UNIVERSITY OF THESSALY

Protocol for node management and message routing in ad-hoc wireless sensor networks

by

Nikos Oikonomou

A thesis submitted in fulfillment for the
degree of Diploma

in the

Department of Electrical and Computer Engineering

October 11, 2018

Περίληψη

Πολύ συχνά έχουμε την ανάγκη να παρακολουθήσουμε τις περιβαλλοντικές συνθήκες προκειμένου να αναλάβουμε δράση σύμφωνα με αυτές. Σ' αυτό το πρόβλημα μπορούν να βοηθήσουν ασύρματες συσκευές, που λειτουργούν με μπαταρία και παρέχουν μετρήσεις που λαμβάνουν από αισθητήρες. Για το συντονισμό τέτοιων συσκευών και την συλλογή των δεδομένων τους, πρέπει να σχηματιστούν ασύρματα δίκτυα αισθητήρων που να επιτρέπουν την εκ των υστέρων τοποθέτηση ή μετακίνηση των συσκευών για να δημιουργηθεί ένα πάντα λειτουργικό και με καλή απόδοση σύστημα.

Ο στόχος της διπλωματικής είναι να δημιουργηθεί ένα πρωτόκολλο που στοχεύει στο να λύσει τα παραπάνω προβλήματα και να προσφέρει στον τελικό χρήστη ένα τρόπο να διαχειριστεί τις συσκευές του χωρίς κόπο. Περιγράφει τα δίκτυα που δημιουργούνται και παρέχει λύσεις για κάθε εργασία που πρέπει να γίνει, π.χ. την αξιόπιστη συλλογή δεδομένων, τη συντονισμένη δειγματοληψία και την ασύρματη ενημέρωση του λογισμικού των συσκευών. Επικεντρώνεται κυρίως στην ελαχιστοποίηση της κατανάλωσης με στόχο την επέκταση ζωής της μπαταρίας των συσκευών και την παροχή δικτύων που μπορούν να λειτουργούν για μεγάλα χρονικά διαστήματα χωρίς ανθρώπινη επέμβαση.

Abstract

Very often often people need to remotely monitor the environmental conditions in order to take actions according to them and wireless battery-powered devices can help us solve this problem by taking samples using sensors. Wireless sensor networks must be created for the coordination of such devices and the collection of their samples. These networks must allow the ad-hoc placement of the devices in order to create a robust and always working system.

The target of this thesis is to create a protocol that focuses on implementing such a task and allows the end user to manage his devices without effort. It describes everything that happens inside such networks and provides a solution for all the related problems, e.g. the reliable data collection, the coordinated sampling and the over the air firmware updates. It focuses mainly on power efficiency, in order to prolong the battery life of the devices so that the networks can operate for long time periods without user intervention.

Acknowledgements

Firstly, I would like to thank my supervisor Dr. Spyros Lalis for his guidance and our collaboration during the development of my thesis.

Secondly, I would like to thank Dr. Sotiris Bantas for providing me the equipment to work with and all my co-workers at Centaur Analytics for their support.

Finally, I would like to thank Chrisa for her patience and love all these years.

Contents

List of Figures

List of Tables

Abbreviations

WSN	**W**ireless **S**ensor **N**etwork
OS	**O**perating **S**ystem
RTOS	**R**eal **T**ime **O**perating **S**ystem
MCU	**M**icro**C**ontroller **U**nit
DCT	**D**ata**C**ollection **T**ree
OTA	**O**ver **T**heir **A**ir
LPM	**L**ow **P**ower **M**ode
RDC	**R**adio **D**uty **C**ycle
MAC	**M**edium **A**ccess **C**ontrol
UI	**U**ser **I**nterface
RTC	**R**eal **T**ime **C**lock

Chapter 1

Introduction

1.1 Motivation

Ad hoc Wireless Sensor Networks

A wireless sensor network is a group of spatially distributed autonomous devices that monitor various phenomenon e.g. temperature or humidity, using sensors and cooperate with each other in order to transfer their data to the outside world. They are called ad hoc because they don't rely on pre-existing infrastructure, but instead, their topology is constructed dynamically. The routing algorithms, inside such networks, are designed to anticipate device movement and allow devices to join the network on the fly.

The devices that participate in WSN are often referred as nodes. Each node usually consists of a radio transceiver, a microcontroller, and some sensors, but might also have other components attached to it e.g. actuators to interact with the environment or powerful processors to do heavy computations. Some of the nodes might also be connected with powerful devices that can communicate with the outside world and transfer messages inside and outside of the WSN.

Although WSN offers zero wiring costs and ease of installation, it has to be robust and in some cases power-efficient because maintenance and/or battery replacement can be very difficult, if not impossible.

Based on the communication protocol and its restrictions, a WSN can consist of dozens, hundreds or even more nodes that can be active from days to years and can self-cure based on current conditions, e.g. a node lost connection or destroyed. That's why WSNs are appealing to many applications, such as environmental monitoring, surveillance, tracking and more.

Environmental Monitoring

Very often people need to remotely monitor the environmental conditions in order to take actions according to them. Of course, there are many different types of environments, which have different characteristics that define the applications that monitor them. Some environments are open to people and can be monitored easily by any type of device, while others are inaccessible, making human intervention hard. Environmental monitoring applications try to detect status changes and create a virtual representation of the environment.

Target application

The system developed in this work must monitor an inaccessible environment, with high temperatures and corrosive, highly-toxical gasses that can harm human health and nodes. Because of those conditions the nodes must be deployed quickly and operate for a long time without human intervention. To solve this problem, the system must target to extend the node's lifetime, by minimizing their power consumption, but must also perform well under difficult conditions, such as metallic structures interfering with the node's signal or dynamic topology changes.

1.2 Thesis outline

This thesis is written as a tutorial for the reader to understand its requirements, its purpose and finally its design and implementation.

Chapter 2 - Background briefly discusses the most important background concepts, whose understanding is prerequisite to follow the key concepts of this thesis.

Chapter 3 - System Overview and User Model is an introduction to the purpose of the system and its main components. It provides a non-technical overview of the system and focuses on the user perspective.

Chapter 4 - System Design is an overview of the design of the system and its components. It provides an abstract description about the most important protocols that the system uses to provide the required features.

Chapter 5 - System Implementation provides information about the architecture of the system and the toolset (hardware/software) that was used to build the firmware for the system's components.

In **Chapter 6 - Experiments**, are referenced the experiments that were done to measure the performance and power efficiency of the system.

Finally, **Chapter 7 - Conclusion** summarizes the results of this thesis and discusses the future work that is needed to improve its usability.

Chapter 2

Background

2.1 IEEE 802.15.4

The IEEE 802.15.4 [1] is a network standard for operation over low-rate wireless personal area networks (LR-WPANs) and is maintained by the IEEE 802.15 working group, which deals with all the WPAN standards. It is the basis for many popular standards, such as Zigbee, 6LowPAN, Thread and 802.15.5 (Mesh) and can be an alternative to 802.11 (WIFI), which offers more bandwidth but also requires more power.

Its purpose is to define the lower network layers of WPANs that focus on low-cost and low-speed communication between embedded devices. The target of the devices that use such a standard is to optimize their communication to achieve the lowest possible power consumption. Additionally, its simplicity makes it very easy to implement and with small memory footprint, optimal for most embedded microcontrollers. Its features include real-time suitability, collision avoidance, integrated security functions and power management functions (link quality, energy detection).

The protocol architecture described by the 802.15.4 standard is based on the OSI model but only its lower layers (Physical, Data Link) are defined. The physical layer defines the electrical and physical specification for the raw data exchange, while the data link layer provides a mechanism to establish and terminate a connection between two devices, with the purpose to exchange a complete data frame.

The network model that is proposed by the standard is quite simple. All the devices inside such a network can either be a full function device (FFD) or a reduced function device (RFD). The FFDs are capable of communicating with any device, can relay other device's frames and can act as a coordinator of the PAN, while the RFDs can only communicate with FFDs.

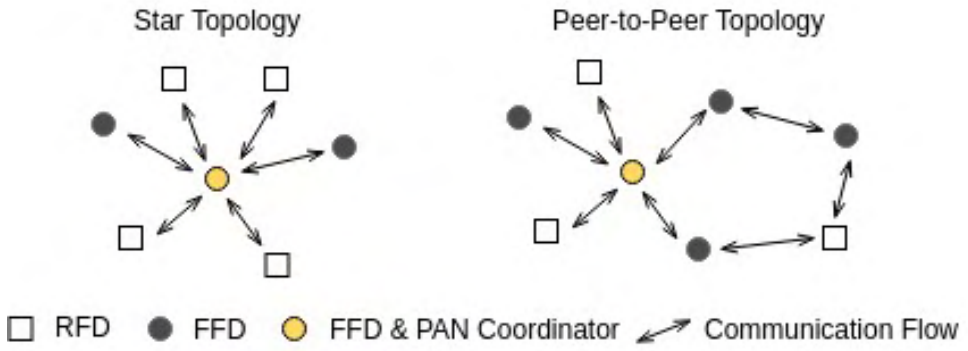

FIGURE 2.1: 802.15.4 topologies.

With this model, the devices can build either a star network or a peer-to-peer (mesh) network. A star network consists of a PAN coordinator and devices (both FFDs and RFDs) connected directly with it, where all devices can only communicate with the PAN coordinator. Same as the star, a peer-to-peer network consists of a PAN coordinator and any other device (both FFDs and RFDs) but with one main difference, the devices can communicate with each other as long as they are in range of one another, thus enabling multiple hops communication to route messages between any device inside the network.

The MAC frames are passed from/to the physical layer as a PHY payload and can be used to transfer data between devices. Below is the structure of the MAC and PHY frames:

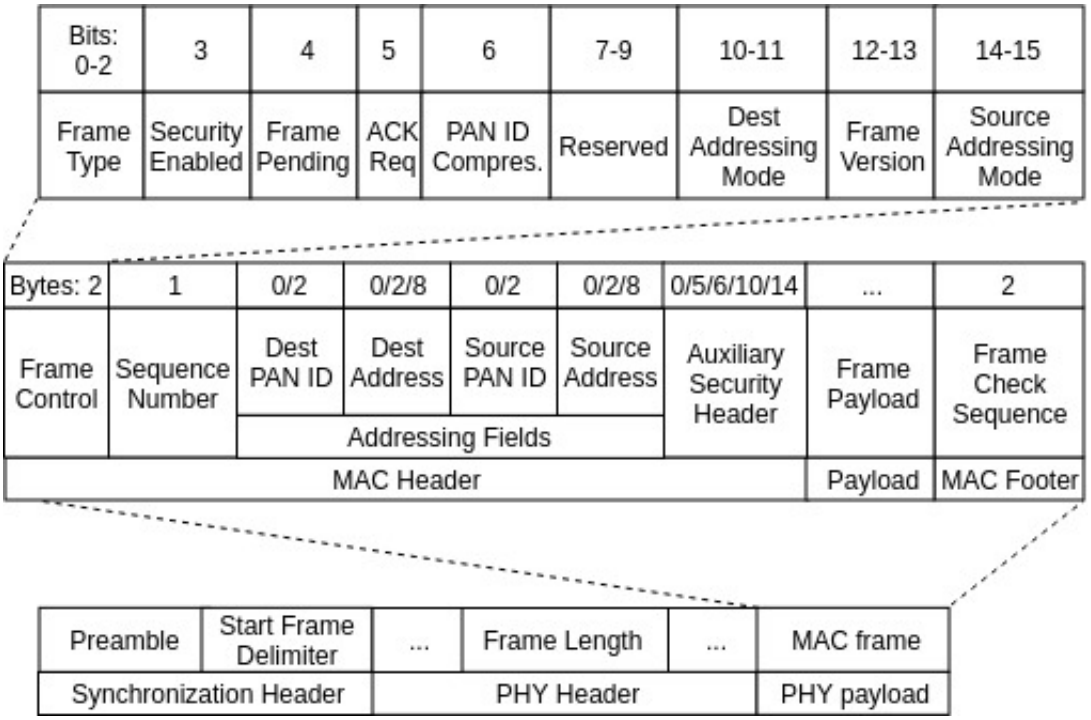

FIGURE 2.2: 802.15.4 frames.

The physical frame fields (the field sizes depend on implementation):

- The **Preamble** field is an altering bit pattern (e.g. 010101...) and is used to determine the amplifier gain.
- The **Start Frame Delimiter** field indicates the end of the synchronization header and the start of the physical header.
- The **PHY header** depends on the implementation and is used by the receiver in order to decode the packet. Its frame length field defines the physical payload size.

The MAC frame fields:

- The 802.15.4 standard defines four MAC frame types, the **Beacon** frame, used to transmit small packets mainly for synchronization, the **Data** frame, used for data transfer, the **Acknowledgment** frame, used for frame reception confirmation and the **Command** frame, used to transmit various MAC commands.
- The **Security Enabled** field defines whether security protection layer is enabled and indicates the existence of the Auxiliary Security Header field.
- The **Frame Pending** field is used to notify the recipient that more data are following, usually used by beacons or beacon-like data frames.
- The **Acknowledgement Request** field indicates that the sender needs an acknowledgment from the receiver.
- The **PAN ID Compression** field specifies whether the MAC frame Addressing field uses one field for both source and destination PAN ID and is used when the source and the destination is the same.
- The **Destination Addressing Mode** and the **Source Addressing Mode** fields specify how the addressing fields are used. The value 0x00 means that the PAN identifier and the address fields are not provided, the value 0x10 means that address fields contain short address (16bit) and finally the value 0x11 means that the address fields contain extended address (64bit).
- The **Frame Version** field specifies the version number of the frame. The value 0x00 means that the frame is IEEE Std 802.15.4-2003 compatible, while the value 0x01 indicates that it is an IEEE 802.15.4 frame.
- The **Sequence Number** fields specifies the sequence identifier for the frame.
- The **Addressing** fields are used to identify the source and the target of the frame.
- The **Auxiliary Security Header** field specifies information required for security processing.

- The **Frame Check Sequence** field is a 16-bit CRC used to check if the frame is valid.

2.2 Contiki OS

The Contiki OS [4, 5] is an open source operating system for low-cost embedded devices. It is a lightweight operating system with an event-driven kernel and a very flexible layered network stack. The Contiki OS is implemented using the C language and is ported in numerous microcontroller architectures. Because it is one of the earliest embedded operating systems, it is mature and well-designed. The applications built using Contiki OS are very small (30KB - 100KB firmware size and 4KB - 8KB RAM) and can fit in most embedded devices. The Contiki OS will be used as the backbone for all the embedded device firmware of this system.

2.2.1 Programming Model

Contiki provides the **Process** library, an event-driven programming abstraction layer that simplifies the procedure of handling events using C language. A process is a piece of code that is executed by the Contiki Runtime Environment.

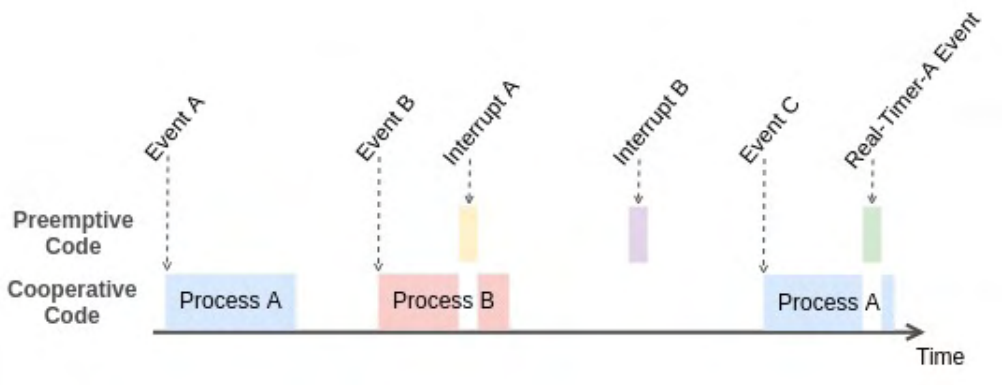

FIGURE 2.3: Contiki Runtime Environment.

The Contiki Runtime Environment offers two execution contexts, the cooperative and the preemptive. When a cooperative code is executed, its execution can be interrupted only by a preemptive code. This means that the cooperative code is executed sequentially, while the preemptive code is executed when it's needed. Processes are always executed in cooperative context, while Real Timer Callbacks and Interrupt Handlers are executed preemptively.

Processes can be treated as real processes and the programs that use them can have multi-process characteristics but without the overhead of synchronization. This programming model helps to build simple applications to manage the hardware and the communication events.

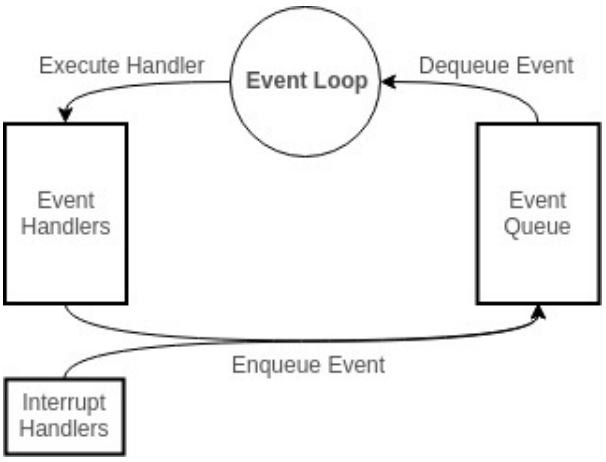

FIGURE 2.4: Contiki Event Loop.

All the events are collected in an event queue and all the processes that wait for an event are registered to an event map. An event loop looks into the queue and for each event notifies the corresponding processes of the map. This type of processing is excellent for an embedded MCU with low power mode because at the end of each loop and until a new event occurs, the MCU can enter this mode to achieve better power efficiency. Contiki can take advantage of the CPU's power saving modes by commanding the CPU to switch into one of these modes when the event queue is empty. For example, when the radio is sending data, the CPU enters sleep mode and will resume after the transmission is complete. Additionally, for even lower power consumption, the CPU can command the peripherals to enter or exit their low power mode or even shut down completely, based on occurring events.

