POLITECNICO
DI TORINO

POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Design and evaluation of an energy-autonomous wireless sensor network

Supervisors
Prof. Andrea Acquaviva
Ing. Michelangelo Grosso

Candidate
Francesco Giavatto

Academic year 2016-2017

Summary

Recent technology improvements in wireless communications, in digital electronics and in micro-electromechanical systems (MEMS) make feasible the design and development of low-power, low-cost multifunctional "nodes" able to interact with each other. This new kind of devices, which include sensing, actuation, data processing and communication, allows the creation of increasingly pervasive wireless networks, enabling and expanding the concept of *Internet of things* (IoT). IoT refers to the inter-networking of physical devices, vehicles, buildings and many other items or objects that embedding some electronics makes them *smart*. Smart in the sense of something able to be part of the environment, reacting to stimuli or actively actuating them, and collecting and exchanging data. From its dawning, the potentiality of such concept was clearly enormous and, despite the actual diffusion, growth potential is still huge. As a natural consequence, the IoT concept can easily take advantage from Wireless Sensor Networks (WSNs). A Wireless Sensor Network is based on the interaction of a large number of nodes of the same type, or also with different characteristics, typically with sensing purposes only. The integration of WSNs into the IoT unleashes an extremely wide range of possible applications and, not surprisingly, IoT and WSN are becoming an integral part of our lives.

In these areas, thanks to the considerable amounts of research efforts of the last years, we have available several sophisticated and extremely efficient communication protocols and software instruments, and some emerging standards enable interoperation within them (e.g., IEEE 802.15.4). Often WSNs are composed of a very large number of sensors nodes, placed in vastly different areas with different requirements in terms of reliability and sensing capabilities. The creation of a robust infrastructure, tailored to specific requirements, can be challenging and needs a careful analysis to find the best solution using the available hardware and software components or developing ad-hoc devices and strategies. These problems are further emphasized when the systems need to be positioned in inaccessible terrains or in conditions where human activity are very limited or not possible at all.

This work has the aim of defining a framework for the development and the evaluation of ambient monitoring wireless sensor networks with low data-rate, based on

an embedded operating system supporting 6LoWPAN (Contiki) and targeting extremely low-power consumption. The flexibility of the platform and of the firmware gives the opportunity to customize network parameters in order to better match application requirements. Within this framework, an experimental case study was developed and empirically validated.

The developed network is composed of a self-forming and self-healing *multi-hop mesh* network for ambient conditions monitoring. The nodes can be entirely powered by a solar energy harvester even in indoor environments not in direct sunlight. The data acquired by each node are collected by a *sink*. The sink is connected to the Internet and provides the Internet access to the other nodes of the network.

The hardware of a node is composed of off-the-shelf devices by STMicroelectronics® belonging to the STM32 Nucleo ecosystem. Each node consists of:

- the core board integrating a low-power microcontroller (ARM® Cortex®-M3) with several integrated peripherals;
- an ultra-low power, Sub-GHz radio module that guarantees the wireless communication;
- an expansion board integrating several sensors for monitoring humidity, temperature and pressure;
- a power module equipped with solar cells and a rechargeable battery.

The first part of my work aimed at characterizing the solar energy harvester used for supplying the system. In order to effectively measure the power output capabilities of the module, a test circuit was specifically implemented to perform the task. In addition to the characterization of the power module, this analysis was used to define some profiles for the ambient light of a typical office and to understand which are the limits and the expected values for the energy that can be collected from the environment.

Then, I focused on the energy consumption of the node, in particular considering the power required by the microcontroller. The several available low-power modes were used to minimize the energy consumption during both active and idle phases, exploiting voltage and frequency scaling as well as selective core and peripheral deactivation. The radio module resulted very critical from the power consumption point of view. In order to guarantee the functioning of the system using the solar harvester, a duty cycling mechanism for the nodes was adopted. It consists in allowing the nodes to go in a low-power mode where the system is frozen, and periodically waking them up for a small amount of time sufficient to acquire, process and send data to the sink, and to maintain the network connectivity. The amount of time in which the system stays on and off can be easily configured to match application requirements and power supply capabilities.

The duty cycling mechanism requires to precisely synchronize nodes to guarantee the consistency of the network and make it work correctly. The time distribution between nodes required some attention so as not to generate excessive traffic on the communication channel but, at the same time, to ensure a sufficient level of synchronization (in the order of few tens of milliseconds).

In parallel with the hardware implementation, all the concepts related to the network were tested using a network simulator that was customized to accurately replicate the behavior of the system. The simulation, based on the Cooja software, was used, at first, to test the goodness of the mechanism adopted for the network management before deploying them in the real node. Then, the software was used to validate the final version of the firmware by analyzing long time periods, unpractical to be tested experimentally.

The average output power of the scavenger during a day is in the order of $180\mu W$. This allows the system to acquire data every 20 minutes relying only on the harvester energy, with a surplus of a few μW useful to compensate small variations in daily light conditions.

Keeping under control overheads, delays and latencies in a heavy constrained system is not a trivial task, as well as guaranteeing reliability in connections between nodes. This work demonstrated the feasibility of an energy-autonomous system and will support the development of WSNs based on the STMicroelectronics® devices. Further development is currently directed to an improvement of the low-power property of the radio module by working on the firmware and driver integration and applying a fine-grained duty cycling. Reducing the power absorbed by the radio during the listening phase is key for enabling an always-on network continuously accessible via the IPv6 protocol.

Contents

Chapter 1

Introduction

Recent technology improvements in wireless communications, in digital electronics and in micro-electromechanical systems (MEMS) enable the design and development of low-power, low-cost multifunctional sensor nodes able to interact with each other. This new kind of devices, which include sensing, actuation, data processing and communicating, allows the creation of increasingly pervasive wireless networks, enabling and expanding the concept of *Internet of things* (IoT). IoT refers to the inter-networking of physical devices, vehicles, buildings and many other items or objects that embedding some electronics makes them *smart*. Smart in the sense of something able to be part of the environment, reacting to stimuli or actively actuating them, and collecting and exchanging data[1]. From its dawning, the potentiality of such concept was clearly enormous and, despite the current diffusion, growth potential is still huge. The IoT concept greatly contributed to the diffusion of Wireless Sensor Networks (WSNs) development. A Wireless Sensor Networks is based on the interaction of a large number of nodes of the same type or also with different characteristics typically with sensing purposes only. Simultaneously to the IoT, the range of possible applications is extremely wide and WSNs are becoming an integral part of our lives.

Thanks to the considerable amounts of research efforts of the last years, we have available several sophisticated and extremely efficient communication protocols and software instruments. They are fundamental in the implementation and deployment of a WSNs. Often WSNs are composed of a very large number of sensors nodes, placed in vastly different areas with different requirements in terms of reliability and sensing capabilities.

The creation of a robust infrastructure, tailored to specific requirements, can be challenging and needs a careful analysis to find the best solution using the available hardware and software components, or developing ad-hoc devices and strategies.

WSNs characteristics make them very suitable for deployment in inaccessible terrains or in conditions where human activity are very limited or not possible at

all. Routing protocols provide a self-forming and self-healing mechanisms for the networks that become fundamental for node sensors randomly and densely deployed.

In traditional networks, performances are often intended in terms of throughput and delay so routing and communication protocol are designed to improve these metrics typically at the cost of a very high power consumption and an important processing power required. WSNs shows very different constraints, hence protocols primarily focus on power conservation and on keeping algorithms as simple as possible. The deployment of WSNs is another factor that is considered in developing protocols. The position of the sensor nodes does not need to be engineered or predetermined. This allows random deployment, e.g., in inaccessible terrains or disaster relief operations. On the other hand, this random deployment requires the development of self-organizing protocols for the communication protocol stack.

All these properties of WSNs present unique challenges for the development of communication protocols, but on the other hand open an extremely wide range of applications from the military ones to the ambient monitoring, healthcare, domotics and industrial application for example in production processes.

Looking at the extremely wide constraints and properties that should be managed and the heterogeneity in the sensor platforms, the difficulty to unify everything should not be surprising. Standardization becomes a major issue in order to improve quality and performance and above all reusability and interoperability. In this respect, IEEE 802.15.4 was formed with the aim to provide a standard physical layer and media access control for *Wireless Personal Area Networks* (WPANs). A WPAN is a computer network used for data transmission among devices with low-range wireless transceiver technology with long battery life and very low complexity. WPAN extension typically varies from a few centimeters to a few meters. For low data-rate networks a more accurate distinction identifies the *Low Rate Personal Area Networks* (LR-WPANs). IEEE 802.15.4 is extended with upper layers by several other specifications such as ZigBee, ISA100.11a, WirelessHART, MiWi, SNAP, and Thread specifications. It can be used also with 6LoWPAN to deliver the IPv6 version of the Internet Protocol (IP) over WPANs.

All the above-mentioned properties and features imply a parallel discussion on the power supply requirements of these devices. For example, the deployment of nodes in the environment can be independent of the availability of power source in the environment itself. This is typically enabled by using batteries with a reasonable capacity for the application or by providing energy harvesting tools to the nodes to allow a form of self-power supply. Others applications may require a strict condition

on the size of the device including batteries. In this case, the trade-off between battery size and battery duration can benefit the extremely low power behavior reached with currently available technology. Merging this two aspects, long battery duration and small-scale dimensions for the devices represent one of the major challenges in the IoT scenario.

This work deals with the analysis on power consumption contributors, the way in which they impact performance and functionalities. These analyses aim to the theorization of some viable strategies to the design of an energy-autonomous wireless sensors network and to the physical realization and validation of a working system adopting one of the discussed strategies. Analyzing the way in which the power is used by the system is the first step to do in order to clearly understand the direction in which to focus attention and try to adopt effective countermeasures or optimizations. The performance of the best solution typically collides with other requirements and trade-off should be evaluated to find an optimal behavior. Considerations of these aspects are actualized by implementing them on hardware using tools and devices provided by the STMicroelectronics®.

The development of an energy-autonomous wireless sensors network is developed starting from off-the-shelf components by STMicroelectronics®. A solar panel is used as energy harvester in conjunction with a small battery that guarantees power continuity over the whole day. The main challenge of the work consists of matching power requirements of the nodes and power supply capabilities of the harvester by properly tuning network features and exploiting as much as possible low power capabilities of the node. Keeping under control overheads, delays and latency in a heavy constrained system may be not so trivial as well as guaranteeing a high reliability in connections between nodes.
The literature provides good starting points for an effective development as well as exploring the backgrounds and the working principles of related protocols and algorithm help to better deal with the problem in a structured manner. In this regard, the next chapter will explore the background of WSNs giving an overview of the conceptual organization of nodes and the routing mechanism that allows inter-node communication. In all the battery-operated applications the power consumption of microcontrollers and radio is a crucial point to investigate. Peripheral components like the radio contribute even more to the whole power consumption and some technique developed by researchers and engineers to reduce it are exposed.

Chapter 3 will show an overview of the system under development describing the crucial point from a conceptual point of view, whereas in chapter 4 all the implementation details are analyzed and described in detail with listings and figures. The firmware development works in conjunction with a software emulation of the system, discussed in chapter 5. In this way, problem analysis and algorithm improvement

could benefit from a previous check or on the contrary after the software deployment a long time simulation could validate the expected result or highlight criticalities hard to detect on the physical device. Results analysis and conclusion will conclude this work.

The development is based on the STMicroelectronics® previous experience on this branch and on its interest in providing reliable and affordable solution in the IoT panorama. The choice of using a set of off-the-shelf components not already forming a full featured node is driven by the will of adopting components that can be easily retrieved and are almost independent of the specific application and hardware equipment provided by the customer. The resulting system tries to be as much as possible customizable and expandable with other features and component. Moreover, STMicroelectronics® products are designed specifically to be a springboard for advanced semi-custom design.

Chapter 2

Background

The design of WSNs requires ample knowledge of a wide variety of research fields including wireless communication, networking, embedded systems, digital signal processing, and software engineering. This is motivated by the close coupling between several hardware and software entities of wireless sensor devices as well as the distributed operation of a network of these devices. Consequently, several factors exist that significantly influence the design of WSNs[2].

A node, the basic element of every WSN, can be sketched with several building blocks in which the main elements are highlighted. Figure 2.1 shows a generic overview of a node divided into the main functional blocks[3].

The simplicity of the schematics reflects the bareness of the system where only the essential module are implemented. The sensors and actuators block may include a very huge variety of sensing devices with different specification and characteristics. They can have both a digital or analog interface to the microcontroller. Common communication protocol for embedded systems, such as CAN, I^2C or SPI, are typically supported by the microcontroller and used for interfacing digital sensors adopting one of them. For analog sensors, a conversion from the analog domain to the digital should be performed. The conversion is typically performed by an ADC peripheral embedded in the microcontroller. If the microcontroller does not provide an Analog-to-Digital Converter between its peripheral, or the embedded one is not sufficiently precise, an external converter can be added in the chain.

The power conversion module is responsible to provide power supply to all the modules of the system. Its output voltages must match all the specifications of each module, hence several voltage domains should be managed. In fact, it may happen that the required voltage for a module is different from the others, for sure it is not the best case but can be managed without particular issues. Moreover, it has to be able to autonomously switch between the available sources in order to satisfy the load requirements. The scavenged energy typically is not so abundant so a high level of efficiency for the converter is required along with a storage mechanism that

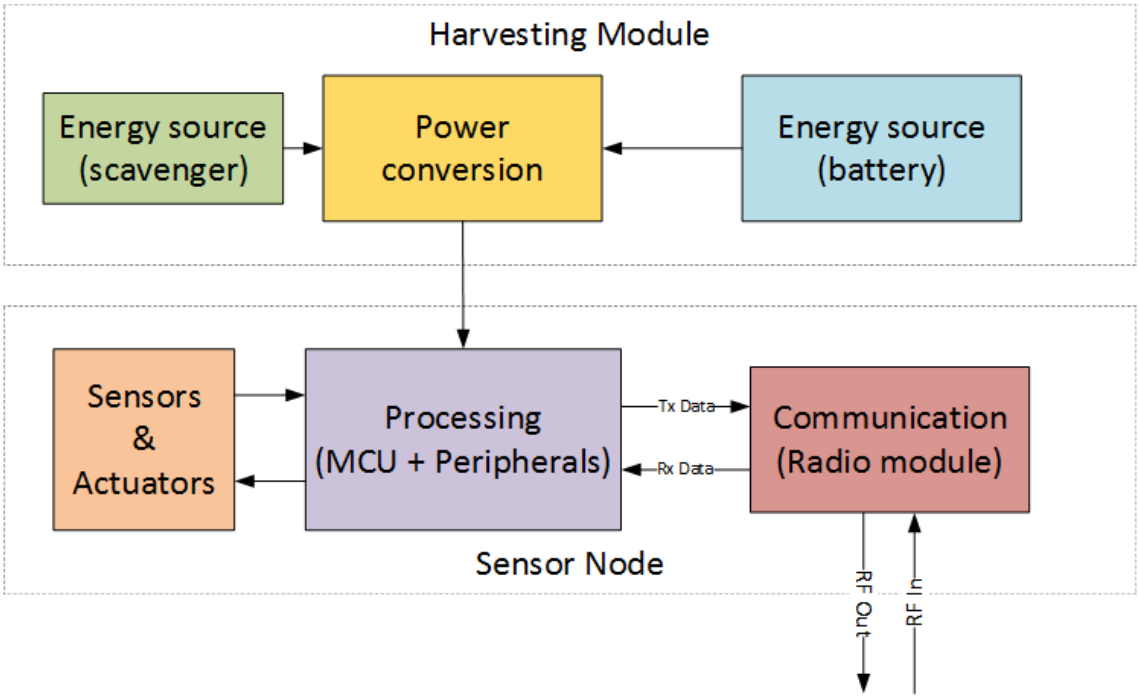


Figure 2.1: Building blocks of a generic node. The supply module may include or not a scavenger.

allows to not waste surplus energy but save for future needs.

In this chapter, all the major factors in the development of a WSN are described including: an overview of the networking aspects like network topology and routing protocol principles, requirements of the operating systems with a brief description of the ContikiOS operating system for WSN, and a review on power consumption factors and some technique typically adopted to reduce them.

2.1 Networking

Starting from an abstract view of the network, this section shows the topology that a Wireless Sensors Network can assume highlighting the benefits that one topology can take with respect to the others and vice versa. Then, the aspects related to the physical layer of the communication between nodes are discussed along with the technique to access the channel and the relative problems. The last two sections deal with two problems related to the network management, in particular routing and time distribution.

2.1.1 Network Topology

A Wireless Sensor Network can assume several network topologies spacing from a very simple star network to an advanced multi-hop wireless mesh network. These two topologies have different characteristics and present advantages and disadvantages depending on the features required by the application.

Star network is one of the most common computer network topologies. It consists of a tree with one internal node and k leaves (see figure 2.2a). The central node typically acts as a bridge toward another network or the Internet and represents the sink of the network, that is the node in charge of collecting the data coming from other nodes. All the *leaf* nodes are able to exchange messages only with the central node and they are directly connected with it through a wireless link. This configuration simplifies a lot the network since routing is not necessary. However, this simplicity presents a substantial drawback. The network spatial extension is strongly limited by the wireless transmission range of both nodes and router. During deployment, the distance between nodes and sink should not exceed its maximum. Due to the power consumption and size bounds, the node transmitting power is usually constrained and, in some circumstances, ranges can be too limiting. A remarkable network specification using this topology is the LoRaWAN™. It is based on the *LoRa* modulation technology and it is intended for wireless battery operated nodes of a *Low Power Wide Area Network* (LPWAN)[1]. Its typical architecture is laid out in a star-of-stars topology in which gateways are a transparent bridge relaying messages between end-devices and a central network server in the backend. Gateways are connected to the network server via standard IP connections while end-devices use single-hop wireless communication to one or many gateways[4]. *Sigfox* technology represents another example of one-hop star topology in LPWAN, unlike LoRa, Sigfox is based on *Ultra Narrow Band* modulation technology.

A mesh network is a computer network topology consisting of a large number of nodes randomly[2] placed more or less close to each other. As opposed to the star topology, nodes can exchange messages with each other. This allows what is known as *multi-hop* communications, that is, if a node wants to send a message to another node that is out of radio communications range, it can use an intermediate node to forward the message to the desired node. Depending on the number of connection we can differentiate *fully connected* meshes, where each node is directly connected

[1]LPWAN and LR-WPAN mainly differs in the spatial scope: the reach of a PAN typically extends to 10 meters, whereas a WAN covers large area such as city, country and even more.

[2]Physical position of nodes does not need to be statically predetermined. Each node can be positioned without constraints except for their wireless range.

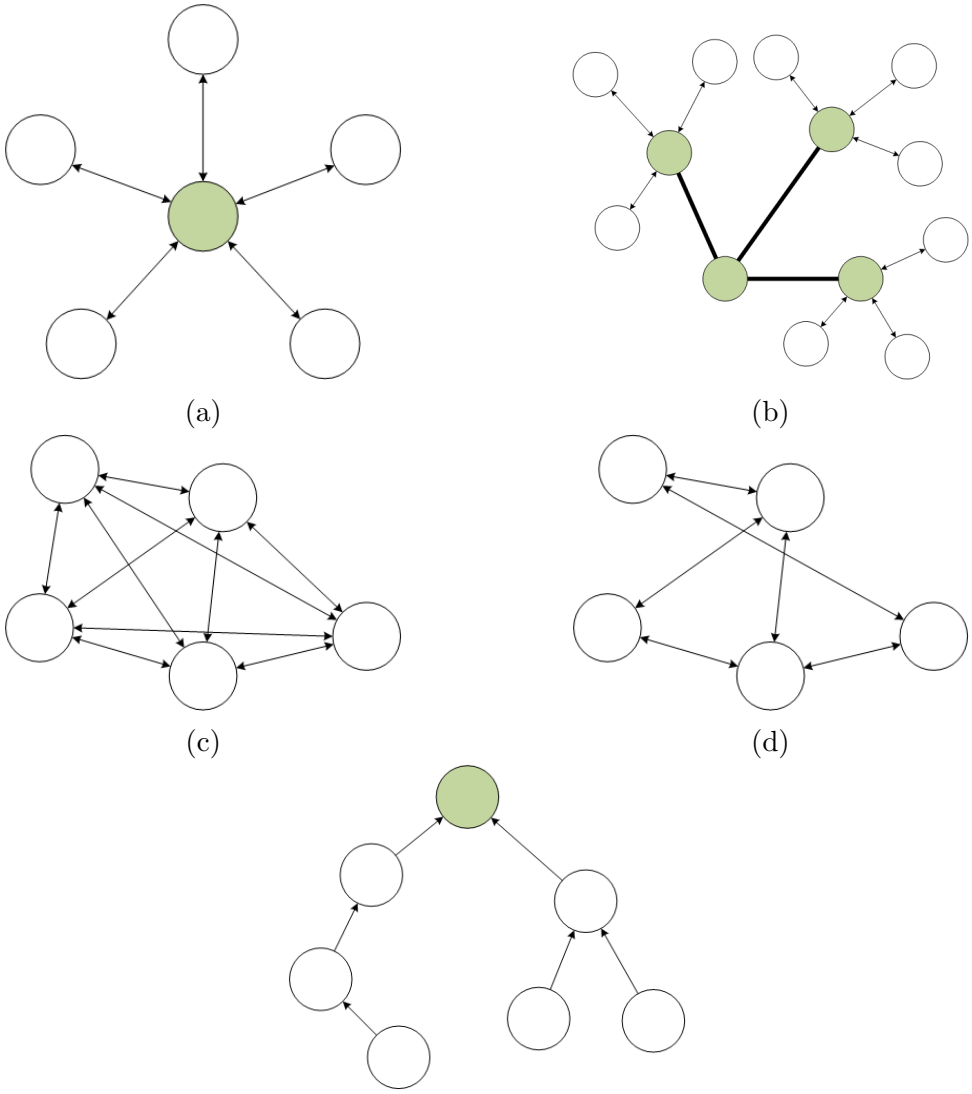


Figure 2.2: Network topology examples: a) shows a simple star topology; b) shows a star of stars topology, where hubs are typically wired and communicate through the IP; c) and d) show respectively a full and a partial mesh network; e) shows a rooted tree-like mesh (DODAG).

to all the others, and *partial* meshes, where only some connection is established. Fully connected networks have the advantages of security and reliability but the complexity of the network grows rapidly as the number of nodes increases. Partial meshes tend to preserve the already said advantages but trying to keep complexity affordable. Mesh network topology has the advantage of redundancy and scalability. If an individual node fails, a remote node still can communicate to any other node in

its range, which, in turn, can forward the message to the desired location. In addition, the range of the network is not necessarily limited by the range between single nodes; it can simply be extended by adding more nodes to the system. The disadvantage of this type of network relates with the power consumption of the nodes that implement the multi-hop communications are generally higher than for the nodes that don't have this capability, often limiting the battery life due to the increased number of messages sent. Additionally, as the number of communication hops to a destination increases, the time to deliver the message also increases, especially if low power operation of the nodes is a requirement. In WSNs environment, as in star topology, one node (or few of them) plays the role of the *sink* and it is in charge of collecting data coming from the network. Often it is physically connected to a DSL router with Internet access. In this configuration, the Internet access is shared by the sink node with the whole network and it acts as a border router. The sink node may be also the one responsible for the network configuration. Client nodes can act as source only or as routers. In the first case, the node is only able to send data to the sink but is not able to forward messages from other nodes. On the contrary, router nodes are able to forward messages and actively participate in the creation of routes to the sink. Both types of node autonomously connect to their neighbors selecting a path that, through other nodes, can reach the sink. The sink node can be considered as the *root* of the resulting graph. The way in which nodes establish links with its neighbors is defined by the routing protocol implemented and can generate different graph structures. Tree-like meshes are widely used thanks to their low complexity, compared with other mesh structures. They organize the network as a *Destination Oriented Directed Acyclic Graph* (DODAG), a graph that is a set of vertices connected by edges, where the edges have a direction associated with them and no direct cycles are present. More complex mesh structures group node into clusters assuming hybrid configurations. Since the communication between two node relies on other nodes functionality, each node contributes to a self-healing mechanism that guarantees to restore communication in case that a node becomes inaccessible due to congestions on the wireless channel or for a failure. Self-healing represent an additional benefit with respect to the star topology, but in case of a single sink, even in mesh network it represents a *single point of failure*[3] for the network.
We will focus mostly on the tree-like mesh topology.

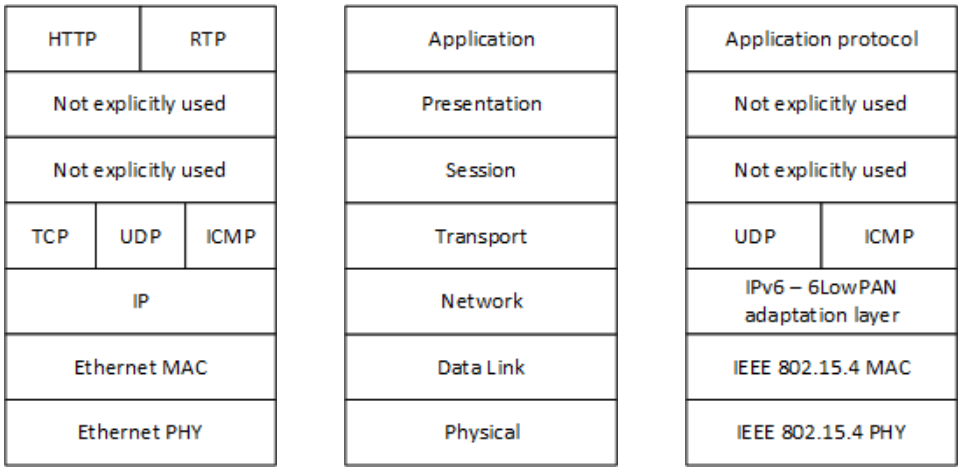
2.1.2 Physical and MAC Layers

The construction of a wide sensor network with nodes densely scattered in a sensor field presents an issue related to the sharing of the communication channel between

[3]A single points of failure, in network science, is any network element that, if it fails, it takes out communication with a section of the network.

several nodes that can result in collision. Collisions are caused by two nodes sending data at the same time over the same transmission medium. To address this problem, a sensor network must employ a *Medium Access Control* (MAC) protocol to arbitrate access to the shared medium ensuring reliable point-to-point and point-to-multipoint connections and to fairly and efficiently share the limited bandwidth resources among multiple sensor nodes.

In figure 2.3 the conceptual stack of the 6LowPAN layers is compared with the ISO/OSI standard model. The similarity with the TCP/IP stack is almost visible, since several concepts of the 6LowPAN protocol are derived from the more complex TCP/IP stack.



(a) TCP/IP stack (b) ISO/OSI stack (c) 6LowPAN stack

Figure 2.3: Comparative schema of TCP/IP and 6LowPAN stack with respect to ISO/OSI model.

For example, medium access control (MAC) has been extensively studied for traditional wireless network, like *Wireless Local Area Networks* (WLANs). Consequently, several methods are available and, depending on the approach adopted, they can be classified into three main classes:

- *contention-based medium access*, if the use of the channel is contented between clients at the same time and an arbitration mechanism design the owner of the medium;
- *reservation-based medium access*, if the access to the channel is regulated in time frame statically or dynamically assigned to clients;
- *hybrid solutions* that merge characteristics of the previous two schemes.

Common medium access methods are: *carrier sense multiple access* (CSMA), *time division multiple access* (TDMA), *frequency division multiple access* (FDMA) and

code division multiple access (CDMA). These protocols do not take into account the unique characteristics and limitations of sensor networks, hence, traditional MAC protocols cannot be applied directly to sensor networks without modification. For example, schemes like CDMA and FDMA are generally not employed in WSNs, whereas CSMA and TDMA are largely adopted, but with several adaptations.

The most used technique in sensor networks is based on *Carrier-Sense Multiple Access* (CSMA) mechanism that has been introduced for WLANs. This scheme belongs to the contention-based medium access class. As the name suggests, the node *senses* the channel listening for a specific amount of time to evaluate the activity on the channel. If the channel is busy the transmission is delayed by a random amount of time referred to as *backoff*.

CSMA protocol offers four access modes of the medium:

- 1-persistent: when the transmitting node is ready to transmit, it senses the channel for idle or busy. If idle, then it transmits immediately. If busy, then it senses the channel continuously until it becomes idle, then transmits the message unconditionally. In case of a collision, the sender waits for a random period of time and attempts the same procedure again;

- Non-persistent: when the transmitting node is ready to transmit data, it senses the channel for idle or busy. If idle, then it transmits immediately. If busy, then it waits for a random period of time (during which it does not sense the channel) before repeating the whole cycle again. This approach reduces collisions and results in overall higher channel throughput but with a penalty of longer initial delay compared to 1-persistent;

- P-persistent: this is an approach between 1-persistent and non-persistent CSMA access modes. When the transmitting node is ready to transmit data, it senses the channel for idle or busy. If idle, then it transmits a frame with probability p. If busy, then it senses the channel continuously until it becomes idle, then transmits with probability p. If the node does not transmit (the probability of this event is 1-p), it waits until the next available time slot. If the channel is still not busy, it transmits again with the same probability p. This probabilistic hold-off repeats until the frame is finally transmitted or when the channel is found to become busy again. In the latter case, the node repeats the whole logic cycle (which started with sensing the channel for idle or busy) again;

- 0-persistent: each node is assigned a transmission order by a supervisory node. When the channel goes idle, nodes wait for their time slot in accordance with their assigned transmission order. The node assigned to transmit first transmits immediately. The node assigned to transmit second waits one time slot

(but by that time the first node has already started transmitting). Nodes monitor the channel for transmissions from other nodes and update their assigned order with each detected transmission.

Some variations of the protocol have been proposed which introduce extensions to the CSMA aimed at improving performance. The most relevant are the collision detection mechanism (CSMA/CD) and the collision avoidance (CSMA/CA). The collision detection tries to improve performance by terminating transmission as soon as a collision is detected, shortening the time required for the retransmission. It adopts a 1-persistent CSMA access mode. The collision avoidance, on the other hand, tries to ensure the use of the channel by sending a small packet to reserve the wireless channel before starting the data transmission in order to avoid at most the probability of collisions. The CSMA/CA protocol adopts a p-persistent access mode.
CSMA/CA technique has the disadvantage of requiring nodes to continuously sense the channel for inactivity resulting in a significant energy consumption. Several energy-aware protocols have been developed starting from the CSMA/CA with different characteristics and drawbacks: S-MAC, B-MAC, CC-MAC and DSMAC, to name but a few[2].

For what concerns the *Reservation-based medium access* protocols, time division multiple access (TDMA) is the reference protocol for several medium access methods adopted for WSNs. It is based on a time-division multiplexing where for one receiver there are multiple transmitters. Examples of reservation-based medium access protocol derived from TDMA for WSN are among the others: TRAMA, PMAC and BMA-MAC[5].

2.1.3 Routing Protocol

Once nodes are identified, routing protocols are in charge of constructing and maintaining routes between distant nodes and withstand failures that may affect the network. Star or other simpler topologies typically do not present criticalities with routing. On the contrary, mesh topologies require a consistent, and desirably energy-aware, routing mechanism. Over the last few years research has led to the development of several routing protocols for wireless sensors network. Each of them is tailored to different requirements that make each protocol appropriate for certain applications. Traditional routing protocols can be classified into three main classes according to the manner in which information is acquired and maintained and the manner in which this information is used to compute paths based on the acquired information:

- *Proactive strategy*, or table-driven, relies on maintaining fresh lists of destinations and their routes by periodically distributing routing tables throughout the network;
- *Reactive strategy*, or on-demand, relies on a dynamic route search to establish paths between a source and a destination. This typically involves broadcasting a route discovery query, with the replies traveling back along the reverse path;
- *Hybrid strategy* combines the advantages of proactive and reactive routing. The routing is initially established with some proactively prospected routes and then serves the demand from additionally activated nodes through reactive flooding.

Another way to classify routing protocol concerns the architecture adopted and the relative routing strategy[6][7]:

- *Flat architecture*: each node plays the same role as peer and sensor nodes collaborate together to perform the sensing task;
- *Hierarchical architecture*: nodes are organized in clusters in which some specific nodes, e.g. the ones with higher energy, assume the role of cluster head. The cluster head is responsible for coordinating activities within the cluster and forwarding information between clusters;
- *Location-based architecture*: the position of the node within the geographical coverage of the network assumes a relevant role for the query issued by the source node. Such a query may identify a specific area where a phenomenon of interest may occur addressing all the nodes near to the specified location.

