition to the periodic task (*perTask* module), modeling sensor network applications require additional classes of tasks. These tasks differ in their execution patterns, but they share some common characteristics. These characteristics are defined in the *absTask* module class. As can be seen in figure 5.2, the task module type is further divided by defining the abstract class *ioTask* inheriting from *absTask* for the IO tasks. In this section, the abstract task modules, *absTask* and *ioTask*, and the implementation module *sporTask* are documented. The *perTask* module is identical to the one of the SoC/NoC framework. The other implementation modules are examples of *ioTask* sub-modules of *ioTask* and are presented in section 5.2.1.

Abstract General Task Module Class

The *absTask* module class describes the fundamental characteristics of all the tasks and provides some functions to generate messages to the operating system model.

- Parameters.
  The tasks have many parameters. Each task is uniquely defined within the node by its `id`. The number of the scheduler to which the task is mapped, `schid`, is also defined. Additionally, the node has real-time parameters (period, priority, deadline, etc.). They are defined in the abstract class because these parameter may be used by the scheduling algorithm (for example EDF schedules tasks depending on their deadline, RM depending on their period) and are therefore required to be able to define the functions for generating the messages for requesting the RTOS.

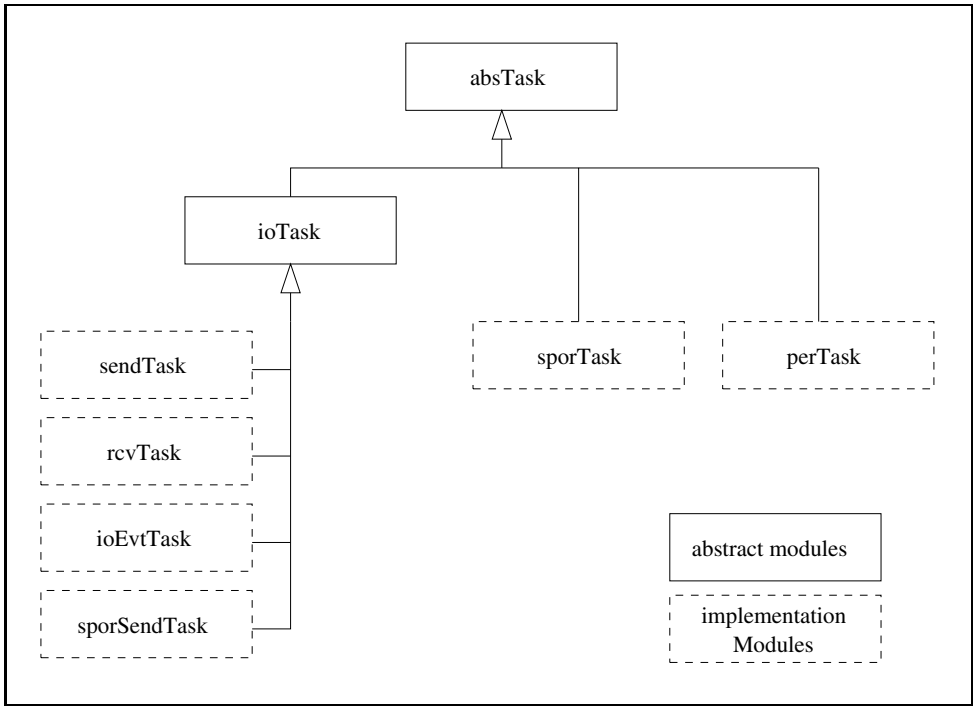

Figure 5.2: Class Diagram for task modules

to enable to define the communication function at this level. A tasks also has a `schid` parameter that indicates which scheduler the task is mapped to. Finally, Each node is given the number of the node it is attached to: `nodenum`; this is mainly used for debugging purposes.

- Task Interface.
  The fundamental interface of the task models is composed of a clock port and a pair of master-slave port for connection to the operating system model. The `out_indication` master port is used to request services from the operating system (requests for execution and requests for resources). The `in_command` slave port receives the commands from the operating system. Additionally, all tasks are equipped with the `out_activate` master port that is used for activating sporadic tasks. Also note that the clock port is not a traditional boolean port, but rather a integer port. This is needed for supporting Dynamic Voltage Scaling techniques. This is further explained in section 5.1.8. All tasks also have a output signal, `sig_state`, indicating their state. It makes it possible to trace the state on the simulation waveforms.
- SystemC Methods.
  There are two main SystemC methods in the *absTask* module class: The first is a clock-sensitive method that has the responsibility of counting time and of notifying the operating system when execution finishes, resources are required, etc. However, the behavior is not represented in the *absTask* class because it is very dependent on the type of tasks. The other is the slave method attached to the `in_command` port. This method handles the commands of the operating system,

such as the `RUN` or the `PREEMPT` commands. Together, these two methods controls the transitions of the task based on the state machine presented in the previous chapter (figure 4.4). However, the functionality of the methods is not implemented in the *absTask* class. Only, the transition to the `s_off` state representing the fact that the node's battery is empty is controlled in this class. The rest of the walk-through of the state-machine is deferred to the member functions `state_machine` and `get_comm_from_rtos` called respectively on clock events and on reception of RTOS commands. These functions are abstract functions. As all the task inherit from the *absTask* module class, they all have to provide an implementation of these two functions for describing their behavior.

- Communication functions.
  The *absTask* module also defines some standard functions. `forwardMessage` generates a copy of the message passed as argument, `sendIndication` generates a request to the scheduler. The argument of the function defines the type of request. The generated request further contains the task parameters (task number, priority, etc.). `sendResourceInfo` generates a request to the allocator. The type of request (resource request or resource release) and the number of the concerned resource are passed as arguments of this function. `activateTask` generates an activation message carrying the data passed as an argument. As the type of data carried by the activation message vary with the application, this function is defined as a template so that the argument of the function can be of any data type.

Abstract Input/Output Task Module Class

The difference between *ioTask* and other tasks is the fact that IO tasks not only communicate with the RTOS, but also with hardware components (despite of the name, input/output tasks can be used not only for IO device managing but also for power management of processor units).

- Parameters.
  In addition to the *absTask* parameters, each *ioTask* has a parameter, `hwcid`, to specify the hardware component it is handling. This parameter is inserted in all requests to the hardware components modules and is also used to filter responses and event from them.
- Interface.
  The *ioTask* also have an additional pair of master-slave ports to communicate with devices. The `hw_request` master port is used to request the hardware component and the `hw_reply` slave port to get responses or events from it.
- SystemC Methods.
  The slave port's SystemC method is attached the `get_comm_from_hw` abstract function which is responsible of handling the messages received from the IO devices.

Sporadic Task Model

The sporadic Tasks are tasks that are not activated periodically but that are activated by an external module through the activation link. Similarly to *ioTask*, the *sporTask* module inherits from the *absTask* class. However, *sporTask* is not an abstract class: it provides

an implementation of the `state_machine` and the `get_comm_from_rtos` functions. It represents a computation activity with no Input/Output activity.

- Parameters.
  The sporadic tasks must specify which task they expect an activation from. The `activatingTask` parameters fulfils this purpose. When receiving an activation message, the sporadic task module compares number of the task sending the activation message with this parameter to determine whether it should activate or not.
- Interface.
  An additional slave port, `in_activate`, is added to connect to the activation port.
- SystemC Method.
  In addition to providing an implementation of the abstract functions of the *absTask* module, the *sporTask* module has an additional SystemC slave method for the `in_activate` port. The function `get_activation` is attached to this slave method and handles the activation messages. The three SystemC methods implemented in the *sporTask* module maintain the state of the task according to the state machine presented in the previous section (figure 4.4).
- Functionality of Sporadic Tasks.
  The functionality of sporadic tasks are described by a `Function_c` object given to the task through its constructor. `Function_c` is a class is composed of the `start_enable` and `execute` methods. Both of these methods operate on the input data, i.e. the data carried by the activating message. `start_enable` is called when an activation message is received. It returns a boolean value specifying whether the task should become ready or ignore the activation message. `execute` is called when the task finishes execution and generates output data. This data is inserted in the activation message generated by the sporadic task when it completes. `Function_c` is an abstract class declaring these two methods.
  To define the functionality of a sporadic task, the user of the framework must define a sub-class of `Function_c`. The functionality of the sporadic tasks was defined as a class rather than functions in order to allow the user to store information about the previous activation received. Thereby, the methods of the functionality class may not only depend on data carried by the current activation, but also on the previous history of the sporadic task.
  It is to be noted, that a sporadic node need not be attached a functionality. If a null pointer is passed instead of a `Function_c` object, the sporadic task gets activated independently of value carried by the activation message, and the value will be passed on in the activation messages generated at the end of the execution of the sporadic task.

A problem of the representation of the sporadic tasks is that a sporadic task can only follow the execution of one instance of the task. For example, if a sporadic task receives an activation before the execution corresponding to the former activation is completed, the second activation is ignored. A solution to this problem would be to have a module dynamically creating sporadic tasks at simulation time. However, this is not supported by the current version of SystemC (version 2.0.1). It is expected that version 3 will support this feature.

5.1.6 Battery Modules

The *battery* module class is an abstract module and has the following characteristics:

- Parameters.
  The global battery model is defined by two parameters: `charge` and `discharge_rate`. These two member variables have floating point values. The constructor of the battery model initializes the charge with the argument that it is given and the discharge rate to zero.
- Interface.
  The battery has a clock input and a pair of master-slave ports for communication with the hardware components of the node. The slave port `in_batRequest` receives requests from the hardware components and the master port `out_batStatus` sends status information to the hardware components and the scheduler.
- SystemC Methods.
  The battery module has two SystemC methods. The first is a clock-sensitive method and it is attached the `updateCharge` function. The implementation of the function describes the discharge profile of the battery. Depending on the battery model selected, the implementation of this function differs. Therefore, this function is abstract and must be implemented by the sub-modules of `battery`. The second SystemC method is the slave method of the `in_batRequest` port. It is attached the `updateEnergyRate` function. This function is called when battery requests from the hardware components are received. These requests specify the variation of the energy consumption rate. The `updateEnergyRate` increments the `discharge_rate` member of the battery module with this value (note that the value can be negative).

5.1.7 Hardware Component Modules

The hardware component module class represent the latency and the energy consumption characteristics of the devices of the sensor node. We distinguish two sub classes of hardware component. They are connected to either IO tasks or other components (for example, the CPU may be connected to the scheduler for DVS scheduling). The module class *HW_ component* is an abstract class from which all hardware components inherit. As for the task modules, the hardware components can be further divided by defining the abstract modules class *IO_ device*. This is illustrated in figure 5.3. In this section we present the *HW_ component* and the *IO_ device* module classes. The implementation modules are example of hardware components and are presented in section 5.2.1.

Abstract Hardware Component Module Class

The *HW_ component* module class describes the characteristics that are common to all the hardware component models.

- Parameters.
  The hardware components are characterized by three parameters. The `hwID` integer is a uniquely specifies each hardware component and is used for filtering the messages from the IO tasks. The `es` data structure is an array of `EnergyState` elements. Each `EnergyState` element defines the characteristics of one energy state

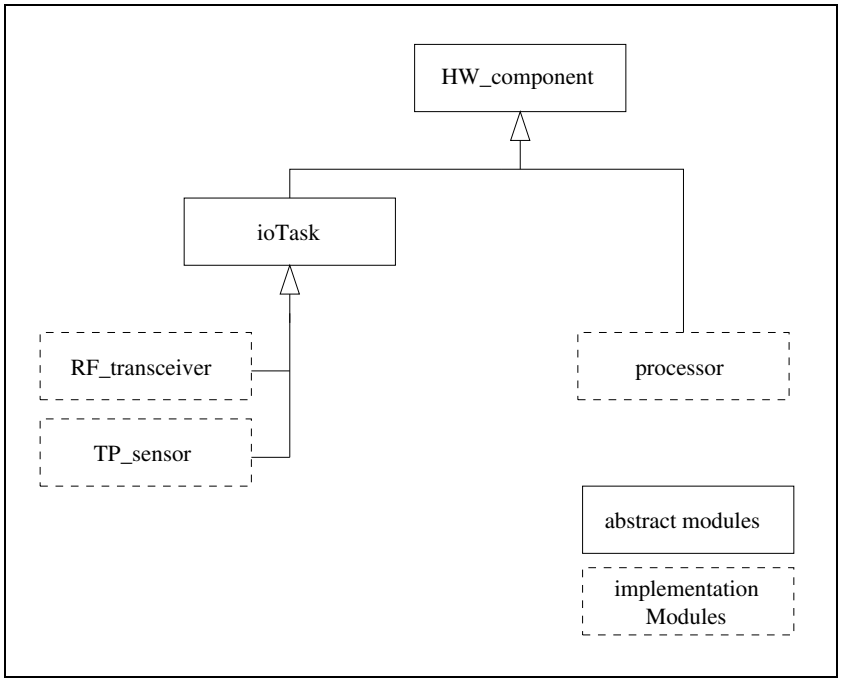

Figure 5.3: Class Diagram for Hardware Components

as the energy consumed per cycle in this state (defined as a floating point value) and the latency and the cycle energy needed for transitioning to a new energy state.

- Interface.
  The hardware components have a clock input (needed for modeling state transition delays) and two master-slave interfaces. They communicate with the ioTask through the `in_request` slave port and the `out_response` master port. The `out_batRequest` master port and `in_batStatus` slave port are used for informing the battery of variations in the energy consumed per cycle, and to receive battery status information (for example a "battery low" message). In addition to these, the hardware components have a float output indicating the cycle energy they consume. This output makes it possible to trace this value on the waveform generator of SystemC.
- SystemC Methods.
  Hardware components have three SystemC methods. The first is the clock-sensitive method. It is attached the abstract function `delay_model` that is responsible for modeling the delays of energy state transition. The two other methods are the slave methods of the task and battery communication. They are attached respectively the `hardware_request` function called on every request from the IO task and the `get_status_from_battery` function called on reception of status messages from the battery. Both functions are abstract.
- Communication Functions.
  A single communication function is implemented within the *HW_component* module: `sendBatteryRequest` generates a message for the battery with a cycle energy

variation passed as argument.