2.2.2 Network Stack

The Contiki provides a layered network stack with the structure shown below:

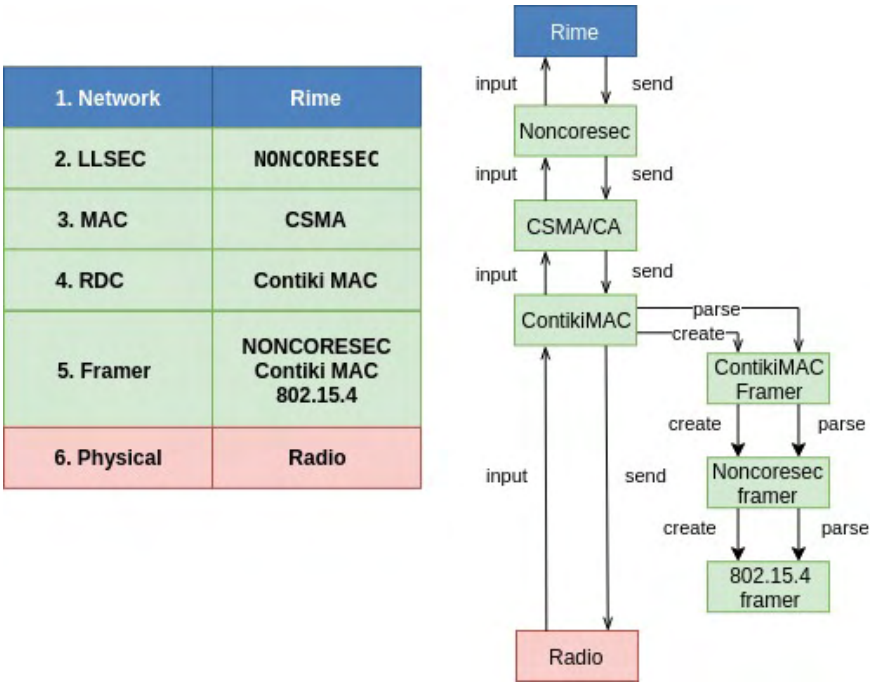

FIGURE 2.5: Network stack.

1. The **Network** layer is a set of tools that help the application manage the entire network communication. The system's active **Network** layer is the Rime [8].

2. The **LLSEC** layer is responsible for the authentication and encryption of the communication. The system's active **LLSEC** layer is the noncompromise-resilient 802.15.4 security.

3. The **MAC** layer is responsible for detecting and fixing problems regarding the exchange of packets. Usually it collects statistics about the packets sent and received by the neighbors. The system's active **MAC** layer is the CSMA.

4. The **RDC** layer is responsible for the radio duty cycling of the radio and decides when the radio is turned on and off. Its main purpose is to optimize the power efficiency of the radio. The system's active **RDC** layer is the Contiki-MAC [6].

5. The **Framer** layer is responsible for constructing and parsing the exchanged packets. The system currently uses multiple framers in a row, starting from the Contiki-MAC framer, which uses internally the noncoresec framer, which finally uses the 802.15.4 framer.

6. The **Physical** layer is the custom API provided by the radio module, in order to send and receive packets and extract helper information (e.g. RSSI). All the supported MCU ports provide a physical layer.

For each layer, Contiki provides an abstract interface, which indicates the functionality that must be provided by an implementation of the layer and multiple ready to use implementations.

Framer Layer - 802.15.4 Framer

This framer is the standard 802.15.4 framer implementation, which is prerequisite for communication using the 802.15.4 standard. Its responsibility is to parse and create 802.15.4 MAC frame header fields.

RDC Layer - Contiki-MAC

The **Contiki-MAC** layer is the power manager of the radio transceiver. It provides a power efficient wake-up mechanism which uses time constrained procedures that keep the transceiver off, for most of the time. This mechanism is based on the **low-power listening** technique, which says that the receivers must periodically turn on their radios and keep it on if they detect activity, while the senders must send some small packets before the actual transmission, with the intent to notify the receivers that an actual packet is following. Additionally the Contiki-MAC offers a framer which can create and parse the Contiki-MAC header fields, needed by the RDC layer. The Contiki-MAC framer internally can use another framer.

MAC Layer - CSMA

The **CSMA** (Carrier Sense Multiple Access) layer implements the addressing, sequence number and retransmission management system. It keeps a list of packets sent by the device and received by its neighbors and for each of them keeps track of the collisions and the retransmissions and tries to fix any problem regarding the packet exchange procedure.

LLSEC Layer - NONCORESEC

The **NONCORESEC** is an noncompromise-resilient 802.15.4 security implementation, which uses a network-wide key. It consists of a framer and a LLSEC layerthat both must be activated at the same time, in order to communicate without problems. The framer is responsible for the encryption and the decryption of the packet sent through the radio, while the LLSEC layer is responsible for updating the required security header fields. The security framer internally can use another framer. This security layer offers various levels of security, each one defining the length of the Message Integrity Code (MIC) and whether the communication is encrypted or not.

Network Layer - Rime

The **Rime** layer is a set of protocols/connections that help with the communication over 802.15.4. The protocols in Rime stack are arranged in a layered fashion and some more complex protocols are implemented over others. The Rime stack contains some basic protocols for single-hop or multi-hop, unicast or broadcast, reliable or unreliable, single-packet or multi-packet (bulk) communication. In addition to the basic protocols, the Rime has a neighbor-discovery protocol, a tree-based hop-by-hop reliable data collection protocol and various other libraries to help the development of even more complex algorithms.

The Rime protocols can be seen as connections with specific parameters each. One of the most important parameter, shared by all protocols, is the channel or channels that they will use to operate. The channel is similar to the Port Number of the TCP/IP stack and must be unique for each connection to work. For example, lets say that two devices opened an anonymous broadcast connection at channel 5 and a third device opened the same connection at channel 6, the first two devices will be able to exchange packets but the third one won't.

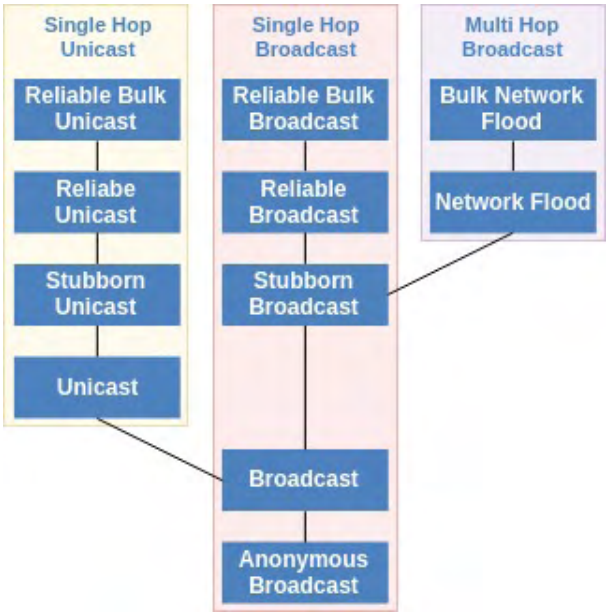

FIGURE 2.6: Rime basic protocols.

Here is a list of the connections that the system is currently using:

- Neighbor discovery mechanisms.
- Best-effort single hop broadcast.
- Single hop unicast.
- Tree-based hop-by-hop reliable data collection.

- Reliable single-source multi-hop flooding (trickle).

There are two available **Neighbor Discovery Mechanisms**, currently supported by Rime. The first one, called **neighbor discovery**, periodically broadcast packets to its neighbors and listens to their responses, while the second one utilizes Rime's **Announcement Layer** [7] to exchange packets with the neighborhood. The **Announcement Layer** is a transparent beacon exchange mechanism. Its main difference with the other packet exchange mechanisms is that the beacons, most of the time, are piggybacked alongside other packets that are sent through the radio and this makes it very cost-effective.

The **best-effort single hop broadcast** sends an identified packet to all the neighbors in its radio radius.

The **single hop unicast** sends an identified packet to a specific neighbor.

The **tree-based hop-by-hop reliable data collection** (or collect), as the name implies, is a protocol that helps a mesh network send data to a specific device. The device that will collect all the packets declares it when it opens the connection. The collect connection uses a unicast connection and one of the neighbor discovery mechanisms to send messages to the neighbors. It's called tree-based because the root of the tree is the target of all packets and all the devices select one parent to create a branch (or route) from the leaves (the devices without children) to the root. The condition that indicates whether a neighbor is appropriate to be selected as a parent is its **depth** (or the distance from the root). By default the root starts with zero depth, while all the other devices start with maximum depth. When a device selects another device as a parent, it computes its depth from the root, which is the parent's depth plus the link estimation with the parent.

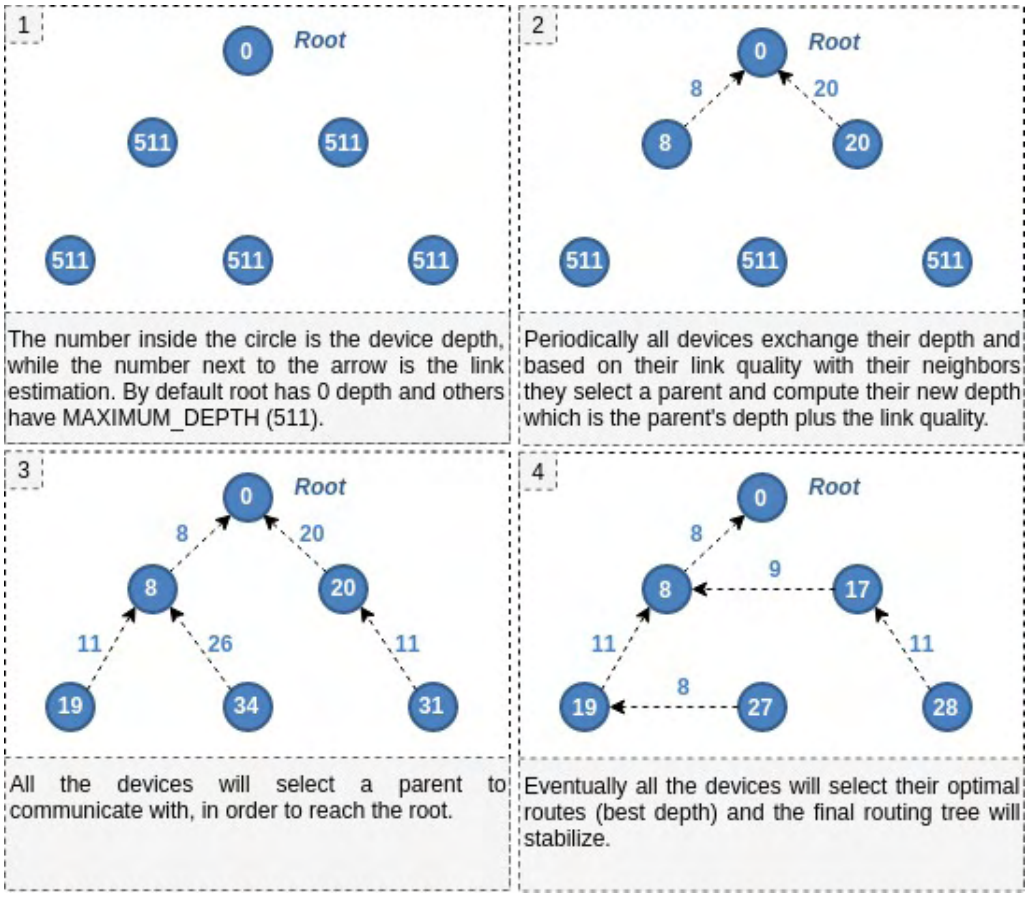

FIGURE 2.7: Collect Connection Routing Tree.

The **reliable single-source multi-hop flooding** (or trickle) reliably sends an identified packet to all the end-nodes inside the network. Trickle's basic idea is that the sender maintains a counter for the sent packets, which becomes a packet field. All devices periodically send their last packet, only if they didn't receive this packet from their neighbors. The devices that didn't receive a packet yet, instead send an empty packet with number zero. If a device receives an old or a new packet, an update is needed. The up-to-date devices send their last packet as an update to the out-of-date ones. This mechanism is often referenced as polite gossip. Trickle can reassure that only the last packet will be propagated to all the devices that are available to receive it. If the sender needs to send multiple subsequent packets, it must add an appropriate delay between them, to reassure that they are propagated correctly; but even then, if a device didn't receive a packet before the next one is sent, it might never receive it.

Chapter 3

System Overview and User Model

3.1 Overview

The purpose of the system is to measure the environmental conditions using various types of sensors. The system will operate inside an inaccessible environment, with gases and heat that are harmful to the system's components and human health. That's why the system must be flexible enough to automatically cure itself and continue to operate with minimum human intervention. If self-cure is impossible, the system provides manual repair processes aiming to minimize the time inside this harsh environment.

System Components

The system consists of four components, the operator, the coordinator, the authenticator and the end-nodes. Each component has a distinct role in the system, and all together create a WSN.

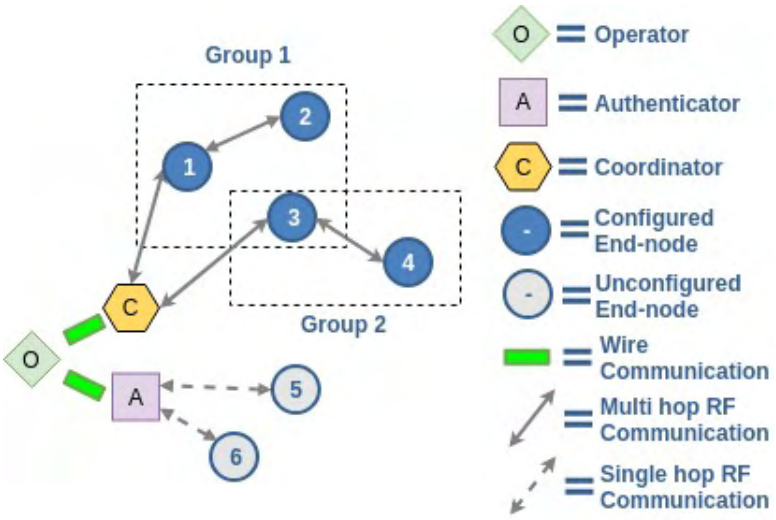

FIGURE 3.1: System components.

The operator's role is to be the manager of the WSN. It can send commands to configure the network or retrieve its state and can receive measurements coming from various sensors.

The authenticator is responsible for detecting unauthenticated and un-configured end-nodes and reporting them to the application operator. The operator can instruct the authenticator to allow an end-node to participate to the private network.

The coordinator is the bridge that connects the operator with the end-nodes. It listens to the operator's commands, in order to configure the network and forwards all the incoming measurements, coming from the end-nodes, to the operator.

The end-nodes have three main responsibilities, firstly to collect measurements from their sensors to send them to the coordinator, secondly to receive commands from the coordinator and send back the responses, and thirdly to help other end-nodes communicate with the coordinator. The end-nodes can be turned on to start their operation and can be turned off to stop their operation and save their battery for later use.

All the end-nodes start as unconfigured and try to communicate with the authenticator (note that the un-configured end-nodes must be located close to the authenticator in order to communicate with it). If the user has selected a configuration about an end-node, the authenticator will update the end-node and it will enable it to participate to a network group. The network group is a set of end-nodes that share common configuration options and can communicate with each other and the coordinator. For example (fig 3.1) the **end-node 1** and the **end-node 2** can communicate with each other but they can't communicate with the **end-node 4**. Some end-nodes can be configured to participate to a specific network group and at the same time be enabled to support other groups, e.g. **end-node 3** participates to network group 2 but also can help the end-nodes of the group 1 to communicate with the coordinator.

Components Interconnection

The operator connects with the coordinator and the authenticator via separate USB cables and exchanges messages with them via a serial protocol.

The authenticator communicates with the end-nodes using an RF module. To ensure that the end-nodes can communicate with the authenticator, the end-nodes must be placed close to it. End-nodes can only communicate with the authenticator directly.

The coordinator also communicates with the end-nodes using an RF module. At their final locations, some end-nodes might not have a direct link with the coordinator but reach it using other end-nodes as intermediates. That's why end-nodes are also responsible for receiving and forwarding messages coming from other nodes, in order to help them reach the coordinator. In other words, the coordinator and the end-nodes create a mesh topology.

System Standard Operation

The operator is connected with the coordinator and the authenticator, and the end-nodes are powered on and placed to the desired location. The user configures the network based on the desired operation, and all the end-nodes sense their environment and send their data to the operator through the coordinator. The user can reconfigure the network even if the end-nodes are already deployed, but the reconfiguration might be delayed until the network is ready.

Steps to Reach Standard Operation

The user must first connect the operator with the coordinator and the authenticator. Then the end-nodes must be placed close to the authenticator and the user must select and configure them with the desired options. The final step is to place the end-nodes to the desired locations and, based on the system feedback, reallocate them until the system reports that all the end-nodes can communicate without problems.

3.2 User Model

3.2.1 Configuration Options

Based on the user requirements, the system provides various configuration options to allow the user select the desired functionality.

Firstly, the user can create end-node groups, choose the type of readings that the end-nodes will take, how often they will take measurements and how often the system will collect those measurements.

Secondly, the user can select which end-node participates in which group and whether an end-node helps other groups to communicate.

Finally the user can enable/disable the sampling, the data collection and the low power mode functionality of the system. The sampling option defines whether the end-nodes take measurements or not, the data collection option defines whether the system collects measurements from the end-nodes and the low power mode defines whether the system tries to save power by commanding the end-nodes to turn off their radios for a while.

3.2.2 Actions and Feedback

The system is designed as a black-box that receives commands and sends back responses and data. Every user action is encoded into a command, which will travel into the system to achieve the desired result. Some of the commands can be directly used through the user interface (e.g. operator's user interface application or end-node's buttons) and others are issued internally. The system provides a network overview where the user can see the link of the WSN and their quality, and occasionally sends notifications to the user when it detects a change in its operation, which needs the user's attention.

