Università di Pisa

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea Specialistica in Tecnologie Informatiche

Tesi di Laurea

SMEPP-Light: A Secure Middleware for Wireless Sensor Networks

Relatori:
Prof. Stefano Chessa
Dott. Michele Albano

Controrelatore:
Prof.sa Laura Semini

Candidato:
Claudio Francesco Vairo

Anno Accademico 2007/2008

A papà, fiamma ormai spenta,
ma sempre accesa nel mio cuore.

Contents

Chapter 1

Introduction

A Wireless Sensor Network (WSN) is a network composed by a large number of tiny, low-power, inexpensive sensor nodes, densely deployed inside or in proximity of an environment that must be monitored. WSNs use no predetermined infrastructure, and self organize into a (multi-hop) ad hoc network. Sensor nodes cooperate to realize an highly distributed application, whose goal usually consists in sensing environmental data and monitoring a variable set of parameters.

These features make a WSN an useful and versatile instrument for several applications, spanning many fields: in Ambient Assisting Living (AAL) project, for example, WSNs are used to enhance the quality of life of people by increasing their autonomy, self-confidence and mobility. In health institution, patients can be equipped with sensor nodes to monitor their physical conditions and be assisted, even at a distance. WSNs can also be used to detect forest fires or monitoring disaster areas.

On the other hand, the features that make WSNs so appealing, are the main causes of their problems too. The resources and energy limitation, the reduced computational power, the unreliability of the links, the dynamic topology of these networks imply non trivial difficulties in designing and developing applications.

One of the critical points to fully exploit WSNs potentialities is the possibility of abstracting these issues by means of a convenient middleware to support the applications development and maintenance, filling the gap be-

tween the applications and the underlying layers. Using a middleware that provides the right primitives, the application developer can focus on the application business logic, not having to implement the layers that provide the access to the hardware and the networking mechanisms.

The purpose of this thesis is to realize a middleware tailored on the WSNs, that offers security services and efficient use of sensors' energy. Security services are used to restrict the access to the groups composing the network, to authenticate peers when they join a group, to encrypt communications, and to ensure the confidentiality and the integrity of messages.

Energy efficiency is necessary since the nodes of a WSN are often battery-powered. We have included an Energy Efficiency module in SMEPP Light, that is responsible of turning off the radio of the nodes, when communication is not supposed to take place. We have observed that the use of the Energy Efficiency module reduces the power consumption of a node of about an half.

The thesis is structured as follows: Chapter 1 introduces wireless sensor networks by describing the features of sensor nodes composing them, and by discussing the main issues of such networks and their typical applications. Then we present the related works on to middlewares for embedded peer-to-peer systems, with a remark on SMEPP (Secure Middleware for Embedded Peer-to-Peer Systems) and on middlewares for wireless sensor networks.

The hardware platform and the software used for the thesis are shown in Chapter 2. Chapter 3 presents the specifications for the development, with regards to architecture, interface and protocols of SMEPP Light.

Chapter 4 is focused on the module responsible for the energy efficiency of the nodes. This chapter also presents the results of the preliminary measures on the Energy Efficiency of SMEPP Light.

The last chapter reports the conclusions and proposes future works pertinent to this thesis.

1.1 Wireless Sensor Networks

A wireless sensor network (WSN) is a network composed by a large number of sensor nodes which self organize into a multi-hop ad hoc network [1]. Sensors are spread in the environment without any predetermined infrastruc-

ture and cooperate to realize an highly distributed application, whose goal usually consists in sensing environmental data and monitoring a variable set of parameters. Sensed data are collected by an external node (*Sink*). The sink node, which could be either static or mobile, provides an interface of the WSN to external networks and it is accessed by the external users to retrieve the information gathered by the WSN. [2]. It is also the mean by which the user can program or query the network: the user interacts directly with the sink, that is responsible to forward the queries into the sensor network.
A typical WSN is shown in Figure 1.1

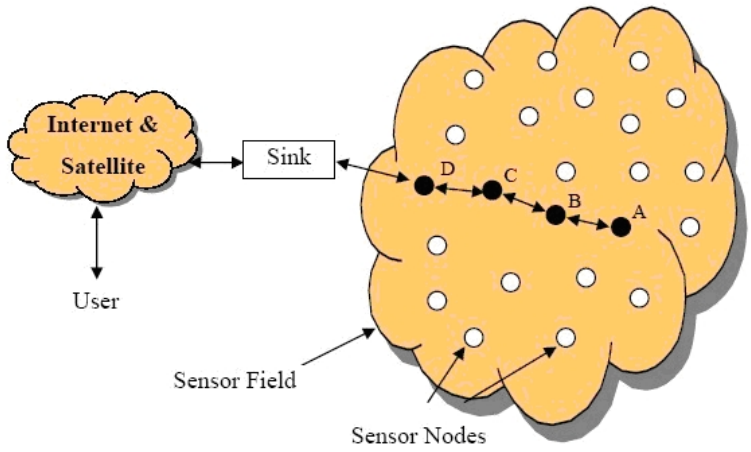

Figure 1.1: A Wireless Sensor Network.

The range of application of WSNs is wide [1] and ranges from e-health, military, environmental and home applications. These features ensure a wide range of applications for sensor networks. Some of the application areas are health, military, environmental and home. In military, for example, the rapid deployment and self-organization characteristics of WSN make them a very useful instrument for intelligence, surveillance, reconnaissance and targeting systems. In e-health, sensor nodes can be deployed to monitor and assist patients. They can also be used to detect forest fires or monitoring disaster areas.

1.1.1 Sensor Nodes

A wireless sensor node is a device characterized by small size, limited computational power, limited memory, that can sample the environment and that can communicate with other sensor nodes by means of a low power wireless transceiver [3].

A sensor node is composed by two main components: the first one (Figure 1.2) is a microsystem comprising a radio transmission system, a processor and a small memory that enables the node to execute a preprocessing of the acquired data, before sending them through the network up to the sink.

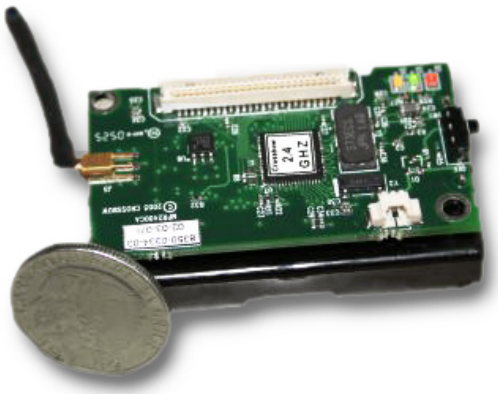

Figure 1.2: MicaZ Mote.

The second component (Figure 1.3) is a sensor board, equipped with one or more sensing units (*transducers*) capable of sampling some environmental data, such as light, temperature, magnetic field, sounds, etc...

1.1.2 Sensor Networks Issues

Wireless sensor networks are a special kind of ad hoc networks however, traditional wireless ad hoc networks protocols and algorithms are not well suited to the features and application requirements of sensor networks. Main differences between WSN and ad hoc networks are [1]:

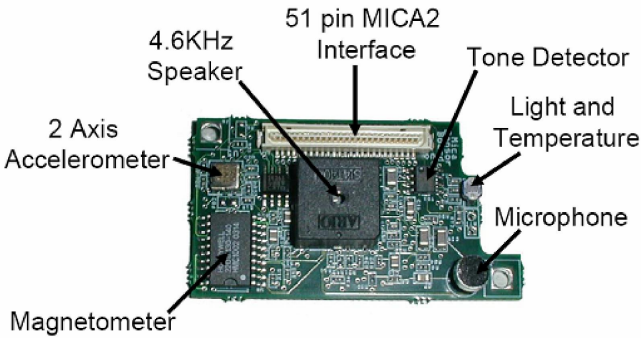

Figure 1.3: Sensor Board for MicaZ.

- The numbers of nodes in a sensor network can be significantly higher than the nodes in an ad hoc network.
- Sensor nodes are densely deployed.
- Sensor nodes are limited in power, computational capacities and memory.
- Sensor nodes are prone to failure.
- The topology of a sensor network could not be predictable and may change frequently.
- Sensor nodes mainly use a broadcast communication paradigm, whereas most ad hoc networks are based on point-to-point communications.
- Sensor nodes may not have unique global ID.

For these reasons, many researchers are currently engaged in developing schemes that fulfill these requirements.

Energy efficiency is one of the most important issue of WSNs, since they often operate in unattended mode. Unnecessary energy consumption must be avoided by careful hardware design as well as low level (i.e., operating system and support middleware) and high level (i.e., application) software programming. The transmission of a packet over the radio, for example, is

quite expensive in terms of energy consumption. Thus most of the research effort is devoted to develop techniques aimed to reducing the power consumption, such as routing protocols that reduce to the minimum the activities of the sensors' radio.

Recent commercial sensor devices provide a high level of flexibility allowing programmers to selectively turn on/off the various hardware components, including transducers, the ADC and the radio. While transducer activation can potentially be handled locally to a node, radio operation requires coordination among neighboring nodes to ensure a proper behavior of the whole network.

Communication inside a WSN is based either on the broadcast paradigm or on a multi-hop routing protocol. Because of the limited range of the radio of a sensor node, only a few nodes communicate directly with the sink. All the other nodes require the cooperation of intermediate nodes of the network to reach the sink (Figure 1.4).

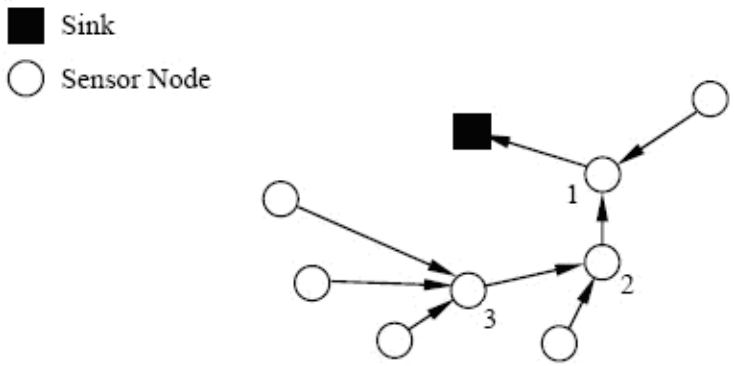

Figure 1.4: A cooperative multi-hop sensor network.

1.1.3 Applications

A diverse set of applications [3] for sensor networks encompassing different fields have already emerged, and they include e-health, environment monitor-

ing, military, home automation, commercial installation, intrusion detection, machine malfunction detection, intelligent toys and many others.

Environmental Applications

One of the most common environmental applications is the monitoring of animal species and collection of data concerning their habits, population, or position.

Sensors can be deployed to continuously report environmental data for long periods of time. This is a very important improvement with respect to previous operating conditions where human operators had to move to the fields and take manual measurements periodically resulting in fewer data, higher errors, higher costs and interference with life conditions of the observed species.

Pollution detection systems can also benefit from sensor networks. Sensors can monitor the current levels of polluting substances in towns, rivers, etc..., to identify the source of anomalous situations if any. Similar detection systems can be employed to monitor rain and water levels and prevent flooding, fires or other natural disasters [26].

Military Applications

The military can take advantage of sensor network technology too.

Sensor networks can be deployed behind enemy lines to observe movements/presence of troops, collect geographical information on the deployment area, survey crucial area, detect the targets.

Other possible applications can be monitoring of allied forces (equipment, ammunitions), detection of nuclear, biological and chemical attacks.

Health Applications

In the medical field sensor networks can be used to remotely and unobtrusively monitor physiological parameters such as heartbeat or blood pressure of patients, and to report to the hospital when some parameters are altered [16]. They can be also used to track the position of medicians and patients inside the hospital, or to realize interfaces for the disabled.

Structure Applications

In structure monitoring applications, sensor networks are deployed on structures of different types including bridges, buildings but also aircrafts, rockets or other equipments requiring continuous monitoring, to ensure reliability and safety. Sensor networks can be used to detect and locate damages as well as forecast remaining life, more effectively and economically with respect to traditional monitoring systems.

Domestic Applications

The diffusion of home automation is increasingly gaining interest, and sensors can be integrated into domotic applications for the comfort and security of the residents. Some functions in home automation include: light and climate control, control of doors and window shutters, security and surveillance systems, control of multimedia entertainment systems, automatic plant watering and pet feeding.

Furthermore, sensor nodes can be placed inside household appliances such as microwave ovens, refrigerators, video recorders, etc... These devices can interact between them and with external networks, enabling the user to remotely control the home appliances.

Wireless Sensor Networks in Automotive

In the automotive field, Vehicular Ad-Hoc Networks (VANETs) aim to improve safety and comfort of passengers. These networks can embed WSNs to provide security functionalities, such as collision warning, automatic payment for parking lots and toll collection.

Moreover, a sensor network can be used to monitor the position of a car, for example to track the vehicle in case of stealing (*Car Tracking*). Furthermore it can be used to supervise the traffic on particularly busy roads.

1.2 Middleware for EP2P

1.2.1 Embedded Peer-To-Peer Systems

A Peer-to-Peer (P2P) network is a completely or partially decentralized network in which nodes share distributed resources (data, computational power, bandwith). It does not have the notion of clients nor servers, but only equal peer nodes that simultaneously act as both clients and servers to the other nodes on the network. The main property of P2P networks is that communications are not based on pre-existing infrastructures, but rather on dynamic ad-hoc networks among peers. The terminals of these networks communicate in a bidirectional and symmetric way with each other. If this type of connection is not provided directly by the underlying physical network, a virtual network (overlay network) is set up on top of the existing physical network. In this sense, the term P2P can also be applied in a more generic context to name the set of communication models that provide end-to-end communication, independently of the application and the network protocols.

An Embedded Peer-to-Peer (EP2P) system [7] is a P2P system where small, low-powered, low-cost embedded systems collaborate in the processing and management of information using wireless channels. The use of P2P paradigms in the design of the application layer for network embedded systems, is very desirable since make these networks very robust against devices mobility, disconnections and failures. Furthermore P2P paradigms are easily adaptable and scalable, thus they are well suited to scenarios featuring mobile devices continuously joining and leaving the network.

Networks of embedded systems range from low-end networks which typically include wireless sensors and actuators networks (WSNs), to high-end networks such as Mobile Ad Hoc Networks (MANETs) where the devices are more powerful and may comprise last generation mobile phones, PDAs or laptops. These networks lack of a centralized control and each node acts as router for the rest of the network. So some mechanism of cooperation and network organization and management are needed, since devices mobility, failures and disconnections may easily disrupt communication paths, thus leading to the disruption of the whole network.

On the other hand these appealing features are the main reason of the major drawbacks of the EP2Ps [2]. Such systems, in fact, present a high degree of heterogeneity (applications may run on different devices, with quite different network bandwidths and computing power) and autonomy (the elements connect into the system and get disconnected in an independent way, causing frequent reorganizations of the system). These drawbacks manifest themselves in terms of technological challenges such as decentralization, links with transitory communications (connections and disconnections happen in an unpredictable and frequent manner) and a constantly changing topology.

In addition, there is a major problem that any EP2P system must deal with: security [4]. Traditional P2P systems have to face attacks on the routing protocols, attacks against the identity of the nodes, and threats against the confidentiality, integrity and authentication of the information that is circulating in the network. These problems in an EP2P context are more troublesome, because the underlying architecture is much more vulnerable than that of traditional P2P systems due to resource constraints (low battery, low memory, low processing power,), lack of tamper-resistant packaging, and the nature of open and public communication channels. Besides this high level of vulnerability, common security techniques are not easily adaptable to a P2P environment, since there are no centralized servers with security databases that can provide security services such as authentication and authorization.