Abstract Input/Output Device Module Class

The *IO_ device* module class is the sub class of the hardware components that are connected to the environment modules. They represent the sensing and acting devices (sensors, transceiver, etc.).

- Parameters.
  In addition to the *HW_ component* parameters, *IO_ devices* identify the environment model they are connected to by the `phen_num` parameter. The `position` parameter points to the position of the node and is passed in the environment requests. Finally, `conversion_delay` defines the latency from the environment is requested and to the value is passed to the connected IO task (represents latency of ADC, demodulation, etc.).
- Interface.
  The *IO_ device* modules are connected to the environment link through a pair of master-slave ports: the `out_request` master port and the `in_response` slave port.
- SystemC Methods.
  The slave method of the `in_response` port is attached the abstract function `environment_response`. This function is responsible to store responses from the environment. The `delay_model` abstract function inherited from the *HW_ component* module class is responsible to delay these responses and forward them to the IO task connected.
- Communication Functions.
  The environment link and the input/output link (link between the IO tasks and the IO devices) do not carry identical data structures. The `sendEnvironmentRequest` converts the input/output link message passed as argument to an environment link message, and the `returnEnvironmentResponse` implements the reverse conversion.

5.1.8 Clock Generator Modules

Concerning the clocking of the nodes, it is important to distinguish the clock signal that are used to synchronize the components of the simulation, and the clocks that represent actual clock signals such as the processor clocks. the global synchronization clock signal distributed to all the nodes represents the notion of time and is used to model latencies, and is used for the battery and the mobility model. Also for the tasks, the real-time characteristics such as the deadline and the period are connected to this absolute time clock. In contrast, the computation time and the resource request times are connected to the rate at which the processor the task is mapped to runs. The tasks are therefore not directly clocked by the clock signal provided to the node.
The *clock_ generator* module generates a clock with a variable frequency based on the absolute time clock. The clock generator implemented here only divides the clock in order to represent the effects of DVS on the task execution. However, it would also be possible to add a clock drift model into this module.
As mentioned above, the tasks need both an absolute clock signal and a CPU clock signal. These two boolean signals are merged into one integer signal: the clock signal

at the output of the clock generator can take four values: 0 to 3. While all the events on this integer clock signal correspond to a absolute time clock event, only the events leading to a value of 2 or 3 correspond to an event on the CPU clock signal. This means that while the real-time characteristics of the tasks are updated on every clock event, the computation time of the tasks are only updated when the value of the clock signal after the event is larger or equal than 2. This can be implemented as it is in the sporadic task module (figure 5.4).

```
void sporTask::state_machine() { // fction of clk-sensitive SC method

  int tempClock = clock;      // copying the signal in a variable

  ...
  dl_counter--;               // actions taken on abs time clk events
  ...

  if (tempClock > 1) {
      comp_time--;
      ...                     // actions taken on CPU clock events
  }

}
```

Figure 5.4: Handling of the Integer Clock in the *sporTask* module

The *clock_ generator* module characteristics are:

- Parameters.
  The behavior of the *clock_ generator* module is dictated by its `divider` parameter that defines the value by which the clock input must be divided by.
- Interface.
  The *clock_ generator* module has a boolean clock input, `clkin`, and an integer clock output, `clkout`. Additionally, the module has a slave port used as a control input, `divider_in`.
- SystemC Methods.
  The SystemC slave method of the `divider_in` port is attached a function which changes the divider parameter of the module. The clock-sensitive SystemC method `clocking` generates the output clock signal.

5.1.9 Module Interconnection

The interconnection of the modules is done using master-slave links as shown in figure 4.3. At the network level, two links exist:

- The environment link is composed of two master slave links: `node2env` passing message from the node to the environment and `env2node` passing messages in the reverse direction.

- The monitoring link is a single master-slave link, `node2mon` passing messages from the nodes to the network monitoring modules.

Furthermore, in each node, the following links exist:

- The RTOS links are the links from the synchronizer to the allocator (`s2aLink`) and from the allocator to the scheduler (`a2sLink`).
- The Tasks and the RTOS are interconnected by a pair of master slave links: `t2sLink` from the tasks to the RTOS (synchronizer module) and `s2tLink` from the RTOS (scheduler module) to the tasks.
- The activation link is a single link, `actLink`, that connect every task to the sporadic tasks.
- The input-output link is composed of a pair of master slave link: `t2HWLink` from the IO tasks to the hardware components and `HW2tLink` from the hardware components to the IO tasks.
- The battery link is composed of a pair of master slave links: `HW2bLink` from the hardware components to the battery and `b2HWLink` from the battery to the hardware components.

While most messages are destinated to a specific module, the links provide a broadcast service. Therefore, most modules filter the messages they receive on the link by reading the ID parameters inserted in the message. For example, a `RUN` message from the scheduler contains the id of the task it is addressed to. When the message is written on the `s2tLink` link, the `get_comm_from_rtos` function will be called on all the tasks. Therefore, this function must start by comparing the id of the destination task against the id of their task. If they do not match the message is rejected. A similar mechanism is used for the environment modules, IO task modules, hardware components, etc. This way of organizing the model may have some performance overhead connected to calling many slave functions when only one is actually concerned, but it makes it easy to add or remove modules: they can simply be plugged on or off these common links without requiring to create or remove links.

5.2 Creating Implementation Modules

The module classes presented above define frames for creating actual implementation modules. For creating a new implementation module, one first has to determine which module class it belongs. The implementation module must inherit from this module class. Then, the header file of the class declaring the new module and defining its constructor can be created. Inheritance for SystemC modules is very similar to standard classes inheritance. The template in figure 5.5 shows how a module named `module_name` inheriting from the module class `parentModule_name` should be defined. The members (functions, ports and variables) that are used in the implementation module but not in the parent module must be declared. The functions that are abstract in the parent class must be declared too, and the functions that are implemented in the parent class but that must be overridden for the implementation module must also be declared. Finally, the implementation of the functions declared in the header file (`<module_name>.h`) must be defined in the implementation file (`<module_name>.cpp`).

In the following, we present some implementation module that illustrate how the framework can be used.

5.2.1 Implementation Modules

To illustrate how the framework can be used, we implement a simple model. The model is inspired by the characteristics of the MICA node for the platform and of Tiny OS for the operating system. In this section, we present this example and show some results of simulating it. The purpose of the example is not so much to provide a realistic example of the sensor node, but rather to illustrate how the model can be used.

Application Representation

in Tiny OS, the execution model is based on two concepts: the tasks that all have identical priority, and the hardware interrupts. It is to be noted that the concept of task of the framework does not represent Tiny OS tasks. Instead, the task modules represent a thread of execution. This thread of execution can be a Tiny OS task (for example for computation tasks) and is then represented by a *perTask* or *sporTask* modules, or it can be a thread executing in a number of execution blocks driven by interrupts is represented by an *ioTask* module.
As communication is fundamental in sensor networks, task modules for representing the communication protocol were implemented. The *sendTask*, *sporSendTask*, and *rcvTask* modules represent the threads of execution for sending and receiving packets at the link layer level using a simple CSMA/CA protocol. This protocol is based on the protocol implemented in Tiny OS by the *SecDedRadioByteSignal* component, except that the synchronization symbol is not considered here.

The ***sendTask*** module represents a thread of execution that periodically sends a packet according to this protocol (figure 5.6). It can for example be used to represent beacon transmissions. For sending a packet, *sendTask* starts by waiting a random number of bit-periods in back-off state (bit-period refers here to the inverse of the bit rate of the transmission). After, the task senses the carrier in order to get medium access: if no communication is heard after a given number of bit-periods, the task can start transmitting. It first transmits the preamble and then the data packet.
In terms of execution on the processor, the thread for sending a message does not run in a single block. Instead, it is driven by the periodic interrupts from the radio transceiver. The *sendTask* models these interrupts by sending interrupt messages to the RTOS at the bit rate of the transmission, and then follows an execution pattern for the bit-period. When the execution pattern has completed, the task self-preempts and waits until the next bit-period. There are different execution patterns for the different protocol states (back-off, carrier sense, preamble transmit and data transmit). These execution patterns are defined in the `state_machine` function of the *sendTask* which controls the transitions of the general task state machine (see figure 4.4). The protocol state machine is implemented as a member function of *sendTask* called `sendProtocol`. This member function is called by `state_machine` at the end of each execution pattern (once per bit period) and updates the state of the protocol. It also maintains a signal with the value of the

```
#ifndef __<module_name>_H
#define __<module_name>_H

#include "<parentModule_name>.h"

void sporTask::state_machine() { // fction of clk-sensitive SC method

class <module_name> : public <parentModule_name> {

  public:

    // declare ports specific to the implementation module if needed:
    <my_port_type> <my_port_name>;

  protected:

    // declare functions that implement the parent abstract functions:
    <return_type> <parent_abstract_function>(...);

    // override parent functions if needed:
    <return_type> <parent_function>(...)

    // declare fctions specific to the implementation module if needed:
    <my_return_type> <my_function>(...);

 public:

     SC_HAS_PROCESS(<module_name>);
     // constructor of the implementation module
     <module_name>(sc_module_name name_, ...) :
            // construct the parent:
            <parentModule_name>(sc_module_name name_, ...),
    // initialize member variables:
    <my_variable>(<variable_value>)
     {

       // construct the implementation module.
       // declare SC_METHODS if needed.

     }

 protected:
     // declare member variables specific to the
     // implementation module if needed:
     <my_variable_type> <my_variable>;
};
#endif
```

Figure 5.5: Template for Declaring an Implementation Module

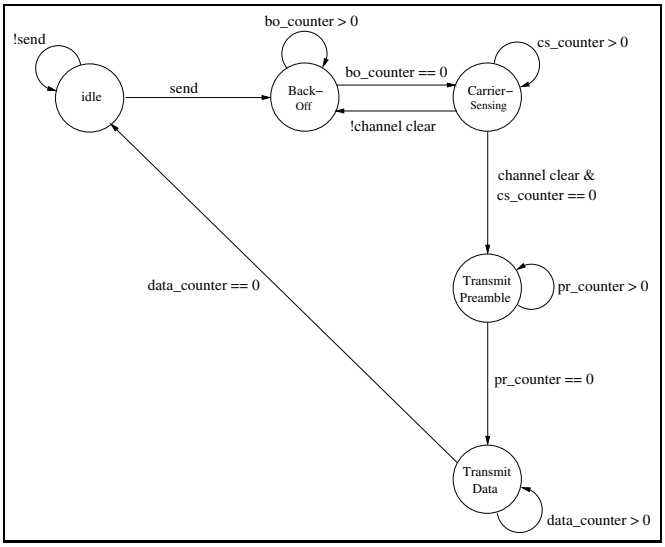

Figure 5.6: Protocol State Machine for *sendTask*

protocol state in order to make it possible to trace it on a waveform. In terms of real time constraints, the *sendTask* module supports both constraints on the task as a whole (e.g. the transmission must be completed before a packet-deadline) and on the bit-periods (e.g. the bit must be transmitted before a bit-deadline). The execution model of the *sendTask* module is illustrated in figure 5.7 and represents the handling of the radio at the bit-level.

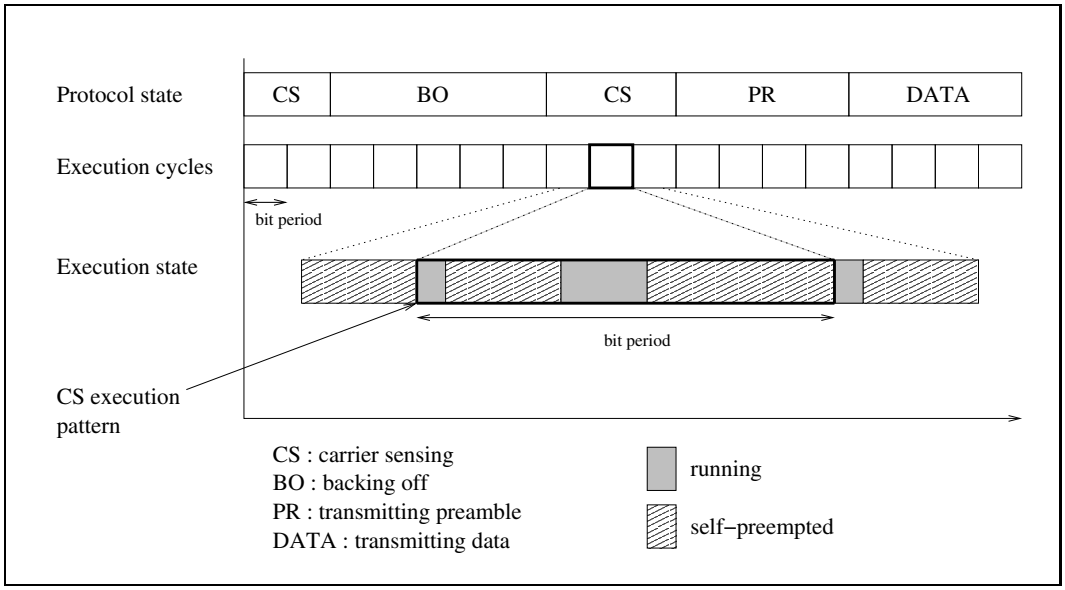

Figure 5.7: Execution Model for *sendTask*

In the sendTask, one execution pattern is defined for each protocol state. For the back-off, preamble transmit and data transmit states, the execution is done in one block.