Actions	**Description**
Create group	Create a new network group, select the desired communication period, the desired reading types and a sampling period for each type.
Update group	Select a network group and update its configuration.
Remove group	Select a network group and delete it.

Table 3.1: User actions for network groups

Actions	**Description**
Turn-on end-node	Turn on the end-node, by pressing its power button.
Configure end-node	Select an end-node and update its configuration, by choosing a main group and the desired groups to support.
Clear end-node's configuration	Select and clear the end-node's configuration.
Turn-off end-node	Turn off the end-node, by pressing its power button again.

Table 3.2: User actions for end-nodes

Actions	Description
Enable sampling	Command the system to allow the end-nodes, to start sensing the environmental conditions.
Disable sampling	Command the system to stop the end-nodes from sensing the environmental conditions.
Enable data collection	Command the system to start collecting measurements from the network.
Disable data collection	Command the system to stop collecting measurements from the network.
Enable low power mode	Command the system to allow the end-nodes to periodically turn their radio off, to save power.
Disable low power mode	Command the system to stop the end-nodes from turning their radios off.
Firmware update	Triggers the system to update the firmware of the end-nodes.

TABLE 3.3: User actions for system

Notification	Description
Missing End-node	The system keeps track of the end-nodes that communicate and notifies the user if an end-node that is expected to communicate, is missing.
Low Link Quality	The system notifies the user when the link quality of an end-node drops below the tolerable limit.
High Consumption Notification	The system notifies the user when an end-node's consumption exceeds the acceptable consumption limit.

TABLE 3.4: System notifications

3.2.3 Management Processes

The management processes are set of user actions, which the user must do in order to interact with the network and achieve the desired result. This section describes the most important processes that address typical user needs.

Create a network

This process is a series of user actions that will result in an fully operating WSN:

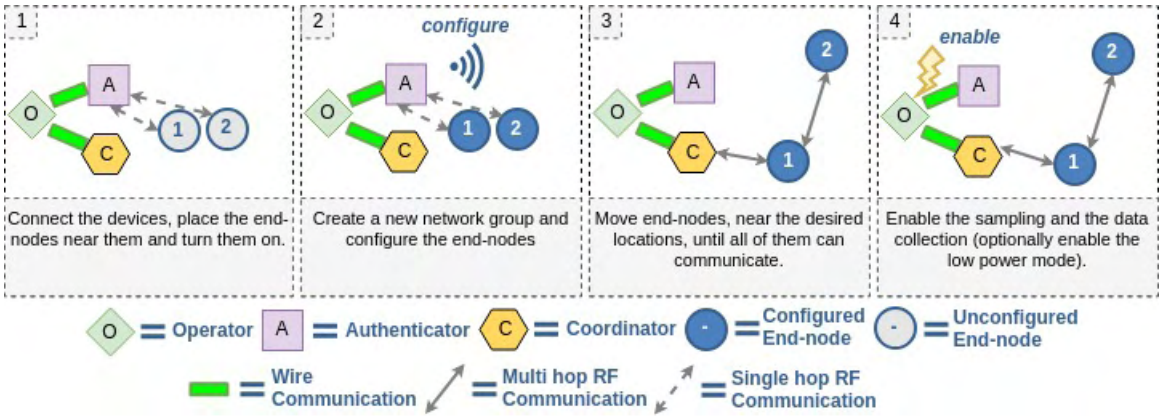

FIGURE 3.2: Create a new network.

1. Connect the **operator** with the **coordinator** and the **authenticator**, place the **end-nodes** near the authenticator and turn them on.

2. Create a new network group and configure the end-nodes.

3. Place the end-nodes to the desired locations and go to the network overview to see if all the end-nodes can communicate without problems. If a node is missing or the system reports that it has problem communicating, move it to a better location and re-do this step.

4. Enable the sampling and the data collection (optionally enable the low power mode, for better power efficiency).

Add a new end-node to a deployed network

This process is a series of user actions, the user must do in order to add a node to a deployed:

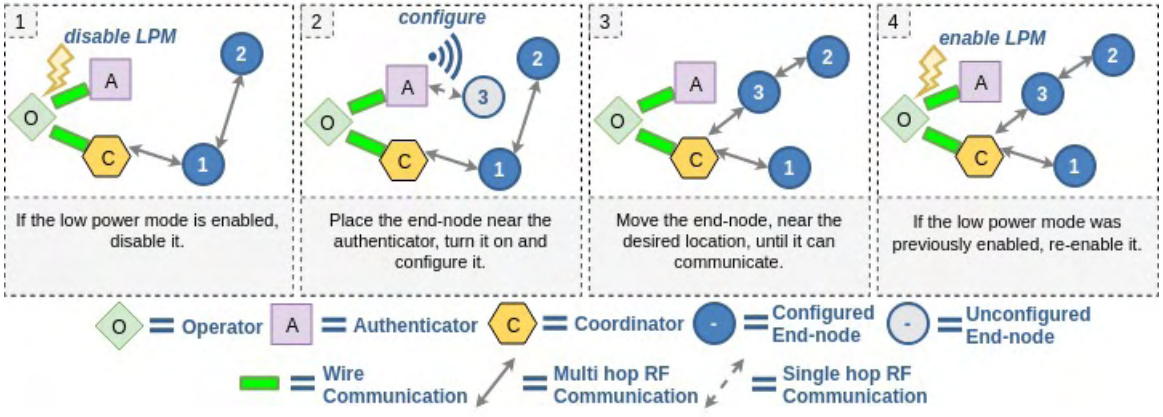

FIGURE 3.3: Add a new node to a network.

1. If the low power mode is enabled, disable it.
2. Place the **end-node** near the authenticator and turn it on and configure it.
3. Place the end-node to the desired location and go to the network overview to see if the end-node can communicate without problems. If a node is missing or the system reports that it has problem communicating, move it to a better location and re-do this step.
4. If the low power mode was previously enabled, re-enable it.

Remove node from a deployed network

This process is a series of user actions, the user must do in order to remove a node from a running network:

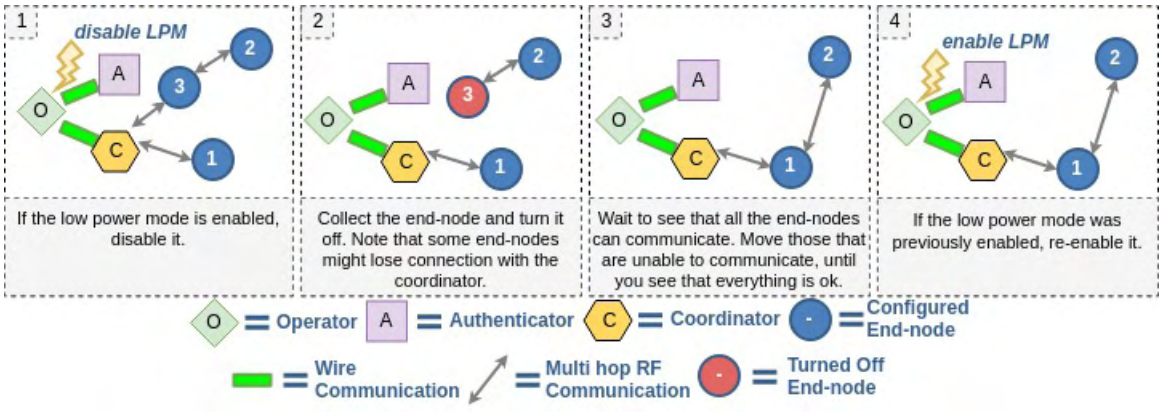

FIGURE 3.4: Remove a end-node from a network.

1. If the low power mode is enabled, disable it.
2. Collect the end-node and turn it off.
3. Go to the network overview to see if all the remaining end-nodes can communicate without problems. If a node is missing or the system reports that it has problem communicating, move it to a better location and re-do this step.
4. If the low power mode was previously enabled, re-enable it.

Destroy a network

This process is a series of user actions, the user must do in order to destroy a network:

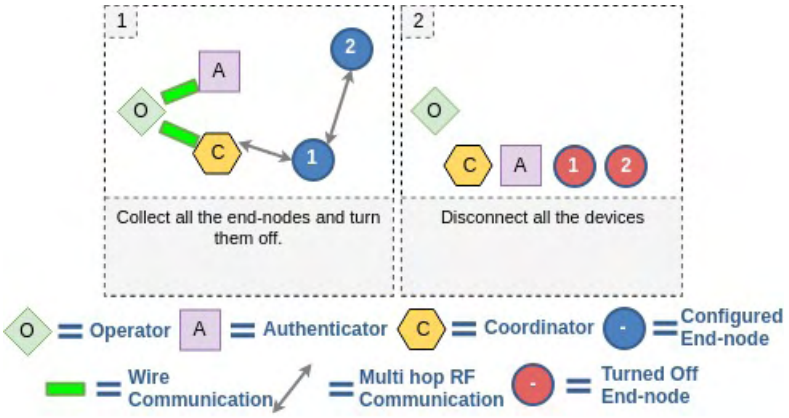

FIGURE 3.5: Destroy a running network.

1. Collect all the end-nodes and turn them off.

2. Disconnect the operator from the coordinator and the network authenticator.

Chapter 4

System Design

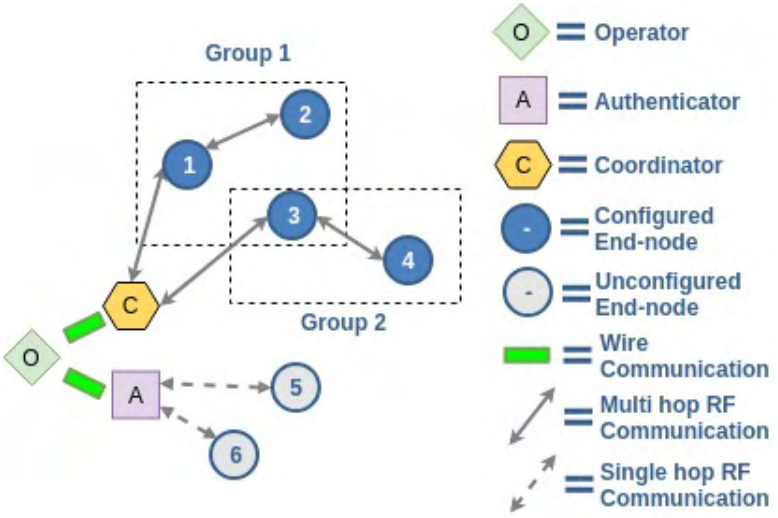

FIGURE 4.1: System overview.

As mentioned before there are four main components in the system, the **application-operator**, the **network-coordinator**, the **network-authenticator** and the **end-nodes**. Each plays a specific role and all together create a fully-operational sensor network. Of course, there are other components that could also be a part of this ecosystem. For simplicity, they left outside this chapter, but there will be a small independent section explaining their role later on (section 7).

4.1 Network Configuration and State

The system maintains a **network configuration** that contains all the user configuration options.

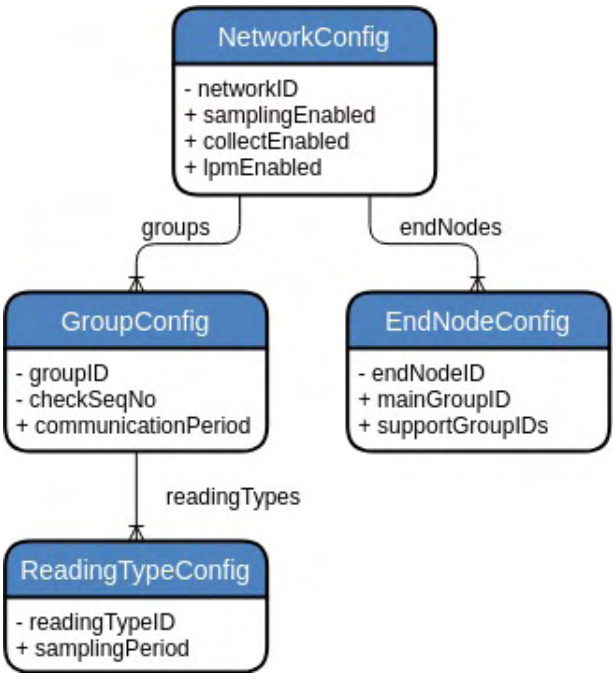

FIGURE 4.2: Network Configuration.

The **NetworkConfig** holds the network credentials (networkID), the functionality flags (samplingEnabled, collectEnabled and lpmEnabled), the configuration options for each network group and the configuration options for each end-node. The network credentials are used to create a private network, where only the devices that have those credentials can communicate. The samplingEnabled flag enables/disables the sampling functionality, which periodically commands the end-nodes to take measurements. The collectEnabled flag enables/disables the data collection functionality, which periodically collects the measurements that are stored at the end-nodes. Finally, the lpmEnabled flag enables/disables the low power mode functionality of the network, which allows the end-nodes to save power.

The **GroupConfig** contains the corresponding group's unique ID, the check sequence number which changes every time a configuration change is done and the communication period which indicates how often will the operator communicate with this group. The operator must communicate with the end-nodes of this group to update their configuration and collect their data.

The **ReadingTypeConfig** the corresponding reading type's unique ID and a sampling period that indicates how often the end-nodes will take measurements.

The **EndNodeConfig** contains the corresponding end-node's unique ID and the group IDs (main, support) used to find the corresponding GroupConfigs to update the end-node's configuration and enable it to participate to those groups.

The check sequence number is 4 bits but the 0xF value is reserved for device reset detection, while the communicationPeriod and the samplingPeriod are 2 bytes and are measured in

tens of seconds (e.g. the value 120 means 1200 seconds). This covers up to 655360 seconds or 7.5 days.

The fields with '-' are not user defined but instead are generated by the system. The IDs are used to create relations between the Network Config and the Network State.

In order to manage the network, the operator maintains a **network state** that contains information about the operation of the network.

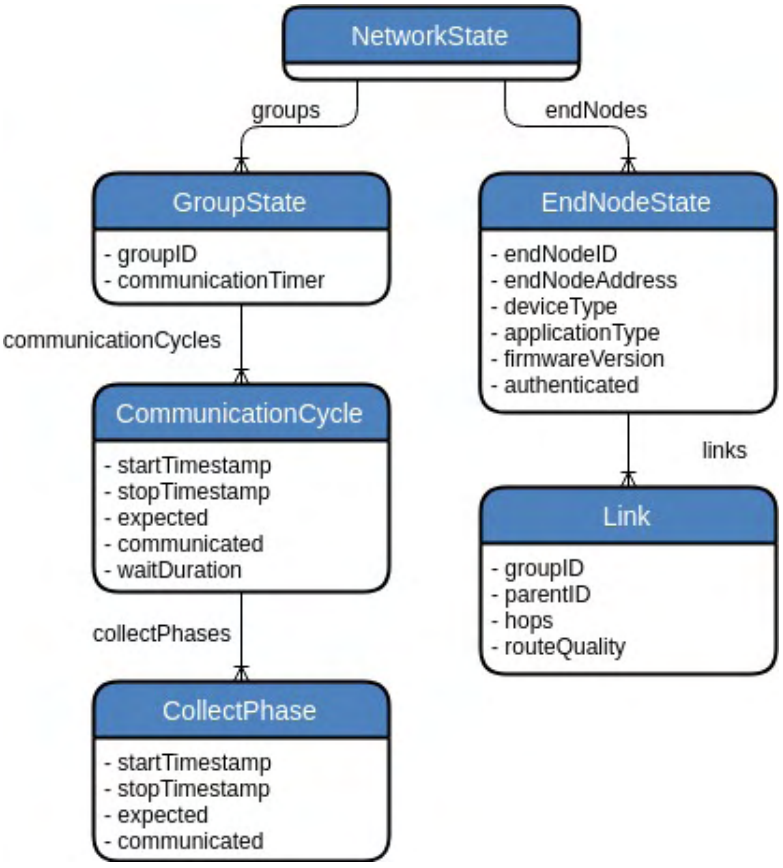

FIGURE 4.3: Network State.

The **NetworkState** holds the state of each end-node and each group that exists at the system.

The **GroupState** contains the corresponding group's unique ID and a timer that is used to keep track of the end-node's current state. The timer's tick means that the all the end-nodes of the group just woke up and are ready for communication. When the operator finishes the communication with the end-nodes, it resets this timer with the current group's communicationPeriod.

The **CommunicationCycle** is a procedure that helps the operator communicate with the end-nodes (more details later on). Each communication cycle has a start timestamp, a stop timestamp, an array of the devices IDs of the nodes that are expected to communicate in

this cycle (expected), an array of device IDs of the nodes that did communicate in this cycle (communicated) and finally the best wait duration of its phases, which can be used at the next wake cycle.

The **CollectPhase** is a procedure that helps the operator collect the data of the end-nodes. Each collect phase maintains a start timestamp, a stop timestamp, an array of the device IDs of the nodes that are expected to communicate at this phase (expected) and an array of the device IDs of the nodes that communicated at this phase (communicated).

The **EndNodeState** contains the device's unique ID (64bit address), the device's unique 16bit Address, the device type, the type of the application it runs, its firmware version and whether its authenticated or not. The ID is used to uniquely identify this device. The Address is the identifier of the device inside the network and every packet that travels from the end-node to them operator will contain this address as a field. The operator uses the end-nodes state to match the incoming packets with the device ID. The deviceType and the applicationType fields are used by the OTA update mechanism to match the firmware with the end-node's hardware. The end-nodes that requested configuration but are not recognized by the user are allowed to register a state but are not configured by the system, until the user decides otherwise. Because the end-nodes participate into Data Collection Trees, one for each group they participate into, they have information about the links with their parent(s).

Each **Link** contains the group ID of the corresponding DCT, the ID of the parent end-node, the hops distance from the coordinator and the quality of route from the end-node to the coordinator.

The network configuration is maintained by the operator and is distributed to the coordinator and the end-nodes.