RPL

A remarkable routing protocol optimized specifically for WSN is the *IPv6 Routing Protocol for Low-Power and Lossy Networks*[8]. IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL pronounced as *ripple*) is a routing protocol for wireless sensor networks consisting of constrained nodes (with limited processing power, memory and energy) typically interconnected by lossy and unstable links supporting only low data rates. As the name suggests it brings IPv6 functionality over the WSN allowing the integration of the network with the Internet. This takes several benefits to the WSN capabilities but at the cost of some new requirements on the nodes like the implementation of an IP stack that typically increases the single node complexity. RPL protocol is specified by the Internet Engineering Task Force (IETF), a large open international community of network designers, operators, vendors, and researchers concerned with the evolution of the Internet architecture and the smooth operation of the Internet. The IETF mission is to produce high quality,

relevant technical and engineering documents that influence the way people design, use, and manage the Internet in such a way as to make the Internet work better. RPL provides a mechanism whereby multipoint-to-point traffic from devices inside the network towards a central control point as well as point-to-multipoint traffic from the central control point to the devices inside the network are supported. Support for point-to-point traffic is also available.

RPL organizes nodes as a Directed Acyclic Graph (DAG) (refer to section 2.1 for details on mesh topologies) that is partitioned into one or more Destination Oriented DAGs (DODAGs), one DODAG per sink (see figure 2.2e).

Communications in the network can be "Up" and "Down" specifying the direction from leaf nodes towards DODAG roots and from DODAG roots towards leaf nodes, respectively. This follows the common terminology used in graph and depth-first-search.
For the Downward traffic RPL supports two modes: Storing and Non-Storing mode. In both cases, packets travel Up toward a DODAG root then Down to the final destination (unless the destination is on the Upward route). In the Non-Storing case, the packet will travel all the way to a DODAG root before traveling Down. On the contrary, in the Storing case the node keeps tracks in a look-up table of node reachable in its Downward routes. In this case, a packet may be directed Down towards the destination by a common ancestor of the source and the destination so a packet does not need to reach a DODAG root. This typically reduces traffic on links close to the root, at the expense of additional memory in the node.

Since no predefined connections between nodes are present, each node needs a discovery mechanism to properly set links with its neighbors. For this purpose, RPL extends the set of ICMPv6[4] control messages introducing 5 new message types:

- DODAG Information Solicitation (DIS): used to solicit a DODAG Information Object from an RPL node. A node may use DIS to probe its neighborhood for nearby DODAGs;

- DODAG Information Object (DIO): allows a node to discover an RPL Instance, learn its configuration parameters, select a DODAG parent set and maintain the DODAG;

[4]The Internet Control Message Protocol for IPv6(ICMPv6) is a supporting protocol in the Internet protocol suite. It is used by network devices, including routers, to send error messages and operational information indicating, for example, that a requested service is not available or that a host or router could not be reached. It is formally defined in Request for Comments: 4443 technical report by IETF

- Destination Advertisement Object (DAO): used to propagate destination information Upward along the DODAG. In Storing mode, the DAO message is unicast by the child to the selected parent(s). In Non-Storing mode, the DAO message is unicast to the DODAG root. The DAO message may optionally, upon explicit requests or errors, be acknowledged by its destination with a Destination Advertisement Acknowledgment (DAO-ACK) message back to the sender of the DAO;
- Destination Advertisement Object Acknowledgment (DAO-ACK): sent as a unicast packet by a DAO recipient (a DAO parent or DODAG root) in response to a unicast DAO message;
- Consistency Check (CC): used to check secure message counters and issue challenge-responses.

A proper exchange of the previously described messages provides RPL with a mechanism to disseminate information over the dynamically formed network topology. This dissemination enables minimal configuration in the nodes, allowing them to operate mostly autonomously.

Often a node can establish links with different nodes of the same DODAG. The goal is to select as preferred parent the node that guarantees the best network configuration. In practical terms, each time a DIO message is received and processed, the RPL protocol checks whether to use that node as preferred parent or not. The goodness of a node can concern several metrics including, for example, available battery energy of a node, or the Rank. The *Rank* of a node is an important information associated with each node. In general, it is a scalar representing the location of that node within the DODAG. The Rank details the hierarchical structure of the DODAG and, consequently, it must monotonically decrease through upward paths, for example, to avoid and detect loops. Rank and path cost are strictly related and correlating these two pieces of information is up to the *Objective function*. In particular, the Objective Function defines the rules of how routing metrics, optimization objectives, and related functions are used to compute Rank. Furthermore, the Objective Function dictates how parents in the DODAG are selected and, thus, the DODAG formation.
The routing metrics, evaluated by the objective function, can take into account several points. They can refer to the node property or to some criteria related to the characteristics of that link. In an environment when reducing to the minimum the number of retransmission is crucial, a node metric related to the number of traversed nodes along a path can be very relevant (known as Hop Count). Some mechanism can also take into account the energy level of a node in order to avoid to load nodes with scarce resources. Routing metrics related to the link are usually expressed in term of throughput, latency and link reliability. Link reliability, in turn, can be

computed in different ways, but, in all the cases, it tries to model how reliable is a connection. A relevant reliability parameter is represented by the Expected Transmission Count (ETX).

The ETX metric is the number of transmissions a node expects to make towards a destination in order to successfully deliver a packet. One of the possible ways in which this value can be calculated[9] is

$$ETX = \frac{1}{Df \cdot Dr}$$

where Df is the measured probability that a packet is received by the neighbor and Dr is the measured probability that the acknowledgment packet is successfully received. Information about the number of transmissions are retrieved from the MAC layer and the contention protocol used.

Analyzing the environment and the requirements for the application it is possible to combine the above showed metrics in order to define a robust Objective Function taking into account all the criticalities of the system. IETF does not mandate to use a specific Objective Function, but it gives guidelines for implementing an effective one. For sure, this is not a trivial task and corner cases may be hard to cover. Often trade-offs should be taken. For example, a node switching repeatedly between two preferred parents with comparable characteristics may lead to waste energy and also to lose some packets. An hysteresis mechanism can be used in this case, which allows a preferred parent to change only if the new parent cost is lower than the current parent cost plus a certain threshold. The threshold value must be accurately evaluated because, in case of problems of the current parent, the node should be ready to quickly swap to another parent to restore the communication but at the same time no useless swap has to be performed. Besides, a good Objective Function guarantees the creation of a stable network able to withstand errors and breaks.

RPL network control

The DODAG generation begins with the DODAG root sending a DIO message to its reachable neighbors, containing information about the instance, its rank, etc. Once a node joins the DODAG, it sends a DAO message upward, to enable downward traffic, and starts sending DIO message to its own neighborhood to propagate DODAG information. A possible sequence of messages exchange is shown in figure 2.4.

A node that wants to join an existing DODAG can send to its neighbors a DODAG Information Solicitation (DIS). Nodes reached by the DIS reply with a DIO message and the new node can establish an Upward link with one of them according to the Objective Function. An example of the use of DIS messages can

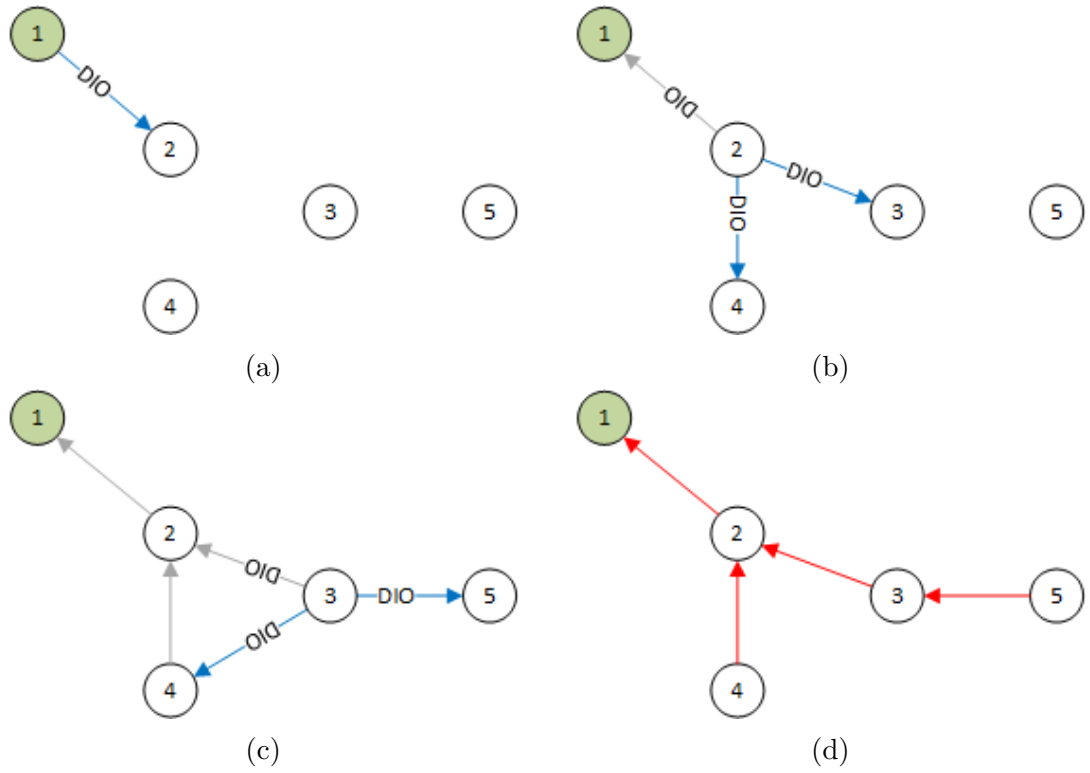


Figure 2.4: DODAG creation: a) shows a DIO message (blue arrows) sent in local broadcast from the sink; b) and c) show DIO messages dispatch by node 2 and 3; d) shows possible upward routes (red arrows).

be seen in figure 2.5.

As said, wireless communication may be unreliable and affected by errors. A typical scenario of connection fault is represented by two devices that previously were able to communicate with each other and are no longer able to. Causes can be different, for example, they moved, one of them lost its power, got broken, or because something blocked their wireless signals, such as a high congestion or a strong radio interference. Independently of the reason of the fault, the network must withstand failures so it has to redefine routing to solve the problem. Depending if the device detects a broken link on the upward or downward route, it acts differently. For upward communication fault, the node can simply pick another node as preferred parent. This typically means that the node changes its rank and rapidly spreads information about its new rank to the network. If a downward route is found to be broken, a more complex procedure takes place. Because each device only maintains a single routing table entry for each downward route, the device cannot by itself pick a new next-hop device for this route but must defer this decision to the root

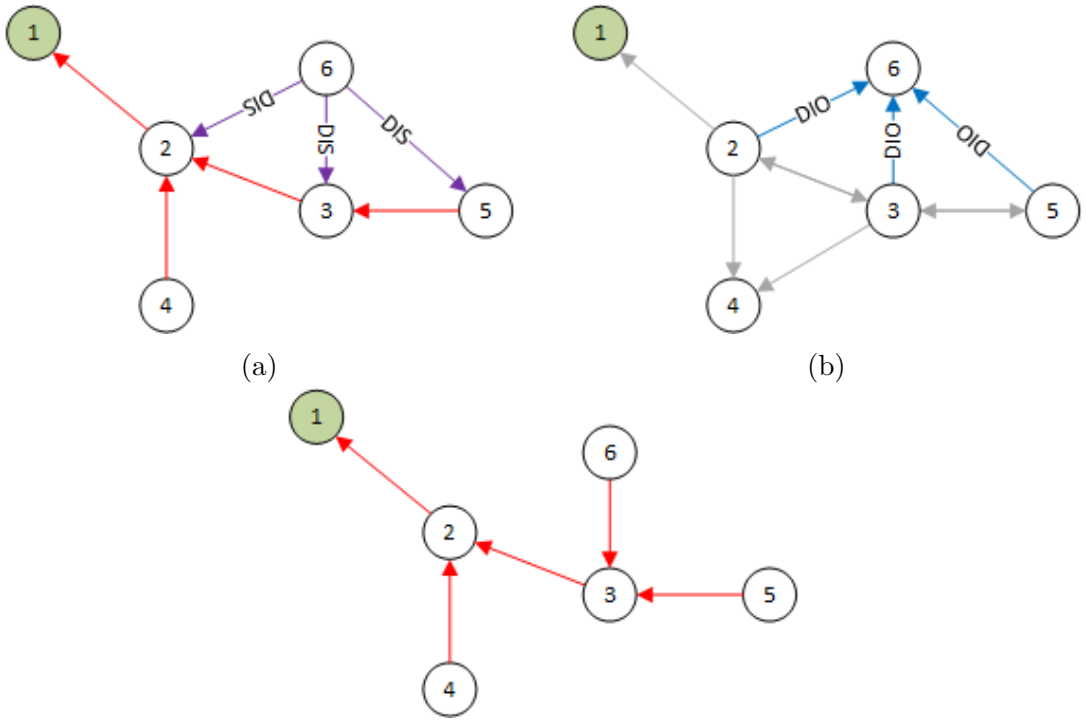


Figure 2.5: Node joining an already formed DODAG: a) shows how the node that wants to join the network sends a DIS message (purple arrows) in local broadcast; b) shows DIO messages (blue arrows) dispatched by node 2, 3 and 5 as response to the DIS message, gray arrows represent DIO messages sent to node not relevant for the current analysis; c) shows possible upward routes (red arrows) once the node 6 has selected a node as preferred parent.

node. This is done by sending a DAO NOACK message towards the root. Because a broken downward route also means that at least one upward link is broken, the root initiates a global repair of the routing tree.

Since we deal with low-power and lossy networks, the overhead introduced by the network maintenance should be as low as possible. From this perspective, the manner in which control messages are sent is regulated by the Trickle Algorithm[10]. Dynamically adjusting transmission windows allows Trickle to spread new information on the scale of link-layer transmission times while sending only a few messages per hour when information does not change. Once the network is configured, if no errors occur, RPL control messages become very rare with a negligible overhead. Unfortunately, this may result in a slow reaction in case of connection fault that is not checked anymore until it is used. In this respect, RPL implements a simple

probing mechanism that force nodes to perform connections and keep neighbors information updated. Sending periodic messages reduces the effectiveness of Trickle and while increasing the number of transmissions, but globally we can notice an improvement in the network behavior.

2.1.4 Time Synchronization

WSNs are distributed systems where each sensor device is equipped with its own local clock for internal operations. Each event that is related to operation of the sensor device including sensing, processing, and communication is associated with timing information controlled through the local clock. Since often users are interested in the collaborative information from multiple sensors, timing information associated with data at each sensor device needs to be consistent. Ideally, local clocks are initially synchronized and evolve at a common pace; however, quartz and circuit non ideality introduce drifts and deviations. The required accuracy depends on the specific application (e.g., seismographic studies need high synchronization between nodes while weather monitoring allows for more relaxed requirements) and guaranteeing it may involve non-negligible costs. These timing requirements make time synchronization an important part of communication for WSNs. Distributed synchronization protocols are required to coordinate the nodes in the network so that they follow the same reference frame. As a result, the following capabilities are provided:

- Temporal event ordering: network with multi-hop and packet-based information are delivered to the sink with variable delay depending on the distance between nodes. For this reason, the order of how packet are received by the sink may differ from the chronological order in which events were generated. Timestamp in the payload may allow the sink to restore the correct temporal order;
- Synchronization to global time: the internal synchronization may be also aligned with the global time, i.e. coordinated universal time (UTC), allowing integration of multiple WSNs through the Internet. This can be performed by a simple conversion between local time and UTC and it is almost trivial;
- Synchronized network protocols: this kind of synchronization is exploited at the MAC protocol level, for example, to provide a common time frame for medium access.

For the local network synchronization, protocols have to address design challenges typically related to the low-cost and low-performance clock with which sensors are equipped, effects of wireless communication, node failures, and in general the resource constraints of nodes. Several protocols are available with different complexity and various performances. Some remarkable protocols are[2]:

- Network Time Protocol (NTP)[11]: in the Internet, NTP is used to discipline the oscillator frequency of each host. NTP relies on a two-way handshake between two nodes to estimate the delay between these nodes and calculate the relative offset accordingly. With modern workstations and fast LANs servers and clients are precise within a few tens of milliseconds with poll intervals up to 36 hours. While NTP provides robust time synchronization in a large-scale network, it is computationally intensive and requires a precise time server to synchronize the nodes in the network; therefore, the characteristics of WSNs make this protocol unsuitable;

- Timing-Sync Protocol for Sensor Networks (TPSN)[12]: TPSN adopts some concepts from NTP. Similarly to NTP, a hierarchical structure is used to synchronize the whole WSN to a single time server. TPSN requires the root node to synchronize all or parts of the nodes in the sensor field. It consists of two phases: the level discovery phase, where the hierarchical structure is built in the network starting from the root node; and the synchronization phase, where pairwise synchronization is performed throughout the network;

- Reference-Broadcast Synchronization (RBS)[13]: RBS synchronizes a set of receivers with one another, as opposed to traditional protocols in which senders synchronize with receivers. In RBS protocol, nodes periodically send beacon messages to their neighbors using the network's physical-layer broadcast. Recipients use the message's arrival time as a point of reference for comparing their clocks. RBS requires message exchanges with all the neighbors, which translates into $O(n^2)$ message exchanges if there are n nodes in a node broadcast range. This increases the energy consumption and may lead to frequent collisions when the network density is high. Another limitation of RBS is that it requires a network with a physical broadcast channel. It can not be used, for example, in networks that employ only point-to-point links;

- Tiny- and Mini-Sync Protocols: Tiny-sync and mini-sync protocols have been developed to provide a simple and accurate time synchronization for WSNs keeping as low as possible the complexity of the synchronization algorithm. Both protocols are based on a hierarchical structure of sensor nodes, where each node is synchronized with its parent node.

Many other protocols exist all with the same aim: to reduce at a minimum the message exchange to avoid collisions, to reach a synchronization error as low as possible at least in the order of milliseconds and to withstand all the problems related to the intrinsic properties of WSNs.

2.2 Operating Systems for WSNs

In a typical system of a certain complexity, resources (including processors, memories, peripherals, network interfaces, etc.) are managed by the Operating System (OS). The OS manages systematically the allocation of these resources to users, providing application programmers with system calls used to invoke different OS services. The resource constraints of typical sensor nodes in a WSN makes OSes not suitable for this task, hence WSNs require a different type of operating system, considering their constrained characteristics. The growing interest in WSN of research and industry allowed the development of several OSes that, with different features, try to exploit the best performance from devices and make them available for developers and users.

The architecture of the kernel represents the first discriminating factor for OSes. An Operating System for a Wireless Sensor Network should have an architecture that results in a small kernel size, hence small memory footprint. Monolithic and modular kernels are two architectural models with advantages and disadvantages that are almost complementary. A monolithic kernel consists of one single program that contains all of the code necessary to perform every kernel-related task. Services provided by an OS are implemented separately and each service provides an interface for other services. Since all the required services are bundled together into a single system image, OS memory footprint should be smaller both in source and compiled forms. An advantage of the monolithic architecture is that the module interaction costs are low. On the other hand, the system becomes hard to understand and modify, unreliable, and difficult to maintain. The lack of portability is also a remarkable disadvantage. A layered (or modular) OS architecture solves some limits of the monolithic kernel implementing services in the form of layers. Advantages associated with the layered architecture are manageability, ease of understanding, and reliability. But the main disadvantage is the poor flexibility from an OS design perspective. The virtual machine is another architectural choice. The main idea is to export virtual machines to user programs, which resemble hardware. A virtual machine has all the needed hardware features. The key advantage is its portability, whereas the main disadvantage is typically a poor system performance.

Another design feature of OSes for WSNs is the programming model. The programming model supported by an OS has a significant impact on the application development. There are two popular programming models provided by typical WSN OSes, namely: event driven programming and multithreading programming. Multithreading is the application development model most familiar to programmers, but it is rather resource-intensive, therefore not considered well suited for resource constrained devices such as sensor nodes. Event driven programming is considered more

useful for computing devices equipped with scarce resources, but not for figure application developers due to its difficulty to extend and the excessive complexity of the application code. Therefore researchers have focused their attention on developing a lightweight multithreading programming model for WSN OSes or a hybrid model combining the two approaches, e.g. with ContikiOS, LiteOS, MANTIS, TinyOS, etc[14].

Depending on the nature of the application also the scheduler should implement a proper scheduling algorithm. Since a Wireless Sensor Network can work both in real-time and non-real-time environment, the OS must be able to provide a suitable scheduling algorithm to accommodate all the application requirements.

Other design issues related to the design of an OS for WSNs involve the memory management and protection, resource sharing and, at a higher level, the communication protocol support for interprocess and internode communications.

2.2.1 ContikiOS

In the previous section, some software designs of Operating System for WSNs were explored. Here ContikiOS architecture and characteristics are exposed with some more details.

> "Contiki is an open source, highly portable, multitasking operating system for memory-efficient networked embedded systems and wireless sensor networks."
> —Contiki documentation[15]

Contiki was created by Adam Dunkels in 2002 and has been further improved and extended by a worldwide team of developers and ported to many different platforms. It was designed to run on types of hardware devices that are severely constrained in memory, power, processing power, and communication bandwidth.

The architecture employed is modular and consists of the kernel, libraries, the program loader, and a set of processes. To combine the benefits of both event-driven systems and preemptible threads, Contiki uses a hybrid model: the system is based on an event-driven kernel where pre-emptive multithreading is implemented as an application library that is optionally linked with programs that explicitly require it. The event-based kernel of Contiki make it completely responsive to real time events and classifies it as a real-time operating system.

The programs in Contiki are called *Processes*. Every process is a piece of code that is executed regularly by the Contiki system and has certain interfaces to interact with other components.

All process invocation in Contiki is done by the process scheduler in response to an event being posted to a process, or a poll has being requested for the process.

The IP communication, both for IPv4 and IPv6, is provided to Contiki through the μIP stack. The μIP implementation is designed to have only the absolute minimal set of features needed for a full TCP/IP stack, such as IP, ICMP, UDP and TCP protocols. It was developed specifically to require very small amount of code and RAM, hence it is ideal for embedded systems applications such as wireless sensor nodes.

A very interesting feature of Contiki is the Over-the-Air Reprogramming. In a wireless sensor network already deployed, manually reprogramming all the node for a bug fix may be a hard or even an unfeasible task. Since Contiki supports dynamic program loading and unloading by keeping the core code separate from the program code in ROM, it has the ability to not only load and unload programs from system memory, but can also load and unload programs over the network connection into RAM or ROM. This allows reprogramming all the nodes in an almost transparent manner.

2.3 Power Consumption

As it has been said repeatedly, nodes of a WSN typically have constrained resources. In an embedded system, but even in general, resources can be intended in terms of processing power, memory and power budget. In particular, the power budget aspect can be very challenging for developers working with battery-powered devices or even with devices without batteries requiring energy harvesting from environmental sources such as heat, vibration, and light. In all those cases the goal is developing a device able to last as much as possible with battery life even in the order of several years.

To better understand the discussions about the power consumption a clarification about terminology can be useful. Speaking about power, energy, total power can be misleading. With the term "power" we intend the rate, per unit time, at which electrical energy is transferred by an electric circuit. It gives information about the instantaneous requirements of energy. The "energy", on the other hand, represents the quantity of electrical energy absorbed by an electrical circuit in a well defined time interval or in a variable time interval associated, for example, with the execution of a task. The "total power consumption" is used, typically, to describe an energy quantity. Typically, the context helps understanding which of the two concepts is referred.

Starting from the physical design of an electronic device up to the software application running on it, techniques for reducing power consumption can cover the whole range with different contributions.
Circuits based on the *Complementary Metal-Oxide Semiconductor* (CMOS) technology dissipate power by charging the various load capacitances (mostly gate and wire capacitance, but also drain and some source capacitances) whenever they are switched. This is referred as *dynamic dissipation* and can be modeled using the formula:

$$P = f_{clk} \cdot C \cdot V_{DD}^2 \tag{2.1}$$

In the equation 2.1, the correlation between power dissipation supply voltage and clock frequency is clarified.

From the technological point of view, CMOS devices have been continuously scaled to achieve higher density, better performance, and lower power consumption. To limit power consumption, the supply voltage (V_{DD}) has been scaled down. This necessitates a corresponding reduction in threshold voltage (V_{th}) to maintain a high drive current and achieve the performance improvement. However, scaling the threshold voltage results in a substantial increase in sub-threshold leakage current. When scaling the channel lengths, it is also necessary to scale the gate oxide thickness nearly proportionally to maintain a reasonable immunity to the short channel effect. The *short channel effect* (SCE) is the decrease in gate threshold voltage as channel length is reduced. The thin gate oxides and the resultant high electric fields across the gate oxides enable considerable current to flow through the gate of the transistor. The total leakage current I_{OFF} is influenced by the threshold voltage, channel physical dimensions, channel surface doping profile, drain/source junction depth, gate oxide thickness, and V_{DD}.

Moreover, simply reducing the frequency may be not sufficient to reduce the total power consumption. In fact, reducing frequency typically slows down the processing, hence, the system requires more time to perform computation.
Relations between various parameters are not linear, therefore, finding the optimal working point where performance and power consumptions (both static and dynamic) are reasonably balanced is the main challenge of manufacturer and designers.

2.3.1 Processor

The voltage and frequency scaling principle is relevant also on a higher abstraction level. As can be intuitive, the faster the MCU is running, the higher the power consumption is. Modern MCU typically offers to application developers mechanism to decrease the operating frequency and the operating supply voltage together making a significant reduction in the dynamic current consumption. The challenge is

to identify the best combination of operating frequency and operating supply voltage for an efficient power consumption while meeting the performance needs of the application. *Dynamic voltage and frequency scaling* (DVFS) is a very effective technique to achieve the best ratio between power consumption and performance. It is based on the modification of the MCU operating supply voltage range and system frequency during runtime through periods when the application does not need significant processing. Dynamic voltage scaling to decrease the core voltage (V_{core}) is known as undervolting[16]. Although at first glance, it is not obvious that high performance correlates with low MCU current consumption, it is a great advantage for many low-power applications to wake-up very quickly, execute software tasks at a high speed and then go back to sleep again as quickly as possible.

To maximize functionality and battery life, developers of battery-powered applications must consider many factors in their system architecture and design. In these applications, the microcontroller is a primary power consumer and developers must carefully consider the way energy is used. Microcontroller power consumption can be identified in four primary power categories[17]:

- Active power: the energy required by the MCU during the run mode;
- Standby power: the energy required by the MCU to keep an inactive state when its action is not required;
- Peripheral power: the energy required by the SoC peripheral like DMA, analog-to-digital converters (ADC), general purpose I/O, oscillator, digital interfaces, etc.;
- Data logging power: the energy required in case of MCU application requiring to record data for future elaboration on flash memory or external storage.

The running phase of an MCU typically consists in the execution of tasks and in idle periods where the MCU waits for the next task to be executed. As said DVFS allows software to change the operating performance point (OPP) in real-time without requiring a reset. By adaptively selecting power supply voltage and clock frequency, DVFS enables software to change SoC processing performance based upon the desired processing tasks to achieve the best performance or lowest power possible. In embedded applications the workload may be known a priori: this allows developers to predict the required processing power and select accordingly the suitable combination between the supply voltage and the clock frequency. This avoids the processing overhead required by a dynamic mechanism that may undermine all the benefits of using it. From a real-time point of view, relaxing MCU performance may be not suitable for applications when hard real-time specifications. In this case,

timing should be accurately evaluated to avoid to miss any deadline.

Modern microcontrollers offer also several low power modes that can be set with different configurations. The peripheral clock can be selectively enabled and disabled while the core is running or, on the contrary, the core can be put into sleep mode while peripherals are running. The latter configuration presents interesting features. During idle periods of the core when it is waiting for interrupt coming from peripherals, the core itself can go in a low-power idle mode until a peripheral makes a request that needs the core intervention. This configuration can enable a duty cycling mechanism that allows the core to rapidly go in low power mode each time its processing is not required. The main problem with this approach relates to the time and power overhead introduced by transitions between states. In order to be effective, wake-up and sleep transition time should be negligible with respect to the time that the core will spend in that state. The *Dynamic power management* (DPM) provides several policies useful to better exploit power and performance capabilities of the system. The DVFS and the core duty cycling are conceptually based on two opposite principles and the preponderance of one over the other depends mostly on the characteristic of the microcontroller considered and on the application. Whereas the former tries to adapt the core capabilities to the workload, the latter tries to maximize idle periods by performing task at the maximum speed for a small time interval. Both mechanisms present overheads during transition or due to the real-time workload assessment.

Often the energy the microcontroller consumes during standby is higher than the active processing energy. This can be very common when microcontroller applications spend the majority of their product life in a low-power standby mode waiting for an internal or external event to wake-up the CPU to process data, make decisions and communicate with other system components or when activities are gathered in small windows distributed over the daytime. Understanding application requirements is fundamental to accurately evaluate the standby current the microcontroller will consume. Typical aspects concern RAM retention, automatic wake-up and interrupt capabilities.

2.3.2 Radio

As part of a wireless network, nodes include a radio peripheral to communicate with other nodes and to be part of the network. Often the radio is the main contributor to the overall power consumption of the system. For this reason, mechanisms to reduce the radio power consumption are developed. The power consumption for transmitting and receiving are almost comparable, but, while transmissions usually

last only a few milliseconds, the radio should be able to receive packets almost continuously so the receiving mode requires a lot of energy.

Several techniques, aimed at keeping radio turned off as much as possible, exploit a duty cycling mechanism. The radio duty cycling (RDC) tries to find a way to keep the radio in receiving mode only when there is an incoming packet on the channel. Designers aim at very low duty cycles ($< 1\%$, meaning that the radio will be active less than 1% of the time), but to achieve this objective they will have to compromise on other network performance goals. The downsides of duty cycling are briefly discussed next. [18]

- End-to-end message delay: data traversing a duty cycling multi-hop network will occasionally have to wait for the next hop to wake-up. This is called sleep waiting and may add significantly to end-to-end latency.
- Collision rates: another side effect of duty cycling is the shortening of transmission and reception time windows. If a contention-based medium access protocol (MAC) is used, these smaller time windows will increase the probability of collisions.
- Control packet overhead: duty cycling may need extra control traffic. The most common source of this overhead is synchronization. Fine-grained synchronization requires frequent resynchronization to deal with clock skews. Designers must check if the added power drain caused by the extra control traffic overhead is compensated by the savings from duty cycling.

Many typologies of RDC process have been studied and developed. They can be classified in three generic groups depending on the timing scheme adopted: synchronous, asynchronous or semi-synchronous.

The synchronous duty cycling implies a common timing reference between nodes. In this way, all nodes are able to wake-up at the same instant so they can successfully exchange messages. If the system is perfectly synchronous, performances are very high, but such system is hard to manage and drawbacks are present. For example keeping nodes synchronized with a sufficient precision requires a higher amount of synchronization messages exchange. Moreover, depending on the network topology, small time drift are always present that may result in *Data Forwarding Interrupt Problem*, meaning that due to a synchronization error, the receiver interrupts the communication even if the packet is not completely received, hence the packet is discarded.

In asynchronous duty cycling, nodes can be time independent from each other and several solutions are used to allow an efficient message exchange. The preamble sampling consists of a long preamble sent by the transmitter that, anticipating the

packets, notifies receivers for the incoming packet. Receivers then wait in ON state for the packet to be received. The two major issues with this technique are the *overhearing* and the *message delay*. In fact, a node should wait for the end of a preamble even if it is already receiving. Moreover, a node does not know if it is the recipient of the packet until the packet is sent. This situation is known as *overhearing*, a node stays ON for a packet that is addressed to another node. Even the duration of the transmission may increase noticeably, since the preamble may be comparable with packet size or even greater. The receiver-initiated transmission is another asynchronous method. In this case, roles are inverted so the transmitter waits for a beacon from the receivers that signals its availability to receive packets. The problem is not completely solved since the transmitter must be always active to listen for receivers availability.

In sufficiently dense deployments, a random duty cycling may be implemented. The idea is that since there is a high probability that there will be enough active nodes anytime, nodes can go to sleep and wake-up randomly. To perform transmissions with a high success probability, this method proposes a random wake-up scheme in which the active time of a node should be inversely proportional to the number of neighbors. A valuable advantage of this approach consists in the fair distribution of traffic load due to randomness and low end-to-end delay.

Some early proposals of asynchronous duty cycling were based solely on the design of the wake-up/sleep schedule. In this category, nodes will divide time into cycles and each cycle will have active and inactive slots. To be used in asynchronous duty cycling, a schedule must guarantee that two nodes will have overlapping active time irrespective of their offset. Even if nodes do not require control messages anymore, such simple approach would typically result in high duty cycle rates.