1.2.2 SMEPP

A key aspect for the success of these systems is the possibility of abstracting all the intrinsic problems of these kind of networks, by means of a convenient middleware [2]. Without a middleware (including the tools and methodologies to support its use), the definition, creation and maintenance of an application could be rather complex. The middleware should hide the complexity of the underlying infrastructure while providing open interfaces to third parties for application development. The development of such a middleware is challenging because of the disappearance of the client and server paradigm, which is currently the base of the most used middleware (CORBA,

J2EE, .NET), and because of the necessity to face other critical requirements, which have to be supported by these infrastructures (mobility, new security problems, identification, discovery and localization protocols, new quality of software criteria, etc).

Secure Middleware for Embedded Peer-To-Peer Systems (SMEPP) [8], is a project whose goal is to realize a middleware that overcomes all the problems described in the previous section. It aims to hide the complexity of the underlying architecture, and at the same time to be secure, generic and highly customizable. With these properties, SMEPP aspires to be used in different application domains (from critical systems to consumer entertainment or communication), and to work on different kinds of devices (from PDAs and new generation mobile phones to embedded sensor actuator systems).

One of the objectives of SMEPP is to provide a high-level, service-oriented model to program the interaction among peers, thus hiding low-level details that concern the supporting infrastructure. The key-aspects of the SMEPP service model are: the organization of interactin peers into *groups*, interaction *service* offered by peers or by groups, big stress on *security*.

Group Characterization

The group is the basic abstraction of the middleware. A group is a set of peers that share services or provide services to other peers. The basic group functions are:

- *creation* and *management* of groups: a peer can create a group by itself, establishing the security requirements for the group. A peer can also search for existing groups and join them if it has the authorization to do it. The middleware has to manage the set of peers belonging to a group, by keeping track of new peers that join the group or existing peers that leave the group.

- *group security*: security is bound to groups. A group must be able to accept or reject a new peer in the group based on the security policies and the credentials of the peer. The group creator is in charge of setting the group security policy.

- *Information broadcast*: the middleware must support secure and ordered multicast of messages (events) at the group level.

- *group search*: The middleware must provide mechanisms to locate services and other peers inside a group.

Service Support

The SMEPP middleware is based on services. Roughly, peers offer services inside groups. A mechanism of service discovery is provided at the group level. A peer must belong to a group in order to use the services offered by the respective group. Service description has to be provided by means of a specially designed language for SMEPP service descriptions (viz., service contracts) that include information to allow the service discovery, adaptation, use or composition with other services.

Security

Security in EP2P is one of the key aspects in the success of this type of system. Traditional security infrastructures cannot easily be adapted to this type of systems, since most of them are based on trustable servers that provide authentication and authorization. On typical EP2P scenarios, this type of services will not be available and the peers must provide their own authentication.

SMEPP provides a scalable key infrastructure that permits a secure flow of information through the entire network, maintaining a low number of keys per node. This infrastructure offers the following services:

- automatic distribution and maintenance of the network keys, using little time and resources.

- a group level security allowing peers of a group to cooperate in a confidential way and fulfill tasks inside the group.

- secure network routing protocols to permit the nodes to communicate with the entire network. These protocols manage the essential require-

ments of network protocols, such as connectivity, coverage, fault tolerance and scalability.

- cryptographic protocols (e.g. authentication, key-exchange) and security primitives (e.g. security key primitives). These services are designed with respect to energy efficiency and take into account the limited computational power of embedded devices.

1.2.3 Middleware for Sensor Networks

The main purpose of a middleware is to support the development, maintenance, deployment and execution of applications, filling in the gap between the application layer and the hardware, operating system and network stacks layers. By using a middleware, the developers can focus on the business logic of the application, not having to implement the underlying layers. In the case of a WSN, this includes mechanisms for formulating complex high-level sensing tasks, communicating this task to the WSN, coordinating sensor nodes to split the task and distribute it to the individual nodes, performing data fusion for merging the sensor readings into high-level results, and reporting it. Moreover, appropriate abstractions and mechanisms for dealing with the heterogeneity of sensor nodes should be provided [2].

Existing Middlewares for WSNs

A seminal proposal in WSN is the Directed Diffusion paradigm [13] in which the WSN is controlled by a special sensor node (*sink*) that programs the network, collects the data and offers an interface to the application. This first effort has evolved into more advanced proposals combining database technology with WSN such as [14], [15]. In these proposals the database to be queried is the physical environment where the WSN is deployed. These approaches (e.g. TinyDB [15]) abstract specific sensor features and offer to the application an SQL-like query language through which it can program the network and access the sensed data. All these proposals exploit a data centric approach in the network organization. In fact all the communications are for named data, that is, the queries describe the data to be fetched by the

WSN rather than the addresses of the nodes that should execute the query.

On the other hand, a different approach may be more suitable to particular application contexts, such as Smart Environments which have in general rather different requirements. In particular the environment is generally indoor, hence also the size of the WSN hardly scales up to hundreds of sensors, and the network often has a small diameter, in some cases it may even be a star (viz., all the nodes can communicate directly with the sink). Furthermore the WSN operates close to the user, hence it can be expected human intervention at least for simple management operations such as battery or sensors replacement. Finally part of the sensors, for example the sensors forming body-area sensor networks, may dynamically join or leave the WSN. Under these requirements the data centric paradigms play a minor role and it becomes more important the ability of addressing individual nodes.

Recently appeared solutions adopt innovative models. TinyLime [9] introduces the concept of tuple spaces. In this middleware, data is represented by elementary data structures called tuples, and the memory is a multiset of tuples, called a tuple space, that is shared among all the sensors. Another proposal, TCMote [10], is based on tuple channels. A tuple channel is a FIFO structure that allows one-to-many and many-to-one communication of data structures, represented by tuples. By using the tuple channels the applications can decide which application components exchange data with each other. In this way the communication overhead can be optimized.

An event-based approach, is employed by the Mires middleware [11]. This middleware exploits a publish/subscribe paradigm, similar to the SMEPP service paradigm, that let the applications specify interests in certain state changes of the real world. Upon detecting a subscribed event, a sensor node sends a so-called event notification towards interested applications.

A different approach to WSN middleware is given by the ZigBee standard [3], [12]. At the network layer it has an inherently node centric behavior, but it offers service-oriented mechanisms to the applications. The ZigBee specification includes mechanisms for the energy management of the nodes, which however are configurable at network creation and that can not be adapted dynamically to the application.

SMEPP Light

Most of the approaches showed in the previous section do not address the issues related to security. This aspect significantly differentiates our work, since our approach inherits from SMEPP the security concepts related to the use of groups, that can be used to build secure multicast and unicast communications with a fine granularity.

SMEPP Light is the version of SMEPP tailored for WSNs, whose aim is focused on monitoring/control applications. Because of hardware constraints of the sensors, a full implementation of SMEPP would face several technical problems. For this reason SMEPP Light addresses a limited but yet significant and coherent subset of SMEPP concepts and primitives. In particular, differently from the full SMEPP specification, SMEPP Light does not support services, on the other hand it provides the same network organization mechanisms and it relies on the SMEPP eventing mechanisms for query dissemination and data collection. These mechanisms are implemented internally according to the Directed Diffusion paradigm [13].

SMEPP Light provides security mechanisms to restrict the access to the group, to authenticate peers, and to encrypt the communication and ensure the confidentiality of the messages flowing through the network. This feature is rather innovative, since existing middlewares does not provides security mechanisms at all, or, if provided, they are not so flexible. ZigBee, for example, provides security domains, but they can be applied only to the whole network, or to single link between two nodes.

In addition, since sensor nodes are battery-powered, SMEPP Light also offers energy efficiency mechanisms that dynamically adapt to the application needs, in order to reduce the power consumption of the nodes.

Chapter 2

Hardware and Software Target Platforms

2.1 Hardware Specification

The most commonly used platform for WSNs is the Crossbow Mote [17]. Motes belonging to this family of sensors, share a common core composed by an 8-bit Atmel microcontroller with RISC architecture, clocked at 8MHz, with an embedded 4KB RAM memory, an external 128KB flash memory, a persistent memory of 512KB, a wireless communication subsystem and three LEDs that can be used to show the operational status of the device.

The new IRIS mote brings some important improvements. It can count, in fact, on a 8KB RAM memory that lets the programmer create bigger and more powerful applications. IRIS mote improves the radio communication system too, by mounting a ZigBee-compliant transceiver, that grants a larger bitrate. In addition the radio range has been increased to an outdoor range of up to 300 meters.

Table 2.1 shows the main differences between the sensors belonging to the Crossbow Mote class.

Another platform for sensor nodes is the SPOT platform by Sun. The Sun SPOT project [18] (Small Programmable Object Technology) is an ongoing research project at Sun Labs. The SunSPOT devices are equipped with a 32 bit ARM-7 processor, 256 KB RAM, 2MB flash ROM, and an IEEE

	Mica2Dot	Mica2	MicaZ	IRIS
Microcontroller	Atmel ATmega 128L			Atmel ATmega 1281
Clock Frequency	4MHz	8MHz		
RAM (KB)	4	4	4	8
ROM (KB)	128	128	128	128
Storage (KB)	512	512	512	512
Radio	Chipcon CC1000		Chipcon CC2420 IEEE 802.15.4	
Radio Frequency	433MHz	916MHz	2.4GHz 250Kbit/s	
Max Range (m)	150		75-100	300
Power	3V Coin Cell	2x AA Batteries		
User Interface	1 LED	3 LEDs		

Table 2.1: Comparison for various sensor architectures.

802.15.4-compliant radio interface. SunSPOT is designed to support Java programming. These devices can count on a small J2ME virtual machine ("Squawk VM") that supports running applications with less overhead. Sun SPOT simplifies development by providing a single tool for programming, configuring, managing and monitoring Sun SPOT devices. Sun SPOT developers can use industry standard Java development tools such as Netbeans or Eclipse to program and debug their applications [2].

2.1.1 Communication System

Early projects about the communication system of a sensor node focused on optical transmission, however the current sensor hardware are based on Radio Frequency (RF) transmission. Optical communication is cheaper, easier to construct and consume less power than RF, but it requires visibility and directionality, which are requirements extremely hard to be met in a sensor network. RF communication is less reliable, suffers of radio interferences and wastes more power, but is more flexible and more suitable to the typical environments of sensor networks.

Crossbow motes employ two kinds of radios. The older one (Chipcon CC1000, embed on the Mica2 and Mica2Dot motes) works in a licence free band (315/433/868/916 MHz) with a bitrate of about 25-50Kbps, and offers a basic Carrier Sense Multiple Access (CSMA) Medium Access Control (MAC) protocol. The newer one (Chipcon CC2420, embed on the MicaZ and IRIS motes) is an IEEE 802.15.4-compliant transceiver operating at the frequency of 2.4GHz with a bandwith of 250Kbps. The motes mounting this transceiver, use an internal antenna that make the sensor nodes more manageable and self-contained than the others, which have a long external antenna. However the former radio type has a radio range of about 300 meters (outdoor), while the 802.15.4-compliant transceiver radio range is limited to 125 meters.

2.1.2 Transducers

A sensor node comprises also one or more sensing units (transducers) to sample the environment. A typical sensing unit is composed by two subunits: a sampler and an analog-to-digital converter (ADC). The ADC converts the analog signal produced by the sampler to a digital signal and then passes it to the processor.

Two possible strategies can be adopted in the design of the sensing unit subsystem. The most general and adopted approach, used by Crossbow, consists in mounting the transducers on stand-alone sensor boards, that can be attached to the main controller board of the node through an expansion bus. In this way the design of the controller part can be separated from the design of the transducers.
A typical Crossbow sensor board is equipped with light sensor, temperature sensor, microphone, sounder, tone detector, 2 axis accelerometer and 2 axis magnetometer devices. More economic sensor boards may mount a reduced set of transducers, and more expensive ones may be equipped with a GPS module for the localization of the sensor node.

The other approach consists in mounting the transducers directly on the micro controller board. In the sensor nodes adopting this approach, the transducers are soldered on the board, or can be mounted if needed, but the available configurations and the expandability is affected [3]. On the other

hand, this approach may reduce production costs and make the sensor nodes more robust than the stand-alone sensor boards which may detach from the controller board in rough environments.

2.1.3 Power

An ideal sensor node would be very small, not replaceable and very cheap. However, since the cost of a sensor is still not cheap, and sensors are mainly used for research purposes, the batteries must have a long life and must be replaceable. So most common sensor nodes are powered by standard AA batteries, that have a reasonable life duration, and are extremely easy to be found. Size is determined by the batteries, so currently sensors are a few cubic centimeters in size.

An exception is represented by the Mica2Dot motes (Figure 2.1) from Crossbow, that have been designed to be an alternative to the common sensors, and to be used when the reduced dimension is a strict constrain. It is powered by a coin cell battery, so its size is about a quarter dollar, but it is more resource constrained than the larger sensor nodes.

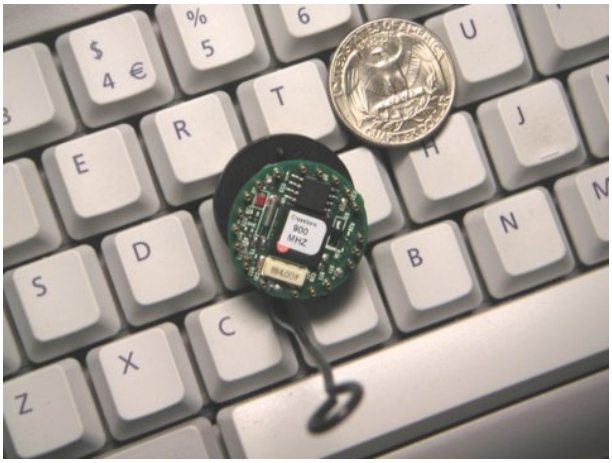

Figure 2.1: Mica2Dot Mote.

Currently researchers are investigating other possible power sources, such as solar cells, but there are some reservations about the effective uses of these

alternatives. Solar cells, for example, would not produce much energy in indoor environments, or under the coverage of tree foliage like in woods or forests.

2.2 Software

Software platforms for WSNs range from minimal operating systems constructed on C-like programming languages such as TinyOS and Contiki, to virtual machines such as Squawk, used in Sun SPOT sensors.

Contiki [20] is a lightweight operating system, which supports the dynamic loading and replacement of individual programs and services. It is built around an event-driven kernel but provides an optional preemptive multithreading mechanism that can be applied to individual processes. Contiki is written in C and it has been ported to several micro-controller architectures. Contiki includes some mechanism to reduce power consumption, and the total size of compiled code fits in 4KB RAM. It can load and unload individual applications or services at run-time. Contiki programs use native code and can therefore be used for all types of programs, including low level device drivers without loss of execution efficiency.

Squawk virtual machine [19] is a fully Java compliant and Connected Limited Device Configuration (CLDC) 1.1-compatible (viz., the Java ME platform) adopted for the SUNSpot sensor devices. The main goals of the Squawk VM are portability, ease of debugging, and maintainability. Squawk architecture is composed by a compact bytecode instruction set and small bytecode for the core classes, statically linked (35% - 45% shorter than equivalent J2ME class files). Once closed, a suite can not be modified anymore, and this results in a significantly reduced start up time when running an application from a suite (as opposed to a dynamically loaded set of classes). Squawk virtual machine supports the concurrent execution of more than one application, but each application is completely isolated from the others. However, given the immutability of suites, different applications can share common suites. This can significantly reduce the memory footprint of each application, that is particularly important in the embedded device space [2].