However, the execution time of the block depends on the state, and the execution time for each states are passed as arguments of the *sendTask* constructor. For the carrier sense state, the execution is represented as 2 blocks: the first represents the requesting of the transceiver while the second represents the handling of the response. It is to be noted that the execution pattern only define when the RTOS is requested. The scheduling of the block is managed by the RTOS model which may for example preempt the task in the middle of an execution block. The execution blocks defined for *sendTask* and the interaction with the transceiver module are shown in figure 5.9 (the patterns represent the ideal case were the task is alone on the processor and is not preempted). Note that the execution patterns for transmitting a preamble or a data bit are identical.
The parameters of the protocol such as the size of a packet, the length of the preamble, and the number of clear cycles in carrier sense state before transmission can start are defined as precompiler constants. and can be changed easily (it requires to recompile the communication tasks though).

The ***sporSendTask*** module is similar to *sendTask*, but instead of being periodic, it is activated externally by another task. The activation message contains the value of the packet to be sent.

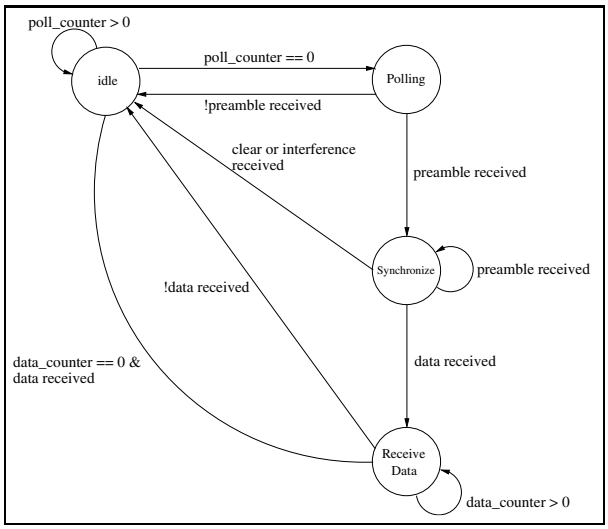

Figure 5.8: Protocol State Machine for *rcvTask*

The ***rcvTask*** module represents a task that periodically polls the RF channel and receive the detected messages. Similarly to the *sendTask* module, it executes as patterns that repeat at the bit-period. The state machine that governs the *rcvTask* module is presented in figure 5.8: initially, it polls the channel at a low frequency, alternating between the polling and the idle states. When a preamble is received, the task goes to the synchronize state, an polls the channel at the bit-period. When it receives the first data bit, it moves to the receive data state and stays there until it has received the number of bits that constitute a packet before moving back to the idle state and restarting polling. When the last bit of the packet is received, the *rcvTask* module sends

an activation message on the activation link so that sporadic tasks can react on the reception of the message. If *rcvTask* reads an unexpected value on the channel (channel clear while expecting a data bit or preamble bit), it leaves its current state, moves back to the idle state and restarts polling the channel. Concerning the execution patterns, the *rcvTask* has a single pattern which is identical to the carrier sense pattern (indeed, they both represent the same operation). As for the *sendTask* module, the execution pattern is controlled by the `state_machine` function, while the protocol state machine is implemented by the `rcvProtocol` function called at the end of each execution pattern. In contrast to *sendTask*, a sensor node only has one *rcvTask*. It represents the ability of the node to detect messages rather than a request from the application.

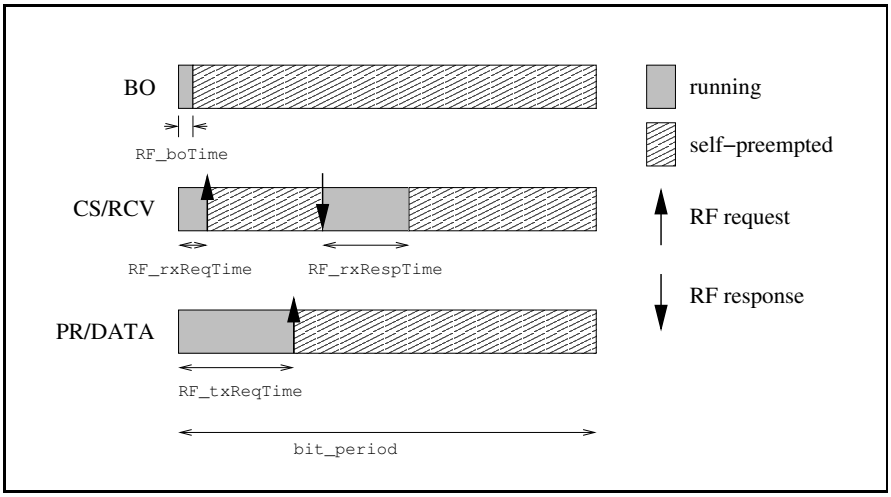

Figure 5.9: Execution Patterns for the RF tasks

The send and the receive tasks are both using the radio transceiver of the node. Therefore, they are not allowed to execute concurrently. This is ensured by having the two tasks request the radio from the RTOS. When a *sendTask* instance is scheduled and starts executing, it requests the radio from the RTOS. If the radio is not available, the RTOS will preempt the task. When the radio is released, the task will be given the radio and will resume execution. The *sendTask* instances hold the radio during their entire execution, from they start in back-off state until they have sent the last bit of the packet. The *rcvTask* is slightly different, because it is activated by interrupts coming from the transceiver. This is represented by a special type of messages to the RTOS that requests the resource at the same time as they get activated. If the radio transceiver is not available at the time of the *rcvTask* activation, the task returns to the idle state of the protocol state machine. When the transceiver is granted to a *rcvTask*, it keeps the transceiver until it returns to the idle state.

The ***tpioTask*** module is a very simple module that represents the handling of interrupts from an event-detecting sensor. When it receives a message from the sensor, it gets activated, and when its execution completes, it sends a message on the activation port. The behavior of this task is in fact similar to a sporadic task, with the difference,

that it is not activated by an activation message, but rather by an message of a hardware component.

Together with the periodic and routing tasks, the task presented here constitute building blocks for representing application. To do so, the tasks can be interconnected either by dependencies resolved by the synchronizer as in the SoC/NoC framework, or through activation of sporadic tasks. The second solution has the advantage to allow to represent data dependent execution by using the functionality property of the sporadic tasks.

Operating System Model

Scheduler and allocator modules were implemented to represent a simple operating system similar to Tiny OS. For the synchronizer, no modification to the synchronizer developed in the SoC/NoC framework are needed.

The allocator module, ***tos_allocator***, models the resource allocation. Resources can represent parts of the memory, but also the radio transceiver, or a sensor of the sensor node. It is very similar to the allocator module developed within the SoC framework, but it does not handle priority inheritance (as there are no priorities in tinyOS). Additionally, the allocator is responsible for cancelling activation messages representing interrupts form hardware resources that are already allocated to another tasks. Indeed, such interrupts cannot occur as the hardware that causes them is used by another task to which it is allocated. In this situation, a cancellation message is sent to the scheduler which forwards it to the concerned task module. The allocator maintains a data structure holding the status of all the resources.

The scheduler module, ***tos_scheduler***, models the scheduling of tasks and the handling of interrupts. The Tiny OS scheduling is used here. It is a simple non-preemptive scheduling algorithm that allocate the processor to the Tiny OS tasks in the order that the Tiny OS tasks are posted (in order to avoid ambiguities between Tiny OS tasks and the task modules of the framework, task refers to the framework task while Tiny OS tasks are referred to as TOS tasks). TOS tasks can be interrupted by hardware interrupt handling routines, and the interrupt handling routines can also interrupt each other (nested interrupts).
The scheduler keeps track of the task that are active (tasks that are not in the inactive state of the state machine of figure 4.4). To do so, it has two data structures:

- the `preempted` structure. It contains references to tasks that are waiting for the processor to become available: tasks that were activated but not allocated the processor yet and the task that were preempted by an interrupt. This data structure is implemented as a vector and can be added elements at either end depending on whether they represent a TOS task (added at the back of the structure) or interrupts (added at the front of the structure). When the processor becomes available, the task at the top of the stack is allocated the processor, and the scheduler sends a `RUN` message to it. Thereby, interrupts are handled in a LIFO manner and TOS tasks in FIFO manner as it is in Tiny OS.
- the `blocked` structure. It contains the references to tasks that wait for a mes-

sage external to the scheduler: tasks that were refused a resource and wait for a `RELEASED` message from the allocator and the tasks that self-preempted and wait for the self-resumption. When one of these messages are received, the data structure is looked up, and the relevant task released. As self-resumption represents interrupts, the task extracted from the `blocked` data structure is scheduled immediately, thus preempting the currently running task. Else, the released task is added to the preempted list.

This handling of the messages from the tasks and from the allocator is done by the `doSchedule` function.

Hardware Platform

We consider a platform composed of a processor, a radio transceiver and an event-detecting sensor. In this section, we present examples of how the power characteristics of the components are represented, and how the behavior of the components is modeled.

Power State	Cycle Energy	To Off		To Idle		To Active	
		Latency	Power	Latency	Power	Latency	Power
Off	0.0	0	0.0	80000	150.0	80000	350.0
Idle	250.0	50	150.0	0	250.0	130	350.0
Active	550.0	50	250.0	90	350.0	0	550.0

Table 5.1: Energy States of the Implemented Processor Model (Cycle Energies is given in $mA \cdot ns$ and latencies are given in clock cycles)

Power State	Cycle Energy	To Off		To StandBy		To Receive		To Transmit	
		Latency	CE	Latency	CE	Latency	CE	Latency	CE
Off	0.0	0	0.0	150000	0.07	150000	100.0	150000	700.0
StandBy	0.07	100	0.07	0	0.07	160	100.0	160	700.0
Receive	180.0	100	100.0	100	100.0	0	180.0	120	750.0
Transmit	1200.0	100	200.0	100	200.0	160	800.0	160	1200.0

Table 5.2: Energy States of the Implemented Radio Transceiver(Cycle Energies is given in $mA \cdot ns$ and latencies are given in clock cycles)

The **processor module** implemented, *processor*, is a sub-module of the *HW_component* class. It defines the power states of the processor and implements the abstract function of its parent class. The processor represented has three states: off, idle and active. These states are characterized by their cycle energy (CE), i.e. the energy consumed in one cycle, and the cycle energy and the latency of the transitions between two states (cycle energies are given in $mA \cdot \mu s$ and latencies in number of clock cycles of the absolute time clock - the values given in the table are for a clock period of 100 ns). These values can generally be found in the data sheets of the devices used. In our example, the power characteristics of the processor are described in table 5.1.

In addition to defining the power states, the processor module must also define the

abstract functions of the *HW_component* module class. The request received by the processor are all of the same type and request to change to a new power state. These requests are handled by the `hardware_request` function. On reception of a request, this function changes its cycle energy to the value of the transition to the new state and informs the battery if the cycle energy has changed. It also resets a counter and stores the new state value. The `delay_model` functions decreases the counter until it reaches 0. When it does so, The state of the processor is changed to the new state, the cycle energy is updated correspondingly, and the battery informed of changes in the cycle energy. It is to be noted that if a request is received while the previous request has not completed yet, the previous request is cancelled and the second is handled.

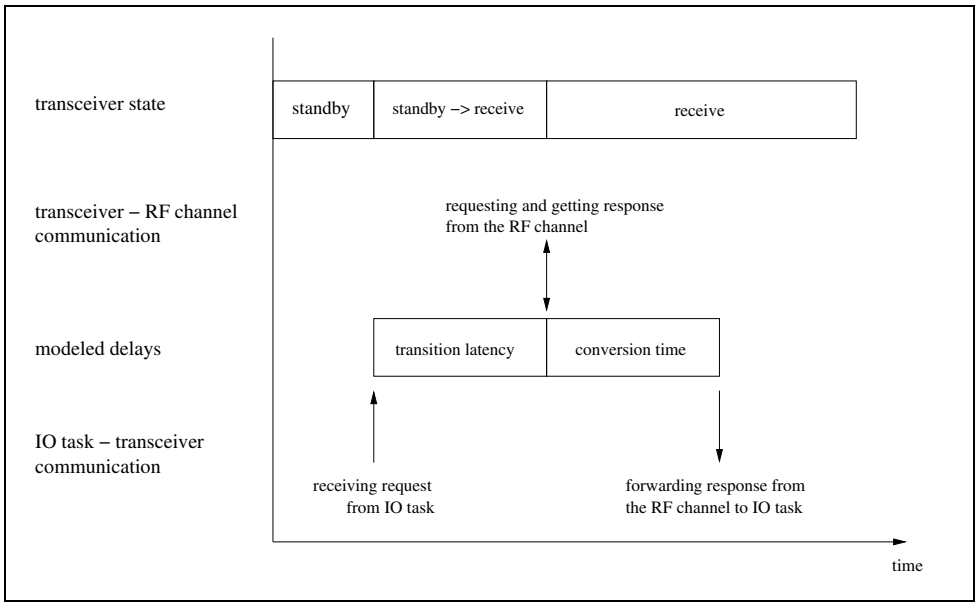

Figure 5.10: Input/Output Model

The **transceiver module** implemented, *RF_transceiver*, is a sub-module of the *IO_device* class. It defines its power states similarly to the processor module. The power states were taken from the data sheet of the TR1000 [18] transceiver used by the MICA node. The transceiver has four different states: off, standby, receive and send. The characteristics of the different states are shown in table 5.2. As the processor, the transceiver modules must handle the power management requests. In addition to that, the transceiver may receive requests for IO operation such as receive or transmit. For such requests, the transceiver module must not only model the power state transition. It must also forward the request to the environment module connected (RF channel), and if a response is received from the RF channel, it must delay the response for modeling the conversion time (including demodulation and AD conversion). The abstract classes are used to model this. Let us look at the example of figure 5.10 an example: the transceiver is in standby state and receives a listen to the RF channel request (which requires the transceiver to be in receive state). When the request is received, the transceiver must first change to the receive state. Once in the receive state, it requests the environment model and gets a response immediately. The transceiver then delays this response for

the time that represents the conversion latency and then forwards the response of the RF channel.
The behavior of the transceiver can be described as the state machine of figure 5.11 (in the figure, PTL refers to the Power Transition Latency and CT refers to the Conversion Time). In terms of implementation, in addition to `hardware_request` and `delay_model`, the transceiver module also implements the `environment_response` function. The transceiver module is not implemented to support multiple concurrent requests, and the IO tasks connected to it must ensure that it waits for the completion of a request before sending the next one.