Finally, a possible mechanism relies on another communication interface, generally called wake-up radio, a low power radio that would listen to a wake-up signal and send an interrupt to the CPU that would activate the primary (or data) radio in response. This is called *on demand wake-up*. Conceptually it is clearly advantageous, but its effectiveness depends on the power consumed by the wake-up radio. The wake-up radio, active all the time, may consume more power than that saved from reducing the active time of the data radio. In addition, the increase in terms of cost and complexity for the device may restrict the adoption of a mechanism like this.

In semi-synchronous proposals, neighbors are grouped into synchronized clusters and clusters interact with each other asynchronously. Since the synchronization between neighbors is easier to achieve than global synchronization, these schemes try to take the best of synchronous and asynchronous mechanisms. On the other hand, cluster maintenance may require a reference node with a consequence control traffic that may make clusters inadequate for dynamic topologies. Depending on how clusters are created we can distinguish between *Spontaneous clustering* and

Elected Cluster-heads. Spontaneous clustering will refer to mechanisms where nodes coordinate themselves without the need of a cluster-head, while Elected Cluster-heads will include the mechanisms where one of the nodes in each cluster (the cluster-head) receives the special assignment of (temporarily, in most cases) coordinating cluster activity.

2.3.3 Sensors and actuators

The power consumption of the sensors modules of a node may be critical from a low power point of view. The market offers an extremely wide variety of sensing devices with different sensing characteristics tailored on specific or more general purpose applications. All the major physical phenomena can be actually monitored, but, depending on the resources required to perform sensing, power requirements may heavily change. For example, capacitive digital sensors for relative humidity and temperature (the typical ambient condition parameters) are very power efficient with current requirements in the order of some μA. Other sensors, such as motion or air quality sensors, may be more power hungry with current consumption even in the order of some mA. The sensor power consumption is typically related even with the sampling frequency required by the phenomenon to be observed. Depending on the characteristics of the phenomenon, the sampling frequency can be relaxed, reducing the total energy required by the sensor. Wireless Sensor Networks provide extensive information from the physical world through distributed sensing solutions. Generally, this results in one-way information delivery, where information from the physical world is imported in the digital domain. With the emergence of low-cost actuators and robots that can affect the environment, a two-way information exchange is possible. As a result, information that is sensed from the environment can be utilized to act on the environment. This expands WSNs functionalities to a Wireless Sensor and Actor Networks (WSANs) that are capable of observing the physical world, processing the data, making decisions based on the observations, and performing appropriate actions. From a power consumption perspective, equipping each node with an actuators module implies considerations analogous to the case of the sensors. Actuation can be performed by transducer in several ways, exploiting different physical phenomena.

2.4 Energy harvesting

Nodes in Wireless Sensor Network are often placed, for many reasons, far from an electrical grid that may offer continuous power supply. In general, even if an electric grid is available, guaranteeing a wired power supply is not feasible because wiring limits the flexibility of the WSN. Using electric battery is very straightforward to provide the power supply to the nodes. Moreover, technology improvement allows

energy density for batteries that can guarantee very long time of functioning with relatively small sizes.

A more sophisticated approach for supplying power to the node relies on energy harvesting. The energy harvesting allows deriving electrical energy from the environment (for example solar power, thermal energy, wind power, kinetic energy, etc). Physical and chemical phenomena can be exploited in many ways and on many scales in order to convert temperature gradients, light beams or vibrations into electrical energy that can be stored into accumulators or used directly to supply devices.

The energy derived from the environment can be classified into several types depending on the ambient energy source[19] (see figure 2.6 for a graphical representation):

- Mechanical Energy. The mechanical energy scavenging converts the kinetic energy of an object. The kinetic energy refers to the motion of the object in general, including vibration or deformation phenomena. The harvesting can be based, for example, on the phenomenon of *piezoelectricity*. Piezoelectric materials have the properties of generating a voltage drop when a pressure is applied due to the internal deformation. This method exploits the electromechanical interaction between the mechanical and the electrical state in crystalline materials. For very scaled application the energy scavenged is very limited, often in the order of some μW or lower[20];

- Radiant energy. The radiant energy is related to the energy associated with electromagnetic waves such as visible light, ultraviolet ray, and radio frequency signal. The solar energy is very popular among green energy. The scavenging requires the so called photovoltaic cells to convert light beam into electric power. This technology scales very well and also for small surfaces of the panel provide an energy density of hundreds of $\mu W/cm^2$ [20][21]. Solar energy has the disadvantage to be only available during daytime for outdoor environment or when lights are turned on indoors. The use of batteries or capacitors as charge storage is fundamental to guarantee continuous power supply the application. The increase of wireless signals surrounding urban areas and domestic settings has motivated also the research on radio frequency scavenging. Its performance extremely depends on the radio frequency fields emitted in the surrounding environment but common value space from some pW/cm^2 to tens of $\mu W/cm^2$[20];

- Thermal Energy. The thermal energy extraction is typically based on the Seebeck effect. It allows generating an electrical field starting from the temperature gradient of two surfaces using a Thermoelectric Generator (TEG). Several methods of thermal energy harvesting for wireless sensor networks exist in the literature exploiting, for example, the temperature difference between

the human body and the environment[22]. Sources of heat energy vary from body heat, which can produce energy density of some $\mu W/cm^2$ to a furnace exhaust stack where surface temperatures can produce energy density in the order of mW/cm^2[20];

- Fluid Flow. The wind and water flow energy can be classified under fluid dynamic or fluid flow. The energy from these sources can be harvested using turbines. Despite the extremely dependence of the wind source from weather conditions, the literature shows some example of wind energy harvesting used to supply a Wireless Sensor Network[23].

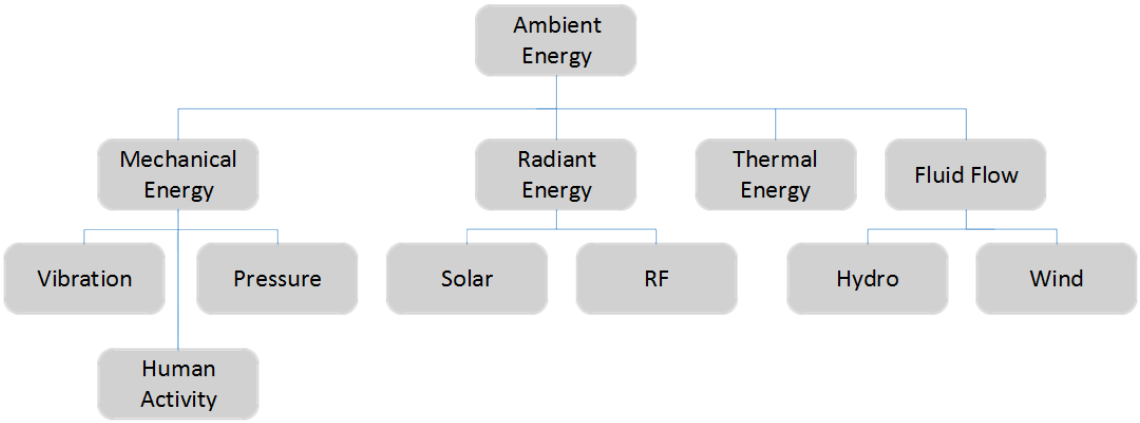


Figure 2.6: Simple classification of the energy scavenged from the environment depending on the ambient energy source.

Depending on the nature of the power source in almost all the cases a power conversion is required. Typically a DC/DC converter is employed but also a AC/DC converter may be required. The needs of a power conversion is driven by the fact that source, storage and load have different electrical characteristics, hence a power matching between them has to be considered. Reasons why electrical domain between modules are different can be several and diverse. One example can be related to the very wide range of values for the voltage that the power source can provide as output or to the magnitude of the voltage that can be really high or extremely small depending to the physical phenomenon associated with.

The power conversion obviously implies an efficiency factor. The ideal efficiency of 100% is not physically possible, hence a certain power is dissipated in the path shown in figure 2.7. Moreover, the harvester has low efficiency if storage device is not impedance-matched to the source. For this reason, the power conversion block on the left of figure 2.7 typically integrates a *Maximum power point tracking* (MPPT) technique. The *maximum power point* (MPP) refers to the point in which the power output characteristics of the harvester assumes its maximum value. ON a I-V graph

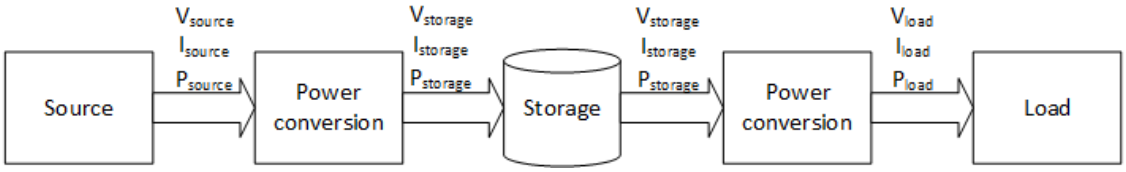


Figure 2.7: Block diagram showing the power conversion path required to match the electrical characteristic of the source with the requirements of the load.

this point is represented by the "knee" on the curve, see figure 2.8 for a graphical example[24].

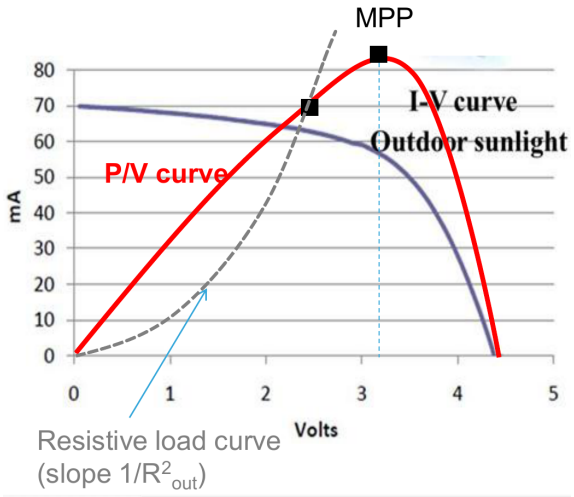


Figure 2.8: Example of a I-V graph of the characteristic of a solar cell. From the power curve superimposed the maximum power point can be easily identified and put in relation with the I-V curve.

In this context, the MPP tracking is an essential component of the solar energy harvester and several techniques can be found in the literature depending on the application domain and performance requirements. Algorithms used for the tracking may be very complex with the consequence of consuming too much energy themselves. The perspective of increasing the efficiency of the harvester should also consider the overhead of the successive module required by the harvesting chain. For this reason the efficiency of the harvester often is expressed as a global parameter instead of on the single module.

Chapter 3

System overview

This chapter presents an overview of the addressed wireless network system. Starting from a top view of the network the network management is described with a description of the node, of the network creation mechanism and of the routing protocol.

3.1 Network structure

The adopted structure for the network consists of a mesh network with a tree-like topology rooted in the sink node that, in addition to the functionality of the other nodes, has a network interface to the Internet and is in turn of collecting the data coming from all the nodes of the network. The sink provides access to the Internet to the whole network. The nodes of the network are composed all by the same modules and have all identical functionalities. They all run ContikiOS as operating system, whereas the hardware is composed of off-the-shelf devices by STMicroelectronics® consisting of:

- the core board integrating a low-power microcontroller (ARM® Cortex®-M3) with several integrated peripherals;
- an ultra-low power, Sub-GHz radio module that guarantees the wireless communication;
- an expansion board integrating several sensors for monitoring humidity, temperature and pressure;
- a power module equipped with solar cells and a rechargeable battery.

The Contiki operating system provides a basic set of functionalities to support the development of a mesh network, on which an application can be designed and

implemented. The message exchange between nodes is based on the User Datagram Protocol(UDP), a simple message-based connectionless protocol. Connectionless protocols do not set up a dedicated end-to-end connection as communication is achieved by transmitting information in one direction from source to destination without verifying the readiness or state of the receiver. By definition the UDP protocol is unreliable. In order to ensure quality, UDP functionality is extended by means of adding a small acknowledgment message of received packets and retransmission of lost packets. Adopting an acknowledge mechanism for data exchange improves the reliability of the connection reducing the risk of packet loss and allow a lower redundancy on sent packets by retransmitting packets only when strictly necessary. Moreover, the acknowledge message enables the use of the ETX metric, discussed in section 2.1.3, since the acknowledge messages provide a feedback on the number of retransmissions required for each packet. Drawbacks, as usual, are always present. Even if acknowledge message are very small, their management requires time and space on the channel increasing the required time for a single transmission. The overhead may be not negligible in the case of a highly congested medium.

3.2 Routing optimization

ContikiOS natively implements the RPL protocol (analyzed in section 2.1.3) providing some functionalities useful to tailor routing behavior according to the application needs.

The choice of an objective function rather than another relies exclusively on the developer's needs. Contiki provides in its firmware two objective functions already implemented following the guidelines provided by the IEEE in their relative Request for Comments documents. They are the objective function zero (OF0) and the Minimum Rank with Hysteresis Objective Function (MRHOF).

The Objective Function Zero (OF0) has a very simple implementation. It is designed to find the nearest Grounded root[25], where Grounded root means a DODAG root providing such connectivity (the sink in our environment).

The Minimum Rank with Hysteresis Objective Function is the objective function[1] adopted in our application. This Objective Function selects routes that minimize the ETX metric while using hysteresis to reduce parent switching in response to small metric changes. The hysteresis threshold recommended by the IEEE working group is $PARENT\_SWITCH\_THRESHOLD$: 192 calculated as the number of transmissions required by a node to successfully deliver a packet multiplied by a 128 factor. This means that a node will switch to a new path only if it is expected to require at least 1.5 fewer transmissions ($1.5 \cdot 128 = 192$) than the current path[26].

[1] See section 2.1.3 for details.

However, this value is not binding so for our application we followed the ContikiOS developer's guideline that preferred to adopt a more aggressive setting for the hysteresis (in particular set to 96 corresponding to a number of retransmission equal to 0.75 $0.75 \cdot 128 = 96$), so a change is more likely to occur with respect to the default configuration. This represents a trade-off between network reconfiguration speed and network stability.

An aspect to be considered concerns with the initial ETX value to be assigned to a node when no information about the connection properties of the newly added link. ContikiOS allows two different mechanisms for initializing the ETX metric: static and dynamic. The static initialization simply assigns a fixed arbitrary value as ETX metric, by default it is set to 3 retransmission required that with the multiplication factor of 128 becomes 384. The problem with the static initialization refers to the complexity of selecting a reasonable value, as close as possible to the effective transmission count. Values lower than the correct one may cause unreasonable parent switch resulting in a network instability and possible messages loss. On the other hand, a too high value may interfere heavily on the rank computation making a "good" parent, with fewer hops, not preferable. The direct consequence is a sub-optimal network routing.

The ETX definition implies a direct acknowledged message exchange between two nodes, this means that to update the ETX value of a node after the initialization a message exchange must occur. Once the network is formed, unless critical errors on the path, a message exchange between two neighbors is very unlikely to happen, hence the stability of the network results in a sub-optimal configuration that may last for a long time. Finding a reasonable value can be non-trivial since it may change from network to network and a common value working, in general, may not exist.

A dynamic initialization tries to find a more realistic ETX value starting from other information about the communication channel. Since we are dealing with other characteristics of the channel partially unrelated with the ETX computation we must accept some assumption as true or valid in a first approximation. A quite realistic ETX estimation can be derived from the assumption that the Received Signal Strength Indicator (RSSI)[2] is somehow correlated with the ETX. This assumption, in general, is not true but gives a sufficiently realistic value for the ETX.

In the eventuality that the RSSI assumes a meaningless value due to unpredictable problems on the link, the same issue encountered for the static initialization

[2]Received Signal Strength Indicator (RSSI) is a measurement of the power present in a received radio signal, in milliwatts. The value is tipically expressed in dBm (logarithmic scale) and common values varies between -100dBm, for a low signal level, and -60dBm for a very strong signal level.

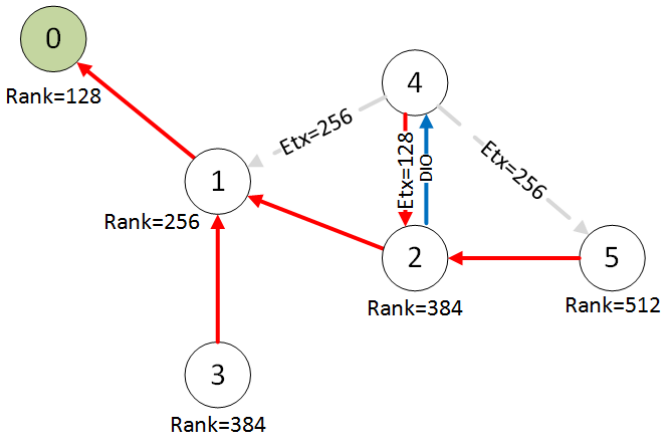


Figure 3.1: Non-optimal network configuration due to static ETX initialization ($ETX_{init} = 2 \cdot ETX\_DIVISOR = 256$).

arise. The node affected by the error in the estimation may be associated with a high ETX value that will prevent its neighbor to use it as preferred parent, even if it may be a good candidate.

A viable solution, valid also for the static initialization, may consider the use of a probing mechanism. A probing mechanism consists in sending unicast message (for example using DIS or DIO messages, but any other custom probing functions are acceptable) to the neighbors of each node with the intent of forcing the update of the ETX metric consequently to the unicast message exchange. The introduction of these supplementary messages clearly impacts on the channel occupancy. As discussed in section 2.1.3, probing too often to keep all the neighbor's statistics updated means a high message traffic on the medium whereas an infrequent probing will make the network very slow to set itself in an optimal configuration. As usual, a trade-off must be adopted.

When a DIO coming from an unknown node is received, the RSSI value associated with the DIO packet just received is used to compute the ETX. The guess is performed by roughly estimate the Packet Reception Rate (PRR) from RSSI, as a linear function. For the radio adopted the linear function suggested by ST developers is:

$$\begin{aligned} RSSI \geq -60dBm \quad results \quad in \quad PRR \quad of \quad 1 \\ RSSI \leq -90dBm \quad results \quad in \quad PRR \quad of \quad 0 \end{aligned}$$

Figure 3.2 shows an example of the behavior of the *guess-etx-from-rssi* approach. In this case, it performs better with respect to the static case shown in figure 3.1, however cases in which the node is associated with a completely wrong ETX value

are still present. In any case, with time the network tends to evolve towards a more reliable state.

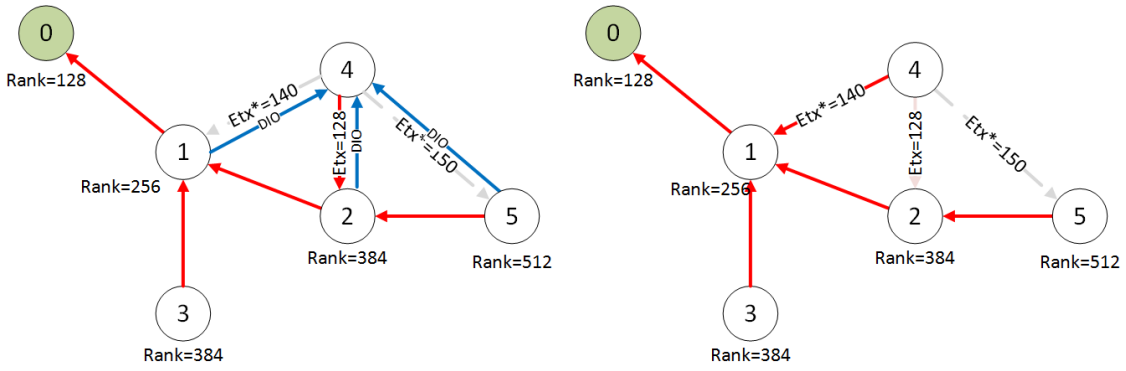


Figure 3.2: Nodes that receive a synchronization message update their time only if the sender is their preferred parent.

3.3 Node synchronization

As frequently said, the Radio Frequency (RF) module is the main source of power consumption for a node. A viable approach to reducing the RF power consumption impact at the application level can rely on a *sleep wake-up scheduling* protocol. The sleep wakeup scheduling protocol adopted consists on grouping processing and radio activities in small time windows. The result is a kind of duty cycling mechanism where the radio is allowed to transmit only during a predetermined time frame. Such mechanism, extended to the whole network, requires a fine time synchronization between nodes. In fact, the useful active time is given by the time interval where all the node active frames overlap, see figure 3.3. To correctly manage all the application and control messages, the useful overlapping time in figure 3.3 should be greater than a threshold that increases with network size and is in the order of few seconds. A very fine-grained synchronization allows to keep the useful active time as close as possible to the total active time of the node, keeping the overhead due to clock skew[3] as low as possible.

To propagate synchronization messages two different approaches were investigated. In both cases the *sink* node acts as a time reference and since it is typically connected to the Internet, it may act also as UTC (Coordinated Universal Time)

[3]On a network such as the Internet, clock skew describes the difference in time shown by the clocks at the different nodes on the network.

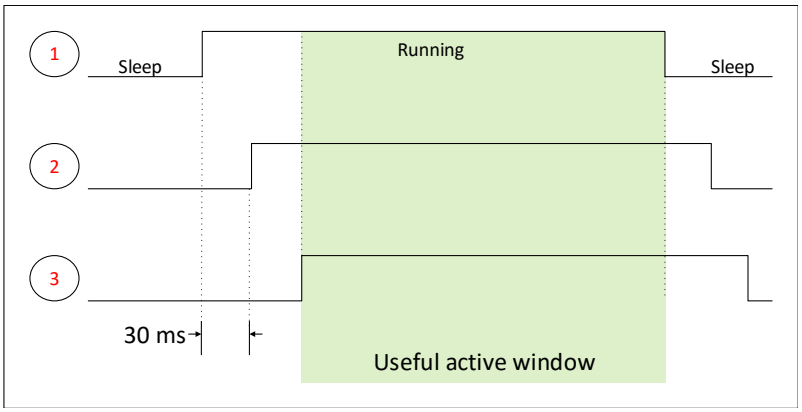


Figure 3.3: Timing activity schema. The clock skew between the nodes reduces the useful active windows, hence the available communication time slot is shrunk.

translator in case a local time synchronization is adopted. The first method uses the Roll Trickle Multicast protocol[4] to propagate messages through the network. The message propagation is shown in figure 3.4 with the timestamp arrival time indicated near each node. The delay with which each node receives a timestamp is proportional to the number of hops, namely the depth level of the node in the network. For big networks the delay may become remarkable. In this case guaranteeing the minimum size to the active windows requires the system to stay active for more time with the consequence of a growth in the power consumption.

The second method, schematically shown in figure 3.5, is based on the local synchronization between a node and its parent. Starting from the root, each node periodically sends in local broadcast its timestamp. When a synchronization message from the preferred parent is received by a node, the internal RTC registers are updated and a packet with the new timestamp is prepared and sent to the neighbors of the node.

An aspect that has to be taken into account when addressing time synchronization relates to the characteristics of the RTC oscillator used on the board. The relative shift between two oscillators clocks, even if it is in the order of a few ppm[5], will require more frequent synchronization messages at the cost of a higher activity on the channel. A preliminary calibration of the RTC clock of each board may be beneficial allowing, for example, to reduce the sync messages frequency.

In fact, even if synchronization messages are small (see section 4.5 for details

[4]This protocol is defined by the IETF as *Multicast Protocol for Low-Power and Lossy Networks* (MPL) in the Request for Comments: 7731 technical report; previously it was addressed as Roll TM protocol and Contiki still use this terminology.

[5]The performance of an oscillator is typically expressed as parts per million (ppm), indicating how much the crystal's frequency may deviate from the nominal value.

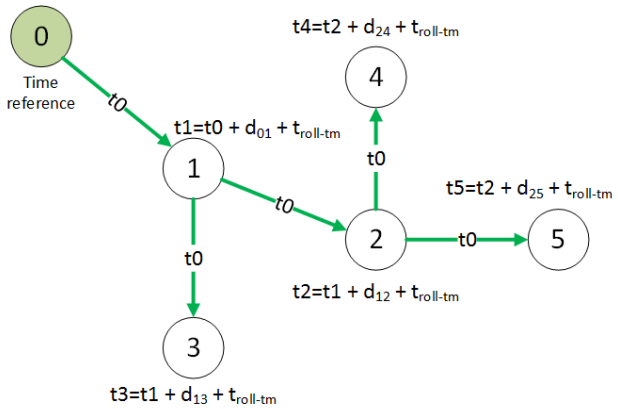


Figure 3.4: Propagation of a timestamp from the root to all nodes using Roll TM protocol. On each branch the value of the timestamp sent is indicated. Node 5 receives the timestamp T0 several milliseconds after it has been sent by the root. This delay is proportional to the depth level of the node and for each hop is in the order of tens up to hundreds of milliseconds. The $t_{roll-tm}$ variable refers to the send delay specified by the Roll Trickle Multicast protocol, whereas d_{xy} is the delay related to the propagation of the message from node x to y.

on the sync message structure) and do not require an acknowledge message to be sent back to the sender, they necessarily require to be propagated through the whole network generating a not negligible channel occupancy. Moreover, each node receives and has to manage all the synchronization messages coming from its neighbors, even if they are not the parents and will be discarded.

3.4 Microcontroller low power modes

From the analysis exposed in section 2.3.1, all the possible aspects to take into account for minimizing the microcontroller power consumption were investigated. For a quantitative analysis refer to section 4.3 The analysis focused on:

- Characterize the active phase, selecting a clock frequency trying to trade-off between power consumption and performance;
- Configure core peripheral (*General Purpose I/O* (GPIO) peripherals give the highest contribution) to minimize power waste in unused peripherals or through unnecessary pull-up or pull-down networks;
- Configure the low power mode in order to minimize the power consumption;
- Configure the radio parameters to match the processor performance capabilities, in particular, define a datarate and a payload size for packet able to

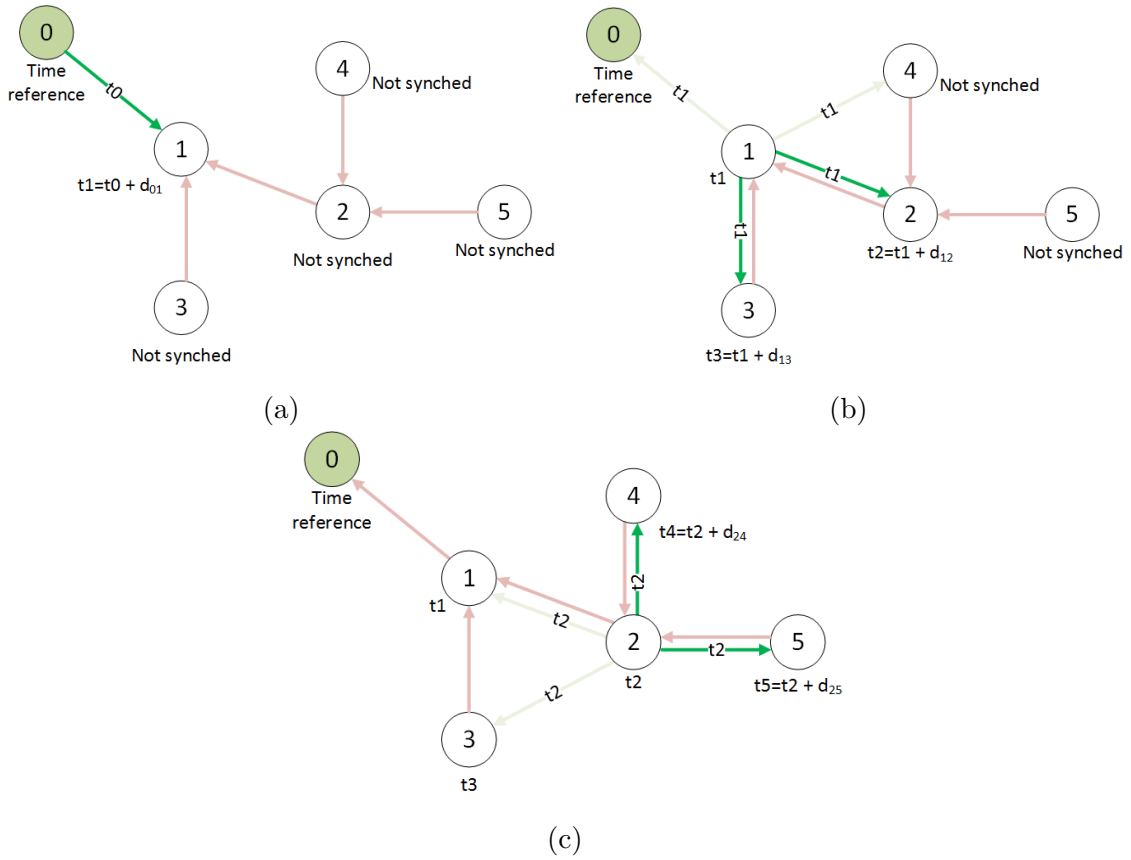


Figure 3.5: Propagation of a timestamp from the root to all nodes using unicast messages. On each branch the value of the timestamp sent is indicated. Nodes that receive a synchronization message update their time only if the sender is the preferred parent for that node and then propagate the new timestamp to neighbors nodes. The d_{xy} variable represents the delay related to the propagation of the message from node x to y.

optimize transmissions and reception operations.

MCU Run mode

The timing for the execution of the ContikiOS threads is the main task for the ContikiOS's scheduler. Depending on the events that are occurring it updates the queue of execution and then the MCU is fed with a ready to run task. After the execution of a task, the MCU may be idle waiting for the next task to execute. During this time the process does not perform any useful task but it is still consuming power. The number of idle periods and their duration clearly depend on the number of tasks to be executed by the MCU (the MCU load) and on the run speed of the MCU.

The most simple way to reduce power consumption relies on minimizing the current consumed by the MCU during the idle periods between consecutive tasks exploiting, for example, a low power mode such as *sleep* mode.

Merging this low power solution with the clock scaling allows to take advantages from both aspects. However, it should be noted that reducing the clock frequency makes the MCU slower and, with the same amount of MCU load, prevents idle periods to be long and frequent as before. An empirical analysis shows clearly the evolution of the power consumption depending on the different combination of clock frequency and sleep mode (see figure 4.5 in section 4.3 for a graphical example). Under a certain clock frequency, the number of idle periods is very shrunk, hence, the benefits introduced by the switch between active and sleep mode does not justify the latency introduced on the scheduler.

GPIO and peripherals configuration

To reach the maximum power saving available it is necessary to enable only the necessary peripheral and keep OFF and with the clock disabled the others. Moreover, during the run mode, the unused GPIO pins are set to the analog mode with no-pull. During low power mode, all the pins are set in the low power configuration except for the pins connected with the radio module that requires a more accurate configuration in order to drive also the radio module to a correct low power mode with a minimum power consumption.

Chapter 4

Implementation

This chapter shows what was done for what concerns the implementation and the software design. Firstly, the energy harvester is described and characterized. Then, the hardware of the nodes is described with a focus on which steps are required to configure a node in order to minimize power consumption in all the power modes used. Finally, the technique adopted for the node synchronization is discussed and described.

4.1 Characterization of the energy harvester

The wireless sensor network analyzed is intended to be powered by harvesting solar energy through a solar panel. The solar energy harvested is stored in a battery after being converted to match the electrical characteristic of the battery.

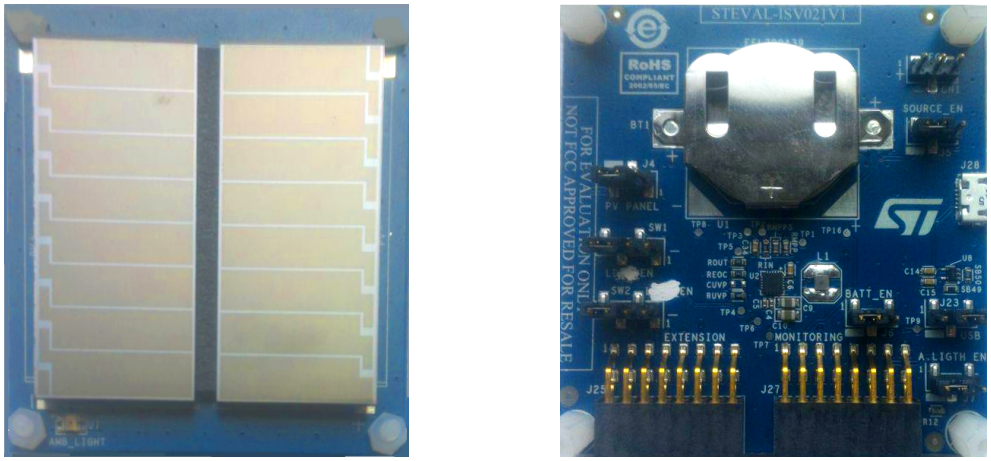
To quantify the capabilities of the harvester in terms of energy, we developed a very simple circuit using an operational amplifier and a transistor in shunt configuration in order to drain all the available current provided by the harvester and measuring the corresponding power output in μW.