2.2.1 TinyOS

TinyOS [21] is a lightweight open-source operating system, developed by the Berkeley University. It is written in nesC (a C-like language with some extensions to support the development of mote applications) and it is the de-facto standard operating system for WSNs. TinyOS is a component-based and event-driven operating system, and it is highly modular and configurable. It has no kernel and it does not allow dynamic memory allocation. TinyOS allows the user to interact with the network by means of a pc connected to the sink and some Java tools, in order to collect data from the network, send commands to the nodes, and monitor network traffic.

Component-Based Model

A typical TinyOS application consists of a set of *components*, both provided by the OS and written by the programmer, assembled (viz., *wired*) together to form an executable. A component is composed by two parts, one for its specification, with the list of its *interfaces*, and another one for its implementation. Components interacts by means of bidirectional interfaces. A component *uses* or *provides* interfaces. The provided interfaces represent the functionalities that the components implements and offers to its users. The used interfaces represent the functionalities that the component needs to use to perform its job.

An interface may contain *commands* and *events*. Commands are the functions implemented by the component providing that interface, while events are notifications *signalled* by the components providing the interface. Events may notify an asynchronous event (e.g. the receipt of a message) or the ending of a split-phase operation (see Split-Phase Operations section).

A component providing an interface has to implement all the commands specified in the interface. A component using an interface has to implement all the events declared in the interface (viz., implements the handle to be executed at the signalling of related event). A single component may use or provide multiple interfaces and multiple instances of the same interface.

Figure 2.2 shows the component-based model in TinyOS.

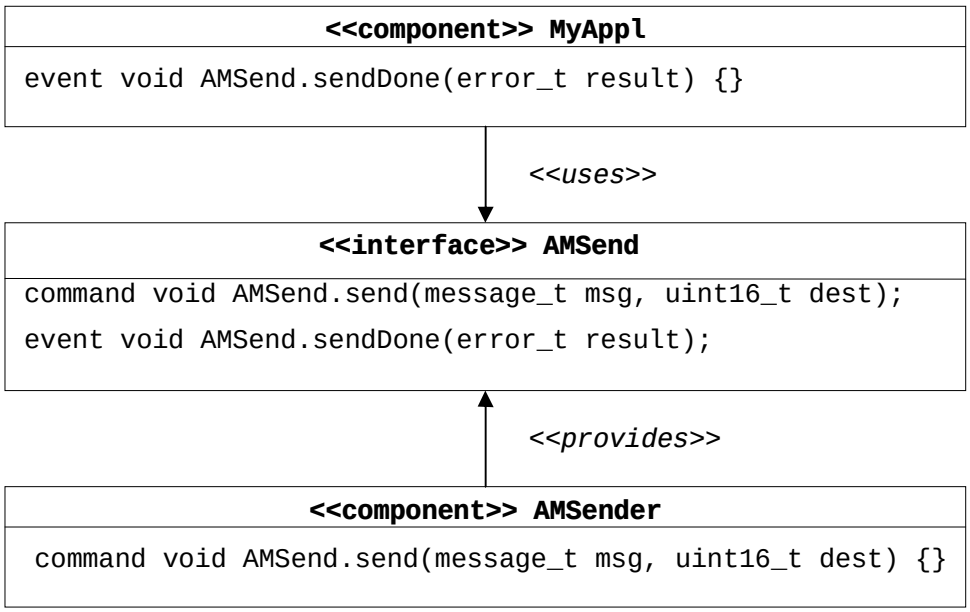

Figure 2.2: Component-based model in TinyOS.

TinyOS Execution Model

TinyOS does not support multi-threading, but it relies on a *task* mechanism to perform long computations. The standard nesC code is synchronous, it runs in a single execution contest without preemption, that is the CPU is kept busy until the completion of synchronous code. This simple mechanism enables the scheduler to reduce RAM consumption, but can affect the system responsiveness, since long-running pieces of code increase the response time to events.

To obviate this problem, TinyOS uses tasks. A task is an execution context that runs in "background", to completion, without being preempted by other tasks but it can be preempted by interrupt service routines (viz., events signalled by the underlying levels such as the receipt of a message). Thus the system can respond to hardware events while continuing its current operations, simulating a sort of parallelism.

A task can be *posted* by a command, an event or by another task. When posted, a task is inserted in a task queue, processed in FIFO order. As

already explained, tasks are used for long-running computation, however a task should not run for a long period anyway because, once scheduled, a task runs to completion before the next task can start (nevertheless it can be preempted by hardware interrupts whose handlers will eventually put new tasks in the task queue). If a series of long operations are needed, more separate tasks should be posted, one for each operation.

Because they are the root of a call graph, tasks can safely both call commands and signal events. Commands, on the other hand, do not signal events to avoid creating recursive loops across component boundaries (e.g., if command X in component 1 signals event Y in component 2, which itself calls command X in component 1). Events, finally, can post tasks, call commands, and signal other events. [21].

Since tasks and hardware event handlers may be preempted by a different asynchronous code, nesC programs are susceptible to race conditions. Races are avoided either by accessing shared data exclusively within tasks, or by having all accesses within *atomic* statements. The nesC compiler reports potential data races to the programmer at compile-time.

2.2.2 nesC

nesC is the most used programming language for WSNs. It has a C-like syntax and exposes a programming model that incorporates event-driven execution, a flexible concurrency model, and component-oriented application design. Its main goal is to allow the developer to build components that can be easily composed into complete, concurrent applications and perform extensive checking at compile-time. There is no dynamic memory allocation and the call-graph is fully known at compile-time. These restrictions make the program analysis (for safety) and optimization (for performance) significantly simple and accurate. nesC's component model and parameterized interfaces eliminate many needs for dynamic memory allocation and dynamic dispatch of methods [2].

There are two types of components in nesC: *modules* and *configurations*. Modules provide the implementations of one or more interfaces. Configurations are used to assemble other components together, connecting interfaces

used by components to interfaces provided by others. Every nesC application is described by a top-level configuration that wires together the components inside.

Interfaces are bidirectional and specify two sets of functions, commands and events. Commands have to be implemented by the provider of the interface and can be invoked (`call`) by the user of the interface. The events have to be implemented by the user of the interface and are signalled (`signal`) by the provider of the interface (see Figure 2.3).

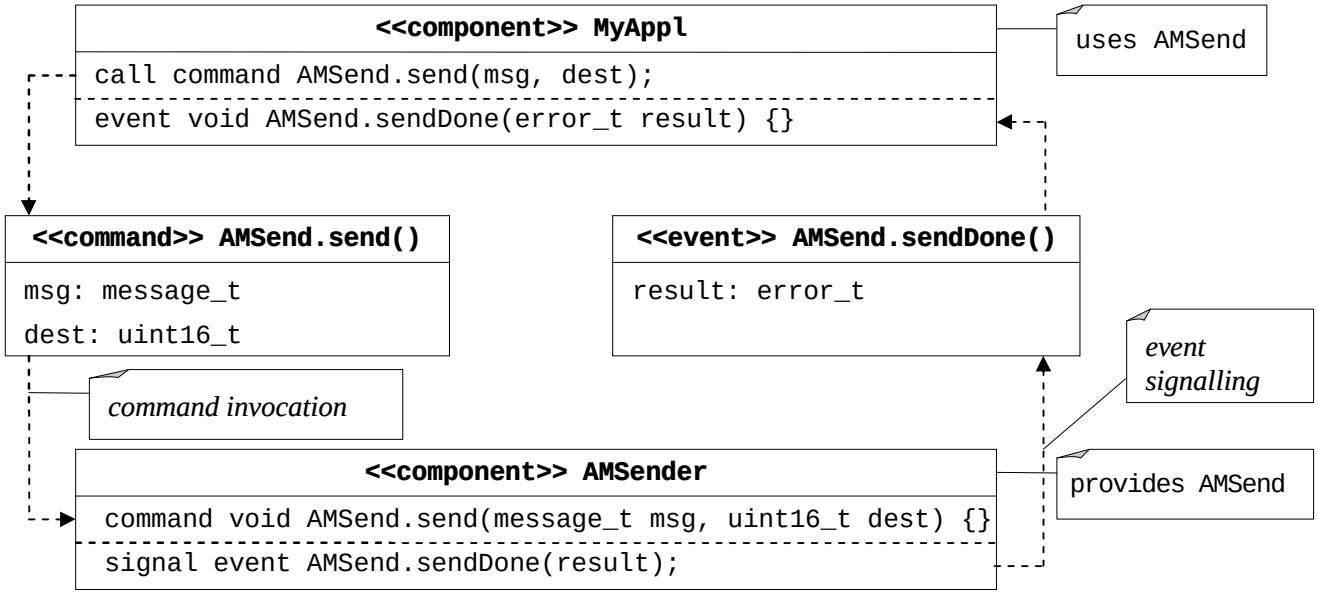

Figure 2.3: Component-usage model in TinyOS.

Concurrency model of nesC is based on tasks and interrupt handlers. Tasks run to completion, and may be preempted only by hardware interrupts, continuing their computation after the interrupts have been handled (this usually results in a new task in the task queue). nesC compiler detects potential race conditions at compile-time.

Split-phase operations

Some TinyOS operations, such as sending a radio message, or read a value from a transducer, takes too much time even if executed in a task, since they would block the execution of tasks until the completion of the operation, keeping the CPU inactive uselessly. nesC provides a *split-phase* mechanism that TinyOS adopts to perform blocking operations.

In such a paradigm, long-running operations are split into two separate phases of execution: invocation and completion. When a program calls a split-phase operation, the call returns immediately and other subsequent instructions can be executed, while the execution of the split-phase operation continues in "background" (e.g. the radio component sends a packet over the radio channel). When finished, a callback is invoked to notify the completion of the split-phase operation.
Here is a simple example of the difference between the two ways to implement the sending of a message with an increment of a counter when the operation ends:

Blocking mode:

```
if (send() == SUCCESS) {
    sendCount++;
}
```

Split-phase mode:

```
// start phase
send();

//completion phase
void sendDone(error_t err) {
    if (err == SUCCESS) {
        sendCount++;
    }
}
```

In the blocking mode the execution is blocked until the sending of the message is completed. On the other hand, in split-phase mode, the control returns just after the calling of `send()` command, and other actions can be performed while the message is being sent. Eventually, the event `sendDone()` is signalled by the radio component and the counter is incremented by the handle of the event.

Split-phase code is more verbose and complex than sequential code, but it has several advantages:

- split-phase calls do not fill up stack memory while they are in execution.
- split-phase operations keep the system responsive: it never happens that an application needs to take an action but all of its threads are tied up in blocking calls.
- it tends to reduce the stack utilization, as creating large variables on the stack is rarely necessary.

Split-phase interfaces enable a TinyOS component to easily start several operations at once and have them executed in parallel. Also, split-phase operations can save memory. This is because when a program calls a blocking operation, all of the state it has stored on the call stack (e.g., variables declared in functions) has to be saved. Since determining the exact size of the stack is difficult, operating systems often choose a very conservative and therefore large size. Of course, if there is data that has to be kept across the call, split-phase operations still need to save it.

Another example of split-phase utilization is the waiting. In sequential programming, blocking operations (such as I/O readings) have an implicit call to a `wait()` function, that blocks the execution until the operation is completed. TinyOS provides a timer mechanism that enables the execution to be continued while performing blocking operations.
In the following example `op1()` is a blocking function:

Blocking mode:

```
op1();
wait(); //blocked
op2();
```

Split-phase mode:

```
op1();
call Timer.startOneShot(500);
...         //other operations can be performed

event void Timer.fired() {
    op2();
}
```

In split-phase mode, during the waiting for the timer to be fired, other operations can be performed.

2.2.3 TOSSIM

TOSSIM [22] is a discrete event simulator for TinyOS applications. It works by replacing real components with simulated implementations of them. TOSSIM can simulate hundreds of nodes simultaneously, but it has the limitation that each simulated node runs the same program. Its main purpose is to faithfully simulate, as much as possible, the TinyOS applications, not the real world. Thus, some functionalities, such as the power consumption, are not featured. On the other hand, TOSSIM simulates the following functionalities:

- *timers*: the component providing the `Timer<Milli>` interface (viz., the component that implementsthe timer system in TinyOS), is replaced by a simulation implementations of millisecond timers.
- *communication*: a packet-level simulation replaces the packet-level communication component, that is acomponent providing the interface to access and manipulate the packets (e.g. fill the message fields, get or set the destination, etc.).

- *radio*: the low-level radio chip component is replaced by a precise simulation of the code execution.

TOSSIM works by pulling an event from the event queue, and executing it. The event queue is sorted by time, and contains events representing hardware interrupts, high-level system events (e.g. packet reception), and tasks.

TOSSIM is a library, so the programmer has to write programs to configure the simulation and run it. TOSSIM supports two programming interfaces: Python and C++. Python allows the user to dynamically interact with the simulation, thus acting as a debugger. C++ interface is supported since the interpreter may result a performance bottleneck, especially for large simulations.

A TOSSIM object has several functions to get information about the simulation like the id of the current node, the current time, or an object representing the radio model.

To enable the nodes of a simulation to communicate with each other, the user has to specify a network topology. TOSSIM is structured in a way that the user can easily change the underlying radio simulation. Its default radio model is signal-strength based and needs the user to specify network topology with the propagation strength of each link and the RF noise and interference each node ears. These information have to be specified by the user in files that are parsed by the TOSSIM library to build the radio model. An entry of the topology file is composed by the source node ID, the destination node ID and the gain, in dBm, of that link.

Here is an example of topology file:

```
1  2 -34.0
2  1 -38.0
2  3 -33.0
3  2 -33.0
3  4 -35.0
4  3 -40.0
```

Figure 2.4 shows the network topology derived from the configuration file above.

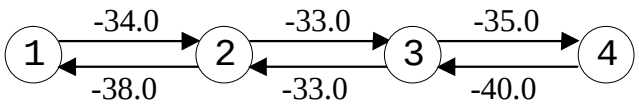

Figure 2.4: Graphic representation of the network topology .

In this case we are simulating symmetrical links (viz., with the same gain value in the two directions of the link) on a multi-hop chain network, in which a node can communicate only with its predecessor and its successor.

There are two ways to configure and perform a TOSSIM simulation: interact directly with it by a command prompt, or by means of a python configuration file. The former implies that the user has to configure the environment (import libraries, create the radio model, add the channels, start the nodes) at each simulation and that he has to call repetitively the `runNextEvent()` function to execute the simulation. The latter enables the user to perform the simulation just by executing a python file.