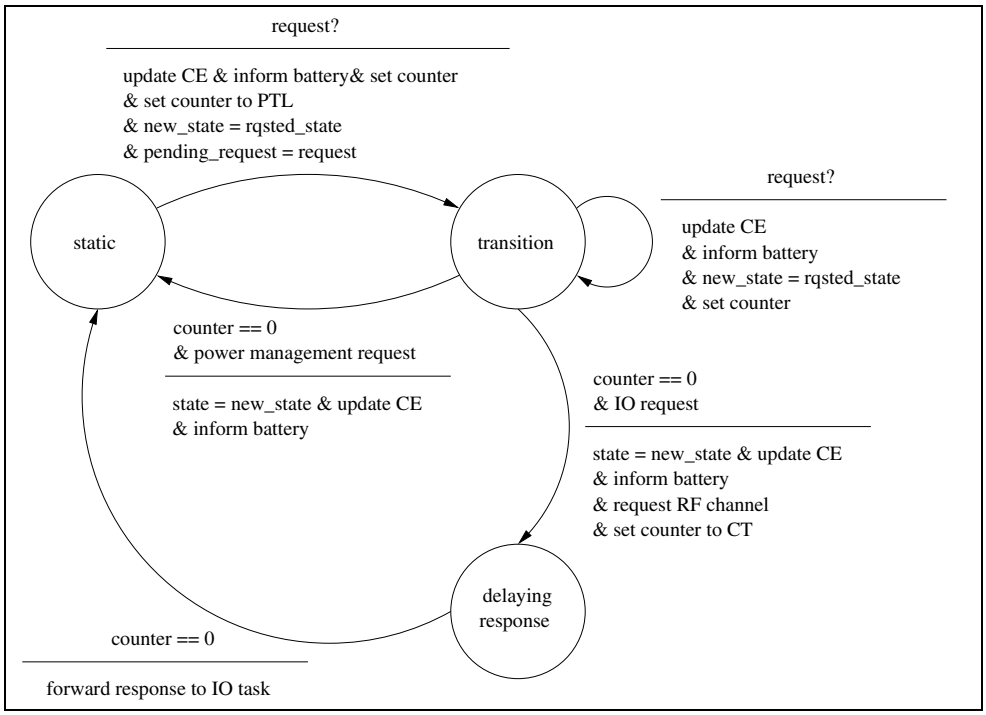

Figure 5.11: State Machine of the Transceiver Implementation

The **event-detecting sensor module**, *TP_sensor*, is a very simple module which purpose is to illustrate that such sensors can easily be implemented in the framework. It has a single power state and ignores requests from the IO tasks that it is attached to. The module is only receiving event messages from the environment model. Similarly to the transceiver, it delays these messages to model the conversion time of the sensor and then forwards the event message to the IO task.

Environment Models

Correspondingly to the IO devices, two environment models are needed: the RF channel and a model of the phenomenon monitored by the event-detecting sensor.

The ***RF_channel*** module represents a ideal radio channel: no propagation delay is modeled, and the transmission occur a null bit error rate. The module maintains a list

of the transmissions in process, the position of the transmitting nodes, its transmission range, and the type of the transmitted bit. Bits can be either preamble bits, or data bits. The *RF_channel* receives requests from the send and receive task through the transceiver modules of the nodes. These request can be of four types:

- `RF_LISTEN` requests the value of the RF signal at a position specified in the request (position of the requesting node). This request causes the *RF_channel* to look through the list of transmissions. If the listening node is not within the range of any transmission, a `CLEAR` message is sent back to the requesting transceiver. If the node is exactly one transmission of the list is within the range of exactly one transmission, the type of bit transmitted is communicated to the requesting transceiver. Finally, if the node is in the range of two or more transmissions, an `INTERFERENCE` message is sent to the transceiver.
- `RF_PREAMBLE` indicates that the transceiver is starting sending a preamble. The *RF_channel* adds this new transmission to its list.
- `RF_DATA` indicates that a transceiver previously sending a preamble is now starting sending the packet. The *RF_channel* will change the status of the transmission from the requesting node from preamble to data.
- `RELEASE` indicates that a transceiver stops sending. The *RF_channel* therefore removes the transmission of this transceiver from its list.

The value of the actual packet is communicated to the channel already at the start of the preamble transmission. The packet is passed to a listening transceiver in the response messages.
This module models some aspects of the communication: for example, if a node misses the preamble of a packet, it is not going to receive the packet. However, the model does not represent bit error: either the whole packet is received without corruption, or it is not received at all. It also abstract from the details of how the preamble is detected or the data coded.

The ***TP_channel*** module represent the phenomenon monitored by the *TP_sensor* module. This Environment model is defined by three parameters: the range of values that it can take, the event threshold, and the inter-event period. This environment model is connected to the global clock. On each event of the clock, it generates a random number within the specified range. If the value is above the even threshold, the module randomly generates a node number and sends an event message with the value phenomenon value generated to it. After an event has been generated, the module waits for the inter-event period before restarting generating values.

The Battery Model

The battery is modeled by the *linearBattery* module which is a sub-module of the *battery* module class. For this example, we adopted the linear model because of its simplicity. The implementation module is very simple to generate. Only the `updateCharge` abstract function of the *battery* class must be implemented. This function simply decreases the battery `charge` member variable by `discharge_rate` energy units. When the battery becomes empty, it sends a message on its master port. More advanced battery modules could use the non-linear models [17][4]. As each implementation module created is a

class, there is no restriction to the complexity of the function updating the charge, and to the amount of parameters needed. For example, an implementation module could have a list in which all the history of the battery discharge could be stored and thus allow to use the model proposed by Rakhmatov et al. [4].

The Mobility Model

In [2], a number of mobility models are presented. They are classified in two classes: entity mobility models, where the nodes move independently of each other and group mobility models where the movements of the nodes are connected. The entity mobility model implemented by the *mob_RandWayPoint* module is a 2-dimensional random waypoint model. The model initially selects a random destination within the deployment area and a random speed and moves in the direction of the destination until it reaches it. When the destination is reached, the node pauses at this position for a random time, before selecting a new destination and speed. In this model the deployment area is rectangular and is defined by two of its corners. Another two parameters define the range of speeds.

5.2.2 Examples

In this section, examples of how the modules presented can be used to model sensor networks and simulate simple applications are provided. For all the examples, the parameters defining the communication protocol such as the length of the packets in bits, the number of preamble bits are set to low values in order to make it easier to read the waveforms. Setting these parameters to real values is done by changing the preprocessor constants of the file *command.h* and recompiling the send and receive task modules.
The simulation results obtained are the waveforms generated SystemC. In the waveforms, the values presented are integers for the state of the tasks. The table 5.3 presents the meaning of the different values. The Execution Task column refers to the state machine of figure 4.4 while columns 2 and 3 respectively refers to the state machines of figures 5.6 and 5.8. Also note that the time values indicated on top of the simulation waveform must be multiplied by 100. Then, the communication modelled are happening at the rate of 50 kbps, and the latencies specified for the HW components match the values from the data sheets.

value	Execution State	Send Protocol State	Receive Protocol State
0	Inactive	Idle	Idle
1	Ready	Carrier Sensing	Polling
2	Running	Back-Off	Synchronize
3	Preempted	Transmit Preamble	Receive Data
4	Self-Preempted	Transmit Data	*unused*
5	Power-Out	*unused*	*unused*

Table 5.3: Correspondence Table for Reading the Waveforms

The two examples illustrate most of the concepts presented in the description of the

implementation. The first example is very simple but illustrates many aspects of the framework:

- the communication mechanism between nodes and the radio channel model,
- the execution patterns of the send and receive tasks,
- the allocation of resources and the mutual exclusion of send and receive tasks,
- the concurrency of the IO operation and the other tasks of the system,
- the battery model.

In the second example, the aspects illustrated are:

- the representation of simple applications,
- the use of sporadic tasks and the representation of routing,
- the overhearing of messages.

Example 1: Link Layer Communication

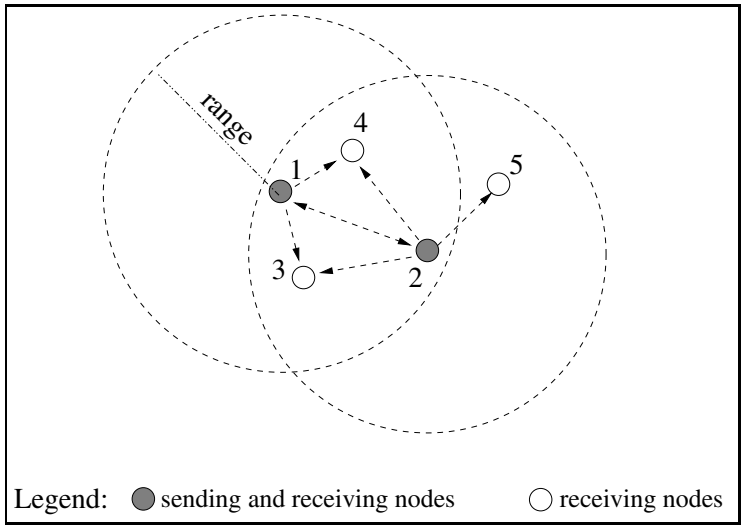

Figure 5.12: Example 1

The First examples illustrates how the model represents the basic mechanism for sending and receiving data from the RF channel. The sensor network simulated here is composed of five fixed nodes deployed as in figure 5.12: all the nodes are in each other's communication range except node 5 that is only in the range of node 2. All the nodes have a instance of the *rcvTask* module. Additionally, nodes 1 and 2 have a *sendTask* instance, and node 1 also has a periodic task. These tasks transmit a packet at the beginning of the simulation. The mobility model is not used, and the *TP_channel* is not connected to the nodes.

The waveform output of the simulation presented in figure 5.13 shows the execution patterns of the tasks as well as the protocol states for the send and receive task. In the waveform, the task signals represent the execution state of the task, while the protocol signal show the protocol of the corresponding send or receive task. The simulation run shows that node 1 and 2 first try to transmit the message. Because the back-off time of

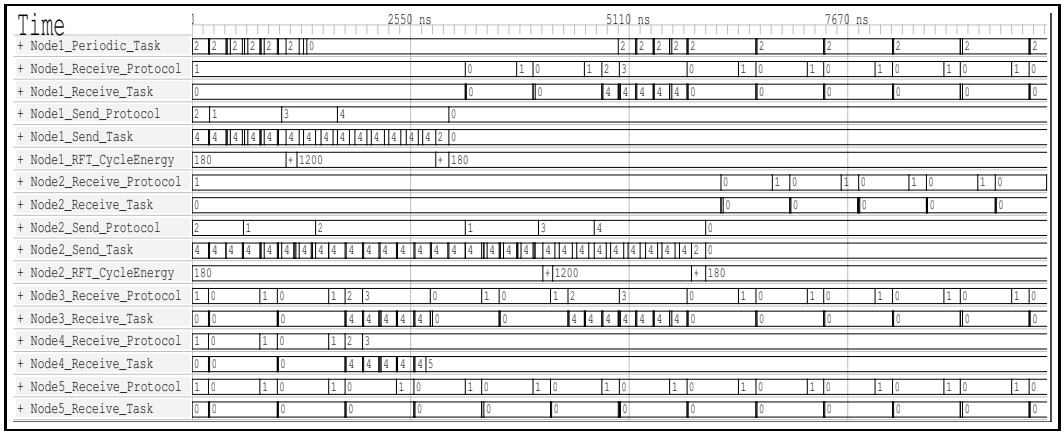

Figure 5.13: Waveform for Example 1

the send task of node 1 was smaller than the one of node 2, node 1 starts sending first. As node 2 sees that node 1 is transmitting, node 2 backs off. It is to be noted that the *rcvTask*s of node 1 and 2 stay inactive because the respective *sendTask*s have reserved the radio. Also, for node 1, the periodic task illustrates the fact that the sending of the packet does not occupy all the CPU time: in the time spent on waiting for an IO interrupt, the periodic task is scheduled. For nodes 3, 4 and 5, it is observed that while the *rcvTask* of node 5 polls the radio channel at a low frequency, nodes 3 and 4 see the preamble, and their *rcvTask*s start reading the channel at the bit rate. At around 3000 ns after simulation start, the sending of the packet by node 1 has completed and the *sendTask* releases the radio transceiver goes to the inactive state. As the transceiver is now available, the *rcvTask* of node 1 starts polling the channel at a slow rate. Node 4 was initiated to have less battery than the other nodes in order to illustrates the death of a node. Indeed, just before the end of the reception of the packet of node 1, node 4 dies its receive task goes in state 5: power-out. After two attempts, the *sendTask* of node 2 eventually see a free channel and start sending. The packet is seen by all the other task, because they all are in its communication range.
In addition to the task and protocol states at the nodes, the waveform also displays the cycle energy of their transceiver (we only presented them for nodes 1 and 2. In the other nodes, the transceiver is only used in receive mode, and its cycle energy is constant.