Such system is used to characterize the behavior of the harvester during the whole daylight in different conditions of sunlight, artificial light exposure and weather condition (sunny or cloudy day). The results are useful mainly for two purposes:

- Find the correlation between luminance level (taken by the light sensor) and the corresponding power extracted. Before deploying the network can be an effective tool to estimate the energy budget that we can expect from the environment using only an approximated luminance measure. Also an information about the ambient temperature was also collected and associated with relative luminance and power samples;

- Extract a general estimation on maximum, medium and minimum power availability starting from the analysis of different working conditions. The maximum power availability is not very useful because it is very unlikely to be observed in real deployments but can be considered as an upper bound. Depending on the deployments and the position of the solar panel with respect to the light source the average and minimum power availability can be very similar and are taken as reference for further analysis.

The solar panel harvester considered is the STEVAL-ISV021V1 demonstration kit. It consists of a complete energy harvesting module based on the SPV1050 Ultra Low Power energy harvester and battery charger (see figure 4.1). The SPV1050 device is a very powerful chip from STMicroelectronics® with several features that allow high performance and very low-power properties. The SPV1050 implements an MPPT function with a minimum accuracy of the 95% and integrates the switching elements of a buck-boost converter. The power manager is suitable for both PV cells and TEG harvesting sources guaranteeing high efficiency for a high range of voltage from 75 mV up to 18 V. It allows configuring the battery charge voltage in the range of 2.6 - 5.3 V level[27].



(a) Upside (b) Downside

Figure 4.1: Upside and downside view of the STEVAL-ISV021V1 board.

Two LDO voltage output (1.8 and 3.3 V) are available for powering external devices like sensors or RF transceivers. The SPV1050 is a very flexible and configurable device that scales very well depending on the application requirements. The functioning of the SPV1050 is related with a $94\mu F$ capacitor used by the chip as the power source for its own operativity and as main power storage. In order to guarantee the lifetime and safety of the battery, the SPV1050 device controls an

integrated pass transistor between the main storage and the battery exploiting both the under voltage (UVP) and the end-of-charge (EOC) protection thresholds.

The board is equipped with two solar panels mounted in series. The solar panels are the *AM-1801* by Panasonic (Sanyo) for indoor applications[21] for a total surface of $26.5cm^2$. They are well suited to work with luminance level in the order of 200lx that represents a common indoor situation with artificial lighting.

The STEVAL-ISV021V1 implements a basic but complete configuration of almost all the functionality of the SPV1050. Moreover, it is distributed with a Power Monitoring Board (PMB) very useful to monitor both PV panel and battery voltages and currents, and system performance like MPPT accuracy and conversion efficiency.

The Power Monitoring Board includes a dedicated software GUI called *SPIDer* useful to analyze statistics on energy harvesting such as:

- Input power extracted from the PV panel;
- Output power carried out to the battery;
- Conversion efficiency (output power / input power);
- Ambient light intensity;
- MPPT accuracy (real maximum power / ideal maximum power);
- Open circuit voltage of the PV panel.

Figure 4.2 shows an example of the graphic interface, in particular it shows the tab relative to the panel efficiency.

Despite the Power Monitoring Board provided is a very effective analysis tool, it does not allow a continuous time analysis of the system.

Unfortunately, the tool can not run for a wide time period, since it provides only instantaneous statistic hard to extend in a continuous form. A simple custom circuit coupled with a STM32F401re probing board was developed and programmed to overcome this limitation. The circuit uses an OPA2340 operational amplifier to drain all the available current provided by the harvester and measuring the corresponding power output. The analysis is performed indirectly by measuring the voltage drop over the load resistor. Then, the current is derived and used to compute the corresponding power involved in the process. It is a dual rail-to-rail operational amplifier powered by the probing board. Both the amplifiers are employed, one for the voltage regulation and the other one as voltage amplifier for better matching the voltage under test with the dynamic of the Analog-to-Digital converter of the

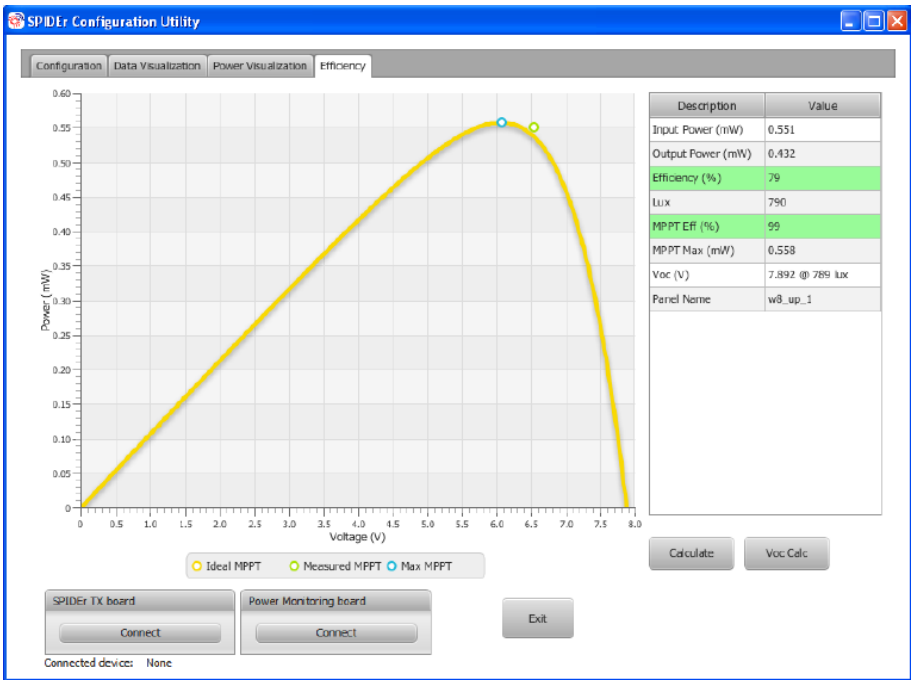


Figure 4.2: Example tab of the software "SPIDer" distributed with the Power Monitoring Board.

probing board. The resulting circuit had a very limited size allowing a simple deployment almost without limitation. The circuit schematic is shown in figure 4.3. It acts as a shunt voltage regulator able to adapt dynamically the impedance seen by the generator to keep its voltage at a fixed value of 4.0V. Its correct behavior was verified using both oscilloscope and multimeter

To better characterize the harvester and its behavior in different illumination conditions, the power measurement is associated with the information about the ambient luminance. To measure the ambient luminance the *SFH 5711* ambient light sensor was used. It is mounted on the STEVAL-ISV021V1, but it is not used by the SPV1050. It is used by the Power Monitoring Board for analogous purposes and since the PMB is not connected, we can easily use the sensors without any conflicts. The custom monitor board is also able to collect ambient temperature information. Since, in indoor application, temperature variations are not so relevant and always in a pretty small range, this quantity does not seem to be remarkable so it is not taken into account in further discussions. The voltage is sampled at a $10Hz$ frequency and then an average of ten samples is performed. The other two magnitudes are sampled with a frequency of $1Hz$ since they are likely to have very small variation in this amount of time and a high accuracy is not strictly necessary. The set of three measures is collected by a PC through the serial communication

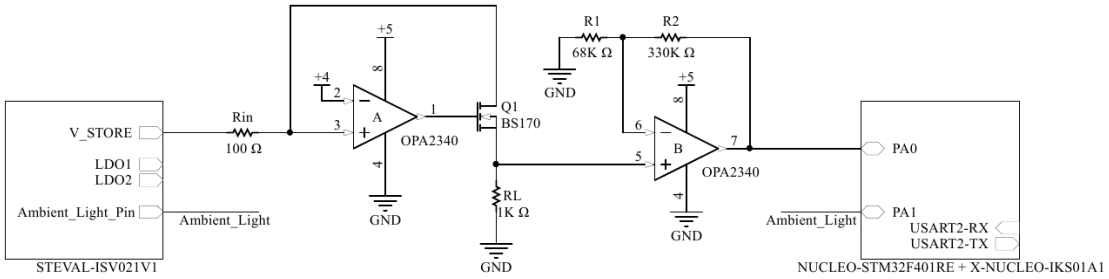


Figure 4.3: Power output tracker circuit. The block on the left represents the STEVAL-ISV021V1, only the pins relevant for the analysis are represented. The OPA2340 is a dual operational amplifier and both amplifiers are employed in the circuit. They are indicated with A and B, A is used as voltage regulator whereas B is responsible for amplifying the voltage drop over the load resistors in order to adjust the dynamic to the ADC requirements. The block on the right represents the probing board used for collecting data. It is connected to a PC through a serial connection (not represented in the schematic).

pins of the board connected to the USB port. Each measurement is associated with a timestamp useful to have also a daylight reference for the measure, The listing 4.1 shows an example of the output generated by the board.

```
[Fri May 19 17:36:09.156 2017] 145.915 25.39 544
[Fri May 19 17:36:10.154 2017] 145.775 25.42 544
[Fri May 19 17:36:11.153 2017] 144.426 25.46 549
[Fri May 19 17:36:12.151 2017] 144.343 25.41 549
[Fri May 19 17:36:13.150 2017] 145.228 25.44 549
[Fri May 19 17:36:14.164 2017] 146.546 25.42 540
[Fri May 19 17:36:15.146 2017] 144.525 25.42 540
```

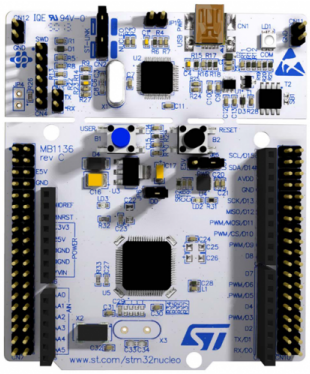
Listing 4.1: Log otput of the energy output tracker. The timestamp is followed by the power output expressed in μW, the temperature in Celsius and the luminance level in lx.

4.2 Node description

Nodes are composed of several devices provided by STMicroelectronics® belonging to the STM32 Nucleo ecosystem. The STM32 Nucleo ecosystem consists of the combination of STM32 Nucleo boards and expansion boards allowing a unified scalable approach with many possibilities for application development, prototyping or product evaluation.

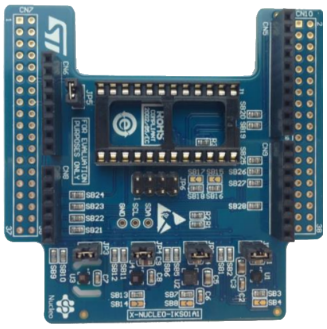
In particular, the following components were involved:

- STM32L152re Nucleo development board equipped with an ultra-low-power ARM® Cortex®-M3 based microcontroller;
- X-NUCLEO-IDS01A4 evaluation board based on the SPIRIT1 RF module SPSGRF-868 expansion;
- X-NUCLEO-IKS01A1 motion MEMS and environmental sensor evaluation board.

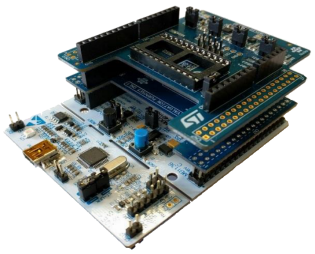


(a) STM32L152re

(b) X-NUCLEO-IDS01A4



(c) X-NUCLEO-IKS01A1



(d) NUCLEO STACK

Figure 4.4: Hardware equipment adopted; d) shows the final node stacked configuration.

The core of a node consists of the Nucleo board and the Radio module. The combination of these two elements forms a node of the network able to join and exchange messages. This basic node can be expanded according to application requirements with all the necessary sensor devices. An X-NUCLEO-IKS01A1 expansion board, for example, can be easily added to the X-NUCLEO stack providing temperature, humidity, pressure sensors, a magnetometer and an accelerometer. STMicroelectronics® provides many others X-NUCLEO expansion boards that can be employed. There are no constraints limiting the use of a single typology of nodes

for the whole network. In fact, the network can be composed of a large variety of nodes sharing the same protocol but differentiating for the sensing or actuating functionality.

All the nodes run ContikiOS 3.0 that is already ported on the STM32 platform by STMicroelectronics® as Open Development Environment (ODE) Function pack[28]. The ODE Function Packs provides some working examples of applications including one with the *X-NUCLEO-IKS01A1* sensors board. This is a useful starting point for any implementation since it presents minimal configuration and easily customizable and expandable structured programming. The border router (sink) uses a different firmware version but fully compatible with the ODE Function Packs. It is the *X-CUBE-SUBG1* communication software expansion for STM32Cube. It is a parallel development branch to the ODE Function Packs with different goals but sharing the same software example and structures. It contains the firmware for a wired border router node to be connected to a PC to interface the network with Internet. A similar example firmware is also present in the ODE Function Pack but that version requires a wireless border router node interfacing with the Internet directly through a traditional Wi-Fi module. The presence of parallel development branches with different goals but that keep a high level of compatibility and interoperability between them emphasize the commitment of STMicroelectronics® to create a well-structured ecosystem where the development opportunities are not restricted to a specific product or application but span several application designs.

4.3 MCU run and low power configuration

The STM32L1re board is equipped with an ARM® Cortex®-M3-based core exploiting an ST's proprietary ultra-low-leakage process technology with an innovative autonomous dynamic voltage scaling and 5 low-power modes.

A useful analysis on the power consumption of a ContikiOS running on a STM32L152re was discoursed previously by R. Russo for STMicroelectronics®[29].
His work represents a good starting point working with Contiki and low power methodologies and gives a good overview of the problem.

The clock frequencies tested were $32MHz$, $16MHz$ and $12MHz$ with a fixed workload. Between $32MHz$ and $16MHz$ there is, as expected, a huge powersave, enabling the sleep mode during idle periods the powersave becomes slightly higher but, since the MCU load is higher the benefit starts to be marginal. At $12MHz$ the system shows some timing error related to the radio peripheral and packet reception. With a slower system all timeouts mechanism for the radio should be reconfigured but since the resulting power saving is not so relevant.

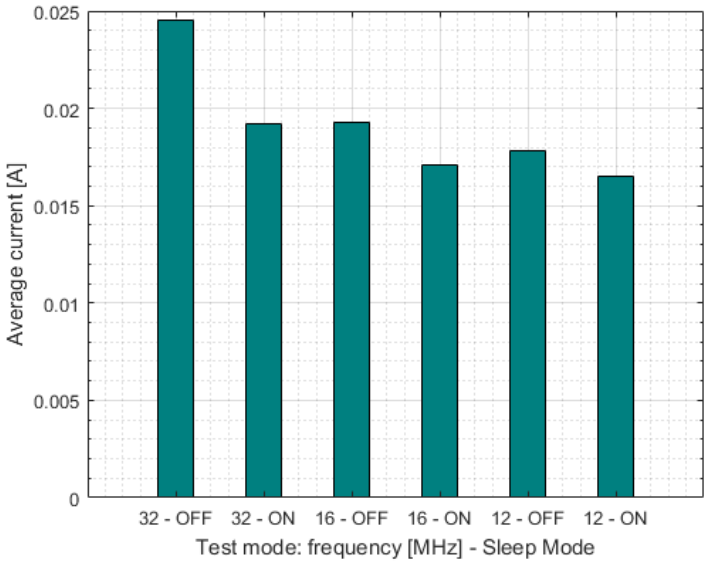


Figure 4.5: Average current consumption with different combination of clock frequency and sleep mode.

```
void SystemClock_Config(void)
{
...
RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
RCC_OscInitStruct.HSEState = RCC_HSE_OFF;
RCC_OscInitStruct.HSIState = RCC_HSI_ON;
RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL3;
RCC_OscInitStruct.PLL.PLLDIV = RCC_PLL_DIV3;
...
__HAL_RCC_PWR_CLK_ENABLE();
__HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE2);
...
}
```

Listing 4.2: Fragment of the system clock configuration function in cube_hal_l1.c file. The HSI oscillator (16 MHz) is enabled and selected as source for the PLL. The PLL is configured with a finl multiplication factor of one so that the resulting system clock frequency is set to 16 MHz. The last two lines are intended to enable the Range 2 for the internal voltage regulator

According to the above analysis, the configuration chosen for the microcontroller consists on a clock of $16MHz$ coming from the PLL with a multiplication factor of

1 (both *PLL_MUL* and *PLL_DIV* are set to 3, code 4.2 shows the actual implementation[1]).

The device family adopted supports dynamic voltage scaling to optimize power consumption in run mode. The voltage from the internal low-drop regulator that supplies the logic can be adjusted according to the system's maximum operating frequency and the external voltage supply[31]. There are three power consumption ranges:

- Range 1 (VDD range limited to 1.71 V - 3.6 V), with the CPU running at up to 32 MHz;
- Range 2 (VDD up to 3.6 V), with a maximum CPU frequency of 16 MHz;
- Range 3 (VDD up to 3.6 V), with a maximum CPU frequency limited to 4 MHz (generated only with the multispeed internal RC oscillator clock source).

Since the clock adopted is set to 16 MHz, the *Range 2* for the internal voltage regulator can be selected for an additional power saving. The last two lines in code 4.2 show how to properly set the Range 2 for the internal voltage regulator. To change the clock speed two other fundamental modification are needed in *platfom-conf.h* file:

```
#define F_CPU    16000000ul
```

and in system_stm32l1xx.c:

```
uint32_t SystemCoreClock = 16000000;
```

An optional adjustment in the baud-rate of the SPI peripheral that interfaces the board with the radio module will guarantee the same communication speed between the two devices avoiding possible timing issues when dispatching interrupt request or sending radio command.

GPIO and peripherals configuration

Before stopping the MCU the custom function *peripheral_lp_enable()* is called by the user thread to configure the GPIO pins, in code 4.3 a fragment of the function with the pins initialization is shown. Setting all the pins to the correct state is fundamental to nullify parasitic current drain through pull-down network or unexpected path. This part required a brief debug due to the presence of an EEPROM memory in the IDS01A4 board that, in case of high impedance input, will cause an unwanted current drain.

[1]Further information about clock trees and clock distribution can be retrieved in the STM32L1xxxx Reference Manual[30].

The following configuration guarantees that the MCU goes in stop state, the radio in standby and the EEPROM keeps a stable state:

- PA10(SDN): OUTPUT_PP + NOPULL + set to 0 (default value);
- PB6(CSN): OUTPUT_PP + NOPULL + set to 1 (default value);
- PA7(MOSI): OUTPUT_PP + NOPULL + set to 0;
- PB3(CLK): OUTPUT_PP + NOPULL + set to 0.

In brackets, next to the pin name, the pin function is shown.

```
static void peripheral_lp_enable()
{
...
GPIO_InitStructure.Mode=GPIO_MODE_OUTPUT_PP;
GPIO_InitStructure.Pull=GPIO_NOPULL;
GPIO_InitStructure.Speed=GPIO_SPEED_HIGH;
GPIO_InitStructure.Pin=GPIO_PIN_10 | GPIO_PIN_7;
HAL_GPIO_Init(GPIOA, &GPIO_InitStructure);

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_RESET);
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_7, GPIO_PIN_RESET);
...
GPIO_InitStructure.Mode=GPIO_MODE_OUTPUT_PP;
GPIO_InitStructure.Pull=GPIO_NOPULL;
GPIO_InitStructure.Speed=GPIO_SPEED_HIGH;
GPIO_InitStructure.Pin=GPIO_PIN_3 | GPIO_PIN_6;
HAL_GPIO_Init(GPIOB, &GPIO_InitStructure);

HAL_GPIO_WritePin(GPIOB, GPIO_PIN_3, GPIO_PIN_RESET);
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);
...
}
```

Listing 4.3: Fragment of code of the function peripheral_lp_enable. Pins 7 and 10 of group A and pins 3 and 6 of group B are set in output mode with no-pull and set to the proper value

The Spirit1 chip by STMicroelectronics® has very effective features in terms of low power in both standby and active transmission/reception state. It implements a *Low Duty Cycling Operation* mode that allows operation with very low power consumption, while at the same time keeping an efficient communication link. The Spirit1 can periodically switch on the receiver to check if a packet is being transmitted, and then go back to sleep in order to save power. This mode allows the reduction of average reception power consumption while allowing the device to continue receiving packets under certain conditions[32]. Moreover, this mechanism allows building a synchronized star network where both transmitter and receiver can sleep periodically to reduce average power consumption. The power reduction is proportional to the duty cycling level adopted. However, since the overhead of adopting this strategy is very low, advantages are noticed even for low rates[33].

Nevertheless, in this phase, we are not able to exploit this kind of feature. The

reason is related to the firmware adopted and the integration between the Spirit1 driver and ContikiOS. The implementation of several basic functions collides with the requirements prescribed by the LDCO mode of the Spirit1. Including this functionality in the driver practically means to rewrite the actual implementation. This is not a trivial task and falls outside the goal of this work. However, our approach and the modularity of ContikiOS keep this issues as a lower level problem that can be easily integrated at a later time.

4.4 Routing optimization

The default RPL configuration provided by ContikiOS sets all the parameters to a state working in almost every condition. To better fit the requirements of the application, only a few adjustments were performed in the code. For example, the time delay before the transmission of a DAO message as answer of a DIO packet was reduced to $1s$ to better fit the time requirements trying to speed up the discovery and maintenance processes.

In addition to minor adjustments, we remark also the use of an RPL network probing mechanism using DIO messages with a probing interval chosen randomly between 60 and 120 seconds. This interval may appear too short but, since it does not take into account all the interval in which the system is in low power mode, the effective number of probing messages sent in an hour is really low and distributed on different active slots.

4.5 Node Synchronization

Once the demo firmware was deployed on nodes and the basic functionality validated, our application idea was implemented by steps. The first step concerns the distribution of a common time reference for all the node. Achieving a low skew with as little message exchange as possible is the desirable goal. Both the solutions analyzed in section 3.3 were implemented and tested on the boards.

The solution based on multicast messages sent by the border router to all the nodes was implemented first. To enable the multicast engines, the following steps are needed [2] :

1. Include the following folder as a resource for the build process:

   `./Middlewares/Third_Party/Contiki/core/net/ipv6/multicast`

[2]See the official documentation for details.

2. Add the following lines in *project-conf.h*:

```
#define UIP_MCAST6_CONF_ENGINE UIP_MCAST6_ENGINE_ROLL_TM
#define ROLL_TM_CONF_IMIN_1 1
```

Between the border router and a node connected directly (no hops) there is a time drift of less than ~50ms. This difference derives from the communication protocol delays and in general it can be treated as a known offset and manually adjusted. However, with this approach, we have to consider also the time required by a node to forward a message according to the ROLL Trickle Multicast engine. Parameters for the ROLL TM protocol can be configured but, Contiki developers' experience suggests to use time interval of 125 ms or higher[34]. This results in a time drift, spread all over the network, proportional to the distance (number of hops) of each node from the border router. Time drifts are in the order of several hundreds of milliseconds. If the active time windows is set to be very short, this could be very critical and the system may be even not working. The second approach is based on the local synchronization between a node and its parent. A node gets synchronization message from its parent instead of the border router, hence, each time a node receives a sync message from its parent, it has to send the updated time to its neighbors. Doing this way the overhead in channel occupancy is almost the same to the multicast version, but this method presents better performance in terms of time drift for nodes deeper in the graph.

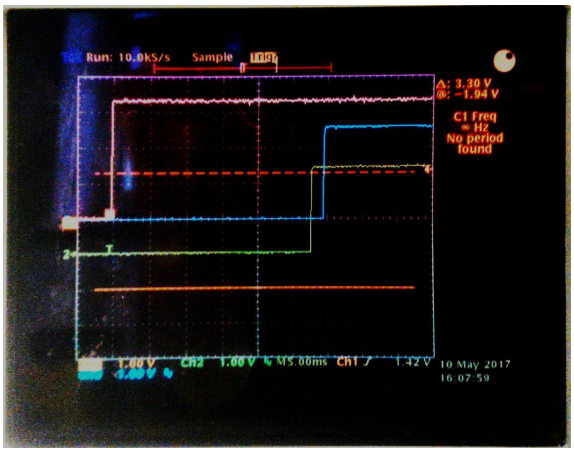


Figure 4.6: Screenshot of the oscilloscope triggering the calibration signal for the Nucleo board. The red trace relates to the preferred parent whereas the blue and green traces are two neighbors already synchronized. The skew between the two neighbors node is very low, whereas the skew between time source and nodes is a bit higher.

The time drift, this time, depends only on the propagation delay of the message since we are removing the multicast forward delay, reducing the delay related to a hop in the network to ~30ms as shown in figure 4.6.

To enable the distribution of the time messages an UDP channel is instantiated in each node and the callback function, shown in code 4.5, is used to manage the incoming message containing the updated timestamp from the parent.

A synchronization message contains all the field corresponding to the RTC calendar register including time format, hours, minutes, seconds, the second's fraction, and its granularity. Other two field are used as control fields, one specifies the type of message (useful in case of future extension) and the other one specifies if the low power mode for the node has to be enabled or disabled. A listing of the sync message structure is shown in code 4.4.

```
typedef struct
{
 uint8_t          MessageType;
 uint8_t          TimeFormat;
 uint8_t          Hours;
 uint8_t          Minutes;
 uint8_t          Seconds;
 uint32_t         SubSeconds;
 uint32_t         SecondFraction;
 uint8_t          RunMode;
}Sync_MessageTypeDef;
```

Listing 4.4: Synchronization message type definition.

A minor issue arose in this phase in both configurations due to a characteristic of the RTC management. In detail, the hardware does not allow to set the sub-second fraction field, but only to set it to 0 when the hour, minute and second values are written, fixing the RTC accuracy to a second. Clearly, this is a huge limitation to our application but a workaround was implemented. In fact, even if this field cannot be written, it can be easily read by the system. To address the problem and reach a higher RTC consistency, a node receiving a synchronization message schedules the update of the timestamp to the beginning of the following second. In this way, the problem is partially overcome. The proposed solution may require some adjustments because in particular condition the chain of sync message from sink to leaf can exceed the working windows and leaf node may not receive some sync message. This is not critical but it is still an unwanted behavior. A viable solution consists in propagating synchronization messages asynchronously. Each node may keep an information about the freshness (consequently the goodness) of its synchronization and propagate its timestamp unless the freshness overcome a certain threshold, becoming outdated. In this way, even the channel occupancy is enhanced since messages are randomly distributed over the whole active window.

A statistical analysis on propagation and message processing delay, in both methods, can lead to better performance in terms of synchronization and consequently

allowing smaller active time frame to reduce the wasted energy. By statically setting an offset, from figure 4.6 25 or 30ms can be a reasonable value, the time drift between nodes can be drastically reduced at a very low cost. However, this assumption may not be valid in general, resulting in a further error introduction. For this reason, we preferred to not implement it for the moment.

To validate the implementations showed above we adopted two different methods. An oscilloscope was used to verify at some interval the synchronization of two nodes and a common reference corresponding to the border router. Three channels were adopted exploiting a characteristic of Nucleo boards that specifically allows to export on a pin a 1Hz calibration signal connected to the Real Time Clock (RTC). An example of the visual feedback from the oscilloscope is shown in figure 4.6 already mentioned. The oscilloscope gives us very precise measures but, due to its characteristics, it is not suitable to collect and elaborate statistics on long time periods. Hence, a parallel software solution is adopted using a fourth Nucleo board. Calibration pins from target board are connected to the probe board and associated to the internal timer for period measurement. In this way, the three calibration signals are expressed according to a common reference, the probe board oscillator, and the result is sent as raw data to a PC through the serial port. A simple Matlab script was used to manage and elaborate the data coming from the probe. Results were, as expected, coherent with previous observation but doing in this way, we have also the ability to analyze the system's behavior on a wide time window greater than the single shot provided by the oscilloscope. Figure 4.7 shows an example of the delays behavior.

The two implementations differ only in some aspects of the source code so the firmware was modified keeping both implementations and allowing the switch between them by using the `SYNC_TYPE` macro.

4.5.1 Application

The user application running on the client nodes consists mainly in a process thread. Process thread in Contiki is a single *protothread* invoked by the process scheduler. Protothreads are the way in which Contiki allows the system to run other activities when the code is waiting for something to happen. Depending on the event received by the system the scheduler invokes the corresponding process polled by calling the function that implements the process thread. Hence, the interaction between processes happen through events that can be synchronous or asynchronous depending on the way they are delivered to the process.

In our application important events are mainly related to the interrupt line associated with the radio module. Incoming packets have to be managed and the system has to react to the information received. An example of the two main user

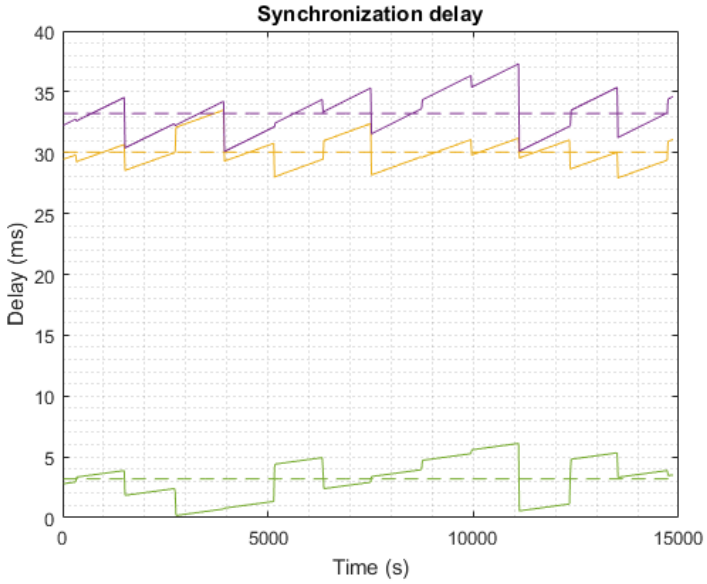


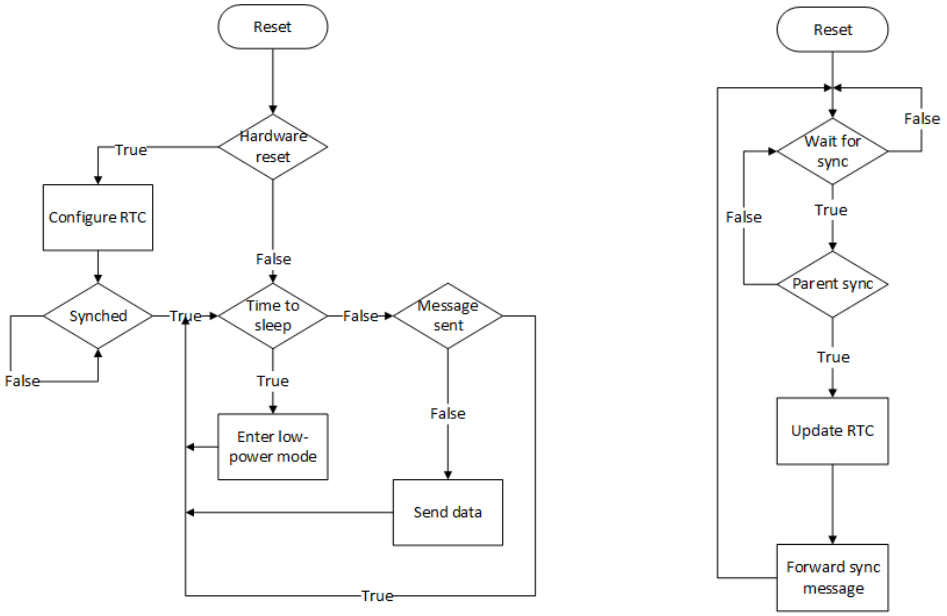
Figure 4.7: Delay behavior of two node over some hours. Purple and yellow lines represent respectively the delay between node 1 and node 2 with respect to the time reference at a distance of one hop. The green line show the relative delay between two nodes. The slopes are caused by a not precise calibration of the internal oscillator.

processes developed for the application is shown in figure 4.8. An additional timer on the client side guarantees the correct timing to the application by informing the scheduler if a process needs to be polled at a certain time.

The same mechanism is adopted by Contiki through interrupt lines for the management of the radio. In that case, the timing requires to be very accurate in order to avoid packets loss or errors in the communications.