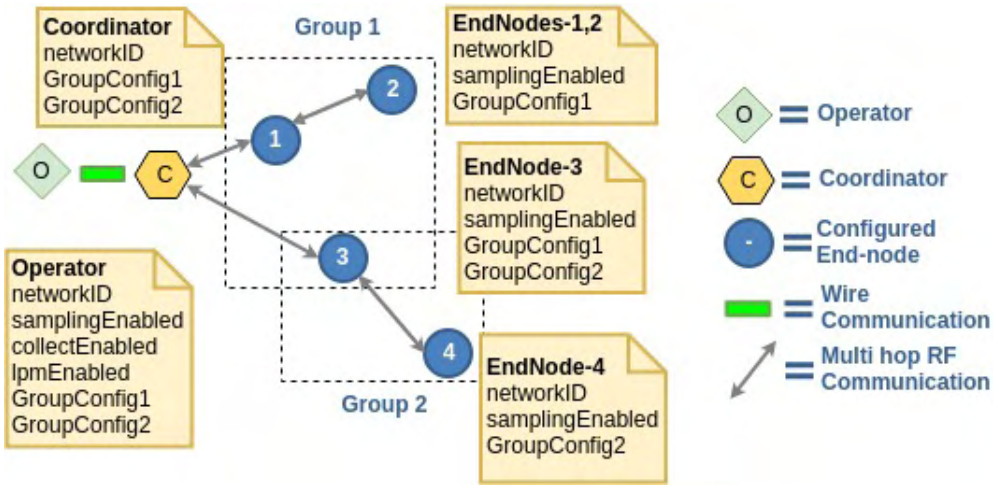

FIGURE 4.4: Configuration Distribution Example.

In figure 4.4, we can see that the operator holds the entire configuration, the coordinator holds the network credentials and the group configurations, and the end-nodes hold the network credentials, the samplingEnabled flag and the group configurations that they participate in. For example, the end-node-1 holds the configuration of the group-1, while the end-node-3 holds the configuration of both groups.

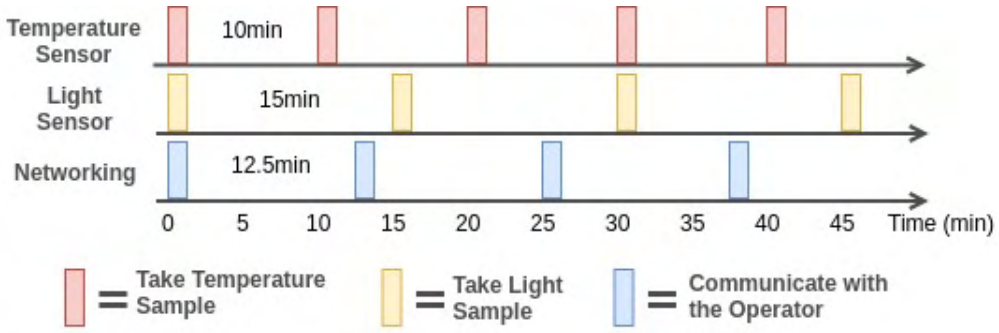

FIGURE 4.5: End-node Life Cycle Example.

Each end-node ends up with a configuration containing information about when it should take samples and each group configuration dictates when the system will enable the communication with the end-nodes. An example configuration is shown in figure 4.5, which shows what happens at the end-node life cycle, based on configuration. The displayed end-node has 10 minutes temperature sampling period, 15 minutes light sampling period and can communicate with the operator every 12.5 minutes. This allows the end-node to asynchronously take samples and store them, until it is able to send them to the operator (more details in section 4.3).

4.2 Communication

4.2.1 Packet Format

All applications share a common generic packet format. The packet format is simple, to be easily implemented but also dynamic enough to help the communication over 802.15.4 radio. Below is the generic packet format, supported by all components of the system:

Type	**Payload**
1 Byte	Max Payload Size - 1 Byte

TABLE 4.1: Packet Format

Each type has specific payload size and payload fields. Based on the communication mechanism (radio protocol or serial protocol), it is wrapped by the needed headers, but those don't affect the payload size.

4.2.2 Radio Communication

The coordinator and the authenticator must be able to communicate with the end-nodes via the radio module. To do that they must be able to exchange packets with them.

Unauthenticated Communication

The authenticator communicates with the unconfigured end-nodes via simple single-hop mechanisms.

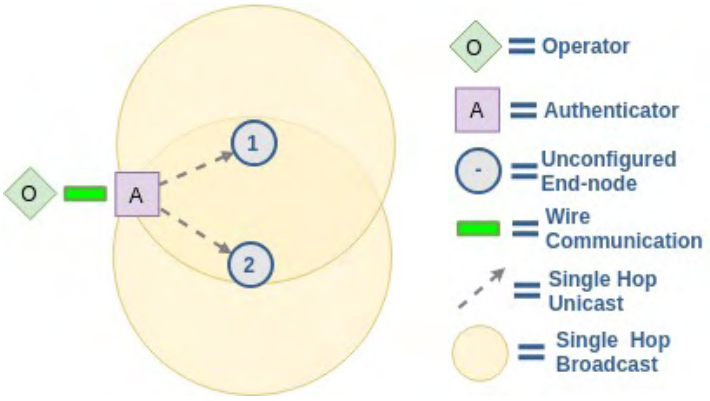

FIGURE 4.6: Unauthenticated Communication.

The end-nodes send unauthenticated configuration request packets to everyone in their radio range (broadcast packets), in hope that the authenticator will be in range to receive their request. On the other hand, the authenticator sends configuration packets directly to a specific end-node (unicast packets).

Group Communication

The coordinator communicates with the configured end-nodes via multi-hop mechanisms.

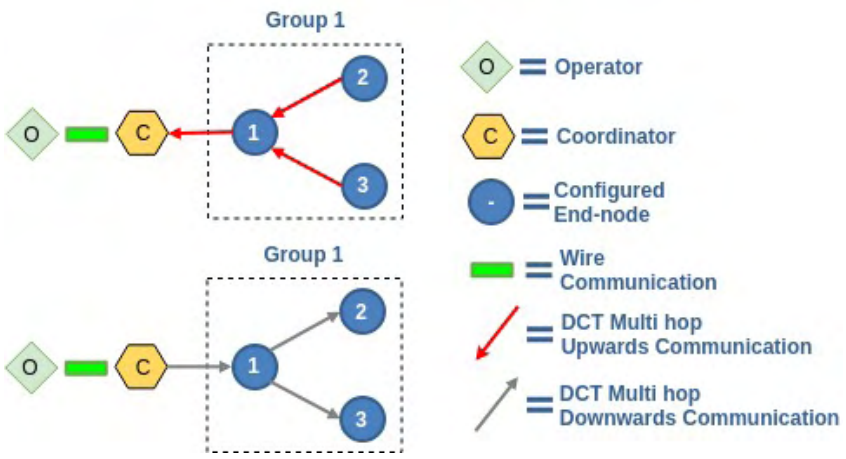

FIGURE 4.7: Group Communication.

The coordinator and the configured end-nodes create data collection trees, one for each group. In all cases, the coordinator is the root of the data collection trees. The data collection tree (DCT) provides two mechanisms for bidirectional communication. The first one is for upwards communication (from the end-nodes to the coordinator) and the second one is for downwards communication (from the coordinator to the end-nodes). The end-nodes use their main group's data collection tree upwards mechanism to send data packets to the coordinator while the coordinator uses the downwards mechanism to send packets to specific groups (and the end-nodes that belong to them). The coordinator sends two types of packets downwards the data collection tree, point-to-multipoint and point-to-point. A point-to-multipoint packet is delivered to all the end-nodes of a specific group, while a point-to-point packet is delivered to a specific end-node.

4.2.3 Serial Protocol Packet Exchange Mechanism

To communicate over a serial protocol, the system provides a custom mechanism to send/receive packets through it. Below is the supported packet format:

Delimiter	**Length**	**Payload**	**Checksum**
1 Byte	2 Bytes	Max Payload Size - 4 Bytes	1 Byte

TABLE 4.2: Serial Protocol Packet Format

Delimiter = '~' (0x7E)

Checksum:

1. Add all data bytes into a number.
2. Keep only the lowest byte.

3. Subtract the result from 255.

To verify a packet, the packet's bytes and checksum are summed up and the lowest byte should equal 255.

From the operator to the coordinator, there are two types of payload, either a command, which is processed directly by the coordinator, or a remote command which is forwarded to the appropriate network group. For the remote commands, in order to find the **target network group**, the last byte must contain its ID. If the **target network group** equals **0xFF**, then the remote command is sent to all groups.

The coordinator enhances all the incoming radio data packets, with the sender's address and forwards them to the operator.

4.2.4 Configuration Updates

Before starting with the configuration of the network, the operator must identify whether a connected device is the authenticator or the coordinator. To do that, the system provides identification packets:

Type	Description	Fields	Direction
Identify Device	Operator requests device identification. Used to distinguish coordinator and authenticator	Empty	Operator ↓ Device
Identify Response	The device informs the operator about its type	Application Type (4 bits)	Operator ↑ Device

TABLE 4.3: Device Identification Packet Types

When the operator connects with a device, it sends a Identify Device packet and receives an Identify Device Response packet. This procedure helps the operator identify the type of the device.

After the operator has detected the authenticator and the coordinator, it uses them to receive requests from the end-nodes and configure them based on the user options. The packets below help the operator to configure the unconfigured end-nodes:

Type	Description	Fields	Direction
Auth Request	The end-node notifies the operator for the need of authentication/configuration	End-Node Address (2 bytes) End-Node ID (8 bytes) Device Type (4 bits) Application Type (4 bits) Version (2 byte)	Operator ↑ End-Node

Auth Response	The operator configures the target end-node	End-Node Address (2 bytes) Network ID (1 byte) Main Group (g-size)[1] Support Groups N (1 byte) Support Groups (N x g-size)[1] Timestamp (4 bytes)	Operator ↓ End-Node

TABLE 4.4: Unauthenticated Packet Types

[1] Group size is dynamic and based on the enabled reading types (details at section 4.1).

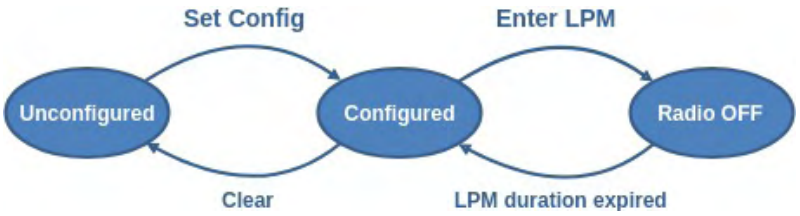

FIGURE 4.8: End-node State.

The end-node, on boot, is unconfigured and periodically sends a Auth Request packet to the authenticator to request a configuration. When the authenticator receives such a packet, it forwards it to the operator, which decides whether the end-node must be configured with a Auth Response packet. After this, the configured end-node stops sending such packets. If the user turns off the end-node, it clears its configuration.

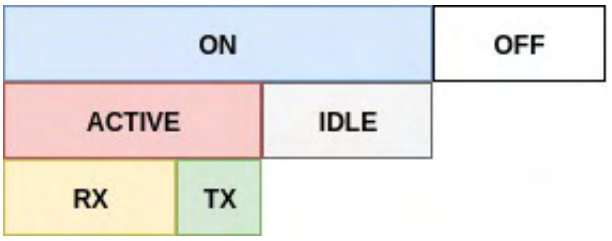

FIGURE 4.9: Radio State.

The CPU can be at LPM or not, but the radio is a bit more complex, because it can be ON and OFF and when it is ON it can either be active or idle. The radio active time is the sum of the time the radio is transmitting and the time the radio is listening. The idle time, is the time that the radio is on and its idle listening.

In some cases, the user might change the configuration of an already deployed network, and the operator must update the coordinator and the end-nodes that are affected. The packets below help the operator to update the network's configuration:

Type	Description	Fields	Direction
Coordinator Update	Operator updates the configuration of the coordinator. Used mainly on boot	N Groups (N x g-size)[1]	Operator ↓ Coordinator
Group Update	The operator updates the configuration for a target group	Group (g-size)[1]	Operator ↓ Group
Config Check	The application operator sends to a network group the configuration check sequence number	Group ID (1 byte) Group CheckSeqNo (4 bits) Sampling Enabled (1 bit)	Operator ↓ Group
Config Report	The end-node reports to the operator its current configuration	End-Node Address (2 bytes) Group ID (1 byte) Group CheckSeqNo (4 bits)	Operator ↑ End-Node
Enter LPM	The application operator notifies a network group that it can enter low power mode for some time	Group ID (1 byte) Timestamp (4 bytes)	Operator ↓ Group

TABLE 4.5: Configuration Update Packet Types

[1] Group size is dynamic and based on the enabled reading types (details at section 4.1).

4.3 Data Flow

One of the most important user requirements is to have minimum data loss. That's why the system focuses on optimizing the data flow, from the end-node to the application operator. The steps below describe the procedure that the system follows in order to take a measurement and make it available to the user:

FIGURE 4.10: Data Flow.

1. The end-node must first acquire a sampling period for a specific reading type in order to periodically enable its sensing functionality.

2. On the right moment (based on sampling period), the end-node powers on its sensor and takes a measurement.

3. After a new measurement is taken, a measurement packet is created containing the **Sensor ID**, the **Sensor Value** (based on sensor it might contain multiple values) and a **Timestamp**.

4. The measurement packets are stored to a persistent storage.

5. On the right time (based on communication period), will start the Communication Cycle, in order to receive the aforementioned measurement packet. The end-node removes the sent measurement packets from the persistent storage.

6. When collected, a measurement is stored at the operator and is available to the user.

4.3.1 Sensing

The user is interested in various reading types but he might be interested in different sampling periods per type (e.g. he needs to see Temperature readings at a 15 minute time interval and Light readings at 1 minute time interval). To provide this feature, the system offers the option to select a sampling period per reading type.

Reading Type	**Sensor ID**
Temperature	TMP_007
Light	OPT_3001

TABLE 4.6: Reading Type and Sensor IDs Matching Example

For each reading type, different kinds of sensors can be used. Some sensors might produce a ready-to-use reading (e.g. a temperature sensor that produces the value 30, which is actually 30 degrees Celsius) and some others might produce raw values that need additional processing to transform into a reading. To solve this issue, the system maintains a match of reading types and sensor IDs that firstly allows the end-node to understand which configuration is for what sensor and secondly the operator to transform the raw sensor values into readings with type. That's why the end-nodes instead of sending the reading type they must send the Sensor ID (see Data packet).

When the end-node needs to take a measurement, it powers on the appropriate sensor, waits until its warmed-up (if necessary), collects the measurement and powers it off again. This procedure is followed every time the end-node needs a specific reading. The power on/off steps are necessary to maintain the power consumption low.

4.3.2 Storage

As previously mentioned, the end-node doesn't send a measurement immediately after it was taken but instead waits for the appropriate time to do it. This asynchronous data collection creates data loss risk, which might occur if the device powers off before sending it. That's why the system provides a persistent **Storage** library, which allows the end-node to store measurements until it is time to send them. This storage can also be used to store the maintenance's state.

4.3.3 Time Library

Most embedded devices don't have built-in support for time management and time synchronization, and also they don't have battery-powered real time clocks (RTC), to retain offline time. The time library is used by the end-nodes, and is simple and easy to implement using any clock technology. It basically keeps track of time using a 4 byte timestamp (the distance in seconds from Unix Epoch) as a reference, and uses a clock to count seconds from it. This way it doesn't affect the clock, and allows better overflow behavior.

In order for the end-nodes to operate and provide measurements with timestamps, they must receive a Unix Epoch timestamp and use it to configure their time library.

Type	**Description**	**Fields**	**Direction**
Get Time	The end-node notifies the operator that it needs to synchronize its time library	End-Node Address (2 bytes)	Operator ↑ End-Node
Set Time	The application operator sends a new timestamp to a network group	Group ID (1 byte) Timestamp (4 bytes)	Operator ↓ End-Node

TABLE 4.7: Time Library Packet Types

There are two ways for this to happen:

- The operator to must send them a Auth Response packet, containing a timestamp.

- The end-node to must send a Get Time packet to the operator, which in return replies back with a Set Time packet.

Those procedures can be triggered again to solve possible time drift issues that might occur at the clock operation.

Note that the system does nothing to solve the possible time error that might be created by the synchronization procedure's propagation delay. There are various algorithms that try to solve this issue, but the solutions will be part of future work (more details section 7).

4.3.4 Data collection with the Communication Cycle

In order for the operator to exchange packets with the end-nodes, it follows a communication procedure. This procedure is called Communication Cycle and is executed, for each group, periodically every **communicationPeriod** time.

Type	Description	Fields	Direction
Link	The end-node sends to the operator link information	End-Node Address (2 bytes) Parent Address (2 bytes) Route Quality (2 bytes) Hops (1 byte)	Operator ↑ End-Node
Get Data	The application operator notifies end-nodes that they must send their data	Group ID (1 byte)	Operator ↓ Group
Get Lost Data	The application operator notifies end-nodes that they must re-send their previous data packet	Group ID (1 byte) N Addresses (1 byte) Addresses (N x 2 bytes)	Operator ↓ Group
Data	The end-node sends to the operator measurements and storage information	End-Node Address (2 bytes) N measurements (1 byte) Measurements (N x m-size)[1] HasMore Flag (1 bit)	Operator ↑ End-Node

TABLE 4.8: Data Packet Types

[1] Measurement size is dynamic and based on the enabled sensor ID.

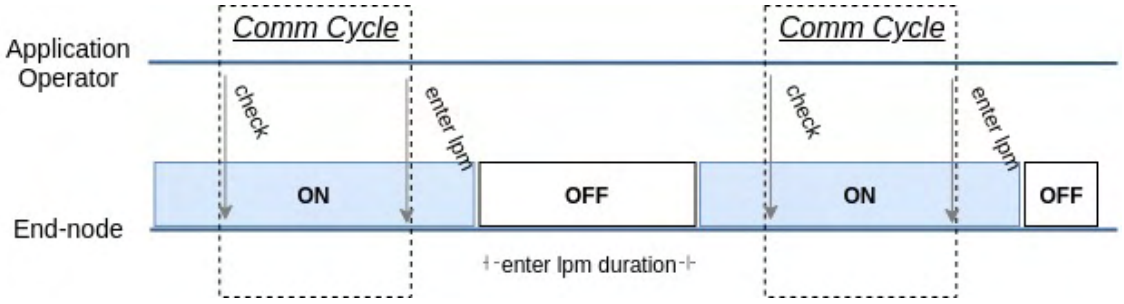

FIGURE 4.11: Radio ON/OFF Cycle.