In order to better see the execution patterns, a zoom of the previous waveforms is presented in figure 5.14. This detailed view shows of when the *sendTask* of node 2 transitions from the carrier sensing state to the preamble transmit state. As seen, the execution patterns for a bit period change. Additionally, the change of state of the transceiver can be seen on the cycle energy change. The transition from the receive to the send state is visible from the changes in the cycle energy: first it changes from 180 to 750 which is the cycle energy of the transition from receive to transmit state for the transceiver, and after the delay of the transition (160 clock cycles corresponding to 16 μs).
On this waveform it can also be seen that after the last carrier sensing operation, the *sendTask* starts the transmission of the first bit of the preamble immediately, thus cutting

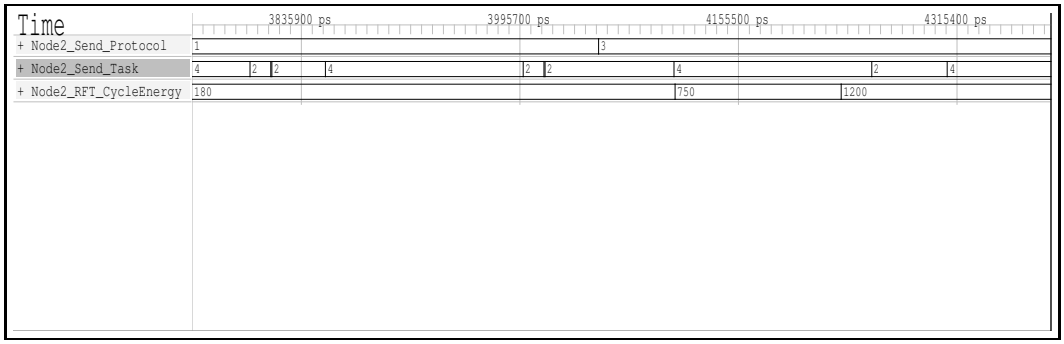

Figure 5.14: Example 1 - Detailed View of the Execution Pattern and Energy Cycle of the Transceiver

the end of the carrier-sense pattern. This detail was not mentioned in the implementation part above to simplify the understanding of the execution patterns of the sendTask. It is however an important aspect for the efficiency of the carrier sensing: after having detected that the channel is available, it is important to start sending as soon as possible to minimize the probability that another node within the transmission range also sees that the channel is free and also starts transmitting.
In this example, the behavior of the network has been looked into at a detailed level. This example validates the representation of the sensor nodes. In the next example, we look at the network at a higher level of abstraction, and present how application can be represented.

Example 2: Simple Ad-Hoc Multi-Hop Routing

In the second example, we consider how routing can be represented. In example 1, we saw that the receive task will be triggered by all the packets they hear independently of their value. The purpose of the routing is to filter out the packets that were not addressed to the considered node. This can be represented using sporadic task. Indeed, as mentioned earlier, *rcvTask*s generate activation messages at the end of a packet reception. These activation message contain the value of the message. A sporadic task connected to the receive task (i.e. the `activatingTask` parameter of the *sporTask* module is set to the ID number of the *rcvTask* of the node) can represent the routing.

In this example, a multi-hop routing is presented. All the data packets are sent up a routing tree to a root node. The algorithm is simple: when a node receives a data packet, it first checks whether the message is addressed to it. If it is, the packet is forwarded by the node to its parent node in the tree.
As indicated by the title the example presents an ad-hoc routing algorithm: i.e. the routing structure is not static. Instead, it is constructed and maintained during the lifetime of the network using network discovery. The discovery mechanism is very simple: the root node of the network periodically sends beacons using flooding. A node receiving a beacon chooses its source as parent and forwards the beacon.
Within each node, three tasks are involved in this algorithm: the *rcvTask* of the node, the routing task (a *sporTask*) and the forwarding task (a *sporSendTask*). The activation

message generated by the *sendTask* at the end of the reception of a packet is received by the routing task which is thereby activated (i.e., the `start_enable` method of the functionality of the routing task returns true independently of the value of the packet). The `execute` method checks whether the packet is relevant. If it is a new beacon packet, `execute` updates its parent ID and returns a copy of the beacon where the source of the message is changed to be the node number. If it is a data packet addressed to the considered node, `execute` returns a copy of the data packet where the destination was changed to the ID of the parent of the considered node. If the packet received does not satisfy either of the previous criteria, `execute` returns a null pointer. The routing task then sends an activation message carrying the return value of `execute`. This activation message is caught by the forwarding task (i.e. its `activatingTask` parameter is set to the ID of the routing task). The `start_enable` method of its functionality returns true if the packet carried by the message is non null, thereby causing this packet to be transmitted.
The data messages are generated on events of the *TP_ channel*: When an event is generated by the *TP_ channel*, it is sent to the *tpioTask* of one node through its *TP_ sensor*. this task generates and sends an activation message carrying the event value. This message is caught by a sporadic task called *nodeSseSporTask*. This sporadic task is identical to the routing task, and it is attached the same functionality. The *sendSensed* is a *sporSendTask* that catches messages from the *nodeSseSporTask* and transmits the packet attached to them.

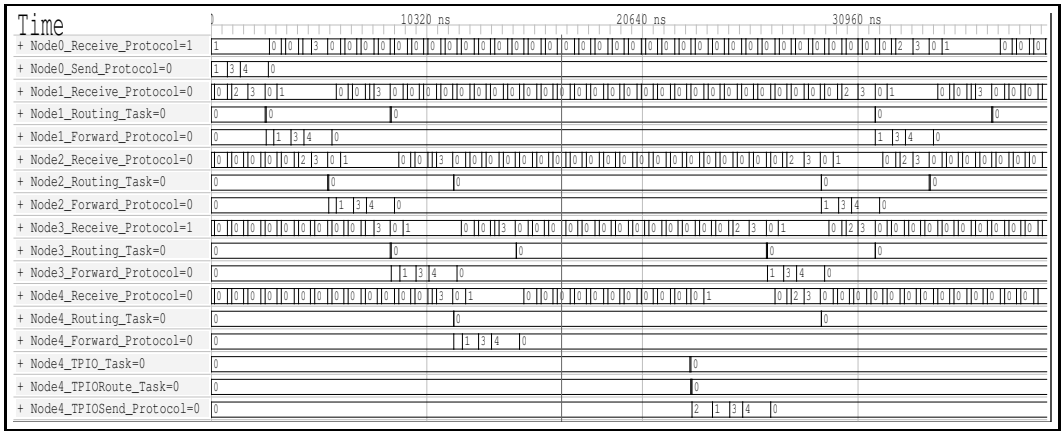

Figure 5.15: Waveform for Example 2

This application was tested on a simple chain topology composed of 5 nodes. Node 0 is defined as the root of the tree. It differs from the other nodes by the fact that it has a periodic *sendTask* sending the beacons. Each node of the chain can here its two closest neighbors. The simulation of this example produced the waveform of figure 5.15. In this waveforms, the details of the execution are abstracted, and only the protocol states of the radio tasks are displayed. In this simulation, one event was generated by the *TP_ channel* at node 4. Therefore, the tasks for handling the event are presented for this node. The other nodes do not see events from the sensor. The first part of the waveform represents the propagation of a network discovery beacon. Around simulation

time 23000ns, the event is generated on node 4, and what follows is the multi-hop routing of the event to the root node.
The overhearing problem can be seen in this example. It corresponds to when the *rcvTask* of a node receives a packet but the router rejects it. Figure 5.16 presents a detail of figure 5.15. It shows the routing of the packet generated by node 4 on the environment event. The packet has reached node 3, which forwards it to node 2. The packet sent is heard by node 2, but also by node 4 that is within the communication range of node 3. This causes useless consumption of CPU time and transceiver time as well as the corresponding energy by node 4.

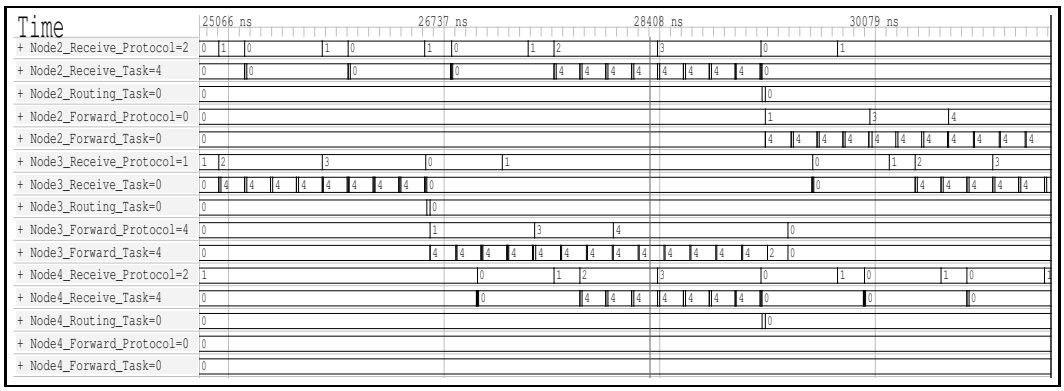

Figure 5.16: Example 2 - Detailed View of the Overhearing Phenomenon

5.3 Coverage of the Design Space of Sensor Networks

Above, examples of how the framework can be used to represent sensor network application were presented. It is believed that the framework allows to represent most possible sensor networks configurations presented in chapter 2.
In terms of hardware platforms, the common platforms as the Mica node are the one used in the examples. However, more advanced platform can be covered. For example multi-processor platforms (for example with a central processor and an accelerator) can be naturally represented using the model proposed by SoC/NoC model which the sensor network framework is based on. Representing packet level transceiver such as the nRF2401 transceiver is done similarly: indeed, such transceiver have processing abilities for supporting parts of the communication protocol and are accelerator that assist the central processor for communication.
The operating systems are represented by their scheduling algorithm and there is no limit to what algorithm can be modeled by implementing new scheduler sub-modules. The common scheduling algorithms such as EDF or RM can be represented. The framework also supports DVS techniques thanks to the clock generator module that controls the rate of the task executions.
The communication protocols are modeled at two levels. IO tasks represent the data link layer while the network layer (mainly routing) can be represented by sporadic tasks as illustrated above. Concerning the data link layer, the send and the receive task modules implemented in this project are very simple. More advanced techniques such as virtual

carrier sensing (RTS/CTS protocols) and Automatic Repeat reQuest (ARQ) can be represented too. Indeed, these protocols can be represented by state machines similar to the ones in figure 5.6 and 5.8 and the send and receive task modules can be modified accordingly. Concerning the routing, the functionality of the sporadic task gives great freedom in the algorithm that are to be represented. In the examples, a routing algorithm where the network topology is discovered and maintained was presented. More advanced protocols can be represented by simply changing the `execute` function of the sporadic task for routing.
Power management can be represented by using IO tasks. These tasks are connected to the different hardware components from which they can receive information (e.g. usage of the resource, average inter-request times, etc.), and to which they can send commands (e.g. go to sleep mode).
The structure of the framework therefore seems to be able to represent very different sensor network designs. The limits of this simulation tool are not so much in the variety of system covered, but more in the simulation performance.

Chapter 6

Performance of Sensor Network Simulation.

The framework developped in this project is aimed at being a design tool allowing a sensor network designer to evaluate various possible implementations and to compare their consequences mainly on the power consumption of the nodes and on the lifetime of the network using simulation of the network. As described in section 2.3, the design space of sensor networks is large, and the number of simulation that have to be run is high. It is therefore critical that the simulation can be run in reasonable time. In this section, we take a closer look at the performance of the simulation of a sensor network using the framework and consider some improvements to it. First we present measurements of the performance of the framework based on the sensor network model implementation presented in this report. Then, the performance of the systemC constructs used by the framework is measured.

6.1 Measuring Performance

The measurements presented bellow are all run on the same platform: a Personal Computer equiped with a Intel Pentium 4 processor running the Mandrake LINUX operating system. The simulation runtime are measured using the *times()* function. Calling the function before and after a piece of code gives access to the CPU time used on behalf of the calling process for the considered code (instruction execution time and system time). These data are given in clock ticks and are converted into seconds. The resolution of the results is 1e-2 seconds. the *times* function also gives the time spent on executing child processes of the calling process. However, SystemC SC_METHODS are run in a single process and the child process time information is therefore null.
For the systemC measurements, the time measured is the actual simulation time and it does not include the simulation setup: instanciation and interconnection of the modules. Therefore, the two calls of times are placed right before and right after the *sc_start* call.

6.2 Reproductability of Time Measurement

Before actually measuring the performance of the sensor network model, some tests were performed to evaluate the confidence that can be given to the outcome of the *times* function. To do so, the runtime of a fixed simulation is measured thirty times. The simulation run has an average runtime of 21.54 seconds. For this simulation, the standard deviation is 0.84 seconds which corresponds to 4 % of the average. the maximum deviation from the average seen in the measurements is 1.45 (7 %). Another set of measurements was made with a longer process with an average runtime of 432.15 seconds. The runtimes of thirty execution of this process have a standard deviation of 0.50. The maximum deviation for this second set of measurements is 0.82.
The variation of the measured runtime is due to the fact that a process does not have a fixed pattern of execution when running on top of an operating system. However, the measurements show that the deviation is not too important and that the reproductability of the runtime measurement increases with the duration of the considered process. The measurements above were conducted with only the measured process running. Some tests were run with two processes running simultaneously. Even if the *times* function measures the time spent on executing the calling process, it is observed that the measurements give larger execution times and that the standard deviation of the measurements is slightly larger to. For example, the process running in 21.54 seconds in average runs with an average of 22 seconds and with a standard deviation of 0.89 seconds. This difference is probably due to side effects of context switching (e.g. cache misses). In order to be able to compare them, we therefore measure the execution time of a process when it is running alone on the operating system.