Minor issues

An unexpected problem was encountered during the validation with the X-NUCLEO-IKS01A1. This board embedded several sensors that interface themselves with the MCU through a I^2C protocol. In our application the 3D accelerometer and 3D gyroscope sensor (LSM6DS0) is not used, hence we disconnected the jumper on the power supply line for this sensor to avoid an unwanted power consumption by an unused component. Despite this is the correct way to turn off an unused sensor, the system was not working correctly. In particular during the startup the initialization of the sensors failed most of the time. After a short study, the reason of the failures was identified in the I^2C communication protocol. Disconnecting a sensor from the X-NUCLEO-IKS01A1 means to not supply it, but the chip still remains connected to the I^2C data and clock lines. Hence, inside the not powered chip, a discharge



(a) Client process flow (b) Synchronization mechanism process flow

Figure 4.8: Flowcharts with the main points of two application processes execution. Figure a relates to the client user application, figure b relates to the node synchronization mechanism. The two flows run in parallel depending on the scheduler activity.

path for the I^2C lines is created, preventing the pull-up resistor to correctly drive the I^2C lines to the high voltage condition. The result is a not working connection between MCU and the other sensors attached to the I^2C lines. To solve the problem we had to physically disconnected also the I^2C wire from the sensor by un-soldering the corresponding 0Ω resistors from the board, figure 4.9 shows the placement of the two resistors on the board.

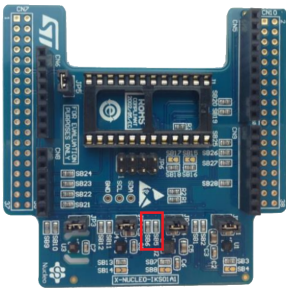


Figure 4.9: Resistor SB5 and SB6 have to be removed from the board to completely isolate the LSM6DS0 chip from the rest of the board, avoiding any interference.

```
static void
receiver(struct simple_udp_connection *c,
const uip_ipaddr_t *sender_addr,
uint16_t sender_port,
const uip_ipaddr_t *receiver_addr,
uint16_t receiver_port,
const Sync_MessageTypeDef *data,
uint16_t datalen)
{
RTC_TimeTypeDef RTC_TimeStructure={0}, RTC_TimeStructure2={0};
RTC_DateTypeDef RTC_DateStructure={0};

static uip_ipaddr_t *pp_address, sa_address;
int drift_table[10];
int i;

HAL_RTC_GetTime(&RtcHandle, &RTC_TimeStructure2, RTC_FORMAT_BIN);
HAL_RTC_GetDate(&RtcHandle, &RTC_DateStructure, RTC_FORMAT_BIN);

sa_address=*sender_addr;

pp_address = rpl_get_parent_ipaddr(dag->preferred_parent);

for (i=0; i<8 && pp_address->u16[i] == sa_address.u16[i]; i++);

if(i==8) {
if (data->MessageType==SYNCHRO){
RTC_TimeStructure.TimeFormat = data->TimeFormat;
RTC_TimeStructure.Hours = data->Hours;
RTC_TimeStructure.Minutes = data->Minutes;
RTC_TimeStructure.Seconds = data->Seconds + 1;
HAL_Delay((uint32_t)(1000-1000*(\
data->SecondFraction-data->SubSeconds)/(data->SecondFraction +1)));
HAL_RTC_SetTime(&RtcHandle, &RTC_TimeStructure, FORMAT_BIN);
enable_stop=data->RunMode;

synch=1;

HAL_RTC_GetTime(&RtcHandle, &RTC_TimeStructure, RTC_FORMAT_BIN);
HAL_RTC_GetDate(&RtcHandle, &RTC_DateStructure, RTC_FORMAT_BIN);

mcast_message.MessageType = SYNCHRO;
mcast_message.TimeFormat = RTC_TimeStructure.TimeFormat;
mcast_message.Hours = RTC_TimeStructure.Hours;
mcast_message.Minutes = RTC_TimeStructure.Minutes;
mcast_message.Seconds = RTC_TimeStructure.Seconds;
mcast_message.SubSeconds = RTC_TimeStructure.SubSeconds;
mcast_message.SecondFraction = RTC_TimeStructure.SecondFraction;
mcast_message.RunMode = LP_MODE_ON;

simple_udp_sendto(&unicast_connection, &mcast_message,\
sizeof(mcast_message), &mc_addr);
}
}
}
```

Listing 4.5: Callback function associated with the receipt of a synchronization message using local synchronization between a node and its parent using unicast messages.

Chapter 5

Network simulation

In the context of creating a functional framework for the design of a wireless network, software tools are fundamental. In particular, a network simulator and a data processing software were adopted. Combining the data coming from the network simulator and the energy harvester monitor, an almost exhaustive description of the system behavior can be elaborated.

This chapter, first, shows an overview on the software employed and its main functionalities. Then, it shows a description of the configuration required for setting up the environment to the conditions of a real deployment. Finally, the simulation is described and results are presented.

5.1 Cooja Simulator

Cooja is a network simulator specifically designed for Wireless Sensor Networks. Cooja allows the simulation of large and small networks of motes[1] running Contiki OS.

Cooja is a powerful tool for Contiki development as it allows developers to test their code and systems long before running it on the target hardware. It allows analyzing RPL and network behavior both at the hardware level, which is slower but allows precise inspection of the system behavior, and at a less detailed level, which is faster and allows simulation of larger networks[35].

The software is very intuitive to use. It is made up of several customizable windows showing different aspects of the simulation. The *Network* window is used to physically place and move physically motes in the environment. Several labels can be added to each node showing, for example, the node ID, the current output of the serial port, the network address, the coordinates, etc. In this window an interesting

[1]Motes is the name used by Cooja to indicate nodes. From now on, 'mote' and 'node' will be used without any distinction.

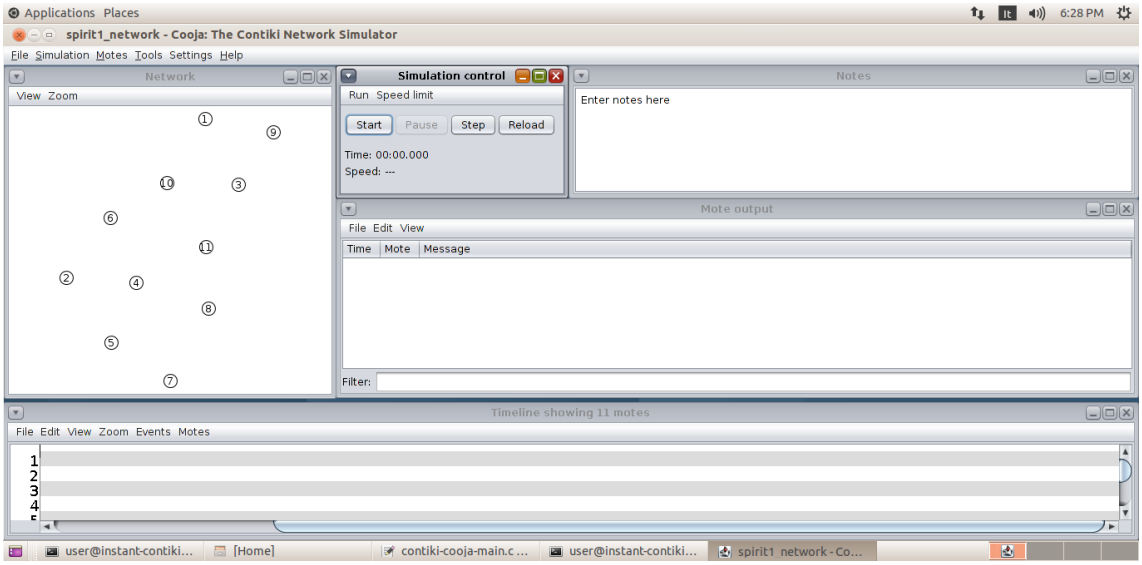


Figure 5.1: Cooja GUI

feature is given by the *mote relation*, consisting in a red directed arrow that joins a mote to its current preferred parent. In this way, at any instant, we can have a snapshot of the network topology and we can easily evaluate the behavior of the objective function used. Another useful section is the *Timeline* window where the activity of each mote is logged, showing information about the radio hardware status (on or off) and its activity (radio transmitting, receiving, idle, interfering). For simulation with huge networks or for a long time, the graphical version may become practically unusable, but statistics can be easily saved in a summary log file that can be analyzed at a later time using other software like Matlab®.
The serial port output of the nodes can be visualized in several manners. The user can choose to see a dedicated window for each mote or a window combining the output of all the nodes sorted by the arrival time. Both views are very useful debug tools and their content is included in the log file previously mentioned.
Other windows include the *Simulation panel* for controlling the simulation and a notes editor.

The simulation speed is a crucial aspect for a simulation software. A real-time simulation can be required in several validation checks, mostly in the earlier phases, but often a check on the long term behavior is needed. The main reason is addressable to the fact that once deployed, nodes can be hard to reach for manual intervention on the firmware, and even if On-The-Air reprogramming is available, minimizing issues related, for example, to not a optimal configuration for the objective function or to application bugs or refinements, is always preferable. Addressing in advance issues is always the best practice, but tools should help. Typically hours

of execution can be simulated in tens of seconds but performance depends mostly on the complexity of the application and the size of the network.

5.2 Software configuration

As said, Cooja allows using the same firmware used by the application for the simulation. Unfortunately, some sections of our application are strictly related with architecture specific drivers not available on Cooja, hence, some modification in the code is required and a way to reproduce the same behavior have to be found.

The hardware equipment adopted in this thesis is not yet listed as a supported platform by Cooja, hence, no hardware level emulation can be used. This is not a major issue since we are interested in simulating the network from a higher level perspective, but a generic node model will be used.

The first issue relates to the concept of RTC clock that in the real application has the central role of scheduling wakeup intervals and it is periodically adjusted for keeping the synchronization between nodes. The emulator does not support variable drift in time reference but only a static offset fixed on the startup. This simplification does not compromise the simulation consistency and we can consider still valid the system. During the sleep periods of the real mote, the *Systick interrupt* is suspended. This means that from the OS point of view, the time is freezed until the next wakeup drived by the RTC wakeup timer. By modifying the *clock_time()* function in *clock.c* of a node we are able to replicate the same behavior observed on the real node when turning to the low power state. Practically we suspend the node clock tick (the really same action is done in the real implementation by disabling the interrupt on the *systick*) of the node. The code shown in 5.1 and 5.2 highlights the modification performed, in particular the variable flag is introduced in the main file to signal the simulator if the node is in low-power mode or not.

The previous modification is enough from the firmware point of view, but from the emulator perspective, the result is not optimal. Since the emulator works in an event-driven fashion, the current configuration will force the simulator to check status variation even during low power phase when the Operating System is inhibited. The result is a low-speed simulation during phases that are practically ineffective. The solution is easy to implement and consists of setting the next expiration timer event to the instant when the system is intended to wakeup. The function to modify is contained in *contiki-cooja-main.c* file, the modified version of the function is shown in listing 5.3. Turning on and off the radio is not an issue since it can be easily managed using the radio driver interface of Contiki in the analogous way done for the real hardware. All the low-level functions used to prepare core peripherals to enter and exit the low power mode are commented out since they are not supported

Listing 5.1: Original code

```
clock_time_t clock_time(void)
{
  return simCurrentTime;
}
```

Listing 5.2: Modified code

```
clock_time_t clock_time(void)
{
static int last;

if(flag==1){
  if(simCurrentTime<expire)
  {
    return last;
  }
}

flag=0;
last=simCurrentTime-delay;
return simCurrentTime-delay;
}
```

```
JNIEXPORT void JNICALL
Java_org_contikios_cooja_corecomm_CLASSNAME_tick(JNIEnv *env,
                                                 jobject obj)
{
        ...

        ...
  /* Save nearest expiration time */
  if (flag==1)
    simEtimerNextExpirationTime = expire;
  else
    simEtimerNextExpirationTime = etimer_next_expiration_time();
}
```

Listing 5.3: contiki-cooja-main.c

by the abstract model of the mote.

When a new simulation is created, see fig. 5.2, a mote delay and the model to use for the Radio Medium must be specified. The mote delay is useful to simulate the small synchronization drift between nodes and avoid the unrealistic perfect synchronization of two nodes that may also result in an increase of unrealistic conflicts on the channel due to the almost identical behavior of the nodes. Cooja is set to pick a random number between 0 and the specified number of milliseconds for each node.

The Unit Disk Graph Radio Medium abstracts radio transmission range as circles. It uses two different range parameters: one for transmissions, and one for interfering with other radios and transmissions. No particular analyses were performed on this aspect.

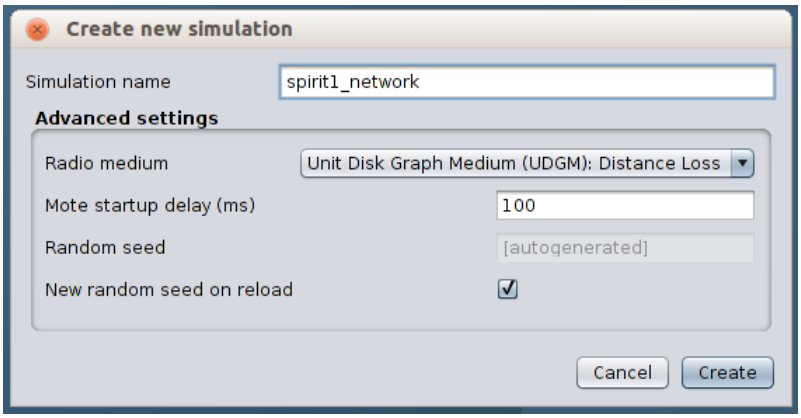


Figure 5.2: Cooja new simulation creation menu

To add a node, the mote type to be selected is the *Cooja mote* and the firmware to be compiled in the same way we do for the real node.
Starting from the `\examples\ipv6\rpl-border-router\border-router` project for the sink node and `\examples\ipv6\simple-udp-rpl\unicast-sender` for all the other nodes. Once added, the nodes can be graphically placed in the *Network* window even using mouse drag and drop.

5.3 Simulation

The simulation can be launched by using the *Simulation control* window and selecting a speed limit, as a percentage. The behavior of the system can be analyzed in run-time, selecting a reasonable speed limit, or at the end of the simulation by collecting the output given by the available tool (mostly using the serial output and the activity log). The mote serial output window gives a visual feedback of the packet sent/received ratio highlighting eventual conflicts on the channel. The activity log, instead, is a textual file storing all the information about a node including the activity of the radio (on, off, receiving, transmitting and idle) and the serial output of the mote.

Several simulations have been performed to understand deeply how the network behaves both in terms of radio activity and in packets received/sent ratio. Both aspects have been inspected analyzing the activity log with Matlab®. Figure 5.3 shows the radio activity of a node during the active phase when the node receives

the synchronization message, forwards the synchronization message to its neighbors and sends to the sink a dump message simulating a sensor measurement.

The duration of a transmission at 100kbps (but even for lower bit-rates) is very short with respect to the total activity windows, hence in the plot, it appears as a spike. Moreover, it's impact on the total power consumed is very marginal.

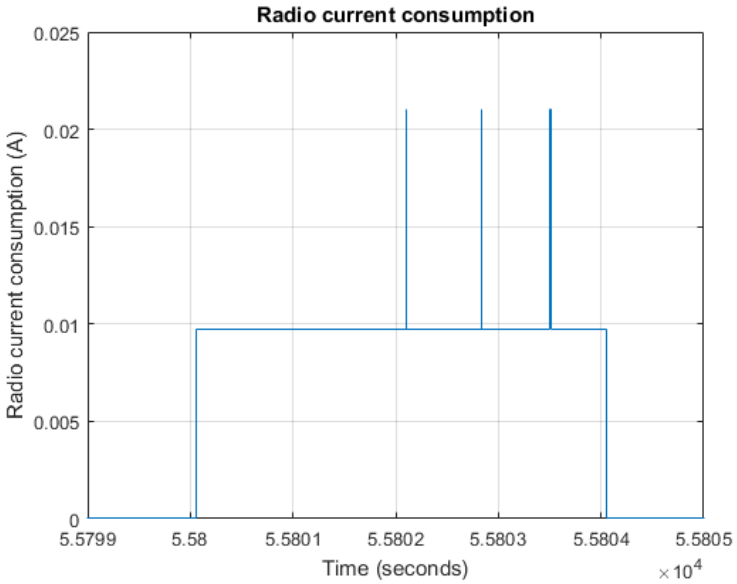
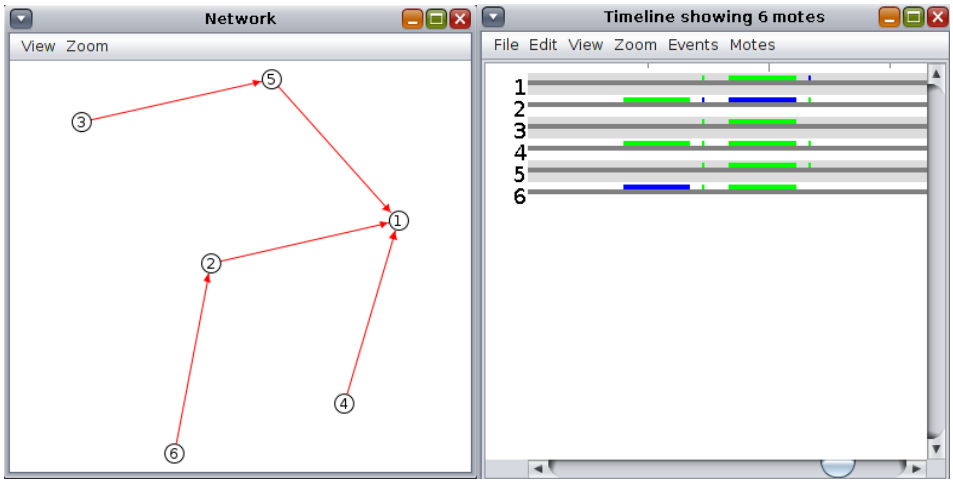


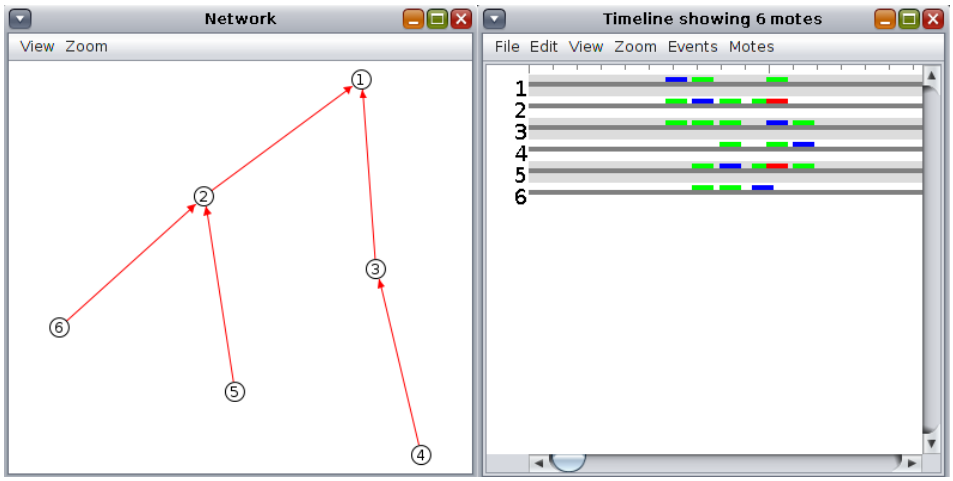
Figure 5.3: Graphical representation of the radio current consumption evolution for a mote during the active window.

Figure 5.4 shows two examples of the timeline windows where two different node activities are inspected (message transmission and synchronization messages distribution). For simplicity, examples show a network composed of a small number of nodes. During functional validation, larger networks (with up to sixty nodes randomly placed) were simulated.

Using the simulation several aspects of the application were investigated. For example the different power consumption between leaf nodes and router nodes that have to manage high traffic. A bit surprisingly, the simulation did not highlight significant differences in power consumption. The reason can be easily identified with the poor contribution that the transmission power takes to the overall with respect to the receiving/idle state component. Exploiting a radio duty cycling mechanism, with a huge reduction in radio power consumption will also increase and make remarkable this aspect.



(a) Message propagation from node 6 to sink (1)



(b) Synchronization message distribution

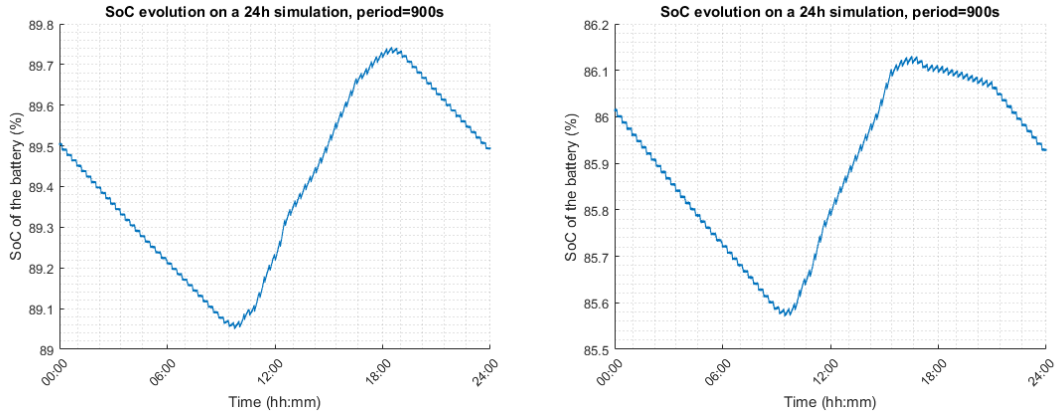
Figure 5.4: Cooja examples showing two kinds of transmission activity through the network and timeline windows. Blue bars represent the transmission of a packet by the relative mote. Wide bars refer to packets containing an application payload whereas smaller ones are the acknowledge message sent by the destination node. Green bars represent motes receiving a packet. Only the recipient at the end of the transmission sends back an *ack* to the sender. The sink sends first its timestamp, and, in sequence, even other nodes forward their timestamps to deeper nodes. Blue bars represent the transmission of a packet by the relative mote, whereas green bars represent motes receiving a packet. More precisely the bar says that the mote is sensing a message on the channel but it can be intended to another mote. The red bar shows a collision on the channel due to multiple nodes sending at the same time. The collision shown in this example has no effect since does not affect the behavior of any node, but, for dense networks, critical collisions are very likely to happen.

The packets received/sent ratio in simulation remarked a very good behavior for the network with percentages higher than 99.5%. However, these results show only how the network behave without any external interference. Environmental interference and sporadic issues affecting the radio and the channel decrease the real packets received/sent ratio, hence, a low-grade redundancy will assure the successful delivery of all the packets.

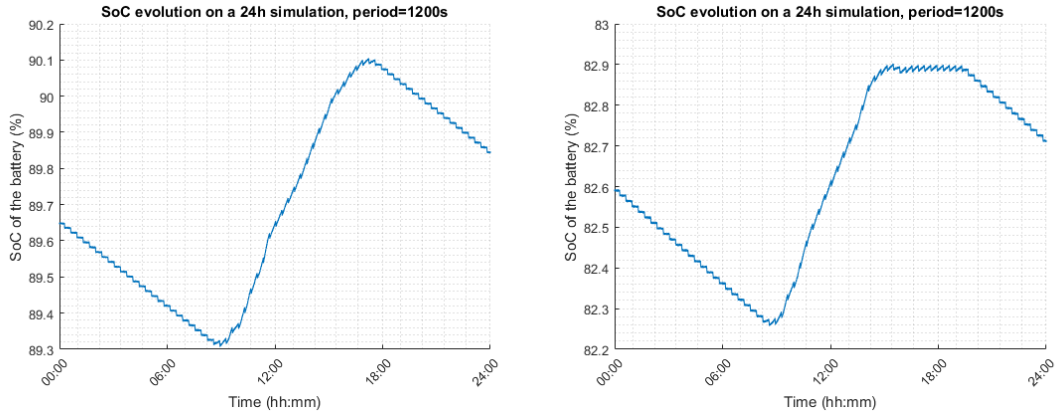
5.4 Simulation analysis

In this work the simulation aspect has a double importance. Firstly, the opportunity to analyze accurately all the nodes composing the network concurrently spanning a very big time window gives an extremely powerful tool to debug the network and to supervise the behavior of each node. Then, a simulator reproducing the network can be used to model the energy consumption of the node and at the same time be used to estimate the capabilities of the network. In this phase, data about the energy consumption and the energy harvested are combined using Matlab® at the end of the simulation. Some different profiles of various configurations and different harvesting profiles are shown in figure 5.5. The plots are realized using 2 discrete simulations adopting respectively a duty cycling period of 900s and 1200s. The curves shown represent the *State of Charge* (Soc) of the battery. It represents as a percentage the residual capacity of the battery over the total. The simulations are then correlated with two different harvesting profiles taken in two distinct days. This kind of representation has a characteristic sinusoidal shape where the minimum corresponds to the light of dawn or to the beginning of a working day when lights are turned on. Then the curve profile tends to raise or at least keep a stable trend until light is no more sufficient to supply the system and the battery is the only power source. The gap between the starting point and the arrival point of the state of charge represented indicates the daily energy balance. Figure 5.5a is characterized by an equal level for the Soc at midnight for the plot on the left, whereas for the harvesting profile used in the right plot it shows a negative energy balance. Figure 5.5b has a very similar behavior but this time it is clear how the energy balance is positive in both the profiles.

The daily surplus energy resulting from such luminance conditions is fundamental to compensate days with very adverse luminance conditions, figure 5.6 shows two very critical cases. The saw-toothed edge of the plotted curves can be easily explained with the alternation of on and off periods typical of a duty cycling mechanism.



(a)



(b)

Figure 5.5: Comparative plots of two harvesting profile with two different network simulations.

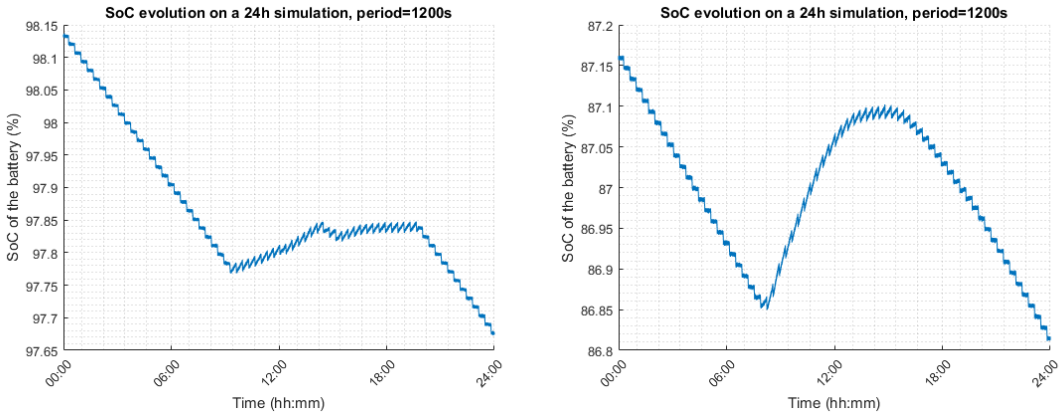


Figure 5.6: Harvesting profiles combined with a network simulation showing a critical behavior with a negative energy balance over a day.

Chapter 6

Experimental results

The work carried out has been analyzed from several points of view in order to obtain an overall view and a summary characterization useful to evaluate the system in its entirety.

In this chapter the experimental results obtained are exposed. First the power harvesting and power consumption aspects are shown separately, then they are merged in an overall concept and some consideration on final results is exposed.

6.1 Energy harvesting

The data collected by our custom power monitoring board show how, in a 24 hour time window, the power budget available varies between 80 and 200 μA depending on the placement of the solar panel. Figures 6.1, 6.2 and 6.3 show a graphical representation of the collected data. They represent three different profiles depending on different light exposures. These values need to be examined accurately, since they represent the crucial point of the whole analysis.

Depending on the requirements of the application the final power budget can be employed for several uses. To achieve a total energy independence from external power source some strategy can be discussed and correlated with the performance requirements.

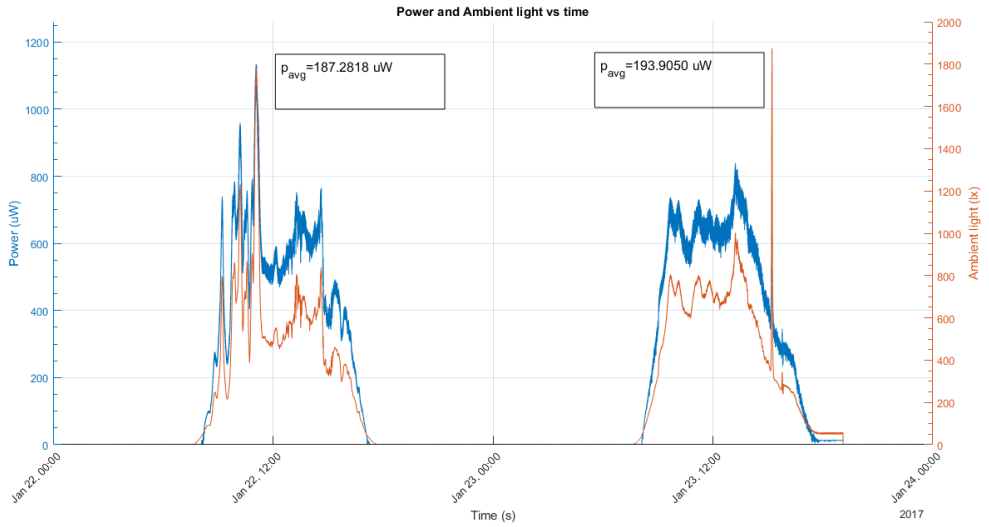


Figure 6.1: Chart of the power monitor output showing the time trend of luminance level and power output during two consecutive days. The acquisition refers to two sunny days in january (9 hours of daylight in Turin) during working days.

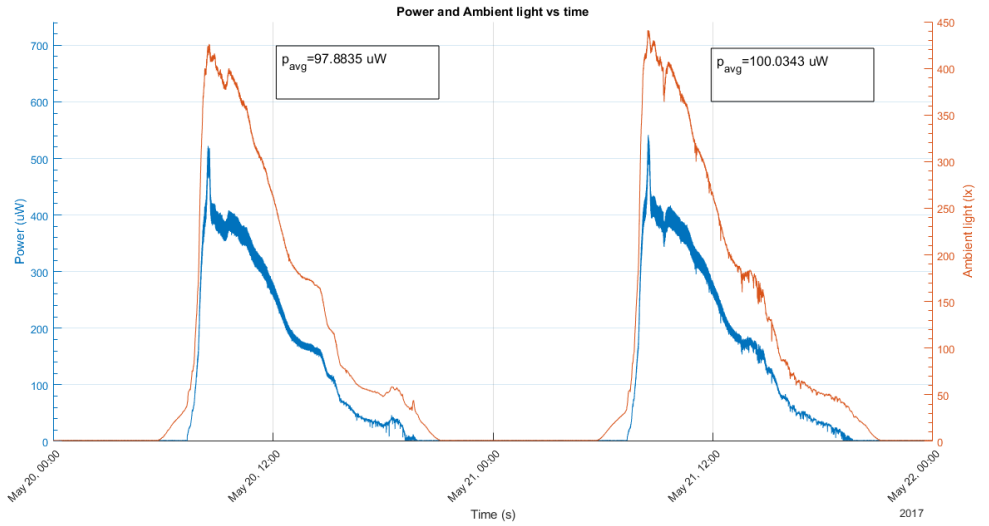


Figure 6.2: Chart of the power monitor output showing the time trend of luminance level and power output during two consecutive days. The PV is positioned on a desk inside the lab far from windows and without artificial lighting.