An example of a configuration file for TOSSIM is the following:

```
from TOSSIM import *
import sys

t = Tossim([])
r = t.radio()
f = open("topology.txt", "r")
...
    r.add(int(s[0]), int(s[1]), float(s[2]))
```

First the needed libraries are imported. Then a TOSSIM object and a radio object are created and the topology model is built adding to the radio object

the gain value of each link specified in topology file. The first two field are integer since they are the identifiers of the nodes, while the third field is a float since it is the gain value if the link.

```
f2 = open("log.txt", "w")
t.addChannel("Boot", sys.stdout);
t.addChannel("SmeppLightP", sys.stdout);
t.addChannel("Security", sys.stdout);
t.addChannel("Boot", f2);
t.addChannel("SmeppLightP", f2);
t.addChannel("Security", f2);
```

Each debug statement in a TinyOS application is put on a channel (in this case three channels are used, Boot, SmeppLightP and Security, to differ the various kinds of debug statements). In order to have these statements printed on the screen, it is needed to link each channel of the application to the standard output. In this configuration, channels are linked to a log file too.

```
t.getNode(i).createNoiseModel()
```

This line creates the noise model and added it to the TOSSIM object (`t`) of each node (`i`) in the simulation.

```
t.getNode(1).bootAtTime(10000000000); //1 sec
t.getNode(2).bootAtTime(10000000000); //1 sec
t.getNode(3).bootAtTime(20000000000); //2 sec
t.getNode(4).bootAtTime(40000000000); //4 sec

for i in range(0, 6000):
    t.runNextEvent()
```

The simulation starts for each node at the time specified as parameter to the `bootAtTime()` function, and then executes 6000 events.

Another way to specify the duration of a simulation is to set directly its time duration:

```
time = t.time()
while (time + 1200000000000 > t.time()): #120 sec
    t.runNextEvent()
```

This simulation runs for 120 seconds.

As seen, TOSSIM provides run-time configurable debugging output, allowing the user to test its applications in a monitored and repeatable environment.

Chapter 3

SMEPP Light Specification

SMEPP Light [5] is a simplified version of the SMEPP [8] middleware, tailored for sensor networks. It aims to be as much as possible compliant with the SMEPP specifications, however because of the resource limitations of the sensors, some features are not provided in SMEPP Light. In particular, this version of SMEPP does not use services for the cooperation of the peers.

The main feature of SMEPP Light is that peers organize themselves in groups, in order to access the functionalities provided in the group. For a WSN this usually means managing data acquired from the environment they are in. The existence of different groups enables the definition of different security and communication domains involving only peers owning the credentials needed to be into the group.

Peers in a SMEPP Light group adopt a publish/subscribe mechanism for their interactions. A peer can perform a subscribe for a certain event in a group. Then, if there is at least one peer providing that event in the group, the subscriber starts to receive automatically the requested events.

A group can be closed or open, private or public. In a closed group, a key, that must be known a priori by the peers, is needed to access the group and the communications are encrypted with a session key, that is established during the join phase.

SMEPP Light provides a two-level security based on symmetric cryptography: network-level security and group-level security. Two keys are used at network level. The first key is used for packets' confidentiality (viz., to

encrypt the data). The second one is used to compute a MAC (Message Authentication Code) field to be attached to the packets to grant the integrity of the message. A message is discarded if it has not the right MAC-field. These keys are set at compile time, but can be changed at run time.

Network-level security provides a first level of security. If the user wishes a stronger security, he should rely on group security, that is based on a session key (that can be refreshed) used to encrypt communications inside the group.

Group-level security exploits three keys, namely the `masterKey`, the `sessionKey` and the `sessionMAC`. The `masterKey` is used to restrict the access to the closed groups, so that a peer can join a closed group only if it owns the right key. The `sessionKey` and `sessionMAC` keys, are transmitted (encrypted with the `masterKey`) to the peers when they join the group, and are used to enforce data confidentiality and data integrity in all the communications within the group. These keys are initially known tothe peer that creates thegroup, and they are set either at compile time, or at run time. However the `masterKey` must be known in advance by all the peers that want to join the group. Since a peer can belong to several groups, the group identifier field is sent in clear text in every message, so that the peer receiving the message can know which key to use to check the integrity of the message and decrypt it.

An important requirement of SMEPP is that each peer must be described by an XML document. In SMEPP Light a peer description consists in the list of the transducers it is equipped with, and consequently in the list of events it can raise. Therefore the peer description is compressed into a bitmask (each bit representing a kind of event), that the nodes can easily store and exchange.

For energy management purposes, each group defines its own duty cycle that drives the radio activity of the peers (a duty cycle imposes to each node a period of activity, followed by a period of inactivity in which the node is practically turned off). Hereafter we call *management* duty cycles the duty cycles used for the group management (i.e. for join, leave and subscribe operations), and *user* duty cycles that are duty cycles related to the handling of the subscriptions (send an event related to a subscription to the subscriber, or unsubscribe of events).

Every node in a group operates the management functions according to

its *management* duty cycle (e.g., it responds to messages only when it is active). Moreover, each subscription has its own rate (e.g., the sink can request the light value every 30 seconds), and this results in a different duty cycle for each subscription to be handled. The middleware uses an energy efficiency module for duty cycle management. It records every duty cycle the node is involved in (both *management* duty cycles and *user* duty cycles) and manages the radio activity of the node by turning on or off the radio whenever necessary. This energy management policy is rather different from analogous common policies. In the case of ZigBee, for example, the duty cycle is defined in advance by the network coordinator and routers (normally according to the expected behavior of the application), and it is common to all the network.

3.1 Interface

SMEPP Light provides primitives for peer initialization, group management, and event transmission. Table 3.1 shows the main primitives of the middleware. The user can configure the middleware by means of two functions, namely `getParam()` and `setParam()`, used respectively to retrieve and to change the status of certain properties of the middleware. Note that, to avoid confusion, hereafter, we call *signals* the TinyOS events, to distinguish them from the SMEPP Light events (that are the events raised by the node in response to a subscribe request).

3.1.1 API

This section presents the SMEPP Light primitives. In particular it discusses their functionalities, the data structures used, and the communications involved, if any.

newPeer

```
command uint16_t smepp_newPeer(credentials networkKey, credentials
networkMAC, int8_t *error)
```

Peer Identification
`command uint16_t smepp_newPeer(networkKey, networkMAC, *error)`
Group Management
`command error_t smepp_createGroup(*groupDescription)`
`command error_t smepp_getGroups(groupDescription)`
`signal void getGroups_result(groupId[])`
`command error_t smepp_getGroupDescription(*groupDescription)`
`command void smepp_joinGroup(groupId, masterKey)`
`signal void peerJoined(*peerDescription)`
`signal void joinGroup_result(groupId, *subsList, result)`
`command peerDescr[] smepp_getPeers(groupId)`
`command error_t smepp_leaveGroup(groupId)`
`signal void peerLeft(peerId, groupId)`
Event Management
`command error_t smepp_subscribe(eventName, groupId, rate?, expTime?)`
`signal void subscribed(eventName, groupId, rate, expTime, offset)`
`command error_t smepp_unsubscribe(eventName?, groupId)`
`signal void unsubscribed(eventName, groupId)`
`command error_t smepp_event(groupId, eventName, value)`
`command error_t smepp_receive(eventName, frequency)`
`signal void receive_result(sender, groupId, eventName, value)`
Middleware Configuration
`command error_t smepp_setParam(key, *value)`
`command void* smepp_getParam(key)`

Table 3.1: Interface of SMEPP Light.

SMEPP Light applications call the `newPeer()` primitive to become peers. This call is used to initialize the components used by the node (e.g. radio component, energy efficiency module, etc). It is only a local call and does not result in any communication.

This primitive takes in input the network keys to validate the peer against the middleware, and returns the peer identifier. This identifier is unique within the network, and correspond to the sensor identifier used by TinyOS and assigned to the node at compile time. In case of failure an error code is reported by the `error` parameter.

Modifications to the environment: it initializes the components and modules used by the node and the `peers[]` list. It is a list containing the peer descriptors of all the peers of a group. A peer descriptor is a data structure containing the identifier of the peer and a bitmask describing the transducers the node is equipped with:

```
struct peerDescr {
    unit16_t peerId;
    uint8_t capabilities;
}
```

The application has to set the `capabilities` field, using the `setParam()` primitive, before invoking the `newPeer()` primitive.

Communications: no communication involved.

createGroup

```
command error_t smepp_createGroup(groupDescr *groupDescription)
```

SMEPP groups are logical associations of peers. Roughly, peers create groups and offer events (typically information sensed by the environment they are in). Then, other peers join existing groups in order to subscribe to one or more events offered in the group. Peers call the `createGroup()` primitive to start new peer groups.

The `groupDescription` parameter specifies the properties of the group to be created. It is a data-structure containing the name, the identifier, the key (viz., the `masterKey`) used to restrict the access to the group, and a bitmask (8 bits) specifying the security policies, in terms of closeness and privacy, of the group.

```
struct groupDescr {
    uint16_t groupId;
    char groupName[DIM_GROUP_NAME];
    uint8_t securityInfo;
    credentials masterKey;
}
```

A group can be closed or open, private or public according to the value of `securityInfo` field:

- if its first bit is 1, it implies that the group is *closed*: cryptography has to be used to communicate inside the group. Only authorized peers can join closed groups, and the `masterKey` field specifies the key to be used to encrypt the join request. Not encrypted requests, or requests encrypted with the wrong key, to join a closed group will be discarded by the middleware (viz., the middleware will return an error to the application issuing the request to join the group). Otherwise the group is *open*: the communication is performed in clear text and the group can be joined with a clear text request.

- if its second bit is 1, it implies that the group is *private*: it is visible by `getGroups()` only by encrypting the `getGroupsMsg` message with the correct `masterKey` (viz., the middleware will return only the `groupId` of the groups whose `masterKey` parameter matches the key used to encrypt the request, see `getGroups()` primitive). Otherwise it is *public* and it can be found by `getGroups()` with a clear text query.
  This second feature has not security purposes, because it would be hard hiding the existence of a private group from an attacker. It is implemented only to avoid the middleware returning not joinable groups.

It is important to note that *open* group implies *public* group: if the user specifies, in the `groupDescription` parameter, a group both *open* and *private* the middleware will return `FAIL`.

Table 3.2 summarizes the possible cases and the key needed by the primitives.

	masterKey	**sessionKey**
open-public	Not used	Not used
public-closed	`joinGroup()`	Every communication within the group
private-closed	`getGroups()` `joinGroup()`	Every communication within the group

Table 3.2: Key usage for each group type.

In SMEPP Light `credentials` is a data-structure containing a 128-bit key.

```
struct credentials {
    uint8_t key[16];
}
```

When the `createGroup()` invocation is successful, `groupId` field of `groupDescription` data-structure is initialized with the identifier of the group that has been created. The application has to allocate the group descriptor to be passed to the `createGroup()` primitive and has to initialize `groupName`, `securityInfo` and `masterKey` fields.

Modifications to the environment: it updates `myGroups[]` list. It is an array of `groupDescr` containing the group descriptors of the groups the peer belongs to.

Communications: no communication involved.

getGroups

```
command error_t smepp_getGroups(groupDescr groupDescription)
```

`getGroups()` is used to retrieve the identifiers of existing groups. However, differently from SMEPP, `getGroups()` is a split-phase call, so it does not return directly a list of `groupId`, but it simply sends the request to the network. The middleware will eventually notify the `groupId`s received by raising the `getGroups_result()` signal to the application.

The middleware waits for the replies to the `getGroups()` call, for a default time interval (settable by `setParam()`). To this purpose, it keeps a timer set to this value that is started when the `getGroups()` request is performed. When this timer expires the middleware raises the `getGroups_result()` signal to the application. This signal contains an array with the `groupId`s received.

The `groupDescription` parameter is used as a search filter (e.g., to retrieve *private* or *public* groups, or groups having a certain name). If the user requests *private* groups, he has to initialize the `masterKey` field of `groupDescription`. In this case the message is sent encrypted with this key. Otherwise the message is sent in clear text and the middleware will return *public* groups.

If the user requests *private* groups and does not specify the `masterKey`, or if he requests groups both *open* and *private*, the middleware returns `FAIL`.

Modifications to the environment: this command updates the following lists at every reception of a `groupMsg` with a new `groupId`:

- `groups[]`: is an array containing the group descriptors of the groups received.
- `groupDutyCycle[]`: is an array containing the *management* duty cycles of the groups received.
- `firstSeen[]`: is used to store the identifiers of the first peer that has replied to `getGroups()` request, for every different `groupId` received.

The middleware will send to this peer the joining request for this group (see joinGroup).

Communications: a `getGroupsMsg` is sent in local broadcast (peers receiving this message do not broadcast it further). This message can be encrypted with the `masterKey` according to `groupDescription` parameter (viz., it is encrypted if the user has requested *private* groups). Peers that are neighbors to the caller (viz., peers who are in the radio range of the caller), and that meet the search criteria, reply with a `groupMsg` (encrypted if the group is *private*), sent in unicast to the caller. This message contains the description of the group (identifier, name and security info) and its *management* duty cycle.

The data structures related to these messages are the following:

```
struct getGroupsMsg {
    uint8_t type;
    uint16_t gNameHash;
    uint16_t peerId;
    uint8_t securityInfo;
}

struct groupMsg {
    uint8_t type;
    uint16_t peerId;
    groupDescr groupDescription;
    dutyCycle groupDutyCycle;
}

struct dutyCycle {
    uint16_t length;
    uint16_t interarrive;
}
```

The `type` field is used to recognize the kind of the message (e.g., if the message is a `getGroupsMsg` or a `groupMsg`), and it is enclosed in every message.

`gNameHash` is a simple hash of the group name, used to identify the group. In this way it is possible to send fewer bytes, than transmitting the whole group name. `peerId` is the identifier of the sender. `securityInfo` denotes the kind of group that has been requested.

As in the other primitives, the `masterKey` field in `groupDescription` data-structure is never sent in any message.

`groupDutyCycle` specifies the activity/inactivity period of the group. It is a data-structure containing the length of the activity window of the node, and its period (viz. the time interval between two successive turning on):

getGroups_result

```
signal void getGroups_result(uint16_t groupId[])
```

The middleware raises this signal to the application when the timer related to the `getGroups()` call expires. The `groupId[]` parameter contains the identifiers of the received groups (viz., the different identifiers of the groups which replied to the `getGroups()` call).

Modifications to the environment: none.

getGroupDescription

```
command error_t smepp_getGroupDescription(groupDescr *groupDescription)
```

Peers call `getGroupDescription()` to retrieve the information characterizing a group, that is, its name and the security properities of the group. This information is already stored in the middleware, so it does not result in any communication.

The application has to allocate a group descriptor to be passed to the `getGroupDescription()` primitive and has to initialize the `groupId` field to notify to the middleware which is the group whose description is requested.

Modifications to the environment: none.

Communications: no communication involved.

joinGroup

```
command void smepp_joinGroup(uint16_t groupId, credentials masterKey)
```

Peers use the `joinGroup()` primitive to become members of groups. The `groupId` specifies the group to join, while the `masterKey` serves to validate the peer, that is, to check whether the peer is allowed to join the group, if it is a *closed* one.

A peer calling `joinGroup()` (say N) sends the request directly to the `firstSeen[i]` (say P), where `i` is the `groupId` group's position into the array `groups[]`. N specifies also its own capabilities.

Communications inside *closed* groups are encrypted with a group key (viz., the couple `sessionKey-sessionMAC`). If N joins a *closed* group, it needs to know these keys, so P will reply with a `groupKeyMsg` and a `groupMACMsg` to transmit the keys to N. If the group is an *open* one, these two messages are not sent.

For management purposes, P will send to N the list of the members of the group (viz., their identifiers and their capabilities). Moreover P will send also the list of the currently active subscriptions of the group. In this way N can set P as its parent for all the subscriptions and it can participate to all the ongoing group activities.

When a node joins a group, it inherits the *management* duty cycle of the group, and the *user* duty cycle of all the subscriptions already present into the group. P will send also time information (viz., the offset between the current time and the next fire of the *management* duty cycle of the group) to let N be synchronized with the joined group.