6.3 Sensor Network Simulation Performance

The example of sensor network simulation chosen for the measurements is composed of a variable number of nodes. The nodes are organized in a routing chain. for each ten nodes, one is sending a packet at the start of the simulation. The packets are received and forwarded along the chain in a multihop manner until the root is reached. This network configuration is illustrated in figure 6.1. This example is very simple, but exercises the interdependence of separate nodes through the radio channel and give thereby a representative picture of the simulation runtime of real simulations. Experiments to estimate the influence of the number of nodes and of the simulated time are conducted. The simulation performance depends on the implementation of the model, of the number of modules in the nodes, etc. However, measurements on this simple example give an order of size of the simulation runtime, and of how it changes with the number of sensor nodes in the network and with the simulated time.

6.3.1 Simulation Runtime Versus Simulated Time

The first set of measurements is done for constant number of sensor nodes and with varying simulated times. Tests were ran with 100 and 1000 nodes and with simulated times varying from 10 time units to 1000000 time units. As expected, the simulation runtime

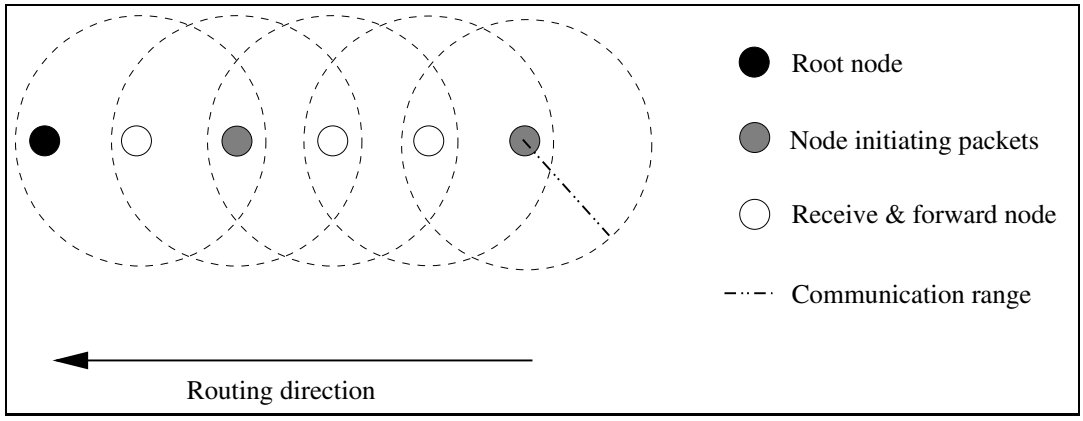

Figure 6.1: Network Configuration for Performance Measurements

increases linearly with the simulated time. Linear regression of these measurements gives a correlation factor larger than 0.9999. The measured simulation runtime are presented in figure 6.2. This property makes it possible to evaluate the time it will take to perform a simulation with a long simulated time based on shorter simulation. For example, in the measurement presented, estimating the simulation time for 1000 nodes and 1000000 time units based on the simulation runtime for 1000 nodes and 10000 time units gives a quite accurate estimation of 5432 seconds against 5346 actually measured.

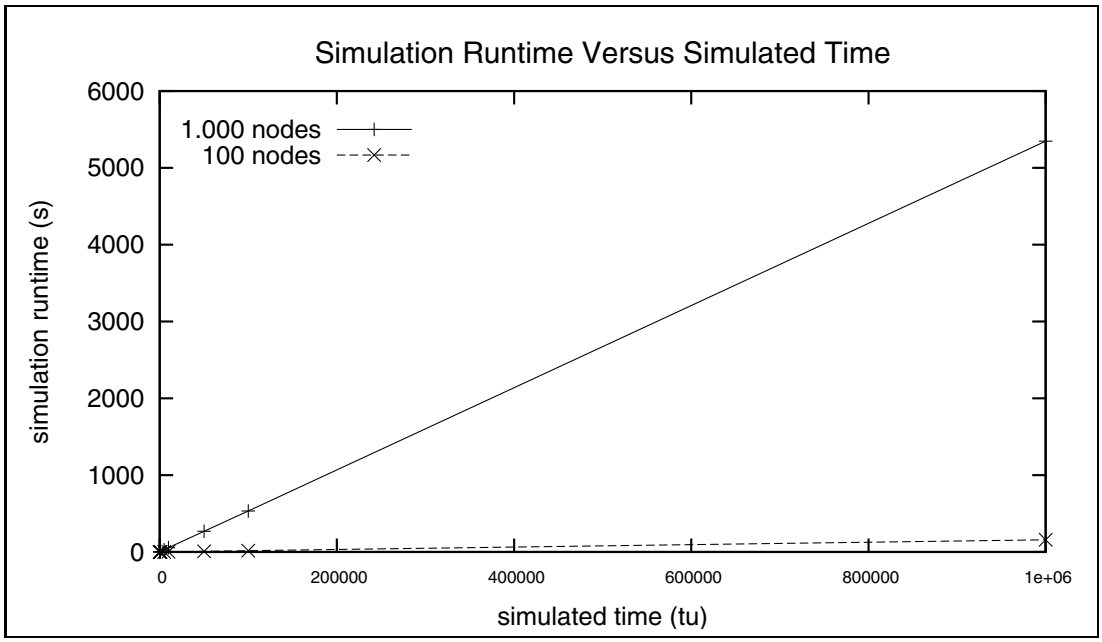

Figure 6.2: Runtime for Simulating a Sensor Network for Variable Simulated Times

6.3.2 Simulation Runtime Versus Node Count

The second set of measurements is done by having a variable number of nodes in the network with a constant simulation time of 1000000 time units. The two lowest nodes in the network are set to transmitting a packet at the start of the simulation. Typical sensor networks consist of tens to thousands of nodes. It is therefore important that the model supports this range. The results of the measurements are shown in figure 6.3.

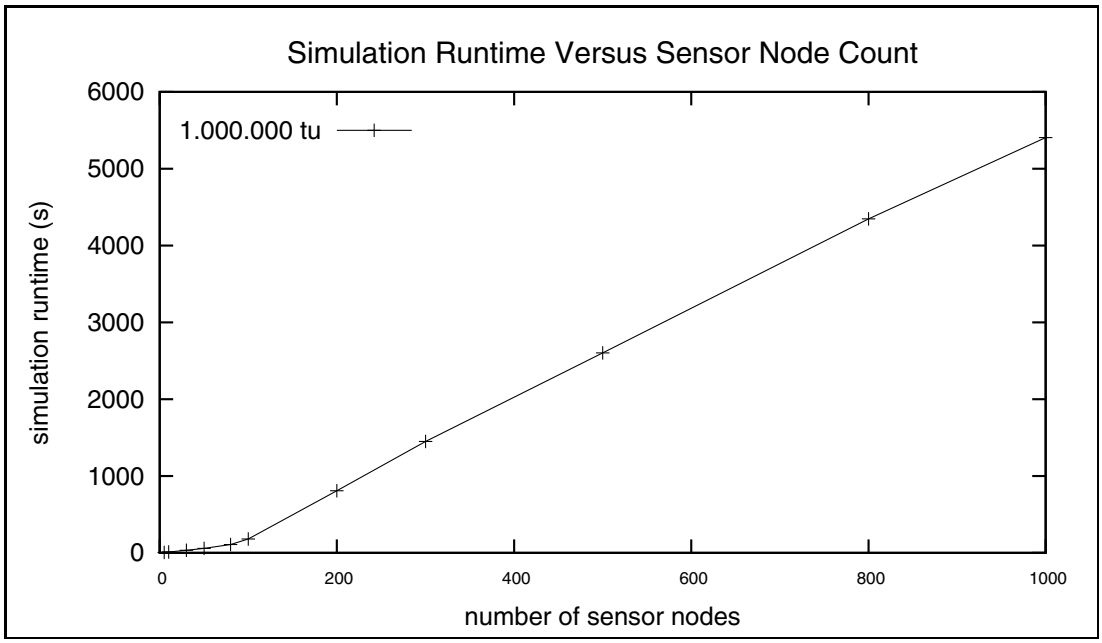

Figure 6.3: Simulation Runtime of a Sensor Network Composed of Variable Number of Nodes

The simulation runtime of the sensor network are in the order of tens of minutes for networks composed of around 100 sensor nodes, and it increases to one and a half our for 10000 sensor nodes. In order to analyze in more details the results of this set of experiments, we have to recall the mechanism used to model the radio communication. The communication is poll-based. Therefore, the action of a node transmitting a packet does not directly trigger activity at the receiving nodes. This makes the nodes activity to be rather independent of each other and we expect the performance to be roughly linear. However, as it can be observed from the figure, for node counts from 0 to 100, the simulation runtime increases exponentially. However, for larger node counts, the slope of the measurement diminishes and tends to be more linear. This shape of the relationship between the node count and the simulation runtime is not caused by the implementation of the sensor network model but by the systemC simulation engine (cf. section 6.4). It is to be noted that a limit to how many nodes may be simulated was observed. Indeed, for the test with 10000 sensor nodes, the simulation is aborted and a "killed" message is sent to standard output. This may be due to the fact that the data structures used in the simulation engine of systemC can only support a limited number of modules. In order to better understand the effect of the systemC library on the observed performance results,

some measurements of the systemC constructs used in the framework are studied.

6.4 SystemC Performance

SystemC constitutes a nice simulation platform that allows to simulate any kind of systems at various level of abstractions, from the gate level to system level. The fact that it covers such an important area have a cost in terms of performance, and when the performance of simulation becomes critical, it may be relevant to develop a more specific simulator optimized for a given type of systems. In this section measurements of the cost in performance of the systemC construct used in the model are conducted, and their result compared to the sensor network simulation presented above. This is done to evaluate which part of the simulation runtime is due to the systemC simulation engine and which part is due to the implementation of the user code. the two fundamental mechanisms of systemC used in the framework are evaluated: the signal sensitive systemC methods and the slave systemC methods.

6.4.1 Performance of Signal-Sensitive Methods

The signal-sensitive methods is the basic construct of systemC for inter-module communication. A method can be made sensitive to a signal when it is created. The function attached to the method will thereby be called every time the signal changes. This mechanism is used in the Sensor Network Framework to synchronize modules together using a global clock. The modules that are clocked in the framework are the task models, the hardware component models, the battery models, and possibly the environment models (For the performance measurements done above, the *RF_transceiver* is not clocked). What is to be evaluated here is the cost of activating the methods. Therefore, we measure the performance of a simulation where a clock is connected to a variable number of modules which contain a single method that is sensitive on this global clock. The function attached to the method is empty.
Measurements are conducted with the number of clocked modules varying from 1 to 20000 (An attempt to simulate 30000 nodes was also done, but the simulation was killed). The results of these measurements are presented in figure 6.4. As can be seen, The simulation runtime first increases in an exponential manner when the number of clocked modules increases from 1 to 3000, but it then grows linearly from 3000 to 10000 clocked modules. It even seems that the slope of the measurements has a tendency to decrease for high clocked module counts. The simulation runtime varies from 1 second for 10 clocked modules to 12 minutes for 3000 nodes and 1 hour and 13 minutes for 10000 modules.

6.4.2 Treeing the Clock

The non-linearity of the simulation runtime when the number of clocked module increases is likely to be caused by the data structures used by systemC to implement the sensitivity lists of modules. In this case, one possible solution to improve the performance of the simulation is to divide the clock signal into multiple copies of it. Thereby, the number of signals is increased, but each signal has a shorter list of sensible modules. To evaluate

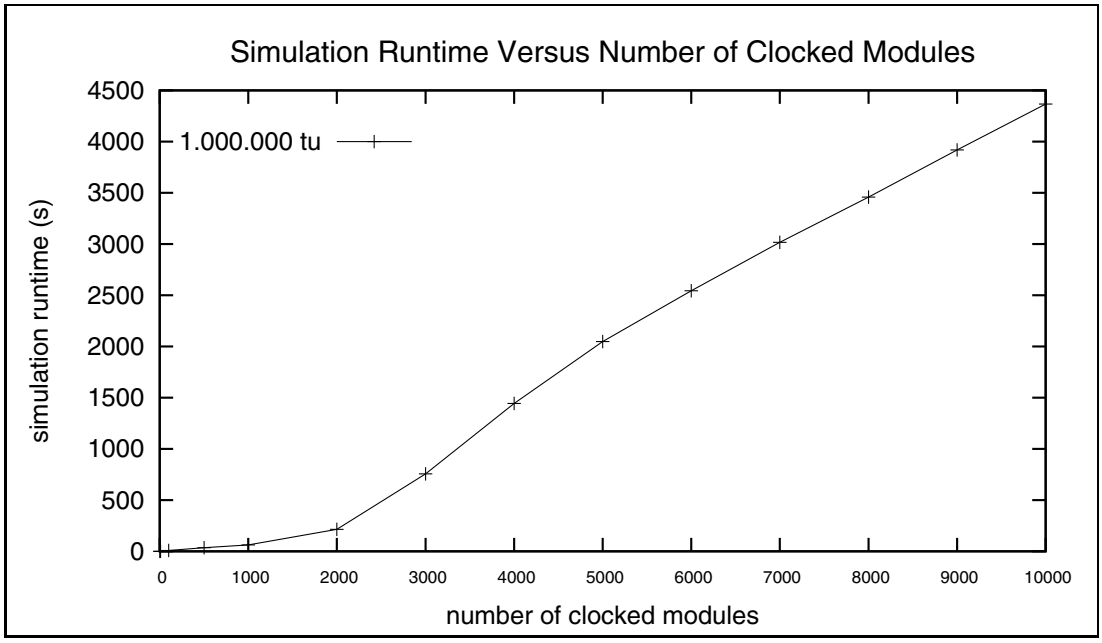

Figure 6.4: Simulation Runtime of Variable Number of SystemC Modules Activated by a Global Clock

the gain that could be obtained by such a technique, the same empty modules as before are connected to a clock tree instead of a common global clock. The clock is copied by branch modules which take a clock signal as input and generate copies of the signal at their output. The number of copies that a single branch module produces (branch fan-out) is a parameter, and an algorithm automatically generates the tree given the number of modules that must be given a clock signal and the fade-out of the branches. For example, figure 6.5 illustrates how a common clock signal (figure 6.5.a) can be replaced by a tree with a maximum branch fade-out of 2 (figure 6.5.b). Simulation are run with different branch fade-out values and the runtime is measured and presented in figure 6.6.
While having a deep tree is significantly increasing the simulation runtime due (the runtime for a branch fade-out of 2 is 2257 seconds against 430 for the common clock implementation), it appears that it is possible to slighthly improve the performance of the simulation. However, the gain obtained (6.7% with a branch fade-out of 2000) is too low compared to the cost of using a clock tree in terms of the complexity of the framework.