6.2 Node power consumption

The power consumption observed in our measurements is coherent in almost all the cases with the theoretical values extracted from datasheets. In particular, the

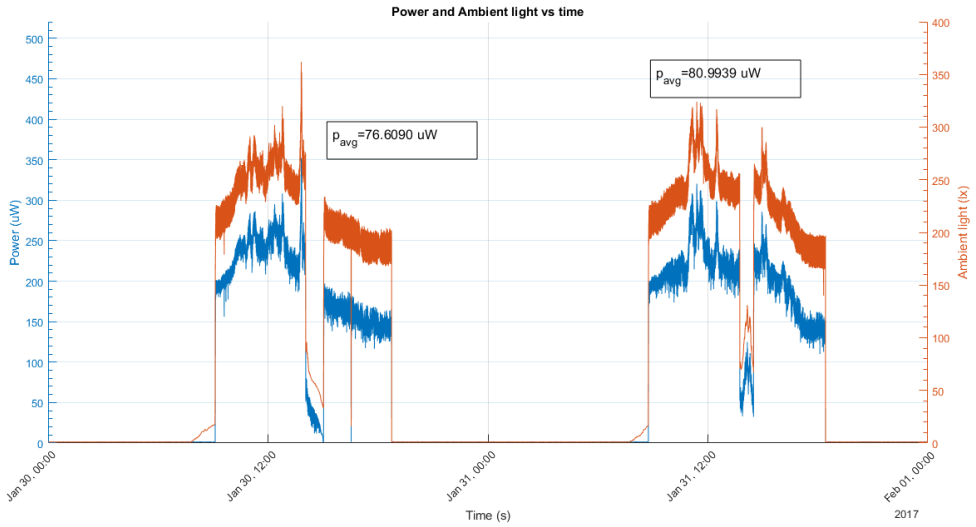


Figure 6.3: Chart of the power monitor output showing the time trend of luminance level and power output during two consecutive days. The PV is positioned on a desk inside the lab far from windows during a cloudy day in january where the light source is exclusively artificial. The two components, artificial and natural can be clearly distinguished in this case. During the hour and half in which the artificial lighting was switched off the natural light luminance is in the order of only 50lx or lower with a few tens of μW harvested.

measures of the current absorbed by the system are shown in the following table:

	MCU	Radio	Radio Tx	Radio Rx	Total
Stop Mode	~0.3 μA	~0.1 μA	-	-	~0.4 μA
Run Mode	3.8 mA	-	21 mA	9.7 mA	24.8mA / 13.5 mA

Values shown in the table highlight what we already know. The critical part of the system is represented by the Radio module power consumption during receiving mode, since the radio never goes idle during on mode, hence we can not exploit the power saving opportunity given by this mode. During transmission the current consumption is very high but, since a transmission lasts only few milliseconds, it does not affect significantly the total power. The radio is in reception mode for all the active time, hence the power consumption during the on time can be approximated with the current consumption of the radio during receiving mode of 13.5 mA. The optimization done on the network management and the use of the already discussed duty cycling mechanism reduce significantly the power required by the system to perform its task. However, the current absorbed by the radio places a considerable limitation on the global performance.

6.3 Results discussion

Once all the needed parameters are collected, several considerations on the final network capabilities can be discussed.

The first goal and the most desirable achievement consists in an always-on and self powered device, able to configure itself at power-on and reset and rejoin the network autonomously in case of error or malfunctioning. ContikiOS and the firmware provided by STMicroelectronics® are able to guarantee an high stability from what concerns first configuration and auto-reset mechanism. The energy capabilities of the harvester and its working conditions dictate some limitations on the always-on aspect.

As discussed in previous chapter, a duty cycling mechanism with long periods is chosen as low power strategy at the cost of a high latency in node reactivity but a very effective powersaving. Guaranteeing an always-on properties means, practically, to find a good dimensioning for the off period whereas the on period is fixed a priori. An example of the correlation between the average luminance level and the period of the duty cycle is shown in figure 6.4. The curve trace is a coarse estimation that gives a good panorama over the magnitude of luminance and time period involved. From measurements we can reduce the portion of the graph in the interval between 50lx and 500lx. Lower values represent very critical condition whereas higher luminance levels, even if they can be easily reached, are still considered special cases. In the considered range the time period to be adopted spaces between 500s and 1800s, as to say between 7 and 30 minutes. To be conservative 30 minutes can be a reasonable value to guarantee an high reliability. In real deployment dark and light day are alternating, hence the storage battery will spread the power budget over several days increasing the daily available budget.

In some applications, adding a constraint can be discussed. The constraint consists in adopting a variable time period depending on the time of the day. For example during the night can be reasonable to relax off periods in order to save power when the system is not required to work. This solution can be very effective if the application allows being silent during dark hours, the improvement depends on the time period adopted during the night and in case of total switch off during night (extreme case) the daily power budget can be roughly doubled. On the other hand, the application must be compatible with a design choice like this and in general, it can be considered a reduction of functionalities that may be not preferable.

Coupling the solar power source with a small battery pack (larger than the 120mAh battery integrated in the harvester) can be an interesting solution to the

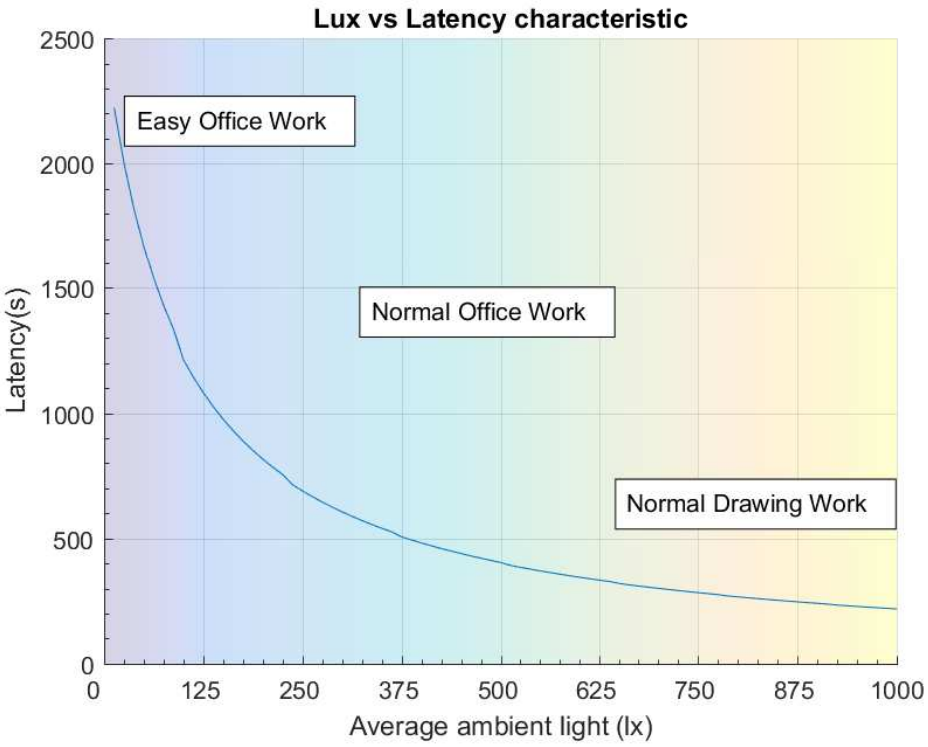


Figure 6.4: Luminance vs latency (duty cycling level) for energy autonomous system

problem of guaranteeing high autonomy and good performance also considering different seasons. In this case, the autonomy is not infinite since it relates to the combination of solar harvesting and battery charge. Since the current provided by the battery is not the primary power source, the system may, theoretically, work for thousands of days, as to say to be almost energy autonomous. Practically, battery performance is not good as expected on time interval so long, therefore the real charge may result less as expected.
The last approach proposed collides in part with the presupposition of this work, namely the design of an energy autonomous system. It has been shown only to be thorough since it practically represents a viable solution when addressing the problem.

In general, a self-configurable network that each 20 minutes wakes-up, collects and sends data to the central node autonomously with a good stability can be considered a satisfying result. This configuration has been extracted from the simulations and verified by an empirical test. An example of a three days simulation is shown in figure 6.5.

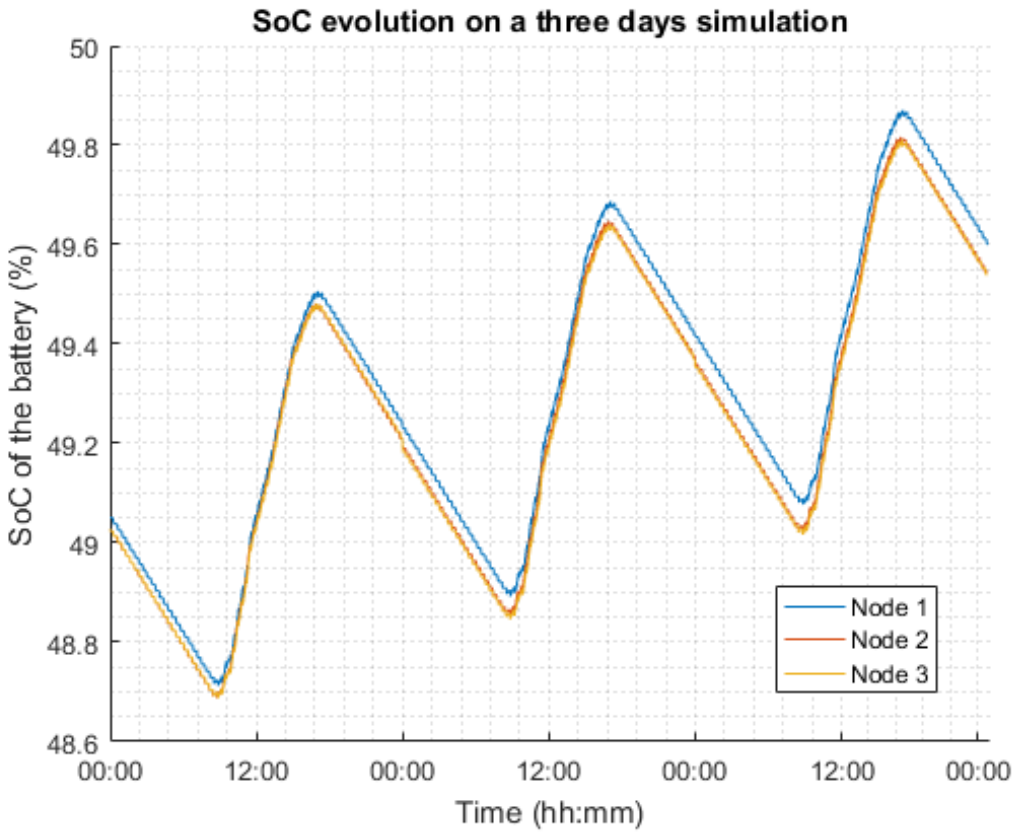


Figure 6.5: Evolution of the state of charge of the battery during a 3 day simulation. The three nodes show extremely similar behaviors and traces are almost overlapped. The final balance is slightly positive for the days considered.

The tools and the models defined constitute the core of a framework that gives the guidelines to the creation of an energy-autonomous wireless sensors network. The energy characterization, the network simulation and all the tunable parameters allow to exhaustively design a trustworthy network giving the ability to the user of balancing as better as possible the application requirements and the restrictions coming from the energy aspect.

Chapter 7

Conclusion

Nowadays, the interaction between humans and the environment in which they move needs more and more electronics and devices. Objects partecipating to the world of the Internet of Things are imposing a new paradigm in the interaction of people with the world. Market motivation and the requirements of a smart society require more and more attention to the development of such technology. On the other hand, recent technology improvements constantly raise the bar of the possibility in innovation, making feasible devices incredible from all the point of view including the size or the ability to dialog with the environment becoming an essential part of the environment itself.

In this work, the focus was on the research and the development of an easy to deploy and trustworthy Wireless Sensor Network able to provide high sensing and even actuating capabilities to an unlimited range of environments with an extremely high level of customization. The point on which we focused our investigation revolves around the power aspect, hence the power consumption requirement of the nodes which make up the system and the way in which the required power supply is retrieved. Beside the goal of creating a stable sensors network, the firm requirement of powering the system by using green energy has been placed, in particular harvesting power from solar energy.

Exploring the potentiality of alternative power sources encourages the commitment to maximize the power harvesting capabilities and at the same time minimize the power requirements to fit the lacking power availability

The first part of the work aimed at characterizing the solar energy harvester used for supplying the system. In order to effectively measure the power output capabilities of the module, a test circuit was specifically implemented to perform the task. In addition to the characterization of the power module, this analysis was used to define some profiles for the ambient light of a typical office and to understand which are the limits and the expected values for the energy that can be collected from the environment.

Then, I focused on the energy consumption of the node, in particular considering the power required by the microcontroller. The several available low-power modes were used to minimize the energy consumption during both active and idle phases, exploiting voltage and frequency scaling as well as selective core and peripheral deactivation. The radio module resulted very critical from the power consumption point of view. In order to guarantee the functioning of the system using the solar harvester, a duty cycling mechanism for the nodes was adopted. It consists in allowing the nodes to go in a low-power mode where the system is frozen, and periodically waking them up for a small amount of time sufficient to acquire, process and send data to the sink, and to maintain the network connectivity. The amount of time in which the system stays on and off can be easily configured to match application requirements and power supply capabilities.
The duty cycling mechanism requires to precisely synchronize nodes to guarantee the consistency of the network and make it work correctly. The time distribution between nodes required some attention so as not to generate excessive traffic on the communication channel but, at the same time, to ensure a sufficient level of synchronization (in the order of few tens of milliseconds).

In parallel with the hardware implementation, all the concepts related to the network were tested using a network simulator that was customized to accurately replicate the behavior of the system. The simulation, based on the Cooja software, was used, at first, to test the goodness of the mechanism adopted for the network management before deploying them in the real node. Then, the software was used to validate the final version of the firmware by analyzing long time periods, unpractical to be tested experimentally.

The average output power of the scavenger during a day is in the order of $180\mu W$. This allows the system to acquire data every 20 minutes relying only on the harvester energy, with a surplus of a few μW useful to compensate small variations in daily light conditions.

A further work may integrate also the power estimation inside Cooja. This can bring a lot of advantages allowing for example the adoption of an adaptive duty cycling depending on the information coming from the nodes, or emulating also critical errors in the network like nodes that at after a certain time are out of charge and simply disappear from the network. Moreover, the interaction between the harvesting profile and the software behavior can be exploited by the system to adaptively modify the duty cycling period to maximize performance without the risk of discharging too quickly the battery. This concept is very promising and can be an excellent starting point for further developments both on the simulator side,

expanding its functionalities, and on a real implementation.

Keeping under control overheads, delays and latencies in a heavy constrained system is not a trivial task, as well as guaranteeing reliability in connections between nodes. This work demonstrated the feasibility of an energy-autonomous system and will support the development of WSNs based on the STMicroelectronics® devices. Further development is currently directed to an improvement of the low-power property of the radio module by working on the firmware and driver integration and applying a fine-grained duty cycling. Reducing the power absorbed by the radio during the listening phase is key for enabling an always-on network continuously accessible via the IPv6 protocol.

Appendix A

Source code

In the following sections, some source files from the developed firmware are listed. In particular, appendix A.1 lists the file containing the source code of the user process of the client node. In this file, highlighting some functions can be useful. The *receiver()* function, for example, at line 300 is the one that manages the synchronization mechanism. It is called when a sync packet is received, updating the RTC register if the packet sender is the preferred parent for the node and, eventually, forward a new synchronization message to the neighbors. Functions *spirit_lp_enable()* and *spirit_lp_disable()* (lines 392 and 398) are used, respectively, to turn the radio off and on, whereas functions *peripheral_lp_enable()* and *peripheral_lp_disable()* (lines 405 and 504) are used to configure the GPIO peripherals to properly go in a low-power mode and to reactivate them when exiting the low-power state. These functions are called by the *enter_stop_mode()* function that prepare the whole system to go in the low power mode, set the wake up timer and call the *HAL_PWR_EnterSTOPMode()* macro. Appendix A.2 refers to the main file for a generic client node. In addition to the configuration procedure for the node initialization, it is interesting to remark the line 144 where, in case of no process ready to be dispatched, in order to save power the MCU is set to go in sleep mode. Finally, appendix A.3 shows the file containing the source code of the user process of the sink node, known also as border router due to its functionality to provide connectivity with the Internet. The sink does not implement any low-power mode, but, since it acts as the time reference, in the user process it implements the periodic time distribution mechanism.