The receipt of these messages results in raising the `joinGroup_result()` signal to the caller's application.

Furthermore, P forwards the message received in group broadcast to notify the joining of N to all the members of the group. The receipt of this

message results in raising `peerJoined()` to the application of each node of the group.

Modifications to the environment: none.

Communications: a `joinGroupMsg` is sent directly to the peer of the group to be joined (viz., `firstSeen[i]`). As consequence, the node that receives this message, will send the following messages to the caller: a `groupKeyMsg` and a `groupMACMsg` (if it is a *closed* group), a `peersMsg`, and, if needed, one or more `peerListContMsg` (if peer list does not fit in the `peersMsg`), and zero or more `subscriptionsMsg`, according to the size of subscription list. All these messages are answered with an `ackMsg` by the joining peer.

Furthermore the node already belonging to the group, that handles the join protocol, sends also a `notifyJoinMsg` in group broadcast, to notify that a new peer joined the group. If the group to be joined is a *closed* one, the first two messages are encrypted with the `masterKey` key, and the other ones are encrypted with the `sessionKey-sessionMAC` key.

The data structures for these messages are the following:

```
struct joinGroupMsg {
    uint8_t type;
    uint16_t groupId;
    uint8_t capabilities;
}

struct ackMsg {
    uint8_t type;
    uint16_t groupId;
    uint8_t ackType : 4;
    uint8_t ackedPkt : 4;
}

struct groupKeyMsg {
    uint8_t type;
```

```
    uint16_t groupId;
    credentials sessionKey;
}

struct groupMACMsg {
    uint8_t type;
    uint16_t groupId;
    credentials sessionMAC;
}

struct peersMsg {
    uint8_t type;
    uint16_t groupId;
    int16_t offset;
    uint8_t nextPeerListPkts : 4;
    uint8_t count : 4;
    uint8_t nextSubsListPkts;
    peerDescr peerList[PEERS_FIRST_MSG];
}

struct peerListContMsg {
    uint8_t type;
    uint16_t groupId;
    uint8_t seqNum : 4;
    uint8_t count : 4;
    peerDescr peerList[PEERS_FOLLOW_MSG];
}

struct subscriptionsMsg {
    uint8_t type;
    uint16_t groupId;
    uint8_t seqNum : 4;
    uint8_t count : 4;
    subsList_entry subsList[SUBSCRIPTIONS_PER_MSG];
```

```
    int16_t nextON[SUBSCRIPTIONS_PER_MSG];
}

struct notifyJoinMsg {
    uint8_t type;
    uint16_t groupId;
    uint16_t peerId;
    uint8_t capabilities;
    uint16_t parent;
}
```

Each node is equipped with a set of transducers and can "sense" some information from the environment it is in (e.g., the light, the temperature). However each node can be equipped differently from the other ones: `capabilities` is a bitmask specifying the transducers available on the node and so the data it can "sense" (viz., the events it can offer).

`ackMsg` contains two 4-bits fields, denoting the sequence number of the packet acknowledged, and the type of acknowledgement, because the same type of message is sent for every acknowledgement.

`peersMsg` contains two fields, namely `nextPeerListPkts` and `nextSubsListPkts`, denoting how many packets will be sent respectively for the list of peers, and the for list of subscriptions. The `offset` field serves to set the *user* duty cycle of the node to be synchronized with the rest of the group. The `count` field denotes how many entries of the array included in the message are valid.

If other packets are needed to complete the transmission of peer list, `peerListContMsg` is used. It contains lesser information than a `peersMsg`, so it can include more entries of the peer list.

An entry of `subsList[]` array in `subscriptionsMsg` is a data-structure `subscription` (see command `smepp_subscribe()`), except that the fields `groupId` and `parent` are not sent. This message contains also the information to synchronize each *user* duty cycle (viz., field `nextON`) that is the offset (in milliseconds) between the current time and the next fire of that subscription.

peerJoined

```
signal void peerJoined(peerDescr *peerDescription)
```

The middleware notifies the joining of a peer to the application of all the members of the group by means of this signal.
`peerDescription` contains also the capabilities of the joined peer.

Modifications to the environment: each node updates its `peers[]` list.

joinGroup_result

```
signal void joinGroup_result(uint16_t groupId, subscription *subsList,
error_t result)
```

The middleware raises this signal to the application, as consequence of a `joinGroup()` call. Two outcomes are possible:

- all the messages described in `joinGroup()` primitive are received. In this case `result` is `SUCCESS`, `groupId` is the identifier of the group joined and `*subsList` is a pointer to the list of subscriptions running into the joined group.
- not all the previous messages are received after a time interval set by the middleware, that is the joining is failed (e.g., wrong credentials used in `joinGroup()` call). In this case `result` is `FAIL`, `groupId` and `*subsList` are null.

Modifications to the environment: in the first case, `myGroups[]`, `subscriptions[]` and `peers[]` lists are updated.

getPeers

```
command *peerDescr smepp_getPeers(uint16_t groupId)
```

The application uses this primitive to retrieve the list of the peers in the `groupId` group. `getPeers()` returns the pointer to the `peers[]` list stored

in the middleware.

Modifications to the environment: none.

Communications: no communication involved.

leaveGroup

```
command error_t smepp_leaveGroup(uint16_t groupId)
```

Peers use the `leaveGroup()` primitive to exit a SMEPP group identified by `groupId`.

The middleware will notify the leaving to the peers of the group by means of the `peerLeft()` signal. This primitive deletes from `subscriptions[]` list all the subscriptions the peer performed into the group that is being left.

Modifications to the environment: it updates `myGroups[]`, `peers[]` and `subscriptions[]` lists on the caller.

Communications: a `leaveGroupMsg`, encrypted with the `sessionKey-sessionMAC` key, is sent in group broadcast.

The data structure for this message is the following:

```
struct leaveGroupMsg {
    uint8_t type;
    uint16_t groupId;
    uint16_t peerId;
}
```

peerLeft

```
signal void peerLeft(uint16_t peerId, uint16_t groupId)
```

The middleware raises this signal to notify to the application of each member of the `groupId` group that the `peerId` peer does not belong to the group anymore.

Modifications to the environment: `peers[]` and `subscriptions[]` lists of each members of the `groupId` group are updated.

smepp_subscribe

```
command error_t smepp_subscribe(uint8_t eventName, uint16_t groupId,
uint16_t rate?, uint16_t expTime?)
```

Peers invoke the `subscribe()` primitive to register themselves as event listeners (viz., subscribers) of the `eventName` events raised in the `groupId` group. When a peer detects an event matching one of the active subscriptions, it sends the value related to that event back to the peer (or the peers) that subscribed it.

Subscriptions do not last forever, they have a default expiration time, configurable by `setParam()`. If the caller needs a different expiration time for a particular subscription, he can use `expTime` parameter that represents the duration of the subscription expressed in seconds (max 65536 sec). Moreover each subscription has a `rate` (if not specified it is the default rate, configurable by `setParam()` too), that defines the sampling rate of the monitoring task associated to the event requested, which also implies the maximum rate at which the events can be sent back to the subscriber.

Each subscription results in the creation of a routing tree spanning on all the peers of the group and rooted in the caller. This tree is used to route the events to the subscriber. After the expiration time, the subscription (and consequently the associated routing tree) expires and the subscriber should issue again another `subscribe()` if it is still interested. Peers offering the event will send the information required by calling the command `event()`.

Once `subscribe()` has been called, the caller has to invoke the `receive()` command to accept the events (see below). As consequence, the middleware raises to the caller's application the `receive_result()` signal at the receipt

of an `eventMsg`.

Modifications to the environment: each node keeps a list (`subscriptions[]`) that contains information about the trees it belongs to.

An entry of this list is:

```
struct subscription {
    uint16_t parent;
    uint16_t subscriberId;
    uint16_t groupId;
    uint8_t eventName;
    uint16_t exp;
    uint16_t rate;
}
```

Communications: a `subscribeMsg`, encrypted with the `sessionKey-sessionMAC` key, is sent in group broadcast to create a tree rooted in the subscriber.

The data structure for this message is the following:

```
struct subscribeMsg {
    uint8_t type;
    uint16_t groupId;
    uint16_t subscriberId;
    uint8_t eventName;
    uint16_t exp;
    uint16_t rate;
}
```

subscribed

`signal void subscribed(uint8_t eventName, uint16_t groupId, uint16_t rate, uint16_t expTime, int16_t offset)`

The middleware raises this signal to the application of each member of the group, at the receipt of a `subscribeMsg`. This signal notifies a subscription for the `eventName` event, that will last for `expTime` seconds, and states that the subscriber needs data (viz., the target node has to call the `event()` primitive) every `rate` seconds.

`offset` parameter serves to synchronize the application with the energy efficiency module, that has to keep the radio on while the application calls the command `event()`.

Modifications to the environment: `subscriptions[]` list of each node is updated.

smepp_unsubscribe

```
command error_t smepp_unsubscribe(uint8_t eventName?, uint16_t groupId)
```

`unsubscribe()` is the dual of `subscribe()`, it cancels a previous subscription. In this case, the caller is not notified of the `eventName` events raised inside the `groupId` group anymore. Omitting the `eventName` parameter, unsubscribes the caller from all events raised in that group.

Receipt of this message results in raising the `unsubscribed()` signal to the application of each peer of the group.

Modifications to the environment: caller updates its `subscriptions[]` list.

Communications: an `unsubscribeMsg`, encrypted with the `sessionKey-sessionMAC` key, is sent in group broadcast to notify the unsubscription to the peers.

The data structure for this message is the following:

```
struct unsubscribeMsg {
    uin8_t type;
    uint16_t subscriberId;
    uint16_t groupId;
    uint8_t eventName;
```

```
}
```

unsubscribed

`signal void unsubscribed(uint8_t eventName, uint16_t groupId)`

The middleware raises this signal when an `unsubscribeMsg` is received. It notifies to the application of each member of the group, that the subscription for the `eventName` event is not active anymore.

Modifications to the environment: `subscriptions[]` list of each node is updated.

smepp_event

`command error_t smepp_event(uint16_t groupId, uint8_t eventName, int32_t value)`

The application calls this primitive when the `rate` timer of the `eventName` event expires. `value` parameter is the payload of the message, that is the current value related to the event requested.

Modifications to the environment: none.

Communications: an `eventMsg`, encrypted with the `sessionKey-sessionMAC` key, is sent through the tree up to the subscriber.

The data structure for this message is the following:

```
struct eventMsg {
    uint8_t type;
    uint16_t groupId;
    uint16_t sender;
    uint8_t eventName;
    int32_t value;
```

```
}
```

receive

```
command error_t smepp_receive(uint8_t eventName, uint8_t frequency)
```

In order to receive the `eventName` events, the application has to call the `receive()` primitive. As consequence, the middleware raises one or more `receive_result()` signals, according to the `frequency` parameter, that specifies how many events the application wishes to receive.

The possible values of the frequency filed are `ONE_SHOT` (the default value) or `FOREVER`. If the user sets the `frequency` parameter to `ONE_SHOT`, the middleware discards all the subsequent events received, until a new invocation of the `receive()` command is executed.

Modifications to the environment: none.

Communications: no communication involved.

receive_result

```
signal void receive_result(uint16_t sender, uint16_t groupId, uint8_t
eventName, int32_t value)
```

When the event reaches the subscriber, the middleware provides the event to the application layer by raising the `receive_result()` signal, that provides the `value` associated with the event along with the `eventName`, the identifier of the peer that detected the event (`sender`), and the identifier of the group where the event was detected (`groupId`).

Modifications to the environment: none.

setParam

```
command error_t smepp_setParam(uint8_t key, void* value)
```

This command is used to set a parameter of the middleware. `key` is the parameter to be set and `value` is the new value to be assigned to it.

Some of the parameters that can be set at the current stage of development of the project, are:

- default `expTime` time of `subscribe()`.
- default `rate` of `subscribe()`.
- default `timer` of `getGroups()`.
- `capabilities` field of the peer descriptor of the node (viz., the list of transducers available on the node).
- `groupDutyCycle`: the *management* duty cycle of the group.
- group `sessionKey`:
- group `sessionMAC`:

getParam

```
command void* smepp_getParam(uint8_t key)
```

This command is used to retrieve the value of `key` parameter from the middleware. Available parameters are the same of the `setParam()` primitive.

3.2 Architecture

SMEPP Light is composed by three main components, namely the Peer Identification, Group Management and Event Management, that implement the SMEPP Light primitives, and three components that provide support to security, networking, and energy efficiency. The interaction between these components is shown in figure 3.1.

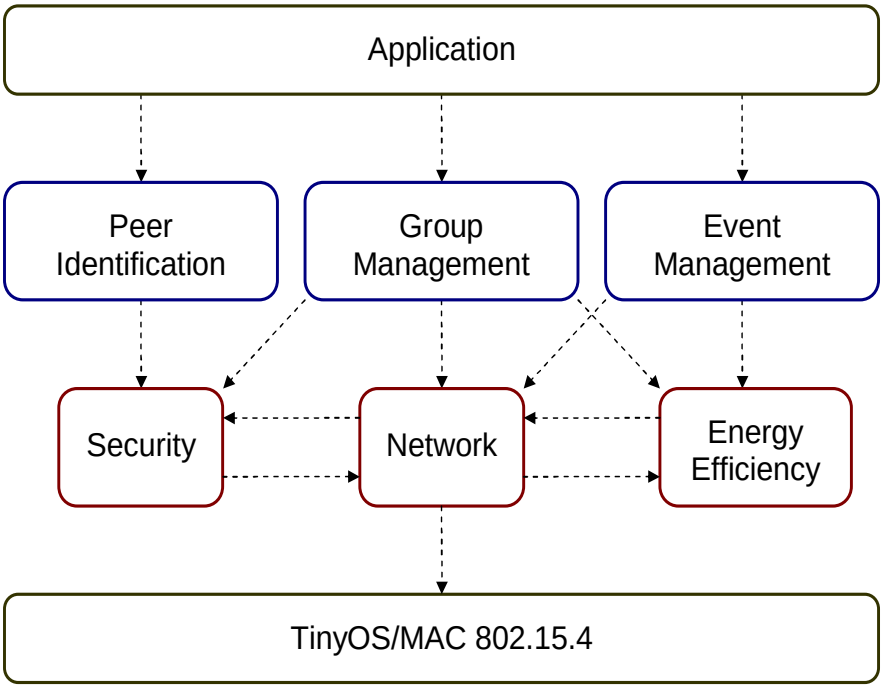

Figure 3.1: Components in the SMEPP Light architecture.

3.2.1 Main Components

Peer Identification Component

The Peer Identification component stores the list of the peers of each group the node belongs to (`peers[]`) and maps to the peer initialization primitives. It interacts with the Security component to set the network keys.

Group Management Component

The Group Management component manages the topology of the groups and maps to the group management primitives. It stores the following data structures:

- lists related to the groups the node belongs to: group descriptors (`myGroups[]`) and *management* duty cycles (`myGroupDutyCycle[]`).

- lists related to the groups received by the `getGroups()` primitive for the discovery of the groups: group descriptors (`groups[]`), identifiers of the first peer replying to `getGroups()` for each `groupId` received (`firstSeen[]`), and *management* duty cycles (`groupDutyCycleReceived[]`).
- an array of pointers to the *management* duty cycles stored into the `channel_table[]` of the Energy Efficiency module, used for the radio management (`EEDutyEntryId[]`, see below).