The end-nodes always keep their radio ON until the operator commands them to enter low power mode (LPM) for a specified duration. The operator ensures that before the communication cycle starts, all the end-nodes have their radios ON and ready to communicate, by providing an appropriate LPM duration. The LPM duration is always one minute plus the worst of the end-node's latency less than communicationPeriod.

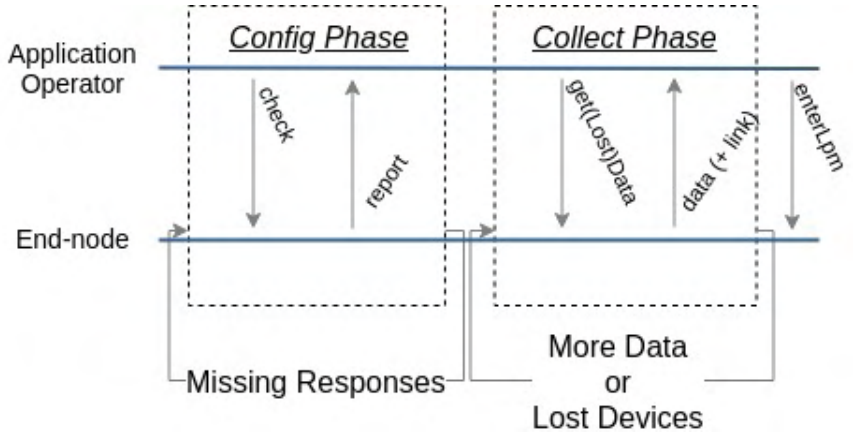

FIGURE 4.12: Communication Cycle.

The **Config** phase of the procedure starts with a Config Check packet sent by the operator. The end-node responds back with a Config Report packet, which helps with the detection of an out-dated end-node. If an end-node is unable to communicate with the operator, it might lose some of the latest updates of its configuration, which will be showw at its current config check sequence number. If the operator received a config report, containing an outdated config check sequence number will send a Group Update packet. Note that in some cases the operator might already know that some end-nodes need an update, even if they didn't report it yet and will send them configuration packets, before the start of this phase. Finally, if the operator detects that an end-node didn't respond, it can restart this phase.

The following phase is the **Collect** phase, where the operator can collect the measurements stored at the end-nodes. It starts with the operator requesting the end-node's data with a GetData packet. The end-nodes will respond with a Data packet that contains some stored measurements and a flag that indicates whether it has more data stored at its storage. If the operator detects that some end-nodes didn't respond, it marks them as lost and restarts the phase but this time it will request data from only the lost end-nodes with a GetLostData. If at least one end-node indicates that it has more data, then the phase restarts. This phase stops when all the end-nodes emptied their storage. Note that the next GetData packet acts as an acknowledgement for the previously sent measurements.

In addition to the data packets, if the end-node detects a change into its route, it also sends a Link packet.

If the Low Power Mode (LPM) option is enabled, the procedure finishes with an Enter LPM packet, which reports to the end-nodes when it is required to wake-up (timestamp).

4.4 Maintenance

The end-node periodically collects information about its operation and its battery, which is sent, alongside the sensor measurements, to the operator. The end-node monitors its operation and counts the time spent in various states (CPU,RADIO).

Operation Statistics:

- Crash Counter (1 byte): the number of crashes occurred from the initial programming.
- CPU up-time (4 bytes): the total time, in seconds that the device is ON.
- CPU LPM perc (1 byte): the percentage of total time, the CPU spent in its low power mode.
- Radio tx-time (4 bytes): the total time, in seconds that the radio was transmitting.
- Radio rx-time (4 bytes): the total time, in seconds that the radio was listening for packets.
- Radio idle-time (4 bytes): the total time, in seconds that the radio was idle listening.

The statistics help compute the consumption of the end-node, evaluate the utilization of the radio and detect possible problems. The radio duty cycle (RDC) is the percentage of time that the radio is on and is equal to the Radio ON time (rx+tx+idle) divided by the uptime. The consumption can be computed by multiplying the time with the average consumption of each state. For example, let's say that the uptime is 30days, the radio is always OFF, the CPU consumes 3,6mW at run mode and 0,06mW at LPM and the CPU's LPM percentage is 60%, then the device consumed 3823J.

Battery Info:

- Voltage (2 bytes): The voltage, in mV, measured by the CPU battery sensor.

The Voltage of the CPU, when the system knows the discharge curve of the battery, can be used to estimate the power level of the device's battery. For example, let's say that the battery's discharge curve is linear, the start is 3.0V and the end is 2.5V. If the CPU reports that the current voltage is 2.9V, the system can assume that the device's battery is at its 80% of its capacity.

Both will be mapped as sensors, in order to be parsed by the operator.

4.5 Over The Air Firmware Update

The OTA firmware update is a must-have feature for any WSN, which provides a way to update the firmware of the end-nodes, without the need to recall them. The problem with the OTA, is that it can't be designed to be fully hardware-agnostic and that causes its implementation to be tailored to specific hardware. Because of that, the hardware-specific features of this mechanism are discussed later (section 5.3).

4.5.1 OTA Firmware Metadata

In order for an OTA mechanism to work, the firmware must be accompanied with various metadata. The **Firmware-Meta** is a data structure that holds such information:

Property	Size	Description
Version[1]	2 bytes	This is the version number of the firmware. This value is used to identify the firmware chunks and of course to keep track of the **latest** firmware.
Size	4 bytes	This is the size of the firmware in bytes.
CRC	2 bytes	This value is used to check for the integrity of the firmware.

TABLE 4.9: OTA Firmware Metadata

[1] The version number is transformed into a **major.minor.patch** format, by using the most significant byte a the major number, the 4 most significant bits of the second byte as the minor, and the rest bits as the patch. E.g. version 289 (or 0x121) is actually the version 1.2.1.

4.5.2 Update Procedure

For an end-node to update its firmware, it needs to download the latest firmware to the latest firmware slot and after a successful download, to command the bootloader to boot from the latest firmware.

While running, the update procedure doesn't affect the other module's operation, but on finish the end-node needs to reboot, which means that the end-node must be reconfigured to make sure that nothing is broken and this might affect the sampling operation.

The table below shows the OTA packet types, with their corresponding fields:

Type	Description	Fields	Direction

Start OTA	The application operator notifies the target end-nodes that the system started the over the air update procedure	Group ID (1 byte) Version (2 bytes)	Operator ↓ Group
Start OTA Response	The end-node notifies operator that it started OTA	End-Node Address (2 bytes) Version (2 bytes)	Operator ↑ End-Node
OTA Firmware chunk	The application operator sends to the target end-nodes a firmware chunk	Group ID (1 byte) Device Type (4 bits) Application Type (4 bits) Version (2 bytes) Chunk Offset (3 bytes) Firmware Chunk (rest bytes)	Operator ↓ Group
Verify OTA	The application operator notifies the target end-nodes that they must verify their stored firmware	Group ID (1 byte) Device Type (4 bits) Application Type (4 bits) Firmware Meta (meta-size)	Operator ↓ Group
Verify OTA Response	The end-node notifies operator that it verified the stored firmware, providing information about whether the firmware is valid (CRC) and how many bytes it received	End-Node Address (2 bytes) Version (2 bytes) Received Bytes (3 bytes)	Operator ↓ Group
Stop OTA	The application operator notifies the target end-nodes that the system stopped the over the air update procedure	Group ID (1 byte) Version (2 bytes)	Operator ↑ End-Node

TABLE 4.10: OTA Packet Types

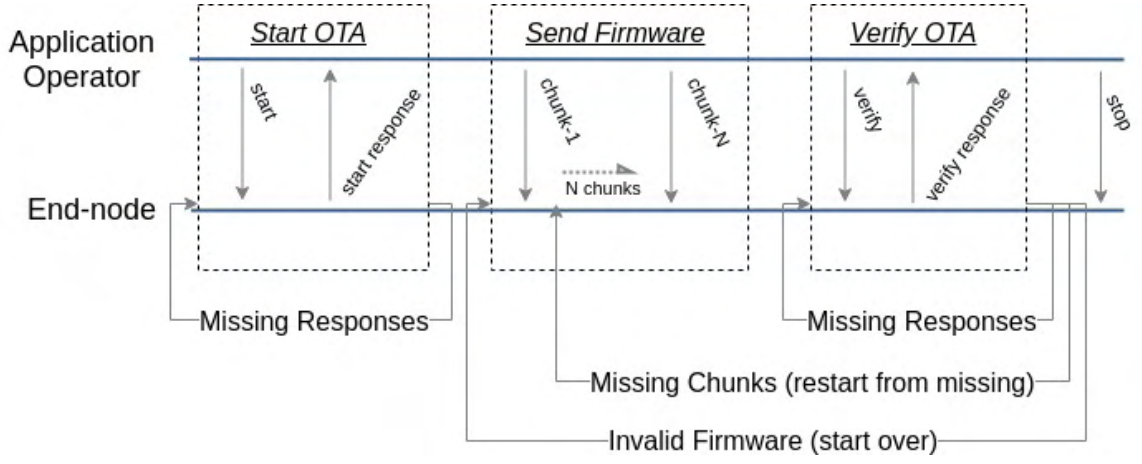

FIGURE 4.13: OTA Firmware Update Procedure.

The procedure starts with a Start OTA packet sent by the application operator. If the end-node's current firmware version is older than the new version it responds back with a Start OTA Response packet, to let the operator know that it started the procedure. The operator keeps track of the end-nodes' firmware version and is resending **Start OTA**, until all the end-nodes that need this update, respond back.

After the start of the procedure, the application operator sends all the firmware bytes (OTA Firmware Chunk packet) in a row and on finish it sends a Verify OTA packet to all end-nodes that participated in the procedure, until they respond with a Verify OTA Response packet.

If an end-node responds with failure, the procedure restarts from the smallest offset of all the end-nodes that responded with failure. In case of invalid firmware (CRC validation), the send procedure restarts.

When all the end-nodes have responded that their firmware update procedure finished successfully, the operator sends a Stop OTA packet and all the end-nodes will eventually reboot with the new firmware.

Chapter 5

System Implementation

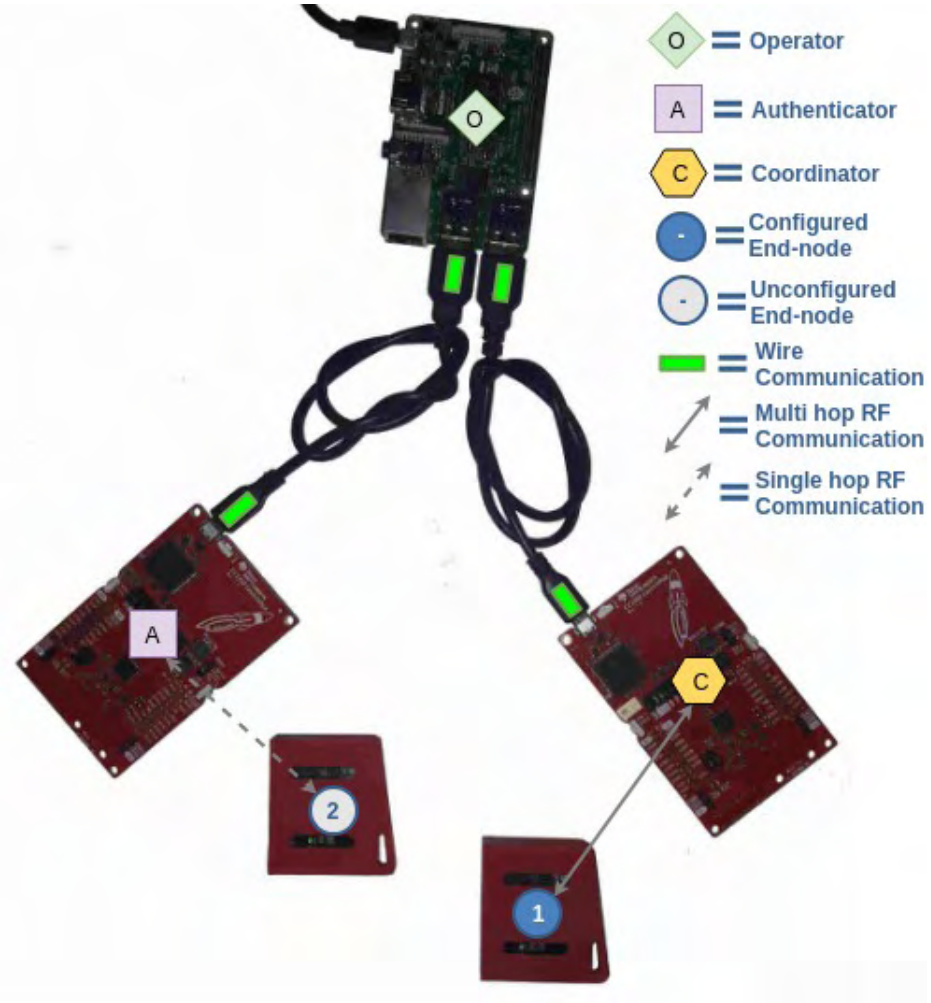

FIGURE 5.1: System's hardware.

All system components are implemented through firmware that runs over specific hardware (or platform). The use case of each component determines the hardware that is required for the platform that will host the firmware of that component. For example, the operator

needs to have two USB ports to communicate with the coordinator and the authenticator. Respectively, these two need one USB port each and an RF module to communicate remotely with the end-nodes. The end-nodes should also have an RF module to communicate with each other, the coordinator and the authenticator, and also must be battery-powered.

5.1 Platform Architecture

The current implementation uses three boards, the **Raspberry Pi 3 Model B** for the operator, the **LaunchPad CC1350** [11] for the authenticator and the coordinator and the **SensorTag CC2650** [12] for the end-nodes.

The SensorTag CC2650 board, the LaunchPad CC1350 board, and their corresponding MCUs are supported out of the box by the Contiki OS, which allows to use it as one of the technologies for the implementation of this system.

5.1.1 Raspberry PI 3 Model B

FIGURE 5.2: Raspberry Pi 3 Model B.

The Raspberry PI 3 Model B features a Quad Core 1.2GHz Broadcom 64bit CPU, 1GB RAM, 4 USB-2 ports and is capable of running NodeJS applications.

The authenticator and the coordinator are each connected to one USB port. Each of those connections serves as a communication channel and power supply for the embedded devices.

This board is chosen because it is a cheap and easy to use solution but any device with at least 128MB RAM, two USB ports and the NodeJS environment can host the operator's firmware.

5.1.2 Texas Instruments CC13xx-CC26xx MCU Families

The CC13xx and CC26xx Ultra-Low Power Wireless Microcontrollers are cost-effective, ultra-low power MCUs that offer very low active RF and MCU current, and low-power mode current consumption.

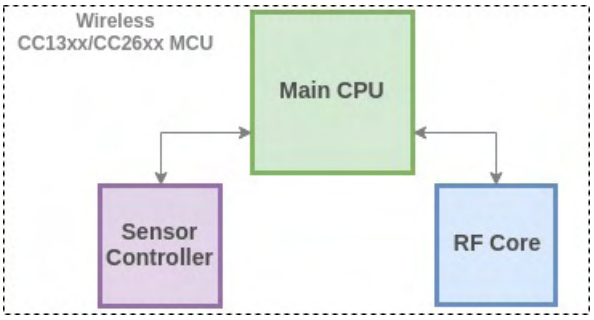

FIGURE 5.3: CC13xx/CC26xx abstract architecture.

Each wireless MCU contains three different processing units:

- A **Main CPU** for the higher layers of the network stack and the application logic.
- A **RF Core** radio peripheral for all the low-level RF handling.
- A **Sensor Controller** for the low-power sensor sampling.

The Main CPU will host and run the actual firmware and will manage the other processors and peripherals.

Section	Size	Addresses
Firmware	130984 bytes	0x00000 - 0x1FFA7
CCFG	88 bytes	0x1FFA8 - 0x1FFFF

TABLE 5.1: Main CPU's Internal Flash Structure

The internal flash of the Main CPU is 128KB big, is split into 4KB blocks (or pages) and is used to store the firmware of the device. The Customer Configuration Area (CCFG) section is a set of registers containing useful information about the device (e.g. the customizable 64-bit MAC address).

The RF Core's firmware implements the 802.15.4 physical layer and provides an interface to transmit and receive 802.15.4 frames. The RF Core's main purpose is to deal with all the time-critical radio events that otherwise would consume much of the Main CPU's processing time. The RF Core's firmware should not be changed by the user because it is tailored to specific hardware and the 802.15.4 protocol.

The Sensor Controller provides a simple interface for asynchronous sampling by the sensor peripherals. To take a sample, the Main Processor triggers the Sensor Controller to enable a specific sensor, and signal back when the sample is ready. This asynchronous operation allows the Sensor Controller, which is a power-optimized CPU, to operate the sensors while the Main CPU sleeps to save power.

In addition to the Main CPU's firmware, a preprogrammed UART/SPI Bootloader is stored to the Main CPU's ROM and allows the reprogramming through the SPI or UART. This bootloader can be used by the XDS110 debugger to program the Main CPU.

The open-source and cross-platform TI Uniflash Standalone Tool is used for transferring the desired firmware, through the XDS110 debugger, to the target MCU.

The main difference between the CC1350 and the CC2650 MCUs is that the first is also able to communicate using the Sub-1 GHz frequencies, but in favor of simplicity this can be ignored and be treated as the same hardware.

5.1.3 SensorTag CC2650

The SensorTag CC2650 is a board that combines the CC2650 wireless MCU used for RF communication, and various peripherals (e.g. external flash, sensors) needed by many IoT solutions.

FIGURE 5.4: SensorTag CC2650.