A.1 client.c

```
1  /**
2   ********************************************************************************
3   * @file     client.c
4   * @author   Central LAB
5   * @version  V1.0.0
6   * @date     20-June-2016
```

```
7   * @brief   lwm2m client
8   ******************************************************************************
9   * @attention
10  *
11  * <h2><center>&copy; COPYRIGHT(c) 2016 STMicroelectronics</center></h2>
12  *
13  * Redistribution and use in source and binary forms, with or without modification,
14  * are permitted provided that the following conditions are met:
15  *   1. Redistributions of source code must retain the above copyright notice,
16  *      this list of conditions and the following disclaimer.
17  *   2. Redistributions in binary form must reproduce the above copyright notice,
18  *      this list of conditions and the following disclaimer in the documentation
19  *      and/or other materials provided with the distribution.
20  *   3. Neither the name of STMicroelectronics nor the names of its contributors
21  *      may be used to endorse or promote products derived from this software
22  *      without specific prior written permission.
23  *
24  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
25  * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
26  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
27  * DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
28  * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
29  * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
30  * SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
31  * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
32  * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
33  * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
34  *
35  ******************************************************************************
36  */
37 #include "ipso-objects.h"
38 #include "lwm2m-engine.h"
39 #include "rest-engine.h"
40 #include "er-coap.h"
41
42 #include "dev/humidity-sensor.h"
43 #include "dev/temperature-sensor.h"
44 #include "dev/sensor-common.h"
45 #include "sensors.h"
46
47 #include "contiki.h"
48 #include "contiki-lib.h"
49 #include "contiki-net.h"
50
51 #include "net/ip/uip.h"
52 #include "net/ip/resolv.h"
53 #include "net/ip/uip-debug.h"
54 #include "net/ip/ip64-addr.h"
55 #include "net/ipv6/uip-ds6.h"
56 #include "net/ipv6/multicast/uip-mcast6.h"
57 #include "net/rpl/rpl.h"
58 #include "net/rpl/rpl-private.h" //dis_output()
59
60 #include "cube_hal.h"
61
62 #include "radio_gpio.h"
63
64 #include "spirit1.h"
65 #include "spirit1-arch.h"
66 #include "stm32l1xx_ll_bus.h"
67 #include "stm32l1xx_ll_exti.h"
68 #include "st-lib.h"
69
70 /** @addtogroup LWM2M_example
71  * @{
72  */
73 #define DEBUG
74 #ifdef DEBUG
75 #define PRINTF(...) printf(__VA_ARGS__)
76 #endif
77
78 #define TEST_MODE 0
79
80 #define MULTICAST_SYNC 1
81 #define UNICAST_SYNC 2
82
83 /* test */
84
85 #define BUSYWAIT_UNTIL(cond, max_time)                                  \
86   do {                                                                  \
87     rtimer_clock_t t0;                                                  \
88     t0 = RTIMER_NOW();                                                  \
89     while(!(cond) && RTIMER_CLOCK_LT(RTIMER_NOW(), t0 + (max_time)));   \
90   } while(0)
91
```

```
92
93  /*Select a sync mechanism*/
94  //#define SYNC_TYPE MULTICAST_SYNC
95  #define SYNC_TYPE UNICAST_SYNC
96
97  /*-------------------------------------------------------------------------------------------*/
98  #ifndef REGISTER_WITH_LWM2M_BOOTSTRAP_SERVER
99  #define REGISTER_WITH_LWM2M_BOOTSTRAP_SERVER 0
100 #endif
101 /*-------------------------------------------------------------------------------------------*/
102 #ifndef REGISTER_WITH_LWM2M_SERVER
103 #define REGISTER_WITH_LWM2M_SERVER 1
104 #endif
105 /*-------------------------------------------------------------------------------------------*/
106 uip_ipaddr_t server_ipaddr;
107 /*-------------------------------------------------------------------------------------------*/
108 /*0 means default port for remote server, otherwise specify a specific port  */
109 #define SERVER_PORT 0
110 /*-------------------------------------------------------------------------------------------*/
111 #undef USE_PUBLIC_LWM2M_SERVER
112 #ifdef USE_PUBLIC_LWM2M_SERVER
113 static uip_ipaddr_t *addrptr;
114 static struct etimer et;
115 static char host[40] = "leshan.eclipse.org";
116 #else
117 /*Define ONE of the following macros*/
118 //#define LWM2M_SERVER_ADDRESS_v4 "192.168.0.11" //for ipv4
119 #define LWM2M_SERVER_ADDRESS_v6 "aaaa::1" //for ipv6
120 #endif /*USE_PUBLIC_LWM2M_SERVER*/
121
122 uip_ip4addr_t ip4addr;
123
124 #define DRIFT_TABLE_SIZE 10
125
126 int calp=511, calm=1;
127
128 /* external variable */
129 extern RTC_HandleTypeDef RtcHandle;
130 extern uint8_t software_reset;
131 extern const struct ipso_objects_sensor IPSO_TEMPERATURE;
132 extern const struct ipso_objects_sensor IPSO_HUMIDITY;
133 /* broadcast synch management */
134 #define PERIOD 1200
135 #define RADIO_RESET PERIOD*2*1000
136 #define ON_TIME 4
137 #define SYNC_UDP_PORT 3001
138 #define TEST_UDP_PORT 3003
139 #define SYNCHRO 2
140 #define LP_MODE_ON 1
141 #define LP_MODE_OFF 0
142
143 uint32_t t_old=0xFFFFFFFF;
144 int drift_table[DRIFT_TABLE_SIZE];
145 int k;
146
147 /* message typedef */
148 typedef struct
149 {
150  uint8_t  MessageType;
151  uint8_t  TimeFormat;
152  uint8_t  Hours;
153  uint8_t  Minutes;
154  uint8_t     Seconds;
155  uint32_t SubSeconds;
156  uint32_t SecondFraction;
157  uint8_t  RunMode;
158 }Sync_MessageTypeDef;
159
160
161 #if SYNC_TYPE == MULTICAST_SYNC
162
163 static struct uip_udp_conn *sink_conn;
164 static void tcpip_handler(void);
165 static uip_ds6_maddr_t * join_mcast_group(void);
166
167 static Sync_MessageTypeDef *mcast_message;
168
169 #elif SYNC_TYPE == UNICAST_SYNC
170
171 static uip_ipaddr_t mc_addr;
172 static Sync_MessageTypeDef mcast_message;
173 static struct simple_udp_connection unicast_connection;
174 static void
175 receiver(struct simple_udp_connection *c,
176   const uip_ipaddr_t *sender_addr,
```

```
177     uint16_t sender_port,
178     const uip_ipaddr_t *receiver_addr,
179     uint16_t receiver_port,
180     const Sync_MessageTypeDef *data,
181     uint16_t datalen);
182
183   #endif
184
185   static struct simple_udp_connection rpl_connection_test;
186   static void
187   receiver2(struct simple_udp_connection *c,
188     const uip_ipaddr_t *sender_addr,
189     uint16_t sender_port,
190     const uip_ipaddr_t *receiver_addr,
191     uint16_t receiver_port,
192     const uint8_t *data,
193     uint16_t datalen);
194
195   static uint8_t synch=0;
196
197   /* low power management */
198   static void SYSCLKConfig_STOP(void);
199   static void peripheral_lp_enable(void);
200   static void peripheral_lp_disable(void);
201   static void enter_stop_mode(int seconds);
202   static void spirit_lp_enable(void);
203   static void spirit_lp_disable(void);
204   static void peripheral_init(void);
205
206   static uint32_t gpioa_moder, gpiob_moder, gpioc_moder;
207   static uint32_t gpioa_otyper, gpiob_otyper, gpioc_otyper;
208   static uint32_t gpioa_ospeedr, gpiob_ospeedr, gpioc_ospeedr;
209   static uint32_t gpioa_pupdr, gpiob_pupdr, gpioc_pupdr;
210   static uint32_t exti_imr_backup;
211   static uint8_t wakeup=0;
212
213   /* sync management */
214   static void increase_ppm(void);
215   static void decrease_ppm(void);
216   rpl_parent_t *p;
217   rpl_dag_t *dag;
218
219   static uint8_t enable_stop = 0;
220
221   static struct etimer periodic_timer;
222   static struct etimer sync_timer;
223   static struct etimer periodic_timer_test;
224   static struct etimer process_pause;
225
226   PROCESS(rd_client, "OMA LWM2M /rd Client");
227   AUTOSTART_PROCESSES(&rd_client);
228
229
230   /*---------------------------------------------------------------------------*/
231   #if SYNC_TYPE == MULTICAST_SYNC
232
233   static void
234   tcpip_handler(void)
235   {
236    RTC_TimeTypeDef RTC_TimeStructure={0}, RTC_TimeStructure2={0};
237    RTC_DateTypeDef RTC_DateStructure={0};
238
239    //take the actual timestamp and compare with the received once
240    //to get the time drift
241    uint32_t t1, t2;
242    HAL_RTC_GetTime(&RtcHandle, &RTC_TimeStructure2, RTC_FORMAT_BIN);
243    HAL_RTC_GetDate(&RtcHandle, &RTC_DateStructure, RTC_FORMAT_BIN);
244
245    if(uip_newdata()) {
246     mcast_message=(Sync_MessageTypeDef *)uip_appdata;
247     if (mcast_message->MessageType==SYNCHRO){
248      RTC_TimeStructure.TimeFormat = mcast_message->TimeFormat;
249      RTC_TimeStructure.Hours = mcast_message->Hours;
250      RTC_TimeStructure.Minutes = mcast_message->Minutes;
251      RTC_TimeStructure.Seconds = mcast_message->Seconds + 1;
252      HAL_Delay((uint32_t)(1000-1000*(mcast_message->SecondFraction - mcast_message->SubSeconds)/(
           mcast_message->SecondFraction +1)));
253      HAL_RTC_SetTime(&RtcHandle, &RTC_TimeStructure, FORMAT_BIN);
254
255      enable_stop=mcast_message->RunMode;
256
257      HAL_RTC_GetTime(&RtcHandle, &RTC_TimeStructure, RTC_FORMAT_BIN);
258      HAL_RTC_GetDate(&RtcHandle, &RTC_DateStructure, RTC_FORMAT_BIN);
```

```
259       PRINTF("SYNCH! Current time: %02d:%02d:%02d.%03d\n", RTC_TimeStructure.Hours,
              RTC_TimeStructure.Minutes,RTC_TimeStructure.Seconds, (1000*(RTC_TimeStructure.
              SecondFraction - RTC_TimeStructure.SubSeconds))/(RTC_TimeStructure.SecondFraction + 1));
260       t1=1000*(3600*mcast_message->Hours + 60*mcast_message->Minutes + mcast_message->Seconds) +
              (1000*(mcast_message->SecondFraction - mcast_message->SubSeconds))/(mcast_message->
              SecondFraction + 1);
261       t2=1000*(3600*RTC_TimeStructure2.Hours + 60*RTC_TimeStructure2.Minutes + RTC_TimeStructure2.
              Seconds) + (1000*(RTC_TimeStructure2.SecondFraction - RTC_TimeStructure2.SubSeconds))/(
              RTC_TimeStructure2.SecondFraction + 1);
262       PRINTF("Time drift since last synch was of t-t_sync = %d ms\n", t2-t1);
263
264       synch=1;
265     }
266   }
267
268   return;
269 }
270
271 /*---------------------------------------------------------------------------*/
272 static uip_ds6_maddr_t *
273 join_mcast_group(void)
274 {
275   uip_ipaddr_t addr;
276   uip_ds6_maddr_t *rv;
277
278   // /* First, set our v6 global */
279   // uip_ip6addr(&addr, UIP_DS6_DEFAULT_PREFIX, 0, 0, 0, 0, 0, 0, 0);
280   // uip_ds6_set_addr_iid(&addr, &uip_lladdr);
281   // uip_ds6_addr_add(&addr, 0, ADDR_AUTOCONF);
282
283   /*
284    * IPHC will use stateless multicast compression for this destination
285    * (M=1, DAC=0), with 32 inline bits (1E 89 AB CD)
286    */
287   uip_ip6addr(&addr, 0xFF1E,0,0,0,0,0x89,0xABCD);
288   rv = uip_ds6_maddr_add(&addr);
289
290   if(rv) {
291     PRINTF("Joined multicast group ");
292     PRINT6ADDR(&uip_ds6_maddr_lookup(&addr)->ipaddr);
293     PRINTF("\n");
294   }
295   return rv;
296 }
297
298 #elif SYNC_TYPE == UNICAST_SYNC
299
300 static void
301 receiver(struct simple_udp_connection *c,
302     const uip_ipaddr_t *sender_addr,
303     uint16_t sender_port,
304     const uip_ipaddr_t *receiver_addr,
305     uint16_t receiver_port,
306     const Sync_MessageTypeDef *data,
307     uint16_t datalen)
308 {
309   RTC_TimeTypeDef RTC_TimeStructure={0}, RTC_TimeStructure2={0}, RTC_TimeStructure3={0};
310   RTC_DateTypeDef RTC_DateStructure={0};
311
312   static uip_ipaddr_t *pp_address, sa_address;
313   int drift_table[10];
314   int i;
315   //take the actual timestamp and compare with the received once to get the time drift
316   int t1, t2, t3, drift;
317   HAL_RTC_GetTime(&RtcHandle, &RTC_TimeStructure2, RTC_FORMAT_BIN);
318   HAL_RTC_GetDate(&RtcHandle, &RTC_DateStructure, RTC_FORMAT_BIN);
319
320   printf("Sync received from ");
321   uip_debug_ipaddr_print(sender_addr);
322   printf("\n");
323   sa_address=*sender_addr;
324
325   pp_address = rpl_get_parent_ipaddr(dag->preferred_parent);
326
327   for (i=0; i<8 && pp_address->u16[i] == sa_address.u16[i]; i++);
328
329   if(i==8) {
330     if (data->MessageType==SYNCHRO){
331
332       HAL_RTC_GetTime(&RtcHandle, &RTC_TimeStructure3, RTC_FORMAT_BIN);
333       HAL_RTC_GetDate(&RtcHandle, &RTC_DateStructure, RTC_FORMAT_BIN);
334       t_old=1000*(3600*RTC_TimeStructure3.Hours + 60*RTC_TimeStructure3.Minutes +
              RTC_TimeStructure3.Seconds) + (1000*(RTC_TimeStructure3.SecondFraction -
              RTC_TimeStructure3.SubSeconds))/(RTC_TimeStructure3.SecondFraction + 1);
335
```

```
336     RTC_TimeStructure.TimeFormat = data->TimeFormat;
337     RTC_TimeStructure.Hours = data->Hours;
338     RTC_TimeStructure.Minutes = data->Minutes;
339     RTC_TimeStructure.Seconds = data->Seconds + 1;
340     HAL_Delay((uint32_t)(1000-1000*(data->SecondFraction-data->SubSeconds)/(data->SecondFraction
            +1)));
341     HAL_RTC_SetTime(&RtcHandle, &RTC_TimeStructure, FORMAT_BIN);
342
343     enable_stop=data->RunMode;
344
345     HAL_RTC_GetTime(&RtcHandle, &RTC_TimeStructure, RTC_FORMAT_BIN);
346     HAL_RTC_GetDate(&RtcHandle, &RTC_DateStructure, RTC_FORMAT_BIN);
347     PRINTF("SYNCH! Current time: %02d:%02d:%02d.%03d\n", RTC_TimeStructure.Hours,
            RTC_TimeStructure.Minutes,RTC_TimeStructure.Seconds, (1000*(RTC_TimeStructure.
            SecondFraction - RTC_TimeStructure.SubSeconds))/(RTC_TimeStructure.SecondFraction + 1));
348     t1=1000*(3600*data->Hours + 60*data->Minutes + data->Seconds) + (1000*(data->SecondFraction -
              data->SubSeconds))/(data->SecondFraction + 1);
349     t2=1000*(3600*RTC_TimeStructure2.Hours + 60*RTC_TimeStructure2.Minutes + RTC_TimeStructure2.
            Seconds) + (1000*(RTC_TimeStructure2.SecondFraction - RTC_TimeStructure2.SubSeconds))/(
            RTC_TimeStructure2.SecondFraction + 1);
350     PRINTF("Time drift since last synch was of t-t_sync = %d ms\n", t2-t1);
351
352     synch=1;
353
354     HAL_RTC_GetTime(&RtcHandle, &RTC_TimeStructure, RTC_FORMAT_BIN);
355     HAL_RTC_GetDate(&RtcHandle, &RTC_DateStructure, RTC_FORMAT_BIN);
356
357     mcast_message.MessageType = SYNCHRO;
358     mcast_message.TimeFormat = RTC_TimeStructure.TimeFormat;
359     mcast_message.Hours = RTC_TimeStructure.Hours;
360     mcast_message.Minutes = RTC_TimeStructure.Minutes;
361     mcast_message.Seconds = RTC_TimeStructure.Seconds;
362     mcast_message.SubSeconds = RTC_TimeStructure.SubSeconds;
363     mcast_message.SecondFraction = RTC_TimeStructure.SecondFraction;
364     mcast_message.RunMode = LP_MODE_ON;
365
366     simple_udp_sendto(&unicast_connection, &mcast_message, sizeof(mcast_message), &mc_addr);
367     }
368   }
369 }
370 #endif
371
372 static void
373 receiver2(struct simple_udp_connection *c,
374   const uip_ipaddr_t *sender_addr,
375   uint16_t sender_port,
376   const uip_ipaddr_t *receiver_addr,
377   uint16_t receiver_port,
378   const uint8_t *data,
379   uint16_t datalen)
380 {
381
382 }
383
384 /*---------------------------------------------------------------------------*/
385 static void spirit_reset(){
386  spirit_radio_driver.off();
387  spirit_radio_driver.init();
388  spirit_radio_driver.on();
389 }
390
391 /*---------------------------------------------------------------------------*/
392 static void spirit_lp_enable(){
393  uint16_t rssi;
394  spirit_radio_driver.off();
395 }
396
397 /*---------------------------------------------------------------------------*/
398 static void spirit_lp_disable(){
399
400  spirit_radio_driver.init();
401  spirit_radio_driver.on();
402 }
403
404 /*---------------------------------------------------------------------------*/
405 static void peripheral_lp_enable()
406 {
407  /* Gpio configuration:
408   *  GPIO: D E F G H are set in the peripheral_init() function as analog-in nopull with clock
          disabled since
409   *  they are not used by the application. (NUCLEO + IDS01A4)
410   *
411   *  IDS01A4 expansion board uses the following pin connected to the nucleo:
412   *  SDN:   PA10
413   *  CSN:   PB6
```

```
414     *  MISO:  PA6
415     *  MOSI:  PA7
416     *  SCLK  PB3
417     *  SPIGPIO3: PC7
418     *  LED:  PB4
419     *
420     *  they are configured by the radio driver but during stop mode they need to be configured in
          the correct state
421     *  in order to be sure that the MCU goes in stop state, the radio in standby and the EEPROM
          keeps a stable state:
422     *  PA10(SDN):  output + nopull + storing a '0'
423     *  PB6(CSN):  output + nopull + storing a '1' (this is the default value)
424     *  PA6(MISO):  analog + nopull
425     *  PA7(MOSI):  output + nopull + storing a '0'
426     *  PB3(CLK):  output + nopull + storing a '0'
427     *
428     *  SPIGPIO3 and LED are set to analog nopull since they are not critical
429     *
430     * Other GPIO used are:
431     * PA5:  Nucleo Green Led
432     * PA0:  TIM2 (used for contiki timers
433     * PC13:  Nucleo Push Button (used by ipso application)
434     * PA2:  TX for USART2
435     * PA3:  RX for USART2
436     *
437     * */
438
439   GPIO_InitTypeDef GPIO_InitStructure= {0};
440
441   /*save the previous configuration*/
442   gpioa_moder = GPIOA->MODER;
443   gpiob_moder = GPIOB->MODER;
444   gpioc_moder = GPIOC->MODER;
445   gpioa_otyper = GPIOA->OTYPER;
446   gpiob_otyper = GPIOB->OTYPER;
447   gpioc_otyper = GPIOC->OTYPER;
448   gpioa_ospeedr = GPIOA->OSPEEDR;
449   gpiob_ospeedr = GPIOB->OSPEEDR;
450   gpioc_ospeedr = GPIOC->OSPEEDR;
451   gpioa_pupdr = GPIOA->PUPDR;
452   gpiob_pupdr = GPIOB->PUPDR;
453   gpioc_pupdr = GPIOC->PUPDR;
454
455   /*GPIOC configuration (not critical for low power mode->all pin in analog mode)*/
456   __HAL_RCC_GPIOC_CLK_ENABLE();
457   GPIO_InitStructure.Mode=GPIO_MODE_ANALOG;
458   GPIO_InitStructure.Pull=GPIO_NOPULL;
459   GPIO_InitStructure.Pin=GPIO_PIN_All;
460   HAL_GPIO_Init(GPIOC, &GPIO_InitStructure);
461   __HAL_RCC_GPIOC_CLK_DISABLE();
462
463   /*GPIOA conf */
464   __HAL_RCC_GPIOA_CLK_ENABLE();
465   GPIO_InitStructure.Mode=GPIO_MODE_ANALOG;
466   GPIO_InitStructure.Pull=GPIO_NOPULL;
467   GPIO_InitStructure.Speed=GPIO_SPEED_HIGH;
468   GPIO_InitStructure.Pin=GPIO_PIN_All & ~GPIO_PIN_7 & ~GPIO_PIN_10;
469   HAL_GPIO_Init(GPIOA, &GPIO_InitStructure);
470
471   /*critical pin: PA10 PA7 */
472   GPIO_InitStructure.Mode=GPIO_MODE_OUTPUT_PP;
473   GPIO_InitStructure.Pull=GPIO_NOPULL;
474   GPIO_InitStructure.Speed=GPIO_SPEED_HIGH;
475   GPIO_InitStructure.Pin=GPIO_PIN_10 | GPIO_PIN_7;
476   HAL_GPIO_Init(GPIOA, &GPIO_InitStructure);
477
478   HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_RESET);
479   HAL_GPIO_WritePin(GPIOA, GPIO_PIN_7, GPIO_PIN_RESET);
480   __HAL_RCC_GPIOA_CLK_DISABLE();
481
482   /*GPIOB conf */
483   __HAL_RCC_GPIOB_CLK_ENABLE();
484   GPIO_InitStructure.Mode=GPIO_MODE_ANALOG;
485   GPIO_InitStructure.Pull=GPIO_NOPULL;
486   GPIO_InitStructure.Speed=GPIO_SPEED_HIGH;
487   GPIO_InitStructure.Alternate=0x00;
488   GPIO_InitStructure.Pin=GPIO_PIN_All & ~GPIO_PIN_3 & ~GPIO_PIN_6 ;
489   HAL_GPIO_Init(GPIOB, &GPIO_InitStructure);
490
491   /*critical pin: PB3 PB6 */
492   GPIO_InitStructure.Mode=GPIO_MODE_OUTPUT_PP;
493   GPIO_InitStructure.Pull=GPIO_NOPULL;
494   GPIO_InitStructure.Speed=GPIO_SPEED_HIGH;
495   GPIO_InitStructure.Pin=GPIO_PIN_3 | GPIO_PIN_6;
496   HAL_GPIO_Init(GPIOB, &GPIO_InitStructure);
```

```
497
498   HAL_GPIO_WritePin(GPIOB, GPIO_PIN_3, GPIO_PIN_RESET);
499   HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);
500   __HAL_RCC_GPIOB_CLK_DISABLE();
501  }
502
503  /*--------------------------------------------------------------------------------------*/
504  static void peripheral_lp_disable()
505  {
506   __HAL_RCC_GPIOA_CLK_ENABLE();
507   __HAL_RCC_GPIOB_CLK_ENABLE();
508   __HAL_RCC_GPIOC_CLK_ENABLE();
509
510   GPIOA->MODER = gpioa_moder;
511   GPIOB->MODER = gpiob_moder;
512   GPIOC->MODER = gpioc_moder;
513   GPIOA->OTYPER = gpioa_otyper;
514   GPIOB->OTYPER = gpiob_otyper;
515   GPIOC->OTYPER = gpioc_otyper;
516   GPIOA->OSPEEDR = gpioa_ospeedr;
517   GPIOB->OSPEEDR = gpiob_ospeedr;
518   GPIOC->OSPEEDR = gpioc_ospeedr;
519   GPIOA->PUPDR = gpioa_pupdr;
520   GPIOB->PUPDR = gpiob_pupdr;
521   GPIOC->PUPDR = gpioc_pupdr;
522
523   HAL_GPIO_WritePin(GPIOA, GPIO_PIN_0, GPIO_PIN_SET);
524   HAL_Delay(10);
525   IPSO_TEMPERATURE.init();
526   IPSO_HUMIDITY.init();
527  }
528
529  /*--------------------------------------------------------------------------------------*/
530  static void peripheral_init(void){
531   /* Deactivate unused GPIO pin */
532   GPIO_InitTypeDef GPIO_InitStructure= {0};
533
534   __HAL_RCC_GPIOD_CLK_ENABLE();
535   __HAL_RCC_GPIOE_CLK_ENABLE();
536   __HAL_RCC_GPIOF_CLK_ENABLE();
537   __HAL_RCC_GPIOG_CLK_ENABLE();
538   __HAL_RCC_GPIOH_CLK_ENABLE();
539
540   GPIO_InitStructure.Mode=GPIO_MODE_ANALOG;
541   GPIO_InitStructure.Pull=GPIO_NOPULL;
542   GPIO_InitStructure.Pin=GPIO_PIN_All;
543
544   HAL_GPIO_Init(GPIOD, &GPIO_InitStructure);
545   HAL_GPIO_Init(GPIOE, &GPIO_InitStructure);
546   HAL_GPIO_Init(GPIOF, &GPIO_InitStructure);
547   HAL_GPIO_Init(GPIOG, &GPIO_InitStructure);
548   HAL_GPIO_Init(GPIOH, &GPIO_InitStructure);
549
550   __HAL_RCC_GPIOD_CLK_DISABLE();
551   __HAL_RCC_GPIOE_CLK_DISABLE();
552   __HAL_RCC_GPIOF_CLK_DISABLE();
553   __HAL_RCC_GPIOG_CLK_DISABLE();
554   __HAL_RCC_GPIOH_CLK_DISABLE();
555
556
557   /* Disable unused peripherals */
558   __HAL_RCC_CRC_CLK_DISABLE();
559   __HAL_RCC_DMA2_CLK_DISABLE();
560   __HAL_RCC_TIM3_CLK_DISABLE();
561   __HAL_RCC_TIM4_CLK_DISABLE();
562   __HAL_RCC_TIM5_CLK_DISABLE();
563   __HAL_RCC_TIM6_CLK_DISABLE();
564   __HAL_RCC_TIM7_CLK_DISABLE();
565   __HAL_RCC_TIM9_CLK_DISABLE();
566   __HAL_RCC_TIM10_CLK_DISABLE();
567   __HAL_RCC_TIM11_CLK_DISABLE();
568   __HAL_RCC_LCD_CLK_DISABLE();
569   __HAL_RCC_WWDG_CLK_DISABLE();
570   __HAL_RCC_SPI2_CLK_DISABLE();
571   __HAL_RCC_SPI3_CLK_DISABLE();
572   __HAL_RCC_I2C1_CLK_DISABLE();
573   __HAL_RCC_I2C2_CLK_DISABLE();
574   __HAL_RCC_COMP_CLK_DISABLE();
575
576   __HAL_RCC_GPIOA_CLK_ENABLE();
577   GPIO_InitStructure.Mode=GPIO_MODE_ANALOG;
578   GPIO_InitStructure.Pull=GPIO_NOPULL;
579   GPIO_InitStructure.Pin=GPIO_PIN_13 | GPIO_PIN_14 | GPIO_PIN_15;//| GPIO_PIN_2| GPIO_PIN_3;
580   HAL_GPIO_Init(GPIOA, &GPIO_InitStructure);
581
```

```
582 }
583
584 /*-------------------------------------------------------------------------------*/
585 static void enter_stop_mode(int sleep_seconds)
586 {
587  int stop_time;
588
589  PRINTF("Stop mode: enter...\n");
590
591  spirit_lp_enable();
592  peripheral_lp_enable();
593
594  HAL_PWREx_EnableUltraLowPower();
595  HAL_PWREx_EnableFastWakeUp();
596
597  exti_imr_backup = EXTI->IMR;
598  EXTI->IMR=0x00000000;
599
600  HAL_SuspendTick();
601
602  stop_time=sleep_seconds*2048;
603
604  while(stop_time>0){
605   /*## Configure the Wake up timer ###############################################*/
606   /*  RTC Wakeup Interrupt Generation:
607         Wakeup Time Base = (RTC_WAKEUPCLOCK_RTCCLK_DIV /(LSI))
608         Wakeup Time = Wakeup Time Base * WakeUpCounter
609                     = (RTC_WAKEUPCLOCK_RTCCLK_DIV /(LSI)) * WakeUpCounter
610         ==> WakeUpCounter = Wakeup Time / Wakeup Time Base
611         To configure the wake up timer to 20s the WakeUpCounter is set to 0xA017:
612           RTC_WAKEUPCLOCK_RTCCLK_DIV = RTCCLK_Div16 = 16
613           Wakeup Time Base = 16 /(~32.768KHz) = ~0,488 ms
614           Wakeup Time = ~20s = 0,488ms  * WakeUpCounter
615           ==> WakeUpCounter = ~20s/0,488ms = 40983 = 0xA017 */
616
617   /* Disable Wake-up timer */
618   HAL_RTCEx_DeactivateWakeUpTimer(&RtcHandle);
619
620   /* Clear PWR wake up Flag */
621   __HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU);
622
623   /* Clear RTC Wake Up timer Flag */
624   __HAL_RTC_WAKEUPTIMER_CLEAR_FLAG(&RtcHandle, RTC_FLAG_WUTF);
625
626   /* Enable Wake-up timer */
627   if(stop_time<=65535){
628    HAL_RTCEx_SetWakeUpTimer_IT(&RtcHandle, stop_time, RTC_WAKEUPCLOCK_RTCCLK_DIV16);
629   }
630   else{
631    HAL_RTCEx_SetWakeUpTimer_IT(&RtcHandle, 0xffff , RTC_WAKEUPCLOCK_RTCCLK_DIV16);
632   }
633   /*## Enter Stop Mode ###########################################################*/
634   HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON, PWR_STOPENTRY_WFI);
635   stop_time-=65535;
636  }
637
638
639  /* Configures system clock after wake-up from STOP: enable HSI, PLL and select
640    PLL as system clock source (HSI and PLL are disabled in STOP mode) */
641  SYSCLKConfig_STOP();
642
643  /* Disable Wake-up timer */
644  if(HAL_RTCEx_DeactivateWakeUpTimer(&RtcHandle) != HAL_OK)
645  {
646   /* Initialization Error */
647   Error_Handler();
648  }
649
650  HAL_ResumeTick();
651  EXTI->IMR = exti_imr_backup;
652  peripheral_lp_disable();
653  spirit_lp_disable();
654
655  HAL_PWREx_DisableUltraLowPower();
656  PRINTF("Stop mode: exit!\n");
657 }
658
659 /*-------------------------------------------------------------------------------*/
660 static void SYSCLKConfig_STOP(void)
661 {
662  RCC_ClkInitTypeDef RCC_ClkInitStruct;
663  RCC_OscInitTypeDef RCC_OscInitStruct;
664  uint32_t pFLatency = 0;
665
666  /* Get the Oscillators configuration according to the internal RCC registers */
```

```
667   HAL_RCC_GetOscConfig(&RCC_OscInitStruct);
668
669   /* After wake-up from STOP reconfigure the system clock: Enable HSI and PLL */
670   RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
671   RCC_OscInitStruct.HSIState = RCC_HSI_ON;
672   RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
673   RCC_OscInitStruct.HSICalibrationValue = 0x10;
674   if(HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
675   {
676    Error_Handler();
677   }
678
679   /* Get the Clocks configuration according to the internal RCC registers */
680   HAL_RCC_GetClockConfig(&RCC_ClkInitStruct, &pFLatency);
681
682   /* Select PLL as system clock source and configure the HCLK, PCLK1 and PCLK2
683       clocks dividers */
684   RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_SYSCLK;
685   RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
686   if(HAL_RCC_ClockConfig(&RCC_ClkInitStruct, pFLatency) != HAL_OK)
687   {
688    Error_Handler();
689   }
690  }
691
692  PROCESS_THREAD(rd_client, ev, data)
693  {
694   PROCESS_BEGIN();
695   static char server_found = 0;
696
697   static uint16_t t;
698   static uint32_t message[10];
699   static int t_now;
700
701   static RTC_TimeTypeDef RTC_TimeStructure;
702   static RTC_DateTypeDef RTC_DateStructure;
703
704   int32_t hum=0, temp=0;
705
706   printf("RD␣Client␣process␣started.\n\n");
707
708  #ifdef USE_PUBLIC_LWM2M_SERVER
709   static int ret;
710   uip_ip6addr_t ip6addr;
711
712   printf("Looking␣for␣LWM2M␣server:␣'%s'\n", host);
713   uip_ipaddr(&ip4addr, 8,8,8,8);
714   ip64_addr_4to6(&ip4addr, &ip6addr);
715   uip_nameserver_update(&ip6addr, UIP_NAMESERVER_INFINITE_LIFETIME);
716
717   /*DNS request for server address*/
718   etimer_set(&et, 2 * CLOCK_SECOND);
719   resolv_query(host);
720
721   HAL_Delay(3000);
722
723   while((ret = resolv_lookup(host, &addrptr)) != RESOLV_STATUS_CACHED) {
724    if (ret != RESOLV_STATUS_RESOLVING){
725     resolv_query(host);
726    }
727    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
728    etimer_reset(&et);
729   }
730
731   server_found = 1;
732   server_ipaddr = *addrptr;
733
734  #else
735  #ifdef LWM2M_SERVER_ADDRESS_v4
736   printf("Looking␣for␣LWM2M␣server:␣'%s'\n", LWM2M_SERVER_ADDRESS_v4);
737
738   // uip_ipaddr(&ip4addr, 192, 168, 0, 1); //Old code
739   if(uiplib_ip4addrconv(LWM2M_SERVER_ADDRESS_v4, &ip4addr)) {
740    ip64_addr_4to6(&ip4addr, &server_ipaddr);
741    server_found = 1;
742   }
743  #endif /* LWM2M_SERVER_ADDRESS_v4 */
744
745  #ifdef LWM2M_SERVER_ADDRESS_v6
746   printf("Looking␣for␣LWM2M␣server:␣'%s'\n", LWM2M_SERVER_ADDRESS_v6);
747
748   if(uiplib_ip6addrconv(LWM2M_SERVER_ADDRESS_v6, &server_ipaddr)) {
749    server_found = 1;
750   }
751  #endif /* LWM2M_SERVER_ADDRESS_v6 */
```

```
752 #endif /*USE_PUBLIC_LWM2M_SERVER*/
753
754   if (!server_found) {
755     printf("ERROR␣with␣the␣Server␣IP␣Address,␣please␣check␣the␣provided␣configuration.\n");
756   } else {
757
758     printf("LWM2M␣Server␣Address:\n");
759     uip_debug_ipaddr_print(&server_ipaddr);
760     printf("\n");
761     dis_output(NULL);
762     /*Notify the address to lwm2m for registration*/
763     //lwm2m_engine_register_with_bootstrap_server(&server_ipaddr, 0);
764     lwm2m_engine_register_with_server(&server_ipaddr, SERVER_PORT);
765
766     //lwm2m_engine_use_bootstrap_server(REGISTER_WITH_LWM2M_BOOTSTRAP_SERVER);
767     lwm2m_engine_use_registration_server(REGISTER_WITH_LWM2M_SERVER);
768
769     /*Initialize objects and start lwm2m engine*/
770     lwm2m_engine_init();
771     lwm2m_engine_register_default_objects();
772
773     peripheral_init();
774     ipso_objects_init();
775   }
776
777   /* custom application for sync*/
778
779   HAL_RTC_GetTime(&RtcHandle, &RTC_TimeStructure, RTC_FORMAT_BIN);
780   HAL_RTC_GetDate(&RtcHandle, &RTC_DateStructure, RTC_FORMAT_BIN);
781   PRINTF("Client␣started␣at:␣%02d:%02d:%02d\n", RTC_TimeStructure.Hours,RTC_TimeStructure.Minutes
        ,RTC_TimeStructure.Seconds);
782
783
784
785   do {
786     dag = rpl_get_any_dag();
787
788     if (dag != NULL)
789     {
790       PRINTF("GATEWAY␣IP:␣");
791       for(uint8_t i=0;i<8;i++)
792       {
793         PRINTF("%.4x␣",dag->dag_id.u16[i]);
794       }
795       PRINTF("\n");
796     }
797     else
798     {
799       etimer_set(&periodic_timer, CLOCK_SECOND/5);
800       PROCESS_WAIT_UNTIL(etimer_expired(&periodic_timer));
801     }
802   } while (dag == NULL);
803
804 #if SYNC_TYPE == MULTICAST_SYNC
805   if(join_mcast_group() == NULL) {
806     PRINTF("Failed␣to␣join␣multicast␣group\n");
807     PROCESS_EXIT();
808   }
809
810   sink_conn = udp_new(NULL, UIP_HTONS(0), NULL);
811   udp_bind(sink_conn, UIP_HTONS(SYNC_UDP_PORT));
812 #elif SYNC_TYPE == UNICAST_SYNC
813   simple_udp_register(&unicast_connection, SYNC_UDP_PORT,
814     NULL, SYNC_UDP_PORT, receiver);
815   uip_ip6addr(&mc_addr, 0xfc00, 0, 0, 0, 0, 0, 0x1337, 0x0002);
816   uip_create_linklocal_allnodes_mcast(&mc_addr);
817 #endif
818
819   simple_udp_register(&rpl_connection_test, TEST_UDP_PORT,
820     NULL, TEST_UDP_PORT, receiver2);
821
822   etimer_set(&periodic_timer, CLOCK_SECOND/5);
823   etimer_set(&periodic_timer_test, 10*CLOCK_SECOND);
824 #if TEST_MODE == 0
825   /* Before starting sensors operation + lp mode process waits for synchronization message
826    * jump if a software reset occurred */
827   if (software_reset==0){
828     PRINTF("Waiting␣for␣sync\n");
829 #if SYNC_TYPE == MULTICAST_SYNC
830     while(etimer_expired(&periodic_timer_test)==0 || !synch){
831       PROCESS_YIELD();
832       if(ev == tcpip_event) {
833         tcpip_handler();//performs the sync
834       }
835     }
```

```
836 #elif SYNC_TYPE == UNICAST_SYNC
837   PROCESS_YIELD_UNTIL(etimer_expired(&periodic_timer_test) && synch);
838 #endif
839   PRINTF("Synched\n");
840  }
841 #endif /* test_mode */
842
843  HAL_RTC_GetTime(&RtcHandle, &RTC_TimeStructure, RTC_FORMAT_BIN);
844  HAL_RTC_GetDate(&RtcHandle, &RTC_DateStructure, RTC_FORMAT_BIN);
845  PRINTF("Sync debug: %02d:%02d:%02d.%03d\n", RTC_TimeStructure.Hours,RTC_TimeStructure.Minutes,
       RTC_TimeStructure.Seconds, (1000*(RTC_TimeStructure.SecondFraction - RTC_TimeStructure.
       SubSeconds))/(RTC_TimeStructure.SecondFraction + 1));
846  etimer_reset(&periodic_timer);
847  message[0]=0;
848
849     PRINTF("Sending message %d to: ", message[0]);
850        IPSO_TEMPERATURE.read_value(&temp);
851        IPSO_HUMIDITY.read_value(&hum);
852        message[1]=hum;
853        message[2]=temp;
854        uip_debug_ipaddr_print(&dag->dag_id);
855        PRINTF("\n");
856        BSP_LED_Toggle(LED2);
857        simple_udp_sendto(&rpl_connection_test, &message, sizeof(message), &dag->dag_id);
858        BSP_LED_Toggle(LED2);
859        message[0]++;
860
861  while(1){
862  RTC_TimeStructure.Hours,RTC_TimeStructure.Minutes,RTC_TimeStructure.Seconds, (1000*(
       RTC_TimeStructure.SubSeconds - RTC_TimeStructure.SecondFraction))/(RTC_TimeStructure.
       SecondFraction + 1));
863
864   PROCESS_YIELD();
865 #if SYNC_TYPE == MULTICAST_SYNC
866   if(ev == tcpip_event) {
867    tcpip_handler();//performs the sync
868   }
869 #endif
870 #if TEST_MODE == 0
871   HAL_RTC_GetTime(&RtcHandle, &RTC_TimeStructure, RTC_FORMAT_BIN);
872   HAL_RTC_GetDate(&RtcHandle, &RTC_DateStructure, RTC_FORMAT_BIN);
873
874   t = (RTC_TimeStructure.Minutes*60 + RTC_TimeStructure.Seconds)%PERIOD;
875
876   if(t>=ON_TIME){
877    HAL_RTC_GetTime(&RtcHandle, &RTC_TimeStructure, RTC_FORMAT_BIN);
878    HAL_RTC_GetDate(&RtcHandle, &RTC_DateStructure, RTC_FORMAT_BIN);
879    if(enable_stop == LP_MODE_ON){
880     PRINTF("time to enter stop mode for %d seconds: %02d:%02d:%02d.%03d\n", PERIOD-t,
          RTC_TimeStructure.Hours,RTC_TimeStructure.Minutes,RTC_TimeStructure.Seconds, (1000*(
          RTC_TimeStructure.SecondFraction - RTC_TimeStructure.SubSeconds))/(RTC_TimeStructure.
          SecondFraction + 1));
881     enter_stop_mode(PERIOD-t);
882    }else{
883     PRINTF("time to be quiet for %d seconds: %02d:%02d:%02d.%03d\n", PERIOD-t, RTC_TimeStructure
          .Hours,RTC_TimeStructure.Minutes,RTC_TimeStructure.Seconds, (1000*(RTC_TimeStructure.
          SecondFraction - RTC_TimeStructure.SubSeconds))/(RTC_TimeStructure.SecondFraction + 1));
884     etimer_set(&process_pause, (PERIOD-t)*CLOCK_SECOND);
885     PROCESS_YIELD_UNTIL(etimer_expired(&process_pause));
886    }
887    wakeup=1;
888   }
889   else if (wakeup==1){
890    if(software_reset==1){
891     dis_output(NULL);
892     software_reset=0;
893    }
894
895    etimer_set(&process_pause, 1000 + random_rand()%2000);//random_rand()%3000);
896    PROCESS_WAIT_UNTIL(etimer_expired(&process_pause));
897
898    PRINTF("Sending message %d to: ", message[0]);
899    IPSO_TEMPERATURE.read_value(&temp);
900    IPSO_HUMIDITY.read_value(&hum);
901    message[1]=hum;
902    message[2]=temp;
903    uip_debug_ipaddr_print(&dag->dag_id);
904    PRINTF("\n");
905    BSP_LED_Toggle(LED2);
906    simple_udp_sendto(&rpl_connection_test, &message, sizeof(message), &dag->dag_id);
907    BSP_LED_Toggle(LED2);
908    message[0]++;
909
910    etimer_set(&process_pause, random_rand()%2000);//random_rand()%3000);
911    PROCESS_WAIT_UNTIL(etimer_expired(&process_pause));
```

```
912
913       PRINTF("Sending␣message␣%d␣to:␣", message[0]);
914       IPSO_TEMPERATURE.read_value(&temp);
915       IPSO_HUMIDITY.read_value(&hum);
916       message[1]=hum;
917       message[2]=temp;
918       uip_debug_ipaddr_print(&dag->dag_id);
919       PRINTF("\n");
920       BSP_LED_Toggle(LED2);
921       simple_udp_sendto(&rpl_connection_test, &message, sizeof(message), &dag->dag_id);
922       BSP_LED_Toggle(LED2);
923       message[0]++;
924
925       RTC_TimeStructure.Hours,RTC_TimeStructure.Minutes,RTC_TimeStructure.Seconds, (1000*(
              RTC_TimeStructure.SecondFraction - RTC_TimeStructure.SubSeconds))/(RTC_TimeStructure.
              SecondFraction + 1));
926       wakeup=0;
927     }
928   #else
929     /* test */
930     etimer_set(&process_pause, 10*CLOCK_SECOND);//random_rand()%3000);
931     PROCESS_WAIT_UNTIL(etimer_expired(&process_pause));
932
933     PRINTF("Sending␣message␣%d␣to:␣", message[0]);
934     IPSO_TEMPERATURE.read_value(&temp);
935     IPSO_HUMIDITY.read_value(&hum);
936     message[1]=hum;
937     message[2]=temp;
938     uip_debug_ipaddr_print(&dag->dag_id);
939     PRINTF("\n");
940     //parent_debug();
941     simple_udp_sendto(&rpl_connection_test, &message, sizeof(message), &dag->dag_id);
942     message[0]++;
943   #endif /* test_mode */
944
945     HAL_RTC_GetTime(&RtcHandle, &RTC_TimeStructure, RTC_FORMAT_BIN);
946     HAL_RTC_GetDate(&RtcHandle, &RTC_DateStructure, RTC_FORMAT_BIN);
947     t_now=1000*(3600*RTC_TimeStructure.Hours + 60*RTC_TimeStructure.Minutes + RTC_TimeStructure.
              Seconds) + (1000*(RTC_TimeStructure.SecondFraction - RTC_TimeStructure.SubSeconds))/(
              RTC_TimeStructure.SecondFraction + 1);
948
949     if((t_now-t_old)>RADIO_RESET){
950       spirit_reset();
951     }
952     etimer_reset(&periodic_timer);
953    }
954
955   PROCESS_END();
956 }
957
958 /**
959 * @}
960 */
961
962 /**
963 * @}
964 */
965
966
967 /************************ (C) COPYRIGHT STMicroelectronics *****END OF FILE****/
```

A.2 main.c

```
1  /**
2  ******************************************************************************
3    * @file    main.c
4    * @author  Central LAB
5    * @version V1.0.0
6    * @date    20-January-2016
7    * @brief   Main program body
8  ******************************************************************************
9  * @attention
10 *
11 * <h2><center>&copy; COPYRIGHT(c) 2014 STMicroelectronics</center></h2>
12 *
13 * Redistribution and use in source and binary forms, with or without modification,
14 * are permitted provided that the following conditions are met:
15 *   1. Redistributions of source code must retain the above copyright notice,
16 *      this list of conditions and the following disclaimer.
17 *   2. Redistributions in binary form must reproduce the above copyright notice,
18 *      this list of conditions and the following disclaimer in the documentation
19 *      and/or other materials provided with the distribution.
```

```
20  *    3. Neither the name of STMicroelectronics nor the names of its contributors
21  *       may be used to endorse or promote products derived from this software
22  *       without specific prior written permission.
23  *
24  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
25  * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
26  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
27  * DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
28  * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
29  * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
30  * SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
31  * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
32  * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
33  * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
34  *
35  ******************************************************************************
36  */
37  /* Includes ------------------------------------------------------------------*/
38
39  #include "stdio.h"
40  #include "string.h"
41  #include "stdlib.h"
42  #include "main.h"
43  #include "cube_hal.h"
44  #include "radio_shield_config.h"
45  #include "spirit1.h"
46  #include "process.h"
47
48  /** @defgroup LWM2M_example
49    * @{
50    */
51
52  /** @addtogroup LWM2M_example
53    * @{
54    */
55
56  /* Function prototypes -----------------------------------------------------*/
57  void USARTConfig(void);
58  void Stack_6LoWPAN_Init(void);
59  void SystemClock_Config(void);
60  int MX_GPIO_Init(void);
61  int RTC_Config();
62  int RTC_TimeStampConfig();
63  /*---------------------------------------------------------------------------*/
64
65  extern RTC_HandleTypeDef RtcHandle;
66  uint8_t software_reset;
67
68  /**
69   * @brief  Main program
70   *   initialises HAL structures and calls the contiki main
71   * @param  None
72   * @retval None
73   */
74  int main()
75  {
76   GPIO_InitTypeDef GPIO_InitStructure= {0};
77
78      HAL_Init();
79      /* Configure the system clock */
80      SystemClock_Config();
81
82      //HAL_EnableDBGStopMode();
83
84      MX_GPIO_Init();
85
86      //enable 3.3v output pin for sensor init
87   __HAL_RCC_GPIOA_CLK_ENABLE();
88   GPIO_InitStructure.Mode=GPIO_MODE_OUTPUT_PP;
89   GPIO_InitStructure.Pull=GPIO_NOPULL;
90   GPIO_InitStructure.Pin=GPIO_PIN_0;
91   GPIO_InitStructure.Speed=GPIO_SPEED_VERY_LOW;
92   HAL_GPIO_Init(GPIOA, &GPIO_InitStructure);
93
94   HAL_GPIO_WritePin(GPIOA, GPIO_PIN_0, GPIO_PIN_RESET);
95   HAL_Delay(10);
96   HAL_GPIO_WritePin(GPIOA, GPIO_PIN_0, GPIO_PIN_SET);
97   HAL_Delay(10);
98      //END enable 3.3v output pin for sensor init
99
100
101     /* Initialize LEDs */
102     BSP_LED_Init(LED2);
103
104     RadioShieldLedInit(RADIO_SHIELD_LED);
```

```
105
106       BSP_PB_Init(BUTTON_USER, BUTTON_MODE_EXTI);
107
108       USARTConfig();
109
110       /* Initialize RTC */
111       RTC_Config();
112       //config the rtc only if a power reset occurred
113       if(RCC->CSR & RCC_CSR_PORRSTF){
114            SET_BIT(RCC->CSR, RCC_CSR_RMVF);
115            RTC_TimeStampConfig();
116            software_reset=0;
117       }else{
118        software_reset=1;
119       }
120
121         /*Compiler, HAL and firmware info:*/
122       printf("\t(HAL␣%ld.%ld.%ld_%ld)\r\n"
123           "\tCompiled␣%s␣%s"
124  #if defined (__IAR_SYSTEMS_ICC__)
125           "␣(IAR)\r\n\n",
126  #elif defined (__CC_ARM)
127           "␣(KEIL)\r\n\n",
128  #elif defined (__GNUC__)
129           "␣(openstm32)\r\n\n",
130  #endif
131              HAL_GetHalVersion() >>24,
132             (HAL_GetHalVersion() >>16)&0xFF,
133             (HAL_GetHalVersion() >> 8)&0xFF,
134              HAL_GetHalVersion()      &0xFF,
135            __DATE__,__TIME__);
136
137       Stack_6LoWPAN_Init();
138
139       while(1) {
140         int r = 0;
141         do {
142           r = process_run();
143         } while(r > 0);
144         HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI);
145       }
146
147  }
148
149  /**
150    * @}
151    */
152
153  /**
154    * @}
155    */
156
157
158  /************************ (C) COPYRIGHT STMicroelectronics *****END OF FILE****/
```