This component interacts with all the support components: it sets the group session keys, it sends and receives data from the Network component when most of the primitives are executed, and it interacts with the Energy Efficiency component to set the information for the management of the peers duty cycle.

Event Management Component

The Event Management component maps to the event management primitives and it is in charge of subscription and event management. It provides commands to subscribe or unsubscribe to a certain event, generate events and set the reception mode for a certain event.

It stores the following data structures:

- the list of the subscriptions the node is involved with (`subs[]`).
- the list of pointers to the *user* duty cycles stored into the `channel_table[]` of the Energy Efficiency module (`EESubsEntryId[]`).
- the receipt-mode for every type of event subscribed (`rec_mode[]`), as set by the application by means of the `receive()` call.

This component interacts with the Network component to access the wireless medium and to set up the routing trees associated with subscriptions. It also interacts with the Energy Efficiency component to configure the *user* duty cycle of the peer according to the subscriptions generated or received.

3.2.2 Support Components

Security Component

The Security component manages the keys for all the security issues related to the network and to the group layers, and provides commands to encrypt/decrypt messages. It keeps the network keys set by the Peer Identification component, the group master key set by the Group Management component, and the group session keys received from the Network component during the join protocol.

At the current stage of development, session keys used for the group communication are fixed and can not be changed at run time. So a ill-intentioned user, hearing the communications, could extract the key by analyzing several messages all encrypted with the same key. Future versions of SMEPP Light will manage the protocols for the dynamic refresh of session keys.
The integration of the cryptography component is currently under development.

Network Component

The Network component implements the communication between peers. It relies on the TinyOS network/MAC component (`CC2420ActiveMessageC`), that simply allows to address a single node, or to broadcast the message to all the network. SMEPP Light Network component provides further mechanisms of communication:

- a group broadcast protocol, used to implement the subscribe and the management of the routing trees associated to the subscriptions.
- a one-hop broadcast protocol, used to implement the group discovery and the join mechanisms. In this way the resulting groups are guaranteed to be connected, that is desirable for routing reasons. reasons.

Energy Efficiency Component

The Energy Efficiency component manages the duty cycles of the peer. In particular it manages the on/off periods of the radio interface according to the

user duty cycles associated to the subscribe messages received or generated by the peer. It stores a list (`channel_table[]`) containing the duty cycles the node is involved with. An entry of this table comprises mainly by the length of the activity window, its period, and its expiration time.

It should be observed that the management of the radio is transparent to the other components, so it is possible that the application calls a command that involves the transmission of a message, when the radio is turned off. In this case the message is buffered, with all the related information (destination, length, and type of the message), and when the radio becomes active, all the messages stored in the buffer are sent.
Further details on this module are reported in Chapter 4.

3.3 Protocols

3.3.1 Creating and Discovering Groups

Figure 3.2, shows the protocols used to create new groups and discover existing groups. Initially, every node has to invoke the `newPeer()` primitive in order to authenticate itself against the network and to obtain a peer identifier.

In the illustrated scenario we considered two nodes, A and B. Node B creates a group by calling the `createGroup()` command. The group creation does not involve communications since it consists in setting a few data structures in SMEPP Light (`myGroups[]` list, that is the array containing the group descriptors of the groups the node belongs to) and in setting the master and session keys in B's Security component, hence the middleware can immediately return the result of the operation.

The application layer of Node A performs the search for existing groups by invoking the `getGroups()` command. This command sends to all the A's neighbors (in local broadcast) a message requesting the group descriptors of existing groups. Local broadcast is implemented simply by not forwarding, on the receiving nodes, the message received. `getGroups()` takes in input a group descriptor that is used by the middleware like a search filter (e.g., only closed groups). In case of private groups, this message is encrypted with the

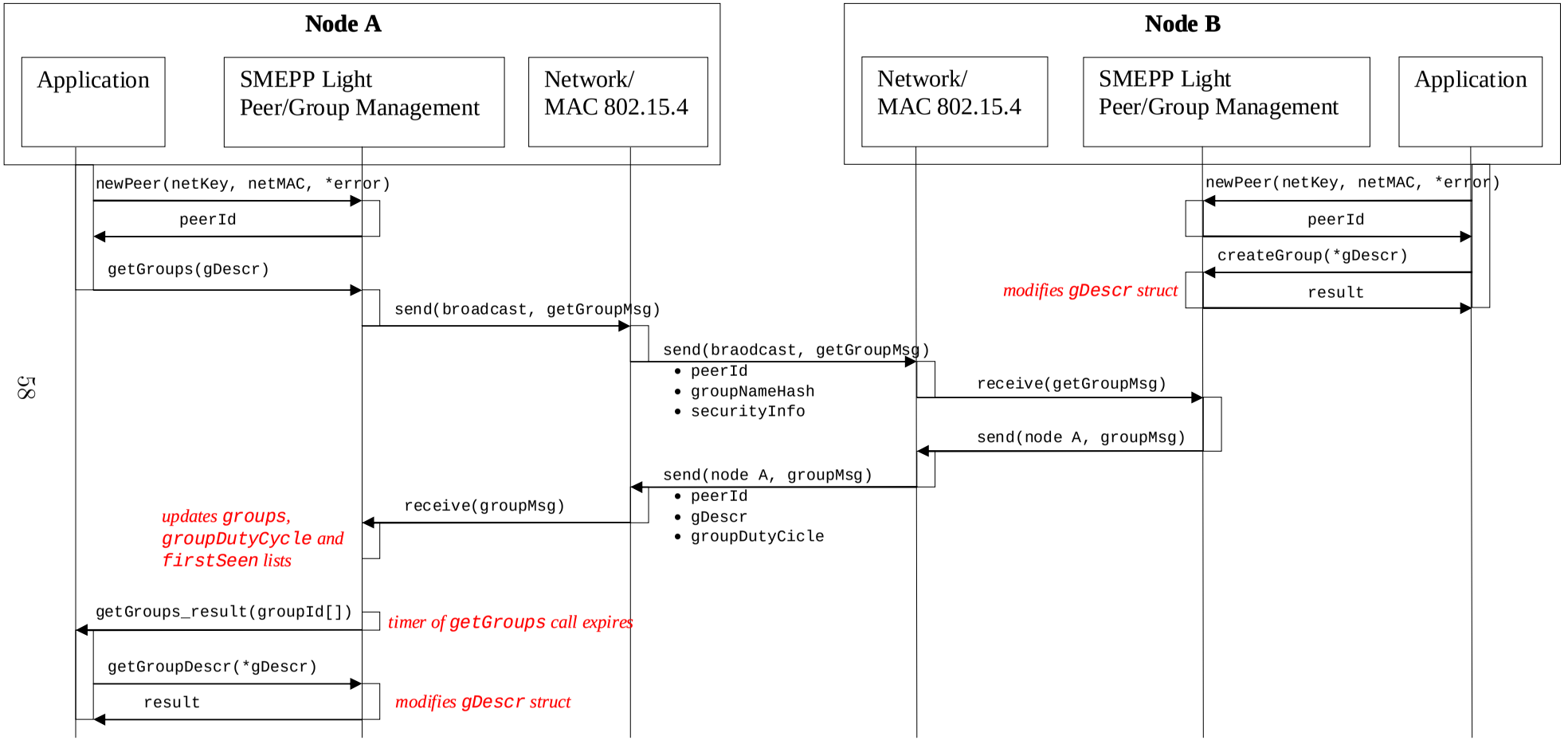

Figure 3.2: Group Creation and Group Discovery Protocols.

master key, contained in the group descriptor that is passed as parameter.

At Node B, SMEPP Light replies to this request (without involving the application layer) by sending to A the description of all the groups known to Node B (in this case only one).

At Node A SMEPP Light stores all the received descriptor, and the *management* duty cycles (`groups[]` and `groupDutyCycle[]` lists), and the identifier of the peer, that first replied to the `getGroups()` request for each different `groupId` received (`firstSeen[]`).

After a timeout the middleware notifies to the application layer the list of identifiers of detected groups. The application at Node A can access the description of each group by means of the `getGroupDescription()` primitive, that takes a `groupId` as parameter, and returns the corresponding group descriptor. Then, according to its necessities, the application can choose the group to join.

3.3.2 Joining a Group

A peer needs to perform a lengthy message exchange to enter into a group, as shown in figure 3.3. This protocol starts when the application layer of Node A invokes the `joinGroup()` primitive, passing the target `groupId` as parameter and, in case of a closed group, the `masterKey`. Then, a join message, containing also the capabilities of node A, is sent in unicast to Node B. In case of a closed group, this message is encrypted with the `masterKey`.

This message is notified to SMEPP Light middleware in Node B, that, in turn, decides whether to accept the request of A or not. At the current stage of development, a peer can handle one join request at a time, so if node B is involved in a join protocol already, it will discard A's join messages. In case of a closed group, an authentication protocol is performed, in addition to that showed in figure 3.3. It consists in a double authentication (peer-to-network and network-to-peer), based on messages encrypted with symmetric 128-bit keys.

If the request is accepted, SMEPP Light in Node B sends to A a set of messages containing the session keys (other two 128-bit keys, used for the cryptography into the group), the list of peers belonging to the group, and

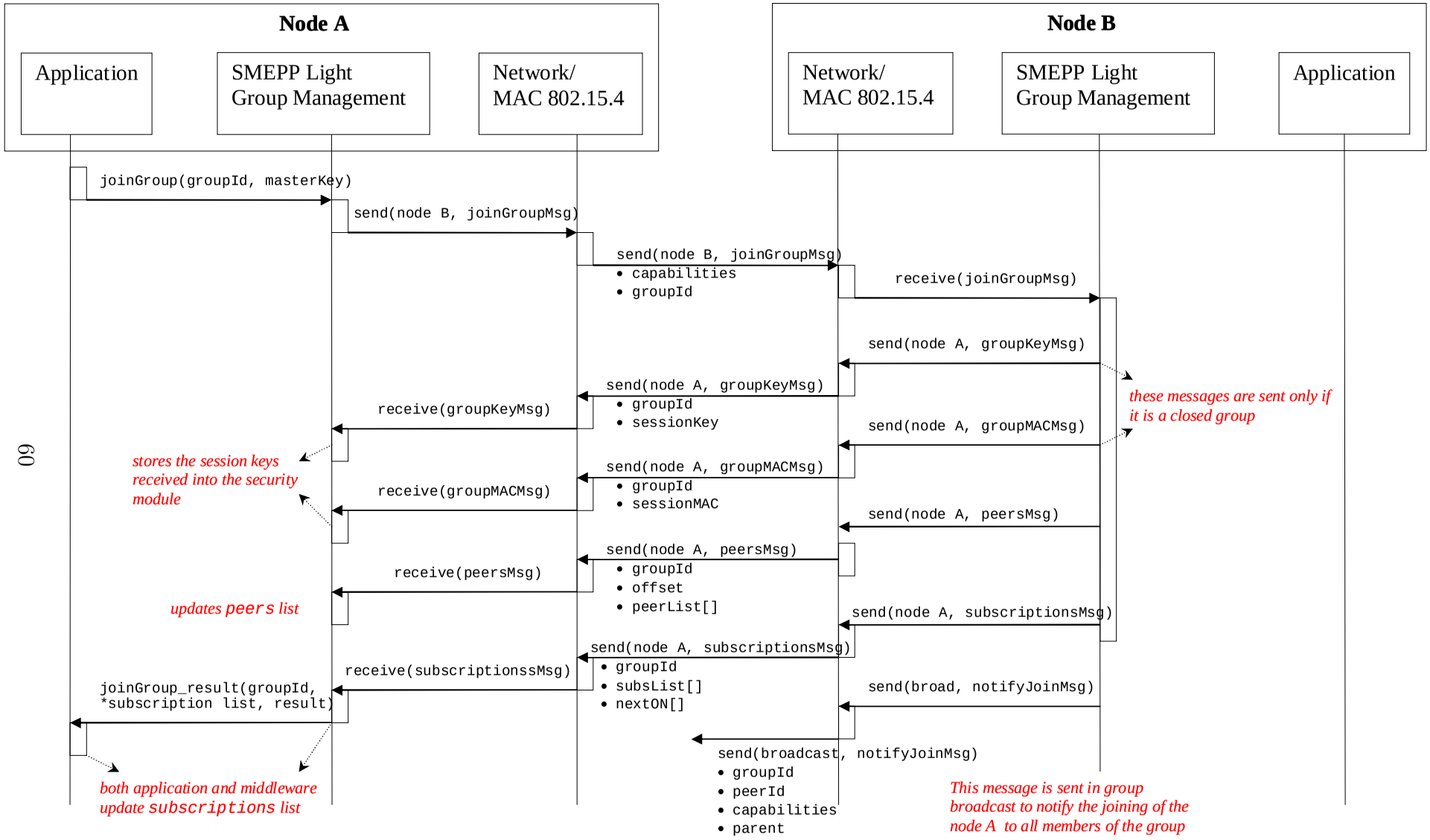

Figure 3.3: Join Group Protocol.

the list of subscriptions that are currently active into the group. In case of a closed group, packets transmitting the session keys are encrypted with the `masterKey`, while the other ones (and the packets for all the subsequent communications within the group) are encrypted with the session keys. In this version of SMEPP Light, the session keys are not updated, but a protocol to refresh these keys is planned, and it will be provided in future versions of SMEPP Light.

Because of the use of an Energy Efficiency module, that turns off the radio when it is not expected to be used, nodes need to keep the radio active at the same time to communicate. Thus both `peersMsg` and `subscriptionsMsg` contain fields (`offset`, related to the *management* duty cycle, for the former, and `nextON`, one for each subscription, for the latter) that are used by the Energy Efficiency module to set properly the entries in the duty cycles' table (`channel_table[]`), so that all the nodes of the group are synchronized.

Peers and subscriptions lists transmission, may result in sending more than one packet per list. All of the packets involved in the join protocol are supposed to be acknowledged by node A.

All these messages are used by SMEPP Light in Node A to update its internal data structures (`peers[]` and `subscriptions[]` lists in the middleware, and `channel_table[]` in the Energy Efficiency module), and once this phase is completed, SMEPP Light raises a `joinGroup_result()` signal to the application layer of the joining node to notify that the join protocol is completed. This signal contains a pointer to the subscription list received, to enable node A to raise previously subscribed events.

In case of failure (viz., not all expected messages are received within a timeout), the middleware notifies the negative result of the operation to the application, by raising the `joinGroup_result()` signal, passing `FAIL` as `result` parameter.

In the meantime, when SMEPP Light in Node B receives the acknowledgement to the last packet transmitted, it sends to all the other peers in the group, a message notifying that Node A joined the group.

3.3.3 Subscribe and Receive Events

The subscribe protocol is initiated by any peer in a group that wants to receive a given type of events generated by other peers in the same group. Figure 3.4 shows a scenario where Node A subscribes for events that are generated by a Node B.

To this purpose Node A invokes the `subscribe()` command, specifying as parameter the identifier of the group in which the subscribe will be executed, the event requested, the rate at which the subscriber needs the data, and the duration of the subscription. As consequence the middleware broadcasts the subscribe request to all the peers in the group.

Nodes receiving this message set a tree, covering the whole group, used to route the events back to the subscriber, that is the root of the tree. Furthermore, the middleware notifies this request to the application layer of each node by means of the `subscribed()` signal. This signal contains the event name for which the subscription holds, with its rate and expiration time, hence it is responsibility of the application to start any relevant monitoring task to detect the events matching the subscrption. This monitoring task should be activated at the sampling rate contained in the subscribe message.