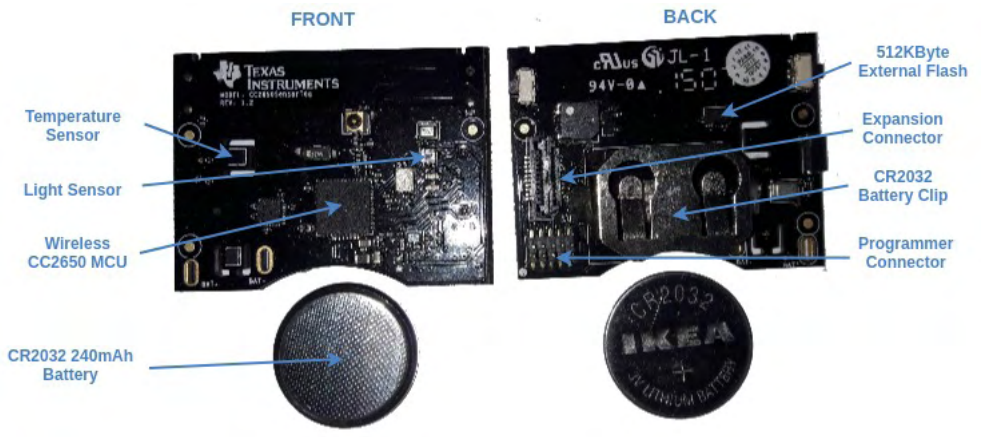

FIGURE 5.5: Inside SensorTag CC2650.

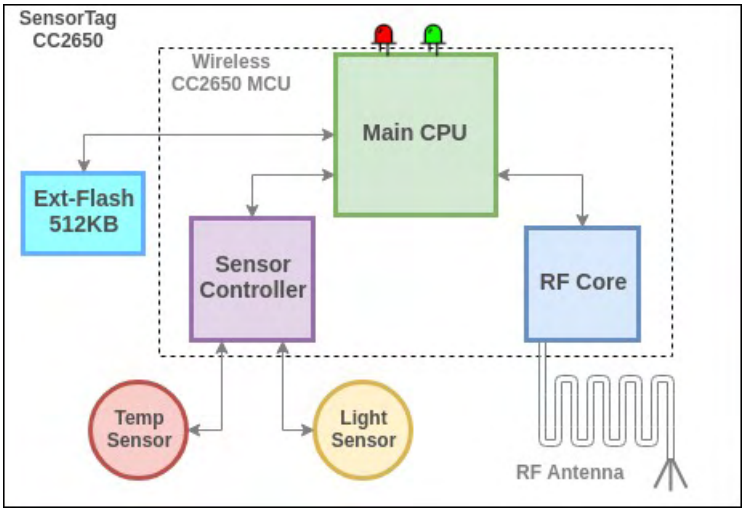

FIGURE 5.6: SensorTag CC2650 abstract architecture.

To extend the storage of the MCU, an external SPI flash (winbond W25X40CLUXIG) will be used. In short this flash is 512KB big and is split into 128 erasable 4KB sectors.

To sense the environmental conditions, there are various sensor peripherals inside this board. Just for the proof of concept, the system utilizes the TMP007 Temperature Sensor and the OPT3001 Light sensor.

The SensorTag provides two operation leds that can be used to inform the user about the device's state and two buttons that can be used by the user to interact with the device. The Power Button, as the name implies, is used to turn the device on and off, while the User Button can be programmed to do anything.

The SensorTag is a battery-powered device that uses a CR2032 240mAh Battery. With this battery the device is expected to operate for months (more details at section 6.2). Of course, the lifetime of the device is critically affected by the user configuration and the size of the network.

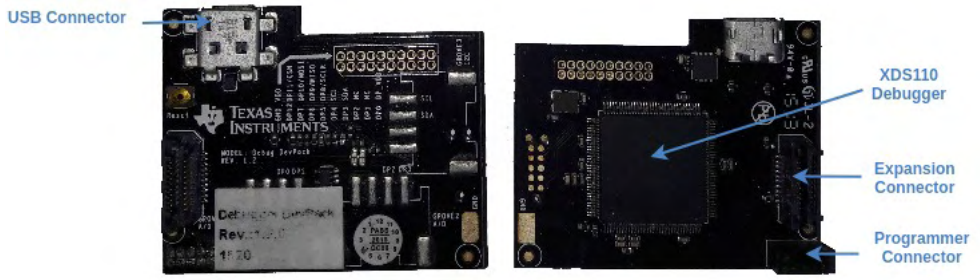

FIGURE 5.7: SensorTag XDS110 Debugger.

The SensorTag XDS110 debugger is connected directly to the SensorTag, using the programmer's connector and the expansion connector, and is used to transfer the firmware to the internal flash of the Main CPU.

5.1.4 LaunchPad CC1350

The LaunchPad CC1350 is a development board that combines the C1350 wireless MCU and the XDS110 Debugger on the same board. The wireless MCU is used for RF communication, while the debugger is used for serial communication and easy reprogramming.

FIGURE 5.8: LaunchPad CC1350.

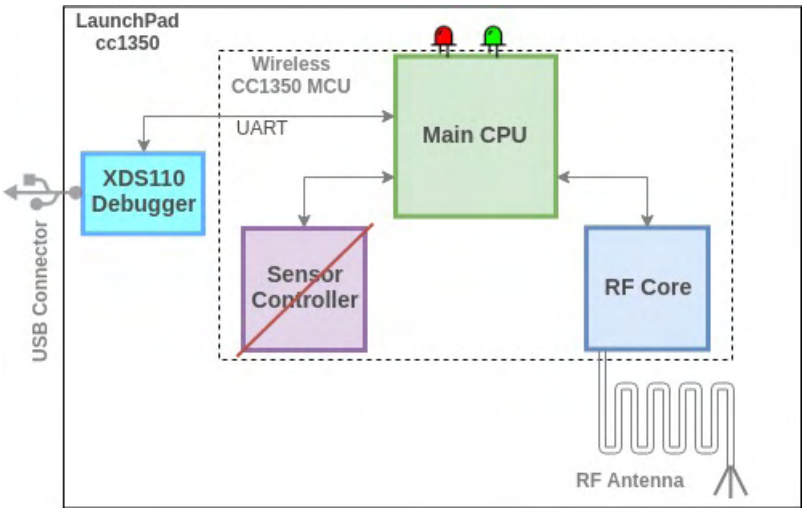

FIGURE 5.9: LaunchPad CC1350 abstract architecture.

The LaunchPad doesn't utilize the Sensor Controller of the MCU but uses the RF Core to communicate with the end-nodes.

The LaunchPad provides two operation leds, which can be used to inform the user about the device's state. The green led indicates that the device is powered on and operates without problems, while the red led turns on when a communication cycle starts and turns off when the cycle stops.

The LaunchPad can connect, and communicate with another device using a micro-USB cable connected to its USB connector. The device is supplied with power through this connection.

5.2 Software Architecture

5.2.1 Application Operator

The operator's firmware is built using **NodeJS** and the **Express Web Framework** while its user interface is built using **AngularJS**. Its role is to keep the network configuration, to distribute this configuration to the network and to collect the data coming from the end-nodes.

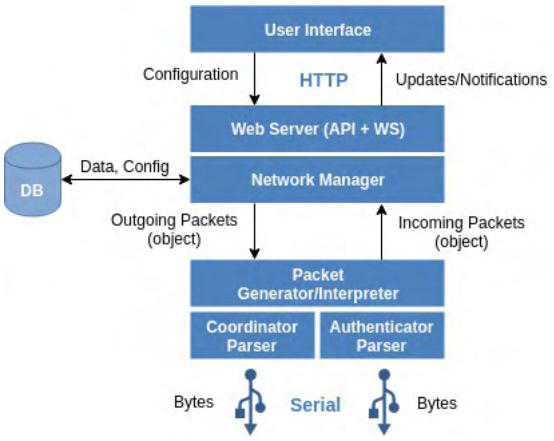

FIGURE 5.10: Application-operator software architecture.

The Packet Generator/Interpreter is a library of constructors/destructors that can receive a valid packet as an object, convert it to a byte stream and vice versa. It contains implementations for every packet described in the previous chapter.

The Coordinator Parser handles the serial port connection with the coordinator, while the Authenticator parsers the connection with the authenticator. Both those parsers implement the serial communication protocol, described in the previous chapter, and transfer the operator's packets to the device connected to their serial port connection. The parsers collect the bytes coming from their serial port connection and try to use the Packet Interpreter to convert the bytes into valid packets. When a valid packet is constructed, it is transferred to the operator for further handling.

The Network Manager maintains the network configuration and uses it to configure the network. It implements the communication cycle and the OTA Update Procedure. With the communication cycle, it manages to receive data packets from the end-nodes, to send the enclosed data to the user interface and to store them to its Database. This library is used by the Network Manager to convert packets into bytes, to send them to the network.

Algorithm 1: Communication Cycle

```
   // Config Phase
 1 waitDuration = prevCycle.waitDuration || (2.5 * maxDepth + config.endNodes);
 2 expected = prevCycle.expected ; // array
 3 communicated = [];
 4 retries = 0;
 5 lostNodes = [];
 6 do
 7 |  prevLostNodes = lostNodes;
 8 |  sendConfigCheck();
 9 |  duration = waitForReports(waitDuration);
10 |  lostNodes = prevCycle.expected - prevCycle.communicated ; // array diff
11 |  if lostNodes.length > 0 then
12 |  |  if prevLostNodes == lostNodes then
13 |  |  |  retries++;
14 |  |  else
15 |  |  |  waitDuration += (lostNodes/expected) * waitDuration;
16 |  else
17 |  |  waitDuration = duration ; // best duration
18 while lostNodes && retries < 5;
19 expected = communicated;
20 communicated = [];
21 if state.collectEnabled then
   |  // Collect Phases Algorithm...
22 if state.lpmEnabled then
23 |  sendLPM();
24 currCycle.waitDuration = waitDuration;
```

Algorithm 2: Collect Phases

```
1  waitDuration,expected,communicated = ... ; // from Config Phase
2  do
       // Collect Phase (Data)
3      sendGetData();
4      waitForData(waitDuration) ; // blocking
5      lostNodes = prevPhase.expected - prevPhase.communicated;
6      retries = 0;
7      while lostNodes.length > 0 && retries < 5 do
           // Collect Phase (Lost)
8          prevLostNodes = lostNodes;
9          sendGetLostData(lostNodes);
10         waitForData((lostNodes/expected) * waitDuration) ; // blocking
11         lostNodes = prevPhase.expected - prevPhase.communicated;
12         if prevLostNodes == lostNodes then
13             retries++;
14     end
15 while hasMoreData;
16 ...
```

The Database (DB) is an instance of LinvoDB and it is used by the Network Manager to save/load the network's configuration and to store/retrieve incoming data.

The HTTP Web Server provides a Rest API and a WebSocket API. Both those APIs can be used by the user interface to exchange information back and forth.

Type	Path	Description
GET	/	Get network's config and state
GET	/config	Get network's configuration
POST	/config/groups	Create a new group configuration
PUT	/config/groups/:groupId	Update a group configuration
GET	/config/nodes	Get list of end-node configurations
POST	/config/nodes	Create end-node configuration
PUT	/config/nodes/:nodeId	Update end-node configuration
POST	/config/enableSampling	Enable sampling
POST	/config/disableSampling	Disable sampling
POST	/config/enableCollect	Enable data collection
POST	/config/disableCollect	Disable data collection
POST	/config/enableLPM	Enable low power mode
POST	/config/disableLPM	Disable low power mode

GET	/state	Get network's state
GET	/state/groups	Get the state of all groups
GET	/state/groups/:groupId	Get group's state
GET	/state/nodes	Get the state of all end-nodes
GET	/state/nodes/:nodeId	Get end-node's state

TABLE 5.2: Operator's Rest API

Type	**Description**
CONFIG	Configuration update
STATE	State update
DATA	Data packet

TABLE 5.3: Operator's WS API

The user interface is a front-end AngularJS application that is served to the user's browser and helps him interact with the operator and, through him, with the network. It communicates with the operator using its REST/WS API.

5.2.2 Network Coordinator and Network Authenticator

The coordinator's firmware and the authenticator's firmware are both built using the **C** programming language and the **Contiki OS**. Both components are a bridge of the communication between the operator and the end-nodes. Their firmware has similar software architecture and they share a lot of code.

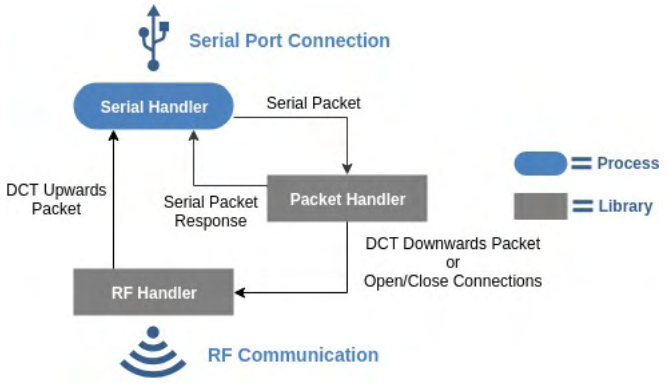

FIGURE 5.11: Network-coordinator software architecture.

The coordinator's role is to handle the traffic in and out the private WSN, to be the root of all the network's DCTs, to forward the operator's packets to the network and to forward the end-nodes packets to the operator.

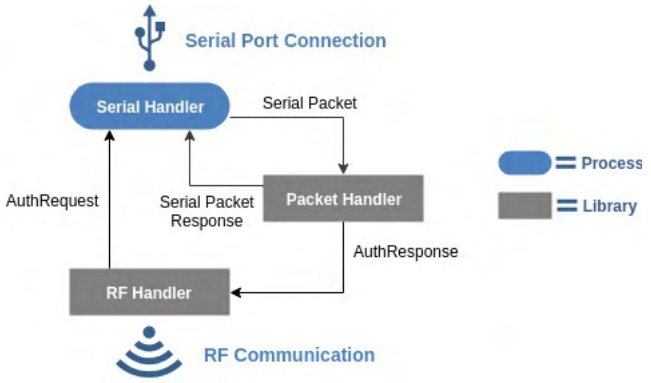

FIGURE 5.12: Network-authenticator software architecture.

The authenticator's role is to communicate with unauthenticated end-nodes in order to transfer their requests to the operator and the corresponding responses back to them.

Serial Handler

The Serial Handler is a Contiki Process dedicated to the serial communication. It handles the device's serial port connection with the operator and implements the serial communication protocol described in the previous chapter. The serial handler collects the bytes coming from its serial port connection, tries to construct valid packets and forwards those packets to the Packet Handler. The Serial Handler forwards any incoming packets to the operator via the serial port connection.

Packet Handler

The Packet Handler interprets the incoming serial packets and sends back serial packet responses that confirm that the packet reception. If the packet has a remote target (group or end-node) it is forwarded to the RF Handler. In the case of coordinator it is also responsible for opening and closing the group connections.

Coordinator's RF Handler

The RF Handler is a library of callbacks that handle the traffic coming in and out the radio.

The coordinator's RF Handler interprets the packets coming from the Packet Handler as DCT downwards packets and forwards them using the corresponding DCT. Respectively, it interprets the packets coming from the radio, as DCT upwards packets and forwards them to the Serial Handler.

The coordinator's RF Handler opens two Rime connections for each active group, a **Trickle** connection and a **Collect** connection. The Trickle connection is used to transfer the DCT downwards packets, while the Collect connection is used to receive the upwards packets.

Algorithm 3: Coordinator's RF Handler Callbacks

```
1 def collectCallback(data: Packet, sender: Address, depth: number):
2     if data.type == LINK then
          // in case of link info packet, also merge sender's depth
3         buffer = [data.type, sender, depth, data.payload];
4     else
5         buffer = [data.type, sender, request.payload];
6     serialPacket = createSerialPacket(buffer);
7     writeToSerial(serialPacket);
```

Authenticator's RF Handler

The authenticator's RF Handler interprets the packets coming from the radio as Broadcast Request packets for the operator and forwards them to the Serial Handler. Respectively, it interprets the packets coming from the Packet Handler as Response packets and forwards them using the single-hop unicast mechanism.

The authenticator's RF Handler opens two Rime connections, a **best-effort single hop broadcast** and a **single hop unicast**. The Auth Request packets are received from the Broadcast connection, while the Unicast connection is used to send the corresponding Auth Response packets.

Algorithm 4: Authenticator's RF Handler Callbacks

```
1 def broadcastCallback(authRequest: Packet, sender: Address):
2     buffer = [request.type, sender, request.payload];
3     serialPacket = createSerialPacket(buffer);
4     writeToSerial(serialPacket);
```

5.2.3 End-nodes

The end-node's firmware is built using the **C** programming language and the **Contiki OS**. Their primary role is take measurements based on the desired configuration and send them to the operator through the coordinator and their secondary role is to act as intermediates, between the other end-nodes and the coordinator, to help them communicate.

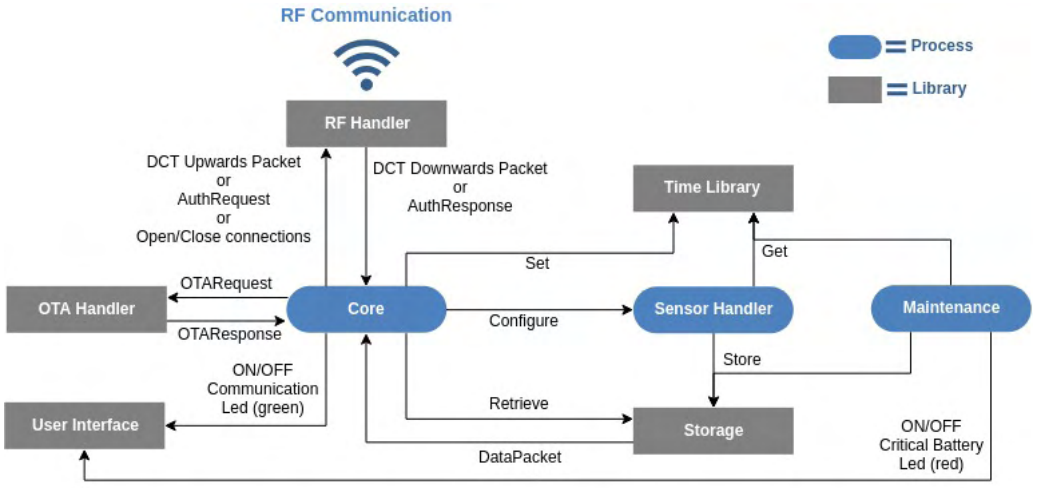

FIGURE 5.13: End node software architecture.