A.3 border-router.c

```
1   /*
2    * Redistribution and use in source and binary forms, with or without
3    * modification, are permitted provided that the following conditions
4    * are met:
5    * 1. Redistributions of source code must retain the above copyright
6    *    notice, this list of conditions and the following disclaimer.
7    * 2. Redistributions in binary form must reproduce the above copyright
8    *    notice, this list of conditions and the following disclaimer in the
9    *    documentation and/or other materials provided with the distribution.
10   * 3. Neither the name of the Institute nor the names of its contributors
11   *    may be used to endorse or promote products derived from this software
12   *    without specific prior written permission.
13   *
14   * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ``AS IS'' AND
15   * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
16   * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
17   * ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
18   * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
19   * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
20   * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
21   * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
22   * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
23   * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
24   * SUCH DAMAGE.
25   *
```

```
26   * This file is part of the Contiki operating system.
27   *
28   */
29  /**
30   * \file
31   *         border-router
32   * \author
33   *         Niclas Finne <nfi@sics.se>
34   *         Joakim Eriksson <joakime@sics.se>
35   *         Nicolas Tsiftes <nvt@sics.se>
36   */
37
38  #include "contiki.h"
39  #include "contiki-lib.h"
40  #include "contiki-net.h"
41  #include "net/ip/uip.h"
42  #include "net/ip/uip-debug.h"
43  #include "net/ipv6/uip-ds6.h"
44  #include "net/ipv6/multicast/uip-mcast6.h"
45
46  #include "net/rpl/rpl.h"
47
48  #include "net/netstack.h"
49  #include "dev/button-sensor.h"
50  #include "dev/slip.h"
51
52  #include <stdio.h>
53  #include <stdlib.h>
54  #include <string.h>
55  #include <ctype.h>
56
57  #include "cube_hal.h"
58
59  /** @addtogroup Border_router
60   * @{
61   */
62
63  #define DEBUG DEBUG_PRINT
64  #if DEBUG || 1
65  #define PRINTF(...) printf(__VA_ARGS__)
66  #endif
67
68  #define MULTICAST_SYNC 1
69  #define UNICAST_SYNC 2
70
71  /*Select a sync mechanism*/
72  //#define SYNC_TYPE MULTICAST_SYNC
73  #define SYNC_TYPE UNICAST_SYNC
74
75  /* external variable */
76  extern RTC_HandleTypeDef RtcHandle;
77
78  /* broadcast synch management */
79  #define SYNC_START_DELAY 2
80  #define START_DELAY 5
81  #define SEND_INTERVAL 450
82  #define ON_TIME 5
83  #define SYNC_UDP_PORT 3001
84  #define TEST_UDP_PORT 3003
85  #define SYNCHRO 2
86  #define LP_MODE_ON 1
87  #define LP_MODE_OFF 0
88
89  /* Mcast message typedef */
90  typedef struct
91  {
92   uint8_t  MessageType;
93   uint8_t  TimeFormat;
94   uint8_t  Hours;
95   uint8_t  Minutes;
96   uint8_t     Seconds;
97   uint32_t SubSeconds;
98   uint32_t SecondFraction;
99   uint8_t  RunMode;
100 }Sync_MessageTypeDef;
101
102 static Sync_MessageTypeDef mcast_message;
103
104 #if SYNC_TYPE == MULTICAST_SYNC
105
106 static struct uip_udp_conn * mcast_conn;
107 static void prepare_mcast(void);
108
109 #elif SYNC_TYPE == UNICAST_SYNC
110 static struct simple_udp_connection unicast_connection;
```

```
111 static uip_ipaddr_t mc_addr;
112 static void
113 receiver(struct simple_udp_connection *c,
114   const uip_ipaddr_t *sender_addr,
115   uint16_t sender_port,
116   const uip_ipaddr_t *receiver_addr,
117   uint16_t receiver_port,
118   const uint8_t *data,
119   uint16_t datalen);
120 #endif
121
122 static struct simple_udp_connection rpl_connection_test;
123 static void
124 receiver2(struct simple_udp_connection *c,
125   const uip_ipaddr_t *sender_addr,
126   uint16_t sender_port,
127   const uip_ipaddr_t *receiver_addr,
128   uint16_t receiver_port,
129   const uint32_t *data,
130   uint16_t datalen);
131
132 static struct etimer synch_timer;
133 static struct etimer rand_timer;
134
135 static void send_synchro_message(void);
136
137 static uint8_t synchro_sent=0;
138 static uip_ipaddr_t prefix;
139 static uint8_t prefix_set;
140
141 #if SYNC_TYPE == UNICAST_SYNC
142   static void
143   receiver(struct simple_udp_connection *c,
144     const uip_ipaddr_t *sender_addr,
145     uint16_t sender_port,
146     const uip_ipaddr_t *receiver_addr,
147     uint16_t receiver_port,
148     const uint8_t *data,
149     uint16_t datalen)
150   {
151    //border router does not receive sync message
152   }
153 #endif
154
155   static void
156   receiver2(struct simple_udp_connection *c,
157     const uip_ipaddr_t *sender_addr,
158     uint16_t sender_port,
159     const uip_ipaddr_t *receiver_addr,
160     uint16_t receiver_port,
161     const uint32_t *data,
162     uint16_t datalen)
163   {
164    PRINTF("Message␣%d␣received␣from:␣", data[0]);
165    uip_debug_ipaddr_print(sender_addr);
166    printf("␣Hum:␣%d␣rH\t␣Temp:␣%d␣C\n", data[1]/1000, data[2]/1000);
167
168   }
169
170   /*---------------------------------------------------------------------------*/
171   static void
172   send_synchro_message(void){
173
174    static RTC_TimeTypeDef RTC_TimeStructure;
175    static RTC_DateTypeDef RTC_DateStructure;
176
177 #if SYNC_TYPE == UNICAST_SYNC
178    static uip_ds6_nbr_t *nbr;
179 #endif
180    int x;
181    HAL_RTC_GetTime(&RtcHandle, &RTC_TimeStructure, RTC_FORMAT_BIN);
182    HAL_RTC_GetDate(&RtcHandle, &RTC_DateStructure, RTC_FORMAT_BIN);
183    x=(1000*(RTC_TimeStructure.SecondFraction - RTC_TimeStructure.SubSeconds))/(RTC_TimeStructure
         .SecondFraction + 1);
184    if(x>=500)
185    {
186     do{
187      rtimer_clock_t t0;
188      t0 = RTIMER_NOW();
189      while( RTIMER_CLOCK_LT(RTIMER_NOW(), t0 + (500)));
190     }while(0);
191      HAL_RTC_GetTime(&RtcHandle, &RTC_TimeStructure, RTC_FORMAT_BIN);
192     HAL_RTC_GetDate(&RtcHandle, &RTC_DateStructure, RTC_FORMAT_BIN);
193    }
194
```

```
195       mcast_message.MessageType = SYNCHRO;
196       mcast_message.TimeFormat = RTC_TimeStructure.TimeFormat;
197       mcast_message.Hours = RTC_TimeStructure.Hours;
198       mcast_message.Minutes = RTC_TimeStructure.Minutes;
199       mcast_message.Seconds = RTC_TimeStructure.Seconds;
200       mcast_message.SubSeconds = RTC_TimeStructure.SubSeconds;
201       mcast_message.SecondFraction = RTC_TimeStructure.SecondFraction;
202       mcast_message.RunMode = LP_MODE_ON;
203
204   #if SYNC_TYPE == MULTICAST_SYNC
205       uip_udp_packet_send(mcast_conn, &mcast_message, sizeof(mcast_message));
206   #elif SYNC_TYPE == UNICAST_SYNC
207       simple_udp_sendto(&unicast_connection, &mcast_message, sizeof(mcast_message), &mc_addr);
208   #endif
209       PRINTF("\nSending␣current␣time:␣%02d:%02d:%02d.%03d\n", RTC_TimeStructure.Hours,
              RTC_TimeStructure.Minutes,RTC_TimeStructure.Seconds, (1000*(RTC_TimeStructure.
              SecondFraction - RTC_TimeStructure.SubSeconds))/(RTC_TimeStructure.SecondFraction + 1));
210     }
211
212     /*---------------------------------------------------------------------------*/
213   #if SYNC_TYPE == MULTICAST_SYNC
214     static void
215     prepare_mcast(void)
216     {
217       uip_ipaddr_t ipaddr;
218
219       /*
220        * IPHC will use stateless multicast compression for this destination
221        * (M=1, DAC=0), with 32 inline bits (1E 89 AB CD)
222        */
223       uip_ip6addr(&ipaddr, 0xFF1E,0,0,0,0,0x89,0xABCD);
224       mcast_conn = udp_new(&ipaddr, UIP_HTONS(SYNC_UDP_PORT), NULL);
225     }
226   #endif
227
228     PROCESS(border_router_process, "Border␣router␣process");
229
230   #if WEBSERVER==0
231     /* No webserver */
232     AUTOSTART_PROCESSES(&border_router_process);
233   #elif WEBSERVER>1
234     /* Use an external webserver application */
235   #include "webserver-nogui.h"
236     AUTOSTART_PROCESSES(&border_router_process,&webserver_nogui_process);
237   #else
238     /* Use simple webserver with only one page for minimum footprint.
239      * Multiple connections can result in interleaved tcp segments since
240      * a single static buffer is used for all segments.
241      */
242   #include "httpd-simple.h"
243     /* The internal webserver can provide additional information if
244      * enough program flash is available.
245      */
246   #define WEBSERVER_CONF_LOADTIME 0
247   #define WEBSERVER_CONF_FILESTATS 0
248   #define WEBSERVER_CONF_NEIGHBOR_STATUS 0
249     /* Adding links requires a larger RAM buffer. To avoid static allocation
250      * the stack can be used for formatting; however tcp retransmissions
251      * and multiple connections can result in garbled segments.
252      * TODO:use PSOCk_GENERATOR_SEND and tcp state storage to fix this.
253      */
254   #define WEBSERVER_CONF_ROUTE_LINKS 0
255   #if WEBSERVER_CONF_ROUTE_LINKS
256   #define BUF_USES_STACK 1
257   #endif
258
259
260     PROCESS(webserver_nogui_process, "Web␣server");
261     PROCESS_THREAD(webserver_nogui_process, ev, data)
262     {
263       PROCESS_BEGIN();
264
265       httpd_init();
266
267       while(1) {
268         PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);
269         httpd_appcall(data);
270       }
271
272       PROCESS_END();
273     }
274     AUTOSTART_PROCESSES(&border_router_process,&webserver_nogui_process);
275
276     static const char *TOP = "<html><head><title>ContikiRPL</title></head><body>\n";
277     static const char *BOTTOM = "</body></html>\n";
```

```
278 #if BUF_USES_STACK
279   static char *bufptr, *bufend;
280 #define ADD(...) do {                                                   \
281   bufptr += snprintf(bufptr, bufend - bufptr, __VA_ARGS__);       \
282 } while(0)
283 #else
284   static char buf[256];
285   static int blen;
286 #define ADD(...) do {                                                   \
287   blen += snprintf(&buf[blen], sizeof(buf) - blen, __VA_ARGS__);       \
288 } while(0)
289 #endif
290
291   /*---------------------------------------------------------------------------*/
292   static void
293   ipaddr_add(const uip_ipaddr_t *addr)
294   {
295    uint16_t a;
296    int i, f;
297    for(i = 0, f = 0; i < sizeof(uip_ipaddr_t); i += 2) {
298     a = (addr->u8[i] << 8) + addr->u8[i + 1];
299     if(a == 0 && f >= 0) {
300      if(f++ == 0) ADD("::");
301     } else {
302      if(f > 0) {
303       f = -1;
304      } else if(i > 0) {
305       ADD(":");
306      }
307      ADD("%x", a);
308     }
309    }
310   }
311   /*---------------------------------------------------------------------------*/
312   static
313   PT_THREAD(generate_routes(struct httpd_state *s))
314   {
315    static uip_ds6_route_t *r;
316    static uip_ds6_nbr_t *nbr;
317 #if BUF_USES_STACK
318    char buf[256];
319 #endif
320 #if WEBSERVER_CONF_LOADTIME
321    static clock_time_t numticks;
322    numticks = clock_time();
323 #endif
324
325    PSOCK_BEGIN(&s->sout);
326
327    SEND_STRING(&s->sout, TOP);
328 #if BUF_USES_STACK
329    bufptr = buf;bufend=bufptr+sizeof(buf);
330 #else
331    blen = 0;
332 #endif
333    ADD("Neighbors<pre>");
334
335    for(nbr = nbr_table_head(ds6_neighbors);
336      nbr != NULL;
337      nbr = nbr_table_next(ds6_neighbors, nbr)) {
338
339 #if WEBSERVER_CONF_NEIGHBOR_STATUS
340 #if BUF_USES_STACK
341     {char* j=bufptr+25;
342     ipaddr_add(&nbr->ipaddr);
343     while (bufptr < j) ADD(" ");
344     switch (nbr->state) {
345     case NBR_INCOMPLETE: ADD(" INCOMPLETE");break;
346     case NBR_REACHABLE: ADD(" REACHABLE");break;
347     case NBR_STALE: ADD(" STALE");break;
348     case NBR_DELAY: ADD(" DELAY");break;
349     case NBR_PROBE: ADD(" NBR_PROBE");break;
350     }
351     }
352 #else
353     {uint8_t j=blen+25;
354     ipaddr_add(&nbr->ipaddr);
355     while (blen < j) ADD(" ");
356     switch (nbr->state) {
357     case NBR_INCOMPLETE: ADD(" INCOMPLETE");break;
358     case NBR_REACHABLE: ADD(" REACHABLE");break;
359     case NBR_STALE: ADD(" STALE");break;
360     case NBR_DELAY: ADD(" DELAY");break;
361     case NBR_PROBE: ADD(" NBR_PROBE");break;
362     }
```

```
363       }
364   #endif
365   #else
366       ipaddr_add(&nbr->ipaddr);
367   #endif
368
369       ADD("\n");
370   #if BUF_USES_STACK
371       if(bufptr > bufend - 45) {
372         SEND_STRING(&s->sout, buf);
373         bufptr = buf; bufend = bufptr + sizeof(buf);
374       }
375   #else
376       if(blen > sizeof(buf) - 45) {
377         SEND_STRING(&s->sout, buf);
378         blen = 0;
379       }
380   #endif
381     }
382     ADD("</pre>Routes<pre>");
383     SEND_STRING(&s->sout, buf);
384   #if BUF_USES_STACK
385     bufptr = buf; bufend = bufptr + sizeof(buf);
386   #else
387     blen = 0;
388   #endif
389
390     for(r = uip_ds6_route_head(); r != NULL; r = uip_ds6_route_next(r)) {
391
392   #if BUF_USES_STACK
393   #if WEBSERVER_CONF_ROUTE_LINKS
394       ADD("<a href=http://[");
395       ipaddr_add(&r->ipaddr);
396       ADD("]/status.shtml>");
397       ipaddr_add(&r->ipaddr);
398       ADD("</a>");
399   #else
400       ipaddr_add(&r->ipaddr);
401   #endif
402   #else
403   #if WEBSERVER_CONF_ROUTE_LINKS
404       ADD("<a href=http://[");
405       ipaddr_add(&r->ipaddr);
406       ADD("]/status.shtml>");
407       SEND_STRING(&s->sout, buf); //TODO: why tunslip6 needs an output here, wpcapslip does not
408       blen = 0;
409       ipaddr_add(&r->ipaddr);
410       ADD("</a>");
411   #else
412       ipaddr_add(&r->ipaddr);
413   #endif
414   #endif
415       ADD("/%u (via ", r->length);
416       ipaddr_add(uip_ds6_route_nexthop(r));
417       if(1 || (r->state.lifetime < 600)) {
418         ADD(") %lus\n", (unsigned long)r->state.lifetime);
419       } else {
420         ADD(")\n");
421       }
422       SEND_STRING(&s->sout, buf);
423   #if BUF_USES_STACK
424       bufptr = buf; bufend = bufptr + sizeof(buf);
425   #else
426       blen = 0;
427   #endif
428     }
429     ADD("</pre>");
430
431   #if WEBSERVER_CONF_FILESTATS
432     static uint16_t numtimes;
433     ADD("<br><i>This page sent %u times</i>",++numtimes);
434   #endif
435
436   #if WEBSERVER_CONF_LOADTIME
437     numticks = clock_time() - numticks + 1;
438     ADD(" <i>(%u.%02u sec)</i>",numticks/CLOCK_SECOND,(100*(numticks%CLOCK_SECOND))/CLOCK_SECOND
          );
439   #endif
440
441     SEND_STRING(&s->sout, buf);
442     SEND_STRING(&s->sout, BOTTOM);
443
444     PSOCK_END(&s->sout);
445   }
446   /*---------------------------------------------------------------------------*/
```

```
447     httpd_simple_script_t
448     httpd_simple_get_script(const char *name)
449     {
450
451       return generate_routes;
452     }
453
454   #endif /* WEBSERVER */
455
456     /*---------------------------------------------------------------------------*/
457     static void
458     print_local_addresses(void)
459     {
460       int i;
461       uint8_t state;
462
463       PRINTA("Server IPv6 addresses:\n");
464       for(i = 0; i < UIP_DS6_ADDR_NB; i++) {
465         state = uip_ds6_if.addr_list[i].state;
466         if(uip_ds6_if.addr_list[i].isused &&
467           (state == ADDR_TENTATIVE || state == ADDR_PREFERRED)) {
468           PRINTA(" ");
469           uip_debug_ipaddr_print(&uip_ds6_if.addr_list[i].ipaddr);
470           PRINTA("\n");
471         }
472       }
473     }
474     /*---------------------------------------------------------------------------*/
475     void
476     request_prefix(void)
477     {
478       /* mess up uip_buf with a dirty request... */
479       uip_buf[0] = '?';
480       uip_buf[1] = 'P';
481       uip_len = 2;
482       slip_send();
483       uip_clear_buf();
484     }
485     /*---------------------------------------------------------------------------*/
486     void
487     set_prefix_64(uip_ipaddr_t *prefix_64)
488     {
489       rpl_dag_t *dag;
490       uip_ipaddr_t ipaddr;
491       memcpy(&prefix, prefix_64, 16);
492       memcpy(&ipaddr, prefix_64, 16);
493       prefix_set = 1;
494       uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
495       uip_ds6_addr_add(&ipaddr, 0, ADDR_AUTOCONF);
496
497       dag = rpl_set_root(RPL_DEFAULT_INSTANCE, &ipaddr);
498       if(dag != NULL) {
499         rpl_set_prefix(dag, &prefix, 64);
500         PRINTF("created a new RPL dag\n");
501       }
502     }
503
504     /*---------------------------------------------------------------------------*/
505     PROCESS_THREAD(border_router_process, ev, data)
506     {
507       static struct etimer et;
508       static uint16_t t=0, i;
509       static RTC_TimeTypeDef RTC_TimeStructure;
510       static RTC_DateTypeDef RTC_DateStructure;
511
512       PROCESS_BEGIN();
513
514       /* While waiting for the prefix to be sent through the SLIP connection, the future
515        * border router can join an existing DAG as a parent or child, or acquire a default
516        * router that will later take precedence over the SLIP fallback interface.
517        * Prevent that by turning the radio off until we are initialized as a DAG root.
518        */
519       prefix_set = 0;
520       NETSTACK_MAC.off(0);
521
522       PROCESS_PAUSE();
523
524       SENSORS_ACTIVATE(button_sensor);
525
526       PRINTF("RPL-Border router started\n");
527   #if 0
528       /* The border router runs with a 100% duty cycle in order to ensure high
529         packet reception rates.
530         Note if the MAC RDC is not turned off now, aggressive power management of the
531         cpu will interfere with establishing the SLIP connection */
```

```
532     NETSTACK_MAC.off(1);
533   #endif
534
535     /* Request prefix until it has been received */
536     while(!prefix_set) {
537      etimer_set(&et, CLOCK_SECOND);
538      request_prefix();
539      PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
540     }
541
542     /* Now turn the radio on, but disable radio duty cycling.
543      * Since we are the DAG root, reception delays would constrain mesh throughbut.
544      */
545     NETSTACK_MAC.off(1);
546
547   #if DEBUG || 1
548     print_local_addresses();
549   #endif
550
551     /* mcast management */
552     /* own address already set*/
553   #if SYNC_TYPE == MULTICAST_SYNC
554     prepare_mcast();
555   #elif SYNC_TYPE == UNICAST_SYNC
556     simple_udp_register(&unicast_connection, SYNC_UDP_PORT,
557       NULL, SYNC_UDP_PORT, receiver);
558
559     uip_ip6addr(&mc_addr, 0xfc00, 0, 0, 0, 0, 0, 0x1337, 0x0002);
560     uip_create_linklocal_allnodes_mcast(&mc_addr);
561
562   #endif
563
564     simple_udp_register(&rpl_connection_test, TEST_UDP_PORT,
565       NULL, TEST_UDP_PORT, receiver2);
566
567     etimer_set(&synch_timer, SYNC_START_DELAY*CLOCK_SECOND);
568     for(i=0; i<10; i++){
569      PROCESS_YIELD();
570      if(etimer_expired(&synch_timer)){
571       send_synchro_message();
572       etimer_set(&synch_timer, SYNC_START_DELAY*CLOCK_SECOND);
573      }
574     }
575     etimer_set(&synch_timer, START_DELAY*CLOCK_SECOND);
576
577     /* calibration giavatto */
578     GPIO_InitTypeDef GPIO_InitStruct;
579
580     GPIO_InitStruct.Pin = GPIO_PIN_13;
581     GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
582     GPIO_InitStruct.Pull = GPIO_NOPULL;
583     GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
584     GPIO_InitStruct.Alternate = GPIO_AF0_TAMPER;
585     HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
586
587
588     if (HAL_RTCEx_SetCalibrationOutPut(&RtcHandle, RTC_CALIBOUTPUT_1HZ) != HAL_OK)
589     {
590      Error_Handler();
591     }
592     /* calibration giavatto */
593
594     while(1) {
595      PROCESS_YIELD();
596      if (ev == sensors_event && data == &button_sensor) {
597       PRINTF("Initiating␣global␣repair\n");
598       rpl_repair_root(RPL_DEFAULT_INSTANCE);
599      }
600
601      HAL_RTC_GetTime(&RtcHandle, &RTC_TimeStructure, RTC_FORMAT_BIN);
602      HAL_RTC_GetDate(&RtcHandle, &RTC_DateStructure, RTC_FORMAT_BIN);
603
604      t = (RTC_TimeStructure.Minutes*60 + RTC_TimeStructure.Seconds)%SEND_INTERVAL;
605
606      if(t < ON_TIME && synchro_sent==0){
607       etimer_set(&rand_timer, random_rand()%2000);
608       PROCESS_WAIT_UNTIL(etimer_expired(&rand_timer));
609
610       send_synchro_message();
611       synchro_sent=1;
612      } else if (t >= ON_TIME){
613       synchro_sent=0;
614      }
615      etimer_set(&synch_timer, CLOCK_SECOND/5);
616     }
```

```
617
618       PROCESS_END();
619     }
620     /*----------------------------------------------------------------------------*/
621
622   /**
623   * @}
624   */
625
626   /**
627   * @}
628   */
629
630
631   /************************ (C) COPYRIGHT STMicroelectronics *****END OF FILE****/
```

Bibliography

[1] International Telecommunication Union. *ITU Internet Reports - The Internet of Things*. Tech. rep. International Telecommunication Union (ITU), 2005.

[2] M. C. Vuran I. F. Akyildiz. *Wireless Sensor Networks*. Ian F. Akyildiz Series in Communication and Networking. Wiley, 2010. ISBN: 9780470036013.

[3] A. Sassone S. Gallinaro S. Rinaudo M. Poncino E. Macii D. Demarchi M. Crepaldi M. Grosso. "A Top-Down Constraint-Driven Methodology for Smart System Design View Document". In: *IEEE Circuits and Systems Society* (2014). DOI: 10.1109/MCAS.2013.2296415.

[4] LoRa Alliance Technology. *LoRa Alliance - Wide Area Network for IoT*. URL: https://www.lora-alliance.org/.

[5] A. Jamalipour J. Zheng. Wiley-IEEE Press, 2009. ISBN: 978-0-470-16763-2.

[6] A.E. Kamal J.N. Al-Karaki. "Routing techniques in wireless sensor networks: a survey". In: *IEEE Communications Surveys & Tutorials* (2004). DOI: 10.1109/MWC.2004.1368893.

[7] T. Znati K. Sohraby D. Minoli. *Wireless Sensor Networks*. Wiley, 2007. ISBN: 978-0-471-74300-2.

[8] T. Winter et al. *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks (RFC: 6550)*. Tech. rep. Internet Engineering Task Force (IETF), 2012.

[9] JP. Vasseur et al. *Routing Metrics Used for Path Calculation in Low-Power and Lossy Networks (RFC: 6551)*. Tech. rep. Internet Engineering Task Force (IETF), 2012.

[10] P. Levis et al. *The Trickle Algorithm (RFC: 6206)*. Tech. rep. Internet Engineering Task Force (IETF), 2011.

[11] D. Mills et al. *Network Time Protocol Version 4: Protocol and Algorithms Specification (RFC: 5905)*. Tech. rep. Internet Engineering Task Force (IETF), 2010.

[12] M. B. Srivastava S. Ganeriwal R. Kumar. "Timing-sync Protocol for Sensor Networks". In: *SenSys '03 Proceedings of the 1st international conference on Embedded networked sensor systems*. 2003. DOI: 10.1145/958491.958508.

[13] D. Estrin J. Elson L. Girod. "Fine-grained network time synchronization using reference broadcasts". In: *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*. 2002. DOI: 10.1145/844128.844143.

[14] T. Kunz M. O. Farooq. "Operating Systems for Wireless Sensor Networks: A Survey". In: *Multidisciplinary Digital Publishing Institute MDPI* (2011). DOI: 10.3390/s110605900.

[15] T. Voigt A. Dunkels B. Gronvall. "Contiki - a lightweight and flexible operating system for tiny networked sensors." In: *IEEE Communications Surveys & Tutorials* (2004). DOI: 10.1109/LCN.2004.38.

[16] STMicroelectronics. *Accurate power consumption estimation for STM32L1 series of ultra-low-power microcontrollers (TA0342)*. 2013. URL: http://www.st.com/content/ccc/resource/technical/document/technical_article/57/8f/9e/f3/7e/d5/42/2a/DM00024152.pdf/files/DM00024152.pdf/jcr:content/translations/en.DM00024152.pdf.

[17] H. Diewald J. Borgeson S. Schauer. *Benchmarking MCU power consumption for ultra-low-power applications*. 2012. URL: http://www.ti.com/lit/wp/slay023/slay023.pdf.

[18] L. C. S. Magalhães C. V. N. Albuquerque R. C. Carrano D. Passos. "Survey and Taxonomy of Duty Cycling Mechanisms in Wireless Sensor Networks." In: *IEEE Communications Surveys & Tutorials* (2013). DOI: 10.1109/SURV.2013.052213.00116.

[19] S. A. Shariff S. S. Yuhaniz N. A. Ahmad O. M. Yusop S. Ismail K. Z. Panatik K. Kamardin. "Energy Harvesting in Wireless Sensor Networks: A Survey". In: *2016 IEEE 3rd International Symposium on Telecommunication Technologies (ISTT), Kuala Lumpur, 28-30 Nov 2016*. 2016. DOI: 10.1109/ISTT.2016.7918084.

[20] J. Drew. *Energy harvester produces power from local environment, eliminating batteries in Wireless Sensors (Design Note 483)*. 2010. URL: http://cds.linear.com/docs/en/design-note/DN483.pdf.

[21] Panasonic. *Amorphous Silicon Solar Cells*. 2016. URL: https://panasonic.co.jp/es/pesam/en/products/pdf/Catalog_Amorton_ENG.pdf.

[22] R. Bensalem R. Kanan. "Energy Harvesting for Wearable Wireless Health Care Systems". In: *2016 IEEE Wireless Communications and Networking Conference (WCNC)*. 2016. DOI: 10.1109/WCNC.2016.7565034.

[23] T. Le A. Jushi A. Pegatoquet. "Wind Energy Harvesting for Autonomous Wireless Sensor Networks". In: *2016 Euromicro Conference on Digital System Design*. 2016. DOI: 10.1109/DSD.2016.43.

[24] M. Grosso E. Macii M. Poncino. "M07 Power efficiency: Power efficiency in the design of Smart IoT devices". In: *Date 16, Design, Automation and Test in Europe*. 2016.

[25] P. Thubert et al. *Objective Function Zero for the Routing Protocol for Low-Power and Lossy Networks (RPL) (RFC: 6552)*. Tech. rep. Internet Engineering Task Force (IETF), 2012.

[26] O. Gnawali et al. *The Minimum Rank with Hysteresis Objective Function (RFC: 6719)*. Tech. rep. Internet Engineering Task Force (IETF), 2012.

[27] STMicroelectronics. *Ultralow power energy harvester and battery charger*. 2015. URL: http://www.st.com/content/ccc/resource/technical/document/datasheet/3e/91/0f/7c/32/3e/46/dd/DM00100984.pdf/files/DM00100984.pdf/jcr:content/translations/en.DM00100984.pdf.

[28] STMicroelectronics. *STM32 ODE Function Packs*. URL: http://www.st.com/en/ecosystems/stm32-ode-function-packs.html?querycriteria=productId=SC2102.

[29] R. Russo. *Tecniche di riduzione dei consumi applicate ad una rete mesh di sensori ambientali*. 2016.

[30] STMicroelectronics. *STM32L100xx, STM32L151xx, STM32L152xx and STM32L162xx advanced ARM®-based 32-bit MCUs (RM0038)*. 2016. URL: http://www.st.com/content/ccc/resource/technical/document/reference_manual/cc/f9/93/b2/f0/82/42/57/CD00240193.pdf/files/CD00240193.pdf/jcr:content/translations/en.CD00240193.pdf.

[31] STMicroelectronics. *Ultra-low-power 32-bit MCU ARM®-based Cortex®-M3 with 512KB Flash, 80KB SRAM, 16KB EEPROM, LCD, USB, ADC, DAC*. 2017. URL: http://www.st.com/content/ccc/resource/technical/document/datasheet/group1/a7/13/6a/ce/1f/f3/40/c1/DM00098321/files/DM00098321.pdf/jcr:content/translations/en.DM00098321.pdf.

[32] STMicroelectronics. *Low data rate, low power sub-1GHz transceiver*. 2016. URL: http://www.st.com/content/ccc/resource/technical/document/datasheet/68/6c/7b/ec/b2/6b/49/16/DM00047607.pdf/files/DM00047607.pdf/jcr:content/translations/en.DM00047607.pdf.

[33] STMicroelectronics. *Low duty cycle operation with the SPIRIT1 transceiver (AN4193)*. URL: http://www.st.com/content/ccc/resource/technical/document/application_note/47/50/91/3a/f5/fd/4d/c1/DM00068699.pdf/files/DM00068699.pdf/jcr:content/translations/en.DM00068699.pdf.

[34] J. Hui et al. *Multicast Protocol for Low-Power and Lossy Networks (MPL) (RFC: 7731)*. Tech. rep. Internet Engineering Task Force (IETF), 2016.

[35] Contiki Community. *Get Started with Contiki, Instant Contiki and Cooja.* URL: http://www.contiki-os.org/start.html.