6.4.3 Performance of Slave Methods

The second, and the main, inter-module communication mechanism used in the Sensor Network Framework is the master-slave communication provided by the systemC master-slave add-on library. This mechanism is similar to Remote Procedure Call: an action of a master (writing or reading) on the master slave link activates the attached function on the slave modules connected to that link. Similarly to the clocked modules, The runtime used by this mechanism is measured by simulating a variable number of slave modules

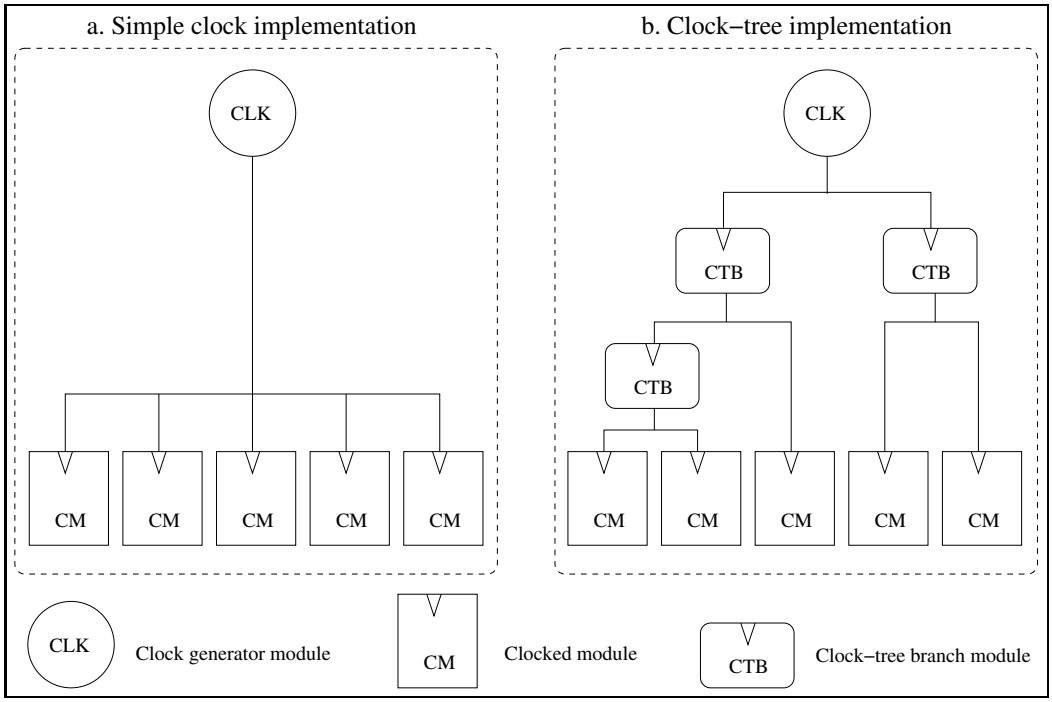

Figure 6.5: Alternative SystemC Implementation of Module Clocking

all connected to a single master-slave link. To activate the slaves, a master must perform an action (in the measurement a write action) on the master-slave link. This master is a clocked module which writes on the master-slave link on every clock event. Here again, the function attached to the slave methods are empty so only the cost of the master-slave call is measured.
The results of these simulations are presented in figure 6.7. The shape of the curve is similar to the one of the signal-sensitive modules: there are two regimes: up to 2000 slaves, the simulation runtime increases in an exponential manner and above 2000 slaves, the slope decreases.
It can also be noticed that the master-slave mechanism is actually more efficient than the signal-sensitivity mechanism: for example, 100 modules sensible on a common clock take 7.34 seconds to simulate, while 100 slave modules connected to a master module (itself clocked and activating the slaves on each clock event) take 4.66 seconds (i.e. master-slave methods are 37 % faster than signal sensitive methods) . This difference increases with the number of modules and reaches 63 % for 10.000 nodes. However, the functionality of both tests is identical: a number of methods are called on every clock event. Thus, replacing the clock signal with a "master-slave clock" could improve the performance of the simulation.

6.4.4 Master-Slave: Broadcast Versus Unicast

In the Sensor Network Simulation Framework, the master-slave communication mechanism is used extensively because it gives a nice and clean structure to the sensor network model. As an example, a single master-slave link is used to connect the scheduler to the

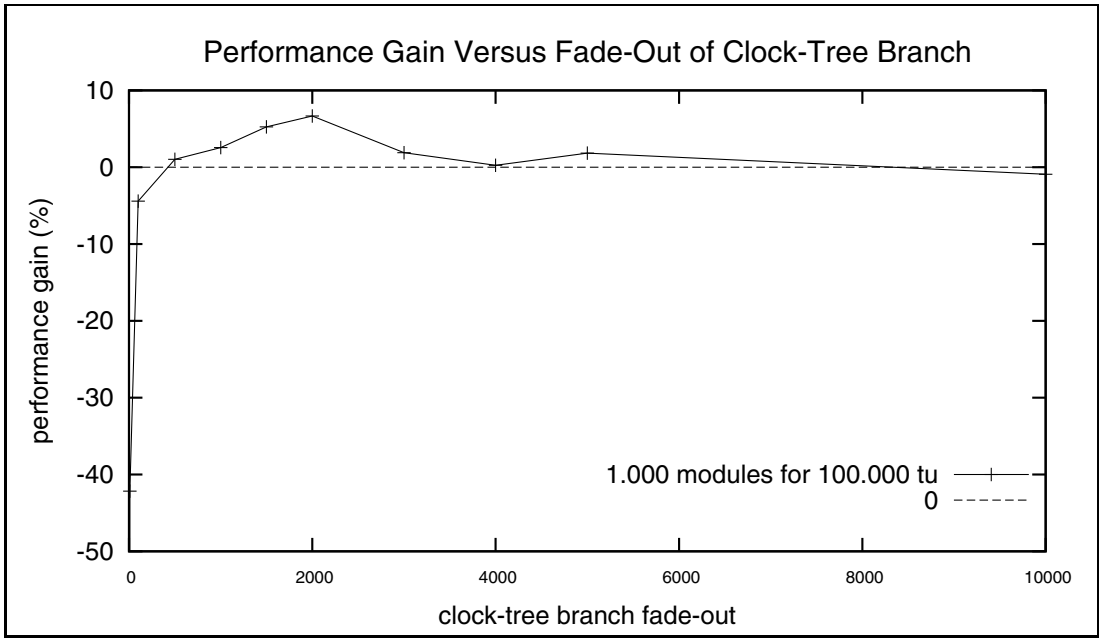

Figure 6.6: Performance Gain Obtained by Replacing the Global Clock by a Clock-Tree

task modules. This makes it possible to add and remove task modules with minimum changes to the simulation file. Having a single link implies that the messages from the scheduler are broadcast to all the task modules. However, in most cases, the communication from the scheduler to the tasks is unicast. Implementing this unicast is done by having all the tasks' slave methods compare the destination of the message with their ID and return if the destination does not match. Thus, most of the slave method calls are useless and decrease the simulation performance.
To evaluate the cost of broadcasting, we measure the runtime needed for simulating a number of slaves each interconnected to a master with their own master-slave link. Figure 6.9 presents the gain obtained by replacing the broadcast master-slave link by a collection of unicast master-slave links (as illustrated by figure 6.8) when all the messages sent by the master are destinated to a single slave.
The results show that using unicast rather than broadcast yields an important gain in simulation performance when more than 10 slaves are connected to a common master. Indeed, the simulation runtime of the unicast implementation increases very slowly with the number of slaves taking only 2.36 seconds when 20000 slaves are connected to the master (in contrast to the broadcast implementation where 10000 slaves take more than more than 20 minutes to simulate the same simulated time). It means that having the master know which slave to activate and activate this slave only instead of activating all the slaves and leave it to them to filter the message will improve the performance of simulating sensor networks. This especially holds with the environment modules that are connected to all the nodes of the sensor networks. For the link between the scheduler of the RTOS model and the tasks of a node, the gain achieved by using the unicast implementation is less important, as the number of tasks to represent an application is in the order of tens of tasks.

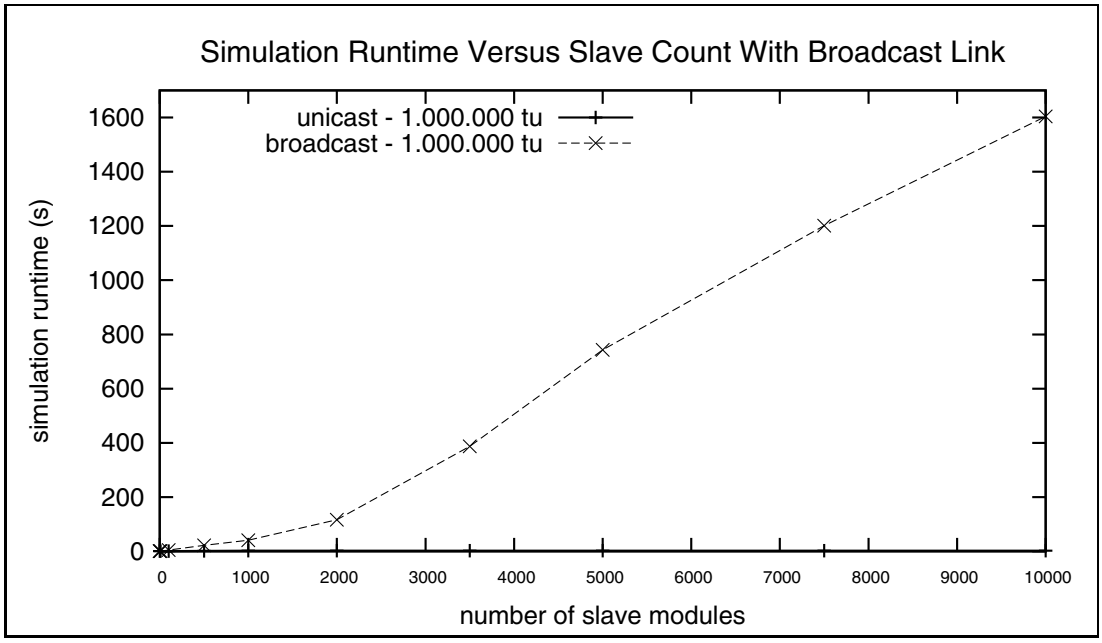

Figure 6.7: Simulation Runtime of Variable Number of Slave Modules for Broadcast and Unicast Master-Slave Communication

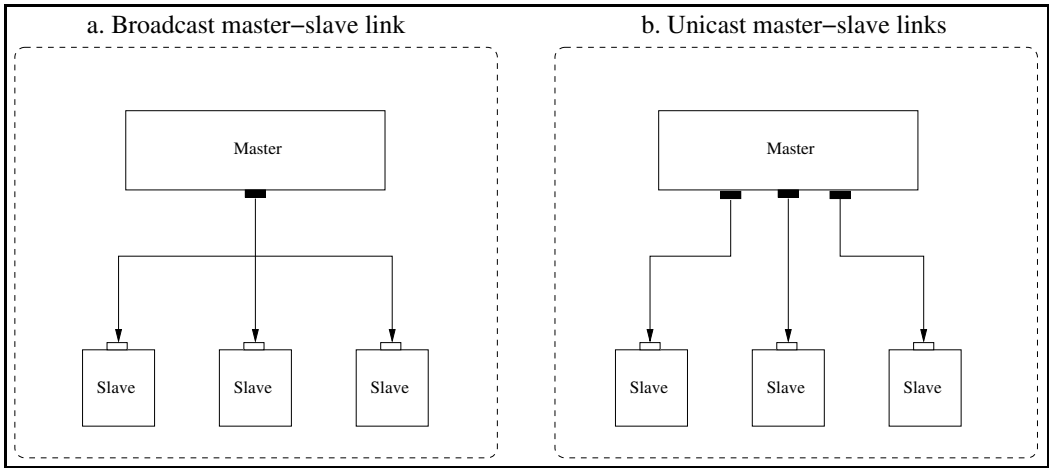

Figure 6.8: Alternative SystemC Implementation of Master-Slave Communication

6.4.5 Effect of SystemC on the Sensor Network Framework

In order top improve the simulation runtime of the framework, an implementation of the model where the broadcast master-slave link between the environment model and the nodes is replaced by individual links to the single nodes. The same simulation was then run for different node counts, and it showed that this does not reduce the simulation runtime. This means that the master-slave call mechanism does not represent an important part of the simulation time. Indeed, with the clock resolution of 1 ns used in the tests above, there are very few master-slave calls compared to the caals triggered