In the case shown in figure 3.4, Node B is the only peer which can provide the event requested. Therefore, when the timer related to the event fires, the application of Node B has to produce a value for the event fired. Sampling an event consist in calling the `read()` command of the component providing the interface related to that event (e.g., `call SensorMts300C.read()` that is the component providing the interface for the light transducer), that converts the analog signal read from the transducer, to a digital value notified to the application layer by means of the `readDone()` signal, raised by the transducer's component. When receiving this signal, the application has to call the `event()` primitive to send the value read to the subscriber.

Node A, on the other hand, has to call the `receive()` command in order to retrieve the received value. The middleware, in fact, does not signal any received event until the application invokes this command. The `receive()` primitive takes two parameters in input, the event name of the requested event, and the number of values the application is prepared to re-

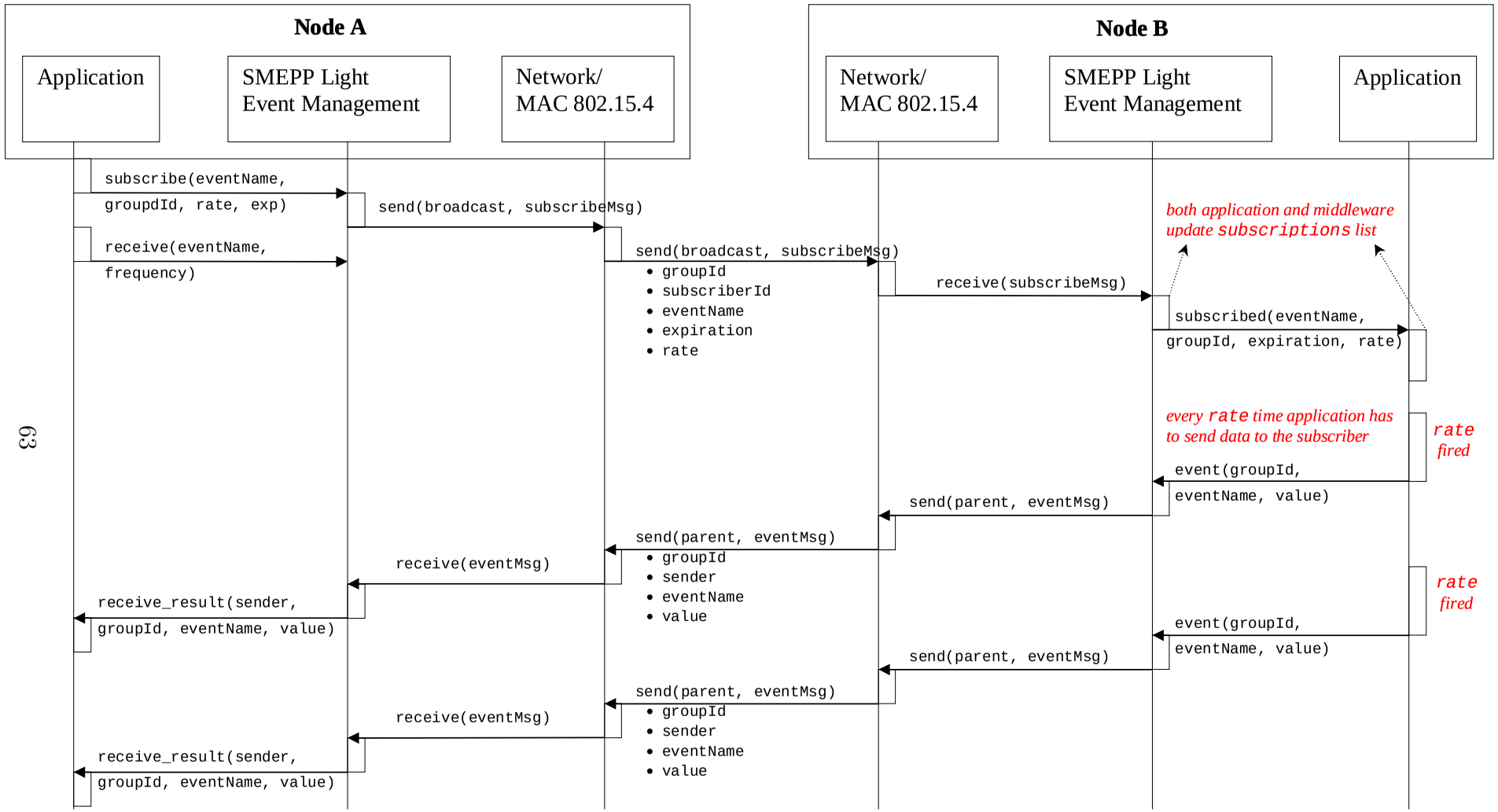

Figure 3.4: Subscribe and Receive Events Protocols.

ceive (`frequency` parameter). This can be `ONE_SHOT` or `FOREVER`. The former means that only one value is to be received, so the middleware will signal only one event value and will discard all possibly subsequent values received. The latter means that the middleware has to signal to the application every value it receives. In the first case, in order to retrieve other values for an event, the application has to call the `receive()` command again.

The middleware notifies the event received by means of the `receive_result()` signal, passing as parameters the identifier of the peer providing the event and the value received. This protocol goes on until the subscription expires or Node A cancels it using the `unsubscribe()` command.

3.3.4 Unsubscribe Events and Leave Group

Figure 3.5 shows the protocols used to retrieve the list of the peers of a group, to leave a group, and to unsubscribe events. The list of peers is obtained by calling the `getPeers()` command, with the identifier of the group as parameter. This is a local call, since it returns the list of the peers stored in the middleware of the same node, and therefore does not involve any communication.

To leave a group, the application has to invoke the `leaveGroup()` primitive, passing the identifier of the group to be left as parameter. As consequence, the middleware deletes all the subscriptions related to that group from the `subscriptions[]` list. This action could break communication paths if node A (the peer that is leaving) is in between a peer (say X) providing an event, and a subscriber (say Y) requesting it. However Y could receive values related to that event from other peers providing it. This is why, subscriptions have an expiration time, that is used to enforce subscription refreshing.

Moreover, when a node leaves a group, the middleware on the Node A updates `myGroups[]` and `peers[]` lists, by removing the entries related to the group left. Eventually SMEPP Light sends the notify of leaving in group broadcast. The middleware on the nodes receiving this message notifies the leaving to the application layer, by raising the `peerLeft()` signal, with the identifier of the peer that left and the group identifier as parameters.

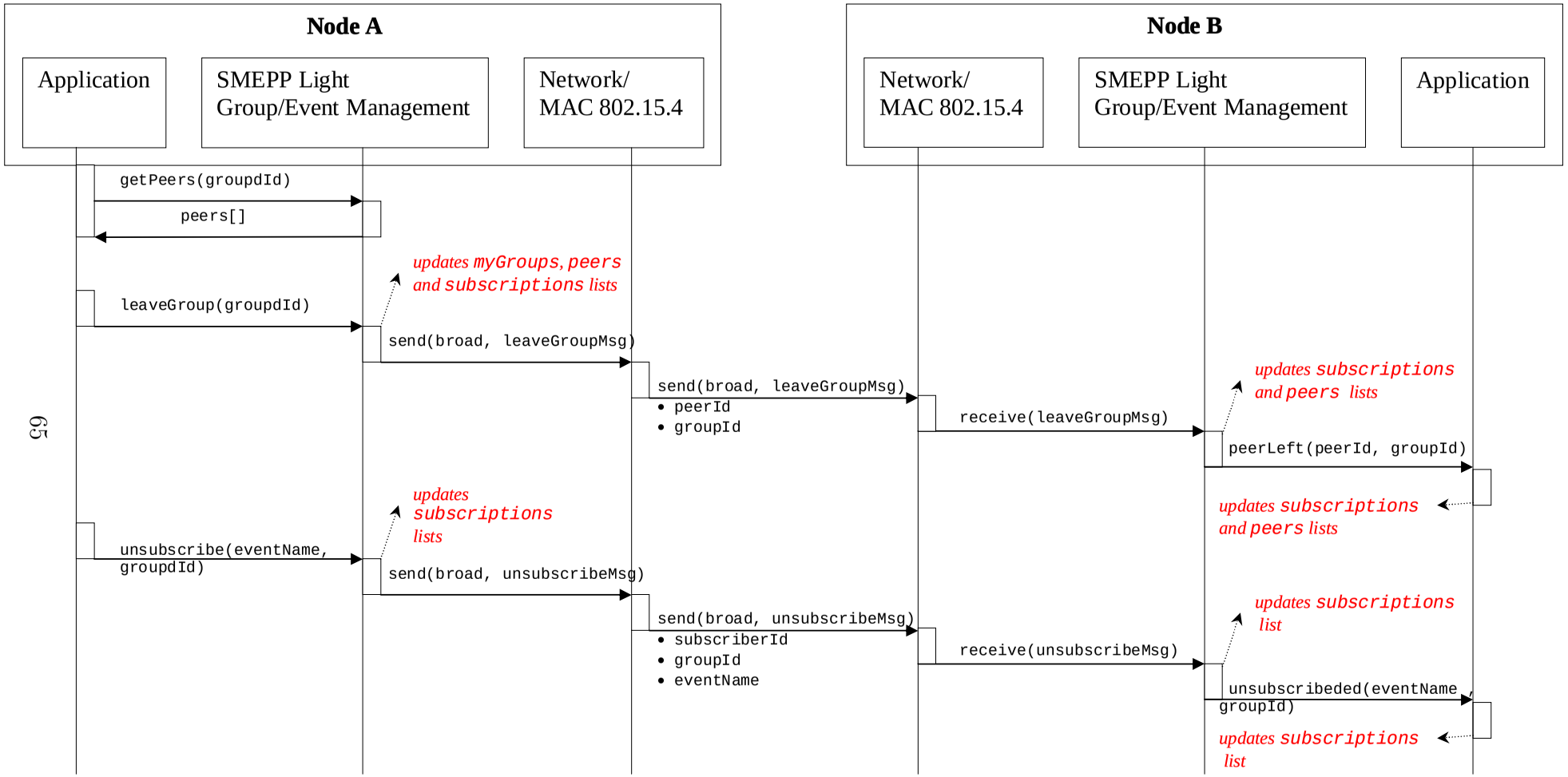

Figure 3.5: Get Peers List, Leave a Group and Unsubscribe Events Protocols.

Unsubscribing an event requires that the event had been previously subscribed. In this case, calling the `unsubscribe()` primitive, specifying the event to be unsubscribed and the group identifier where the event is running, results in updating the `subscriptions[]` list in the caller, and in sending an unsubscribe message in group broadcast. The SMEPP Light middleware of the nodes receiving this message, updates its `subscriptions[]` list by removing the entry related to that subscription, and raises the `unsubscribed()` signal to the application layer, passing the group identifier and the received event type as parameters.

Chapter 4

Energy Efficiency

The energy management is a crucial issue in WSNs, since they often operate in unattended mode. Sensor nodes composing a WSN are battery powered, and usually they can not be replaced. Thus, adopting techniques to reduce the energy consumption, and save the sensor energy, is very desirable.

SMEPP Light includes an Energy Efficiency module that is responsible of turning off the radio when the node does not expect to send or receive data. The Energy Efficiency module manages the radio by means of *user* duty cycles, that are the duty cycles determined by the subscribe messages, and one *management* duty cycle per group that enables the sensors in the group to exchange control messages (in particular join and subscribe messages). Each of these duty cycles defines periodic intervals when the radio should be turned on by all the sensors, and, in consequence, periodic intervals in which the radio can be kept inactive.

The Energy Efficiency module keeps the information related to all these duty cycles and determine whether the sensor can turn the radio off, by calculating the union of the activity windows produced by the duty cycles.

Figure 4.1 shows an example of how this module works, by showing the status of the radio of a node belonging to a group having two active subscriptions (`subs1` and `subs2`). The three lines on the top show the activity windows of the radio for the *user* duty cycles corresponding to the two subscriptions and for the *management* duty cycle. The line on the bottom shows the overall radio activity (portions of this line not marked by a colored bold

line, mean radio turned off). In this example `subs1` has a duration of 2 seconds, and a period of 9 seconds. `subs2` has a duration of 2 seconds and a period of 15 seconds. Finally, the *management* duty cycle has a duration of 5 seconds, and a period of 20 seconds.

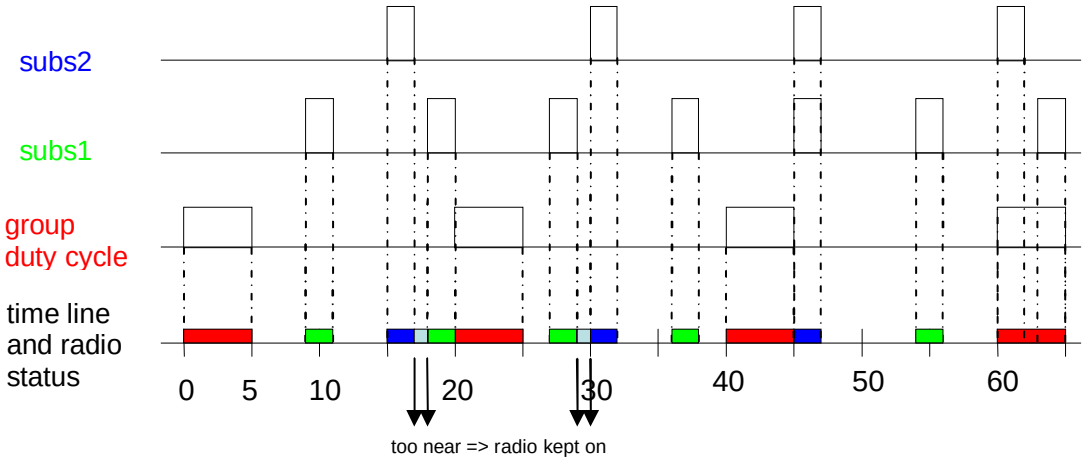

Figure 4.1: Energy Efficiency module functioning.

The Energy Efficiency module calculates the union of all the duty cycles, and decides when the radio should be turned off and on, according to two additional parameters that provide some flexibility to the system. One parameter (`tolerance`) specifies the minimum distance (in milliseconds) between the end of a window of radio activity and the start of the next one: if two windows are too close the radio is kept on until the second window ends (as happens in figure at second 17 and at second 29). The logic behind this behavior is that turning the radio on and off consumes additional energy, so two commutations of the radio waste more energy than keeping the radio on.

The other parameter (`radioDelay`) expresses the time used to delay the radio commutations to compensate the delays introduced by the transmission of the packets through a multi-hop network. In a multi-hop network, all the nodes have activity windows synchronized to begin on the same instant, and of the same duration. Routing a packet from a node to another requires some

time. The duration of such windows is set in a way that enable a message to traverse the longest path in the network and reach the destination before the ending of the window. However it can happen that, due to the delays introduced by the intermediate nodes (to store, to handle and to forward the message) duration time of the window could not suffice, and the message could not reach the destination. So each activity window is turned off a little after the expected ending time, to enable the message to reach the destination anyway.

As a side-effect of using inactivity periods to save energy, when a sensor executes the `getGroups()` primitive, it needs to keep sending request messages (polling) until one of these messages is sent during a period of activity of a group. However during the `getGroups` protocol the sensor receives enough information to synchronize with the other sensors of the group, so the next communications can happen according to the *management* duty cycle of the group. To make this approach effective, the sensors need to be synchronized, for this reason synchronization information is periodically exchanged among the peers in the group.