User Interface

The user interface is a library that provides a way to notify the user about the end-node's status. It turns the platform's leds on and off based on the end-node's state. The green led (or Communication LED), turns on when a communication cycle starts and turns off when the cycle stops. The red led (or Critical Battery LED) turns on when the battery of the device is at critical levels.

Definition	Description
void setLed(uint8_t ledID, bool status)	Turns the specified led (ledID) ON/OFF based on status. The acceptable led IDs are the BATTERY_LED and the COMMUNICATION_LED

TABLE 5.4: User Interface Library API

Storage

The Storage library is using the target platform's external flash to persistently store data.

Definition	Description
void storeMeasurement(uint8_t *bytes, uint8_t size)	Stores the measurement to the Measurements State section
void storeStats(stats_t *stats)	Saves the provided statistics to the Device State section
uint8_t createDataPacket(uint8_t *bytes, bool maintenance)	Creates a new empty data packet and fills it with measurements and if requested, the maintenance statistics. This operation removes the retrieved measurements and clears the maintenance statistics

TABLE 5.5: Storage API

The Storage utilizes the last 276KB (or 69 sectors) of the external flash and leaves the rest sectors for the OTA Handler and the OTA Bootloader. It splits these sectors into two sections, one for the Device State (e.g. maintenance stats) and one for the Measurements. Below is the external flash structure with only the storage's sections specified:

Section	Size	Addresses
...	236KB	0x00000 - 0x3AFFF
Device State	16KB	0x3B000 - 0x3EFFF
Measurements	260KB	0x3F000 - 0x7FFFF

Table 5.6: Storage's External Flash Structure

The Storage, with average size of 13bytes per measurement and average sampling period 15minutes, can store more than 20000 measurements and can cover the end-nodes disconnected operation for more than 200 days.

Time Library

The Time Library implements the API that is mentioned at the previous chapter. It utilizes the platform's RTC module, to keep track of passing time.

Definition	Description
void setTime(uint32_t timestamp)	Sets the time of the system
uint32_t getTime(void)	Returns the number of seconds from Unix Epoch. Warning: If called before **setTime** is called at least once, it will return the number of seconds the device is turned ON
uint32_t getBootTime(void)	Returns a the boot time, in seconds from Unix Epoch. Warning: If called before **setTime** is called at least once, it will return '0'

Table 5.7: Time Library API

When the end-node receives a timestamp from the operator, it calls setTime to update its time. The library doesn't affect the operation of the RTC, but instead uses the known set delay to compute the correct time, which is the difference of the current time (timestamp + RTC value) minus the set delay. For example, if the received timestamp is 1529557832 seconds (or 2018-06-21T05:10:**32**.487Z) and the set delay was 300 seconds and the RTC reports that the passed time is 310 seconds, then the actual time is 1529557832 + 10 seconds (or 2018-06-21T05:10:**42**.487Z).

Maintenance

The Maintenance process periodically collects the aforementioned operation statistics and adds them to the storage. It utilizes the Contiki's Energest library, which monitors the time spent (in CPU ticks) in various CPU and Radio modes. To collect the statistics and the battery info, it uses two timers, one for each. Also, with the help of the OTA bootloader, the end-node records the possible crashes, caused by unexpected behavior (e.g. bugs or misconfiguration).

Algorithm 5: Maintenance Process

```
1  while true do
2      event,eventData = waitForEvent();
3      if event == TIMER_EVENT then
4          if eventData != sensors.battery.timer then
5              enableSensor(sensors.battery) ; // synchronous sampling
6              data = [sensors.battery.type, sensors.battery.value];
7              storeMeasurement(data, data.length);
8              disableSensor(sensors.battery);
               // check if the battery's voltage is below acceptable limits
9              if sensors.battery.value < MIN_VOLTAGE then
10                 setLed(BATTERY_LED, TRUE);
11         if eventData != stats.timer then
               // collects the current statistics and updates the variable
12             collectStats(stats);
               // saves the maintenance state to storage
13             storeStats(stats);
14 end
```

Sensor Handler

The Sensor Handler is a Contiki Process that is responsible for periodically turning the sensors ON, in order to take measurements and store them to the storage. All the sensors can be briefly triggered to take measurements and automatically deactivate themselves on

finish or they can stay always-on to skip warmup.

Algorithm 6: Sensor Handler Process

```
1  while true do
2      event,eventData = waitForEvent();
3      if event == TIMER_EVENT then
           // woke up from a timer event, need to find the timer
4          if eventData != sensors.temperature.timer then
               // it was the temperature timer, so enable temperature sensor
5              enableSensor(sensors.temperature) ; // asynchronous sampling
6      if event == SENSOR_EVENT then
           // woke up from a sensor event, need to find the sensor
7          if eventData != temperature then
               // it was the temperature sensor, so collect the temperature
8              data = [sensors.temperature.type, sensors.temperature.value];
9              storeMeasurement(data, data.length);
10             disableSensor(sensors.temperature);
11             if samplingEnabled then
                   // sampling is still enabled, reset the timer
12                 resetTimer(sensors.temperature.timer);
       // ...  this replicated for all sensors ...
13 end
```

To manage this procedure, the Sensor Handler uses one timer for each reading type, which is configured to tick periodically based on the samplingPeriod of this reading type. When the user sets the samplingEnabled flag to false, all the timers are stopped. In contrast, when the user sets the samplingDisabled to true, the timers of the configured reading types are updated started.

Definition	Description
void setSampling(bool status)	Starts/stops the sampling timers based on status.
void configureSensors(group_t *group)	It uses a group configuration to update all the samplingPeriods of the sensors. If a sensor doesn't match any reading type inside the group configuration, it is disabled until the next configuration. Note: that the active sensors' timers will be reset using the new sampling period

Table 5.8: Sensor Handler API

RF Handler

The RF Handler handles the incoming and outgoing RF packets. It forwards the incoming packets to the appropriate handler and the outgoing packets to the radio. It opens the same connections as the coordinator and the authenticator, receives DCT downwards Packets and Auth Response Packets and sends DCT upwards Packets and Auth Request Packets, through those connections. Finally, it provides to the Core the ability to open and close DCT connections.

Definition	Description
broadcast(uint8_t *packet)	It broadcasts a packet to the neighbors
void openConnections(group_t *group)	It opens a collect connection and a trickle connection, both using a unique channel based on group ID
sendGroupPacket(group_t *group, uint8_t *packet)	It sends a packet using a specific group's connections. Warning: This must be called after openConnections
void closeConnections(group_t *group)	It closes the group's connections

Table 5.9: RF Handler API

Core

The Core is the main process and the heart of the end-node's firmware, it holds and maintains the end-node's configuration and configures the other modules. It implements all the Communication Packet handlers and the Communication Cycle. It periodically sends Auth Request packets, through the RF Handler, to the operator, to request for authentication, until it receives backs an Auth Response packet. After an Auth Response packet it opens/closes the appropriate group connections and based on configuration it starts/stops and configures the Sensor Handler. When the operator requests for data, it collects them from the storage and send them through the RF Handler. Finally, it forwards all the OTA specific packets to the OTA Handler and toggles the user interface's leds based on its state.

Definition	Description
handlePacket(uint8_t *packet)	It calls the appropriate packet handler based on its type. If the packet is for the OTA update procedure, then it forwards it to the OTA Handler
handleAuthResponse(uint8_t *packet)	It updates the device configuration, opens/closes the group connections and sets the time library
handleGroupUpdate(uint8_t *packet)	It updates the group configuration
handleSetTime(uint8_t *packet)	It uses this packet's timestamp to set the time library
handleConfigCheck(uint8_t *packet)	It checks the samplingEnabled flag to update the Sensor Handler and sends back a Config Report packet
handleGetData(uint8_t *packet)	It checks the storage to retrieve a new Data Packet to send it to the operator
handleGetLostData(uint8_t *packet)	If found inside the lost devices it responds with the previous Data Packet
handleEnterLPM(uint8_t *packet)	It turns the radio OFF until the provided timestamp, for better power efficiency

Table 5.10: Core API

OTA Handler

The OTA Handler is a library that implements the OTA Update Procedure by providing the handlers for the OTA Packets (more details later on). It receives and accepts the OTA Packets and if needed, it sends back OTA Response Packets. To accept an OTA packet, it must have same device type and application type as the device.

Definition	Description
handleStartOTA(group_t *group, uint8_t *packet)	If incoming firmware's version is greater than current, then the update procedure starts and a Start OTA Response packet is sent back to the operator
handleOTAChunk(group_t *group, uint8_t *packet, uint8_t length)	If the firmware chunk's version is the currently updated one, then the chunk is stored and an OTA Chunk Response packet is sent back to the operator
handleVerifyOTA(group_t *group, uint8_t *packet)	If the requested verify is for currently updated firmware, it checks the firmware's validity and responds back with a Verify OTA Response packet with the result
handleStopOTA(group_t *group, uint8_t *packet)	It reboots the device after a successful OTA update procedure

Table 5.11: OTA Handler API

5.3 Over The Air Firmware Update

One requirement for the end-node's platform to support an OTA update procedure is to be able to store at least two times the size of the end-node's firmware. Because the end-node's

internal flash (128KB) isn't big enough to store multiple firmware (expected end-node's firmware size is around 70KB), the external flash (512KB) will be used to store them.

Another requirement is that the MCU must be able somehow to boot from the newly updated firmware. A side-effect of using the external flash, as a storage for the firmware, is that the MCU has no mechanism to boot from it, so the firmware must somehow be loaded to the internal flash, before it can run.

Of course, it's not safe for the CPU to write at the same position that is currently reading, so it is unable to replace its running firmware. To solve this problem, a custom bootloader has been created and is responsible, first copy the new firmware to the internal flash and then check its integrity. The internal flash contains two firmware regions, one for the **Bootloader** and one for the **OTA Firmware**, thus enabling the CPU to run either one and safely replace the other.

5.3.1 OTA Firmware

As mentioned in the previous chapter, the OTA procedure demands metadata for the transferred firmware. To make it easier for the CPU to handle these metadata, a new **OTA Firmware** structure is created based on the internal flash architecture and the expected maximum size of the firmware.

Field	Size
Firmware Meta	255 bytes[1]
Firmware Code	118529 bytes (116KB - meta)

Table 5.12: OTA Firmware

[1] The actual size of the **Firmware-Meta** structure is 8 bytes, but because the Vector Table (VTOR) of the platform must be 256bytes aligned, a padding is added after the metadata.

The OTA Firmware consists of two parts, the first one is the firmware metadata and the second one is the actual firmware code that will be executed by the platform.

Because the internal flash's size is 128KB and there must be space left for the bootloader, a decision must be taken for the maximum supported firmware size. Also, to simplify the implementation, the page containing the CCFG will be left unused (page alignment), leaving 31 pages for the Bootloader and the OTA Firmware. With all that in mind, the maximum supported firmware size will be 116KB (or 29 pages of the internal flash) big, leaving 8KB (2 pages) for the bootloader.

In order to build this new OTA Firmware, the system provides a **firmware-meta-generator** tool (written in C) that parses the firmware and computes the metadata. This tool takes two arguments, the first one is the target firmware binary file and the second one is the desired version number for this firmware.

5.3.2 Bootloader

Because the final firmware takes up most of the space of the internal flash, the bootloader's size must not exceed the 8KB limit (2 pages). For this to happen, a new compilation toolchain is provided which will compile only the Main CPU's drivers (srf06-cc26xx library) and the bootloader's functionality. Below is the new internal flash structure, designed to support the OTA:

Section	Size	Addresses
Bootloader	8KB	0x00000 - 0x01FFF
OTA Firmware	116KB	0x02000 - 0x1EFFF
Unused	4008 bytes[1]	0x1F000 - 0x1FFA7
CCFG	88 bytes	0x1FFA8 - 0x1FFFF

Table 5.13: OTA Internal Flash Structure

[1] To avoid messing with the CCFG sector, the remaining bytes of the 32nd sector will be left unused.

As mentioned above, the bootloader's main responsibility is to keep the current firmware, which is stored at the internal flash, up-to-date and check its integrity when it changes.

Also, it keeps track of reboots and crashes.

Algorithm 7: Bootloader

```
1  getBootloaderState(state);
2  if !state.isValid then
       // if state is invalid, then its first boot
3      state.bootCount = 0;
4      state.crashCount = 0;
5      state.bootTarget = current;
6  else
7      state.bootCount++;
   // check if this reboot was caused by an error
8  if state.error then
9      state.crashCount++;
   // check if need to update current firmware
10 if state.bootTarget == latest and current != latest then
11     updateCurrentFirmware(latest);
12     state.bootTarget = current;
   // check if need to reset to golden firmware
13 if !isValid(current) then
14     updateCurrentFirmware(golden);
15     reboot();
16 jumpToCurrentFirmware();
```

5.3.3 OTA Storage

The external flash will be used as a storage for the firmware that the OTA mechanism downloads, and for the bootloader state. Because the external flash memory is big enough to store multiple OTA Firmware, there will be two firmware slots, one used for a golden firmware and one used to store the latest firmware that will be updated by the OTA mechanism. Below is the new external flash structure, designed to support the OTA:

Section	Size	Addresses
Golden Firmware	116KB	0x00000 - 0x1CFFF
Latest Firmware	116KB	0x1D000 - 0x39FFF
Bootloader State	4KB	0x3A000 - 0x3AFFF
Storage State	276KB	0x3B000 - 0x7FFFF

Table 5.14: OTA's External Flash Structure

The purpose of the golden image is to make sure that if something goes wrong with the new firmware, there will be always a "factory" image to reset from. In fact, the current firmware can select to reboot and load from the golden image, in case it needs to.

5.3.4 Contiki OTA Fork and Firmware Bundler

Because the Contiki's build-system (as is), is incapable of supporting the OTA procedure, a fork of it is used instead, which has all the needed changes. In short, it contains two new LD scripts, the first one used for the compilation of the bootloader and the second one for the OTA Firmware. The bootloader's script generates all the needed sections (e.g. CCFG), for the target platform to run, and leaves untouched the OTA Firmware's section. The firmware's script generates only the necessary sections, skipping all the sections generated by the bootloader.

The system provides a bundler, which helps with the creation of the initial binary image that is used to program the end-node. It separately compiles the **Bootloader** and the **OTA Firmware**, generates the OTA Firmware's **metadata**, and finally merges all of them together into one binary file.

5.4 Project Configuration

The project provides various configuration options that can change the behavior of the system. Here is the list of the most important options:

1. Sub 1GHz operation mode.
2. Maximum Radio Packet Payload Size.
3. Maximum Serial Packet Payload Size.
4. Maximum End-node Address Size.
5. Maximum End-node ID Size.

Note that the power efficiency and the performance of the network are associated with those options and each option has a trade-off.

The **Sub 1GHz** operation mode forces the dual-band radio modules (CC1350) to operate using their Sub 1GHz radio frequency. By default is disabled.

The **Maximum Radio Packet Payload Size** is the number of bytes that can be used as payload inside a radio packet. By default equals 128bytes.

The **Maximum Serial Packet Payload Size** is the number of bytes that can be used as payload inside a serial protocol packet. By default equals 512bytes.

The **Maximum End-node Address Size** is the size of all the end-node's address. This is a unique address that used by the network stack to identify each end-node inside the network. By default equals 2 bytes.

The **Maximum End-node ID Size** is the size of all the end-node's ID. This is a unique ID that helps the application operator identify the end-node. The difference with the address is that the ID is unique between all the end-nodes and not only inside the network. By default equals 8 bytes.

Many of those options are set to much by the underlying 802.15.4 protocol specification. If this protocol is replaced with another, they must be changed accordingly.

Chapter 6

Experiments

6.1 Tools

The experiments in this chapter help evaluate the behavior of the sensor network, depending on the user options. Besides the maintenance library severla tools/mechanisms are used in order to emulate the network state and collect statistics about the operation of the WSN.

6.1.1 Sniffer

To monitor the live traffic inside the WSN some sniffing tools are used to collect packet statistics. Firstly, each end-node utilizes Rime's build-in sniffing layer that monitors the incoming and outgoing packets and with the help of the operator the latency of each packet is computed. Secondly, a device with sniffing firmware (sensniff [15]) is used to capture the traffic of the WSN and forward it to the Wireshark [14] utility. This tool is used as a secondary recording tool, to validate the traffic monitoring of the WSN.

6.1.2 Link Quality Emulator

Because we need to create and evaluate specific scenarios that are hard to achieve using the real link quality (e.g. RSSI) of the physical radios, an emulation library is used to create links between the desired end-nodes and emulate the packet loss probability of each link. Each device can specify the desired link for the data collection tree (DCT) and the probability of the packet losses when receiving packets through that link.

6.1.3 Power Consumption

A Digilent Analog Discovery 2 scope is used to power up the target device and measure its consumption via a simple 10 ohm resistor connected in series with the device. The scope collects 10 Million measurements with 5kHz sampling frequency that result into 33 minute sampling duration. These measurements are used to compute the actual consumption of the target device.

6.2 Power Profiling

The target of this experiment is to measure the power consumption of the end-node and to compute its battery lifetime. The maintenance library will be used to collect statistics about the end-node's operation, which will lead to an estimated consumption. The actual consumption will be measured using the aforementioned multimeter and will be compared with the estimated one.

FIGURE 6.1: Power Profiling Experiment.

In this experiment, only one end-node is used and its power consumption is measured for different communication periods. Each test lasts for 5 hours and results in an estimated and an actual consumption.