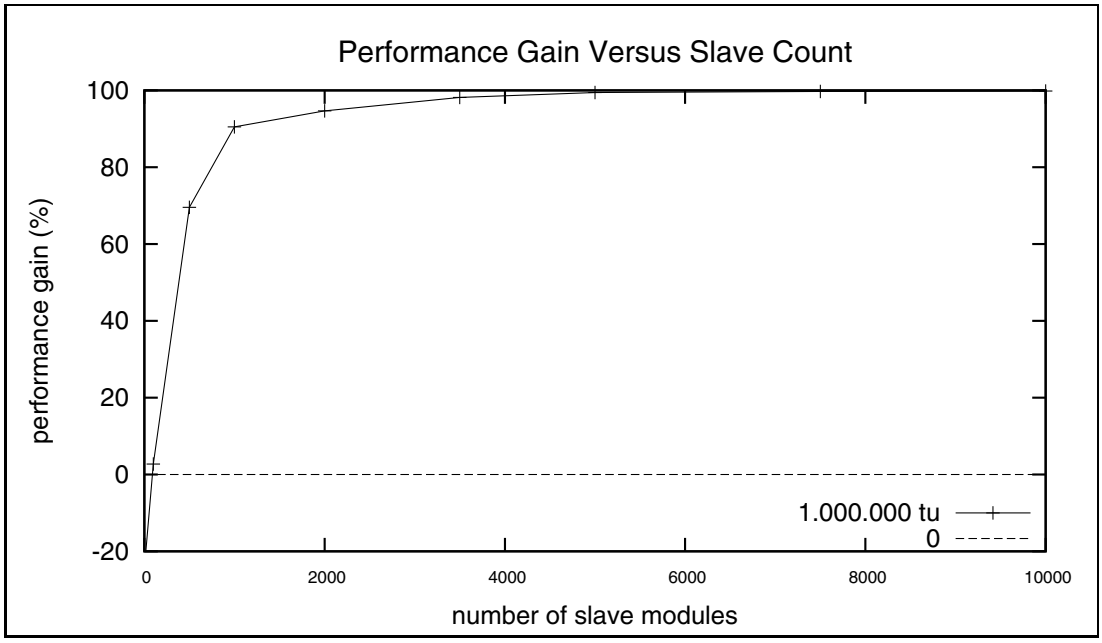

Figure 6.9: Performance Gain Obtained by using Unicast instead of Broadcast

by the clock. Instead of changing the interconnection of the modules, an improvement can be achieved by changing the time resolution: instead of having a clock-tic represent 1 ns, it could represent 100 ns. This implies that all the time parameter of the model are divided by 100 and rounded to an integer value. Time values are found in many modules: tasks have computation time, resource request times, periods, deadlines, etc., hardware components have transition time and IO devices have conversion delays. The scheduler has a switch-context time. As the period of the clock cycle changes, the cycle energies of the HW components and the speed of the mobility model must also be changed accordingly.
In the example used for the time measurement, these changes were made, and the performance of the simulation with a lower time resolution was measured. The results of this experiment are presented in figure 6.10 and are compared to the results with the 1 ns resolution. The simulation runtime for node counts from 5 to 1000 nodes range from 0.20 to 174 seconds against 5 to more than 4000 s for the 1 ns resolution. This confirms the fact that the clocking of modules represent a large fraction of the simulation time. Therefore, to improve the simulation time, one should consider which level of time resolution is needed as it tremendously affects the simulation runtime.

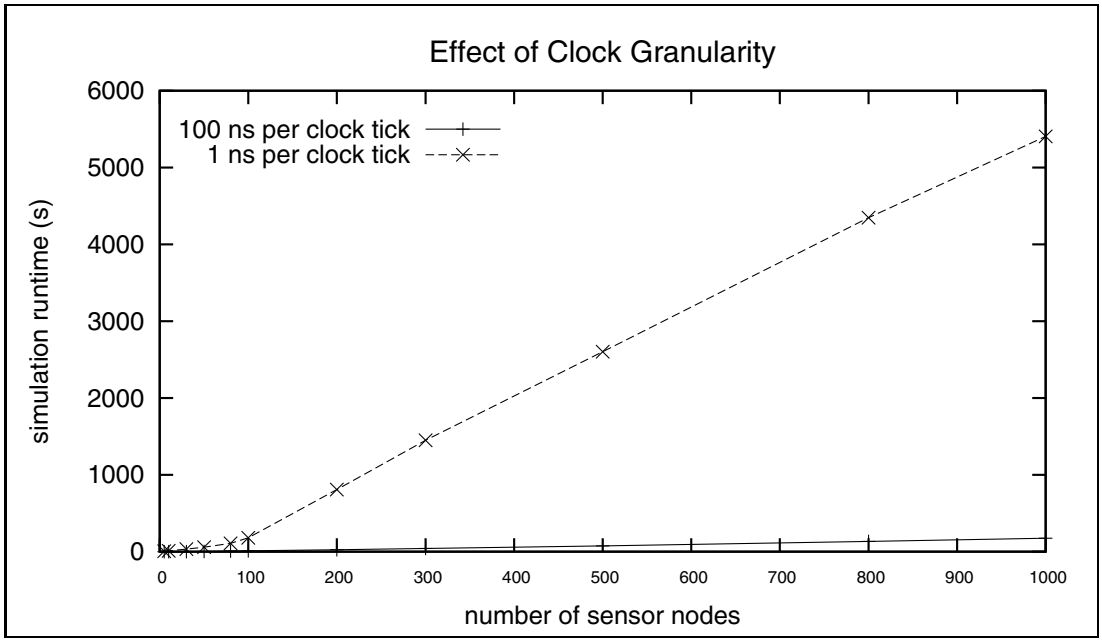

Figure 6.10: Effect of Time Resolution

Chapter 7

Conclusion.

During this Master Project, I have developped a framework for modelling and simulating sensor networks. The framework was developped in SystemC version 2.0.1 using its master-slave add-on library. The purpose of the framework is to make it possible for users to explore the design space of sensor nodes and to understand the effect of design decisions on the execution of an application on a sensor network. The design decisions concern different levels of the system: within the nodes, it is possible to experiment with different design solutions for the hardware, the operating system and the application itself. Each of these components are indeed represented in a modular way allowing the user to easily change them. The behavior of a node as its mobility and its energy resources are modeled within the nodes. The models for the battery and the mobility of the nodes can be adapted the application considered or the design decision to be made. At the network level, it is also possible to implement different classes of nodes to model heterogeneous networks. Models of the environment represent the behaviour of the physical phenomena monitored by the sensor nodes. These models also depend on the environment the network is designed for. In this project, the goal of providing a structure that allows simple modeling of different designs was achieved. Example modules were implemented to demonstrate the validity of the framework and to illustrate how it can be used. These examples were inspired by the Mica node and the Tiny OS embedded system. These modules represent the behavior of the different component of sensor networks at a rather low level, especially concerning the radio communication and the IO operations for which an interrupt model was defined.

With the structure of the framework is defined in this report, modules representing different design solutions for the different parts of the system can be added: the most common schedulers, hardware components, tasks, etc. Thus, the user could select from these building blocks to define a nodes hardware and operating system. The application can be represented using the task models. This simply requires the user to obtain execution time characteristics of the functions of the application. These characteristic can be provided using the processor simulators. Alternatively to using existing modules, the user can also defined his own modules. The structure of the modules is defined by the abstract modules, and the user can implement sub-modules of these thus garanteeing the compatibility of the custom modules with the structure of the framework. While most

current sensor networks are based on similar architecture (one processor, one transceiver and a collection of sensors), some new application of sensor networks start emerging that require high bandwidth, or have hard real-time deadlines. For such application, more complex platforms could be considered. The structure of the framework was constructed such that it can support these more advanced platforms. For example, the processing unit of the sensor nodes can easily be defined as a number of processing elements (microcontrollers, accelerators, signal processors, etc.) using the model of the SoC/NoC framework within the nodes. This ability of the framework to be extended and adapted to the future trends of sensor networks design is a key characteristic, and is specially important in the field of sensor networks where there is currently a lot of research activity going into different directions: from having larger sensors providing high bandwidth hard-real-time applications, to developping tiny and entirely integrated sensors such as smart dust. The limits of the framework are more connected to the simulation runtime of large models than to the architecture and applications that can be represented.

The level of abstraction of the implementation modules presented in this rapport is relatively low, representing for example the communication at the bit level. This may set limits on the scalability of the simulator (number of nodes that can be simulated) and the simulated time that can be simulated within reasonable time. From the performance measurements done on the examples, it appears that the simulation of up to 1000 nodes for a simulated time of 1 second would be of the order of 10 hours for a very simple application. However, in order to represent power consumption accurately, and in order to be able to catch differences in chosing for example different scheduling algorithms, the simulation is required to represent what happens in real systems closely. It is also likely that conclusions on the lifetime of the network do not necessarily require simulation of the whole lifetime, but could be drawn from observation on shorter periods. Moreover, the framework is opened, and users can define the implementation of the modules of it at the level of abstraction they select depending on the type of result they need from the simulation. Ideally, the simulation framework could thus follow the designer through the different steps of the conception of the sensor network, progressively adding details to the model.

Bibliography

[1] G. Asada, M. Dong, T. S. Lin, F. Newberg, G. Pottie, W. J. Kaiser, and H. O. Marcy. Wireless Integrated Network Sensors: Low-Power Systems on a Chip. In *Communications of the ACM*, pages 51–58, May 2000.

[2] Tracy Camp, Jeff Boleng, and Vanessa Davies. A Survey of Mobility Models for Ad Hoc Network Research. *Wireless Communication and Mobile Computing*, 2(5):483–502, 2002.

[3] David Cavin, Yoav Sasson, and Andre Schiper. On the Accuracy of MANET Simulators. In *Proceedings of the 2nd ACM International Workshop on Principles of Mobile Computing*, pages 38–43, October 2002.

[4] Rakhmatov Daler and Sarma Vrudhula. Energy Management for Battery-Powered Embedded Systems. *ACM Transactions on Embedde Computing Systems*, 2(3):277–324, August 2003.

[5] Jason L. Hill and David E. Culler. Mica: a Wireless Platform for Deeply Embedded Networks. *IEEE Micro*, 22(6):12–24, November 2002.

[6] Jason L. Hill, Robert Szewczyk, Alec Woo, Seth Hollar David Culler, and Kristofer Pister. System Architecture Directions for Networked Sensors. *ACM SIGPLAN Notices*, 35(11):93–104, November 2000.

[7] Hogthrob group. Hogthrob Project. http://www.hogthrob.dk.

[8] Jan Madsen and Kashif Virk and Jair Gonzalez. A SystemC Based Abstract Real-Time Operating System Model for Multiprocessor System-on-Chip. Multiprocessor Systems-on-Chips, Morgan Kaufmann Publishers, October 2004.

[9] Jan Madsen and Shankar Mahadevan and Kashif Virk. Network-Centric System-Level Model for Multiprocessor System-on-Chip Simulation. Interconnect-Centric Design for Advanced SOC and NOC, Kluwer Publishers, 2004.

[10] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, October 2002.

[11] Joseph M. Kahn, Randy Howard Katz, and Kristofer S. J. Pister. Emerging Challenges: Mobile Networking for "Smart Dust". *Journal of Communications and Networks*, 2(3):188–196, September 2000.

[12] Brad Karp and H. T. Kung. GPSR: Greedy Perimeter Stateless Routing for Wireless Networks. In *Proceedings of the Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 243–254, August 2000.

[13] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: Accurate and

Scalable Simulation of Entire TinyOS Applications. In *Proceedings of the First International Conference on Embedded Networked Sensor Systems*, pages 126–137, November 2003.

[14] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless Sensor Networks for Habitat Monitoring. In *Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications*, pages 88–97, September 2002.

[15] OPNET Technologies Inc. OPNET Modeler. http://www.opnet.com/products/modeler.

[16] Sung Park, Andreas Savvides, and Mani B. Srivastava. SensorSim: A Simulation Framework for Sensor Networks. In *Proceedings of the 3rd ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pages 104–111, August 2000.

[17] Ravishankar Rao, Sarma Vrudhula, and Daler n. Rakhmatov. Battery Modeling for Energy-Aware System Design. *IEEE Computer*, 36(12):77–87, December 2003.

[18] RF Monolithics. TR1000 data sheet. http://www.rfm.com/products/data/tr1000.pdf.

[19] Amit Sinha and Anantha Chandrakasan. Dynamic Power Management in Wireless Sensor Networks. *ACM Transactions on Embedde Computing Systems*, 2(3):277–324, August 2003.

[20] University of California, Berkeley. Network Simulator-2. http://www.isi.edu/nsnam/ns/.

[21] Guoliang Xing, Chenyang Lu, Robert Pless, and Joseph A. O'Sullivan. Co-Grid: an Efficient Coverage Maintenance Protocol for Distributed Sensor Networks. In *Proceedings of Information Processing in Sensor Networks*, April 2004.

[22] Wei Ye, John Heidemann, and Deborah Estrin. An Energy-Efficient MAC Protocol for Wireless Sensor Networks. In *Proceedings of the twenty-first International Annual Joint Conference of the IEEE Computer and Communications Societies*, June 2002.

[23] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. GloMoSim: A Library for Parallel Simulation of Large-Scale Wireless Networks. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*, pages 154–161, May 1998.

Appendix A

Code.

The source code of the framework is available from the CD bellow in the directory `src`. It includes both the codes for the modules and some simulation example files named `SN_xxxxx.cpp`.