However, this synchronization does not avoid that application invokes a command that comprises a radio transmission when the radio is turned off, since the management performed by the Energy Efficiency module is completely transparent to the application layer. To obviate to this problem, the Energy Efficiency module provides a command to retrieve the status of the radio. Thus, SMEPP Light checks whether it is possible to start the transmission, before sending any message. In case the radio is inactive, the middleware stores the pending messages, and sends them when the radio becomes active (the Energy Efficiency module will raise the proper signal).

4.1 The Energy Efficiency component

`EnergyEfficiencyP.nc` is the component that implements the Energy Efficiency module. It stores a list (`channel_table[]`) containing the duty cycles the node is involved with, that are *management* and *user* duty cycles.

An entry of this table is composed by the following fields:

- `duration`: length of the activity window (viz., the time the radio has to be kept on).
- `interarrive`: period of the activity window (viz., the time between two successive activity windows).
- `expiry`: expiration time of the duty cycle. The *management* duty cycles never expire, so the corresponding entries are deleted from `channel_table[]` only when the peer leaves the group. *users*' duty cycles entries, on the other hand, are deleted when the time denoted by this field has passed, or when the application calls the `unsubscribe()` primitive.
- `nextON`: next instant of time in which the radio has to be turned on.
- `nextOFF`: next instant of time in which the radio has to be turned off.

The component handles two timers (`timer_radioON` and `timer_radioOFF`), one used to determine the next turning on, and the other one used to determine the next turning off. At initialization phase the radio is active, and neither of the two timer is set. When the first entry in the `channel_table[]` is added, the `timer_radioOFF` is started to fire at the `nextOFF` time of the entry added. When this timer fires (in general every time the `timer_radioOFF` fires), the component updates the `channel_table[]` by updating the `nextON` and `nextOFF` fields of every entry according to the current time (after the update all of these fields are bigger than the current time). Then the entry with the minimum `nextON` is selected and it is checked whether this time is distant enough from the current time, to avoid two consecutive commutations that are too close. If the difference between the selected `nextON` time and the current time is bigger than the `tolerance` parameter, then the radio is turned off, and the two timer are set according to the `nextON` and `nextOFF` values of the selected entry. Otherwise, the radio is kept on, and only the `timer_nextOFF` is updated.

4.1.1 Interface

The `EnergyEfficiencyP.nc` component provides an interface to handle the entries of the `channel_table[]` (viz., adding and removing entries), to set

the `tolerance` and the `radioDelay` parameters' values, and to retrieve the value of some variables stored in the module:

set

```
command uint8_t set(uint32_t length, uint32_t interarrive, uint32_t
expiry, int16_t offset)
```

When a new duty cycle is needed (either because the peer joined a new group, or because it received or performed a new subscription), application invokes the `set()` primitive to add a new entry in the `channel_table[]`. The command returns the handler to the entry, i.e. the position into the table where the entry has been added.

The `offset` parameter is the time to the next turning on of the added duty cycles. It is used to set the `nextON` field in `channel_table` to ensure the synchronization among the peers.

reset

```
command void reset(uint8_t id)
```

This command deletes the entry in the `channel_table[]` corresponding to the handler `id`. This can happen when the application invokes the `unsubscribe()` primitive, or when a peer leaves a group.

setTolerance

```
command void setTolerance(uint32_t newTolerance)
```

The application can modify the minimum distance between two subsequent radio commutations, according to the energy consumption values of the sensor nodes currently used in the network.

setRadioDelay

```
command void setRadioDelay(uint32_t newRadioDelay)
```

This command is used to change the value of the `radioDelay` parameter, that is the delay to add at the end of the activity window to turn off of the radio. This parameter is used to enable a node to receive a message that is traveling through the network, compensating the delays introduced by the storing and forwarding of the message through a multi-hop network. If the network is small, this parameter can be set to a low value, since the diameter of the network is short, and the latency for a message travelling all the network is not big. A small value of this parameter results in a lesser power consumption.

getRadioState

```
command bool getRadioState()
```

The middleware uses this command to check whether the radio is active, before sending a message. If the radio is turned off, the middleware buffers the message, and it will send it after the next `radioOn()` signal.

getNextON

```
command int getNextON(uint8_t pos)
```

This command returns the `nextON` field of the selected entry. It is used during the join protocol, by the peer handling the join request, to retrieve the time of the next turning on both for the *management* and *user* duty cycles. Sending these information to the joining peer, enables it to be synchronized with the other peers of the group.

getRadioOff

```
command uint32_t getRadioOff()
```

The join protocol comprises a long messages exchange. To successfully complete this protocol, it is needed that all the expected messages are received by the joining peer. Therefore the peer handling the join request, calls this command to retrieve the time of the next turning off, to know if the radio will remain active for a time sufficient to send all the messages.

posticipate

```
command void posticipate(uint16_t offset)
```

At the beginning of the join protocol, the peer handling the join request, checks if the radio is going to be turned off before the completion of the protocol. In this case, the middleware invokes this command to posticipate the turning off of the radio for a time sufficient to complete the join protocol.

Finally, two signals are provided by the interface: `radioOn()` and `radioOff()`, that notify respectively the turning on and off of the radio to the middleware.

4.2 Test and Simulation

The performance of the energy efficiency mechanism was evaluated by measuring the periods of radio activity of a sensor. First we performed several simulations on the TOSSIM simulator, and then we replicated some of these tests on a real sensor network. In the experiments we used 4 MicaZ motes [17] (mote1, mote2, mote3, mote4) connected in a line to form a multi-hop network, i.e. mote1 is connected to mote2, mote2 to mote3 and mote3 to mote4. Mote1 is the node that creates the group and produces the subscriptions and mote4 is the only mote that raises events. We measured the radio activity on mote2, that is on the path of all the messages directed to the event subscriber (mote1).

We repeated five sets of experiments with a number of subscriptions ranging from 0 to 4. Each experiment was repeated 10 times and ran for 180 seconds. The length of the activity window of each subscription was 2 seconds, that is a time large enough to permit the events to reach the subscriber,

while each rate was set randomly in a range from 7 seconds, to 25 seconds. In all cases, the *management* duty cycle is fixed with a rate of 20 seconds and a duration of 5 seconds.

For each experiment we measured the average period of time of the radio's state of sensor mote2. The transceiver of MicaZ has four different states, each with its value of power consumption:

- *down* (radio off): 0,02 mA.
- *idle* (radio on, but not used): 0,426 mA.
- *receive* (radio on, used for the receipt of a message): 18,8 mA.
- *send* (radio on, used in sending a message): up to 17,4 mA.

From these data we computed the average energy consumption of sensor mote2 (expressed in mA-hr) in all the set of experiments, as shown in Figure 4.2. For a comparison the figure reports the energy consumption estimated with the TOSSIM simulator and the energy consumption in the case where the energy efficiency module is disabled (viz., radio always active). The figure shows that the energy efficiency strategy enables significant energy saving, and that the energy consumed grows sublinearly with the number of subscribes.

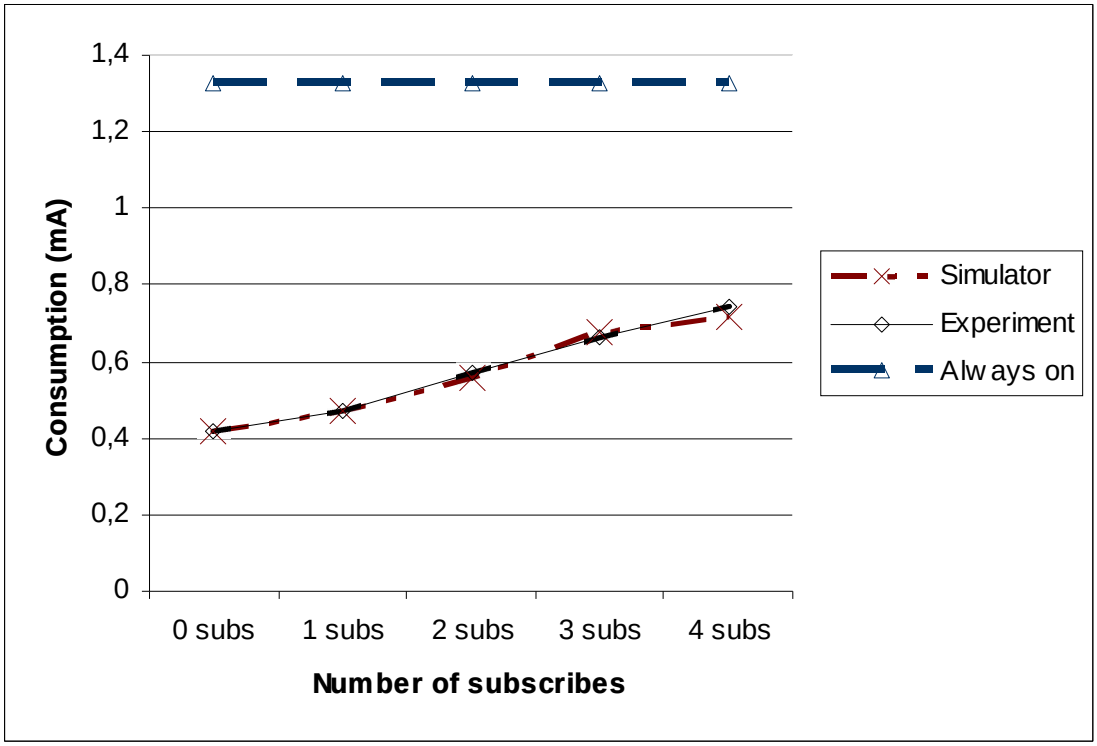

Figure 4.2: Energy consumption (in mA-hr).

Chapter 5

Conclusions and Future Works

The purpose of the thesis was the realization of SMEPP Light, a middleware for wireless sensor networks, that supports the development of applications, by hiding the complexity of the underlying infrastructure, and by providing security services and energy saving mechanisms.

We described the main requirements that are at the base of the SMEPP Light design, namely the sensors' organization in terms of groups, the security model also based on groups, the query injection and data collection based on a subscribe/event model, and the energy efficiency strategy.

At the current state of development, the security module is under development (since the primitive security mechanisms are under development by other parties of the SMEPP project), but the component-based design of SMEPP Light grants an easy integration of fully-working security modules.

We performed several simulations with TOSSIM to prove both the correctness and the efficiency of SMEPP Light, and we confirmed the simulation results by means of tests executed on the MicaZ and IRIS motes. We performed some preliminary measurement on the performance of the energy efficiency strategy, showing that SMEPP Light can significantly contribute to energy savings.

Future works include the extension of SMEPP Light to a service oriented interaction model, in order to make the middleware more flexible and more compliant with the SMEPP specification.

Concerning the security, the cryptography component needs to be tested,

and a mechanism for the dynamic refresh of the session keys can be added to strengthen the robustness of the group-level security. In addition, a session key at the network level could be introduced, even though this would make the middleware more complex, but it would give to the network-level security the same robustness of the group-level security.

Acknowledgements

Grazie

List of Figures

Bibliography

[1] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Capirci. *"Wireless sensor networks: a survey"*. Computer Networks, N. 38, 2002, pp. 393-442.

[2] Antonio Brogi, Stefano Chessa, Angel Dianes, Manuel Diaz, Razvan Popescu, Bartolomé Rubio. *"Service and Interaction Models for Networked Embedded Peer to Peer Systems: A Survey"*. (SMEPP), http://www.smepp.org

[3] Paolo Baronti, Prashant Pillai, Vince Chook, Stefano Chessa, Alberto Gotta, Y. Fun Hu. *"Wireless Sensor Networks: a Survey on the State of the Art and the 802.15.4 and ZigBee Standards"*. In: Computer Communications, 30 (7), 26 May 2007, pp. 1655-1695.

[4] Michele Albano, Antonio Brogi, Razvan Popescu, Manuel Díaz, José A. Dianes. *"Towards Secure Middleware for Embedded Peer-to-Peer Systems: Objectives & Requirements"*. Ubicomp 2007, Innsbruck, Austria, 16 Settembre 2007.

[5] Claudio Vairo, Michele Albano, Stefano Chessa. *"A Secure Middleware for Wireless Sensor Networks"*. Middleware for Mobile Embedded Peer-to-Peer Systems, First International Workshop, 21 - 25 July 2008, Trinity College, Dublin, Ireland.

[6] Claudio Vairo. *"Realizzazione e Sperimentazione di Protocolli di Rete in Reti di Sensori"*. Bechelor Degree Thesis, 2005, University of Pisa.

[7] P. Costa, G. Coulson, C. Mascolo, G. Pietro, S. Zachariadis, *"The RUNES Middleware: A Reconfigurable Component-based Approach to*

Networked Embedded Systems". Third IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS 06), 9 - 12 October 2006, Vancouver, Canada.

[8] EU FP6 Strep project *"Secure Middleware for Embedded Peer-to-Peer Systems"* (SMEPP), http://www.smepp.org

[9] C. Giani et. Al. *"Data collection in Sensor Networks: the TinyLime Middleware"*. J. of Pervasive and Mobile Computing, 4(1), pp. 449469

[10] M. Díaz et. Al. *"A Coordination Middleware for Wireless Sensor Networks"*. IEEE SENET 05, Montreal, Aug 05, pp. 377382.

[11] E. Soutoet. et Al. *"A Message-Oriented Middleware for Sensor Networks"*. MPAC 04, Toronto, Oct. 04, pp. 127134,

[12] ZigBee Specifications 2006, http://www.ZigBee.org

[13] C. Intanagonwiwat, R. Govindan, D. Estrin. *"Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks"*. MobiCom 2000, Boston, pp. 5667

[14] Giuseppe Amato, Paolo Baronti, Stefano Chessa. *"MaD-WiSe: Programming and Accessing Data in a Wireless Sensor Networks"*. IEEE Eurocon, Belgrado, Serbia-Montenegro, November 2005.

[15] S. Madden, M. J. Franklin, J. M. Hellerstein, W. Hong. *"Tinydb: an acquisitional query processing system for sensor networks"*. ACM Trans. Database Syst., 30(1):122-173, 2005

[16] Giuseppe Amato, Stefano Chessa, Fabrizio Conforti, Alberto Macerata, Carlo Marchesi. *"Health Care Monitoring of Mobile Patients"*, ERCIM News No. 60, January 2005.

[17] Crossbow motes. http://www.xbow.com/Products/wproductsoverview.aspx

[18] SunSPOTWorld. www.sunspotworld.com

[19] Sun Squawk http://research.sun.com/projects/squawk/

[20] Contiki http://www.sics.se/contiki/

[21] TinyOS, NesC, Tossim documentation. http://www.tinyos.net/tinyos-2.x/doc/

[22] Philip Levis and Nelson Lee. *"TOSSIM: A Simulator for TinyOS Networks"*. pal@cs.berkeley.edu September 17, 2003.

[23] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, David Culler. *"The nesC Language: A Holistic Approach to Networked Embedded Systems"*. http://nescc.sourceforge.net

[24] David Gay, Philip Levis, David Culler, Eric Brewer. *"nesC 1.1 Language Reference Manual"*. May 2003.

[25] Al Kelley, Ira Pohl. *"C Didattica e Programmazione"*. Addison-Wesley, 1996. cap. 9, pp. 345-365.

[26] David C. Steere, Antonio Baptista, Dylan McNamee, Calton Pu, Jonathan Walpole. *"Research Challenges in Environmental Observation and Forecasting Systems"*. (MobiCom 2000), pp. 292-299, Boston, MA, USA August 2000.