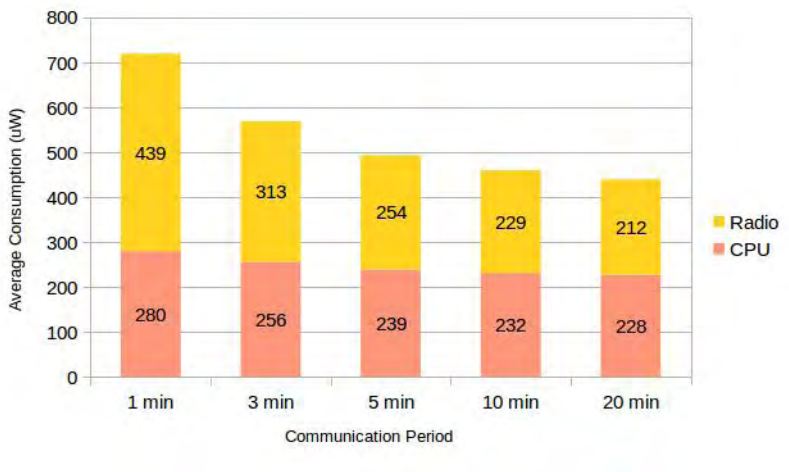

FIGURE 6.2: Power Consumption.

In figure 6.2 we can observe that as the period lowers, the radio and the CPU consumption decrease logarithmically. The radio consumption decreases more aggressively with a 50% decrease in the last case, while the CPU consumption decreases by 18% in the same case.

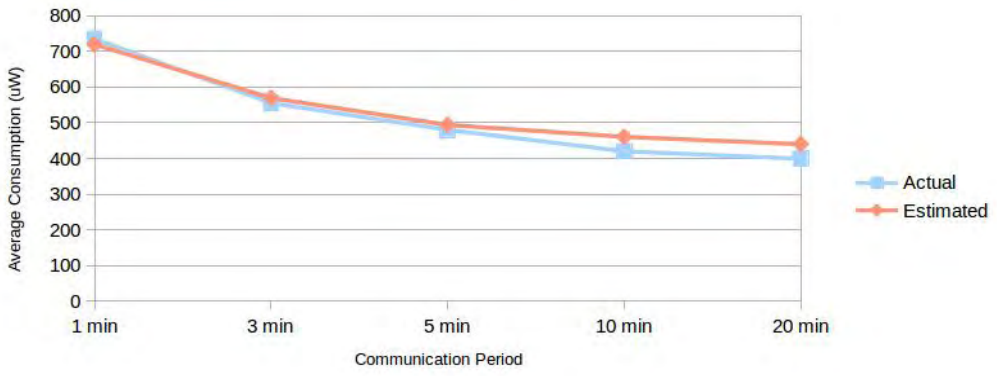

FIGURE 6.3: Estimated Consumption vs Actual Consumption.

The power consumption model that is used by the system to compute the end-node's consumption outputs values close to the actual ones (figure 6.3), with 2% up to 10% error percentage for really low values.

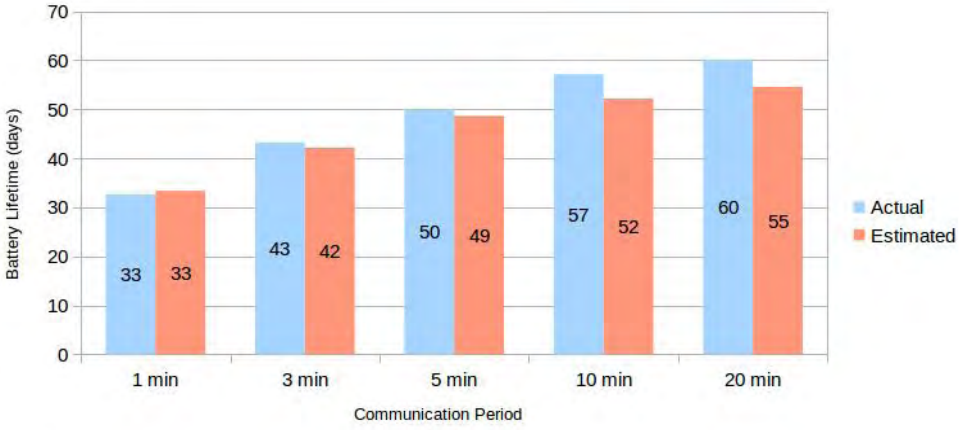

FIGURE 6.4: Estimated Lifetime using 240mAh battery.

Finally, in figure 6.4 we can see that the end-node can operate for many days using a 240mAh coin cell battery (with an estimated 80% useful capacity of 192 mAh).

6.3 Topology Profiling

The target of this experiment is to create profiles of the end-nodes that act as routers and as leaves (devices that don't route messages of other devices).

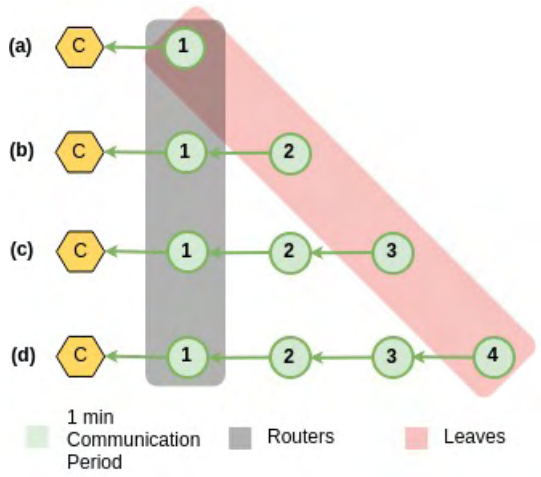

FIGURE 6.5: Topology Profiling Experiment.

In this experiment, various cases are created and are used to create the aforementioned profiles. The first end-node next to the coordinator is the router for all the other end-nodes. In contrast, the last end-node of the connection chain is the leaf, which means that it doesn't act as a router for other end-nodes. The first case (a) uses only one end-node, which is profiled as a router for zero end-nodes and as a leaf one hop away from the coordinator. Each subsequent case adds one more end-node, which increases the number of supported end-nodes for the router and the number of hops for the leaf. In all cases, the network is operating using one minute sampling/communication periods and lasts for 5 hours.

As expected, in the figures below we can clearly observe that the consumption and the latency of the end-nodes increase aggressively as the network depth increases.

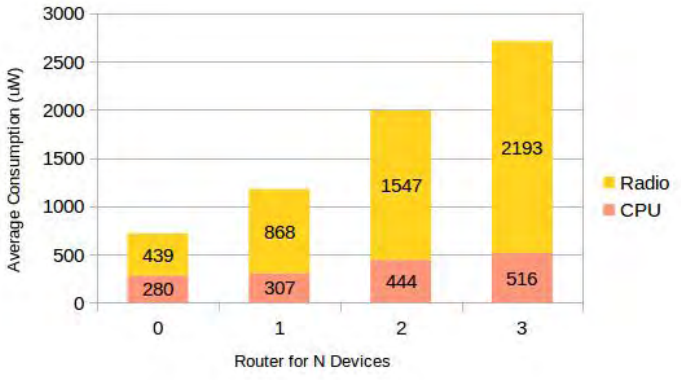

FIGURE 6.6: Router Consumption.

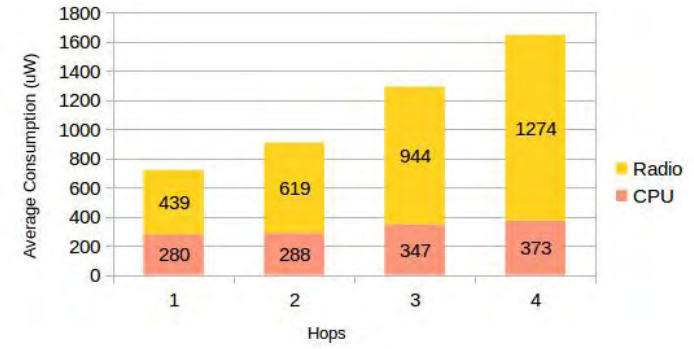

FIGURE 6.7: Leaf Consumption.

The radio is clearly the main factor for the total consumption and at the last case, it increases 5 times for the router and 2.9 times for the leaf, while the consumption of their CPU remains at the low side, with an increase of 1.85 and 1.33 times respectively.

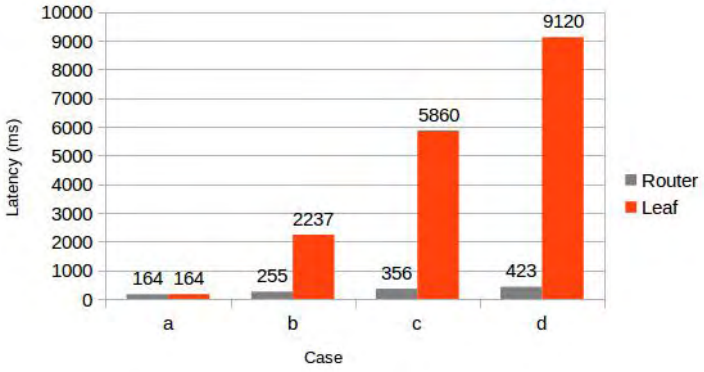

FIGURE 6.8: Topology Latency.

The router's latency increases linearly (2.6 times increase at case d) times but still its sufficient for any use-case. On the other hand, the leaf's latency increases exponentially (increased 55 times at case d) and this means this means that the current protocol's performance is not suitable for high data throughput.

6.4 Silo

The objective of this experiment is to show how the system's group functionality creates alternate routes for the end-nodes and allows them to organize their routes based on their link quality.

FIGURE 6.9: Silo Experiment.

In this experiment, the user is interested in monitoring two different levels of a silo, in order to evaluate the condition of the stored product. The user places the end-nodes inside the silo and fills it with the desired product, which lowers the communication quality inside the silo and prevents the user from interfering with the end-nodes. The end-nodes at the 2-hop range area, are unable to reach the coordinator directly, but instead must use other end-nodes as intermediates.

The silo must be monitored for long time periods without user intervention, with five minute sampling/communication periods.

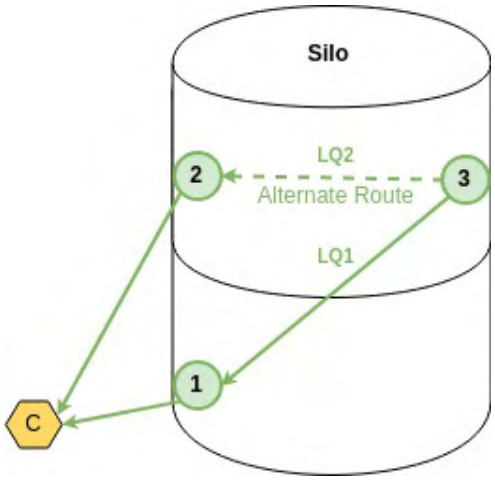

FIGURE 6.10: Silo Scenario.

In this scenario, the user must create a group and assign all the end-nodes to this group.

The end-node 3 selects the best parent based on the quality of its links (LQ1, LQ2). If for some reason the active link's quality (LQ1) degrades, the end-node will change its parent. This is a dynamic behavior and can occur at any time.

At the beginning of the scenario all the links have near 0% packet loss probability and the end-node 3 communicates via the end-node 1 (LQ1 > LQ2). The silo is filled with product and the packet loss probability of the end-node 3's link slowly increases from 0% to 40% and this means that at some point, it will change its parent and select end-node 2 as intermediate (LQ1 < LQ2).

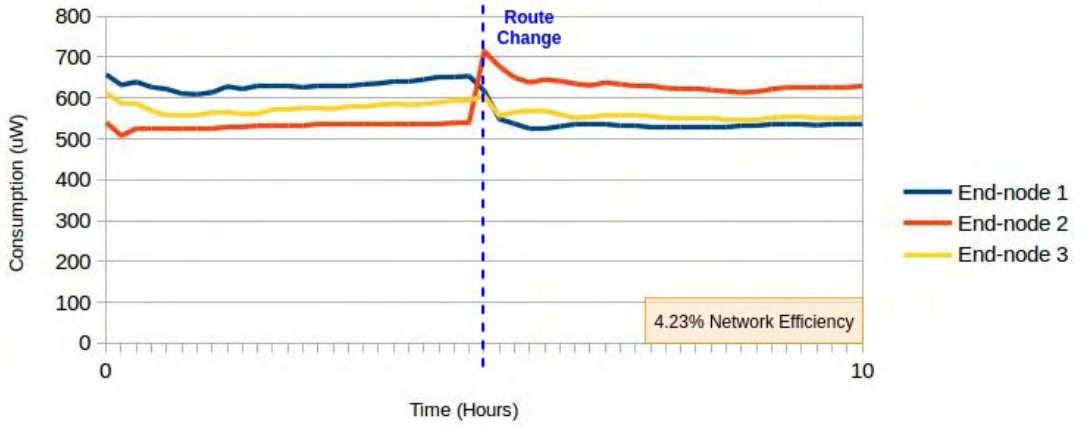

FIGURE 6.11: Silo Consumption.

After the first 5 hours, the route change occurs which results in 4.23% less consumption because the end-node 3 now sends fewer packets to reach the coordinator. The traffic is distributed to the end-nodes based on the link quality of the end-nodes, which results in less consumption.

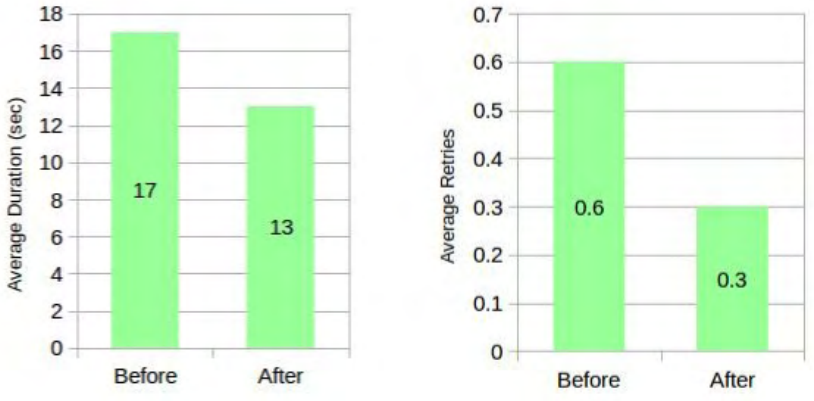

FIGURE 6.12: Silo Average Communication Cycle Duration and Retries.

The decrease in the average consumption is due to the change of the communication cycle duration that happens after the route change. The duration of the cycle is closely related to the number of times that the operator must request data (retries).

6.5 Container

This experiment is designed to show the importance of the system's support-group feature and how it can be used to help the end-nodes with communication problems, reach the coordinator.

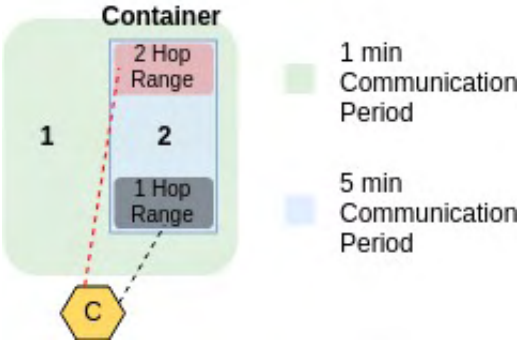

FIGURE 6.13: Container Experiment.

In this experiment, a fumigation process, using a highly-toxical gas (e.g. Phosphine), is done in a container and must be monitored to evaluate its progress and results. The user places the end-nodes inside the container and fills it with the desired product (e.g. wheat), which lowers the communication quality inside the container and prevents the user from interfering with the end-nodes. There are two discrete areas that have different monitoring requirements.

The outside area of the container (Area 1) must be monitored to detect gas losses. This is required firstly to reassure the safety of the people working near this container, and secondly to notify the fumigator that actions are required for the success of the fumigation. The end-nodes that monitor this area must use at most one minute sampling/communication periods because the user must be promptly informed about the conditions.

The inside area of the container (Area 2) must be monitored to evaluate the concentration of the gas and various other conditions that might help with the fumigation (e.g. Temperature, Humidity). The end-nodes that monitor this area must remain alive for long time periods and cannot be easily replaced in case their battery is drained. They use five minute sampling/communication periods.

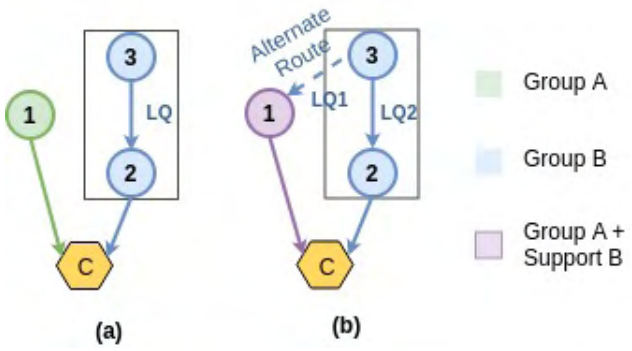

FIGURE 6.14: Container Scenarios.

The user must create a group for each area, based on their requirements and assign the end-nodes to these groups. The end-node 1 is assigned to group A and the end-nodes 2 and 3 are assigned to group B.

In contrast to the previous experiment, the group configuration of this experiment doesn't provide an alternate route to the end-node 3 that is two hops away. This means that if its link quality degrades it will face communication problems and its parent will have higher power consumption.

In the first scenario (a), the end-node 3 is using the end-node 2 as a parent in the DCT and their link quality (LQ) is affected by the product that is placed inside the container. To emulate the real conditions inside the container, the LQ starts from 0% packet loss probability and ends at 40%. All the other links have near 0% packet loss probability.

In the second scenario (b), the end-node 1 is also set to support group B to participate in its DCT. This allows end-node 3 to create an alternate route using end-node 1 and avoid communication with the end-node 2 when their link quality degrades. Same as the previous experiment, when the quality of the link between end-node 2 and 3 degrades (0% -> 40%), the end-node 3 will change its parent and select end-node 1 as intermediate (LQ1 > LQ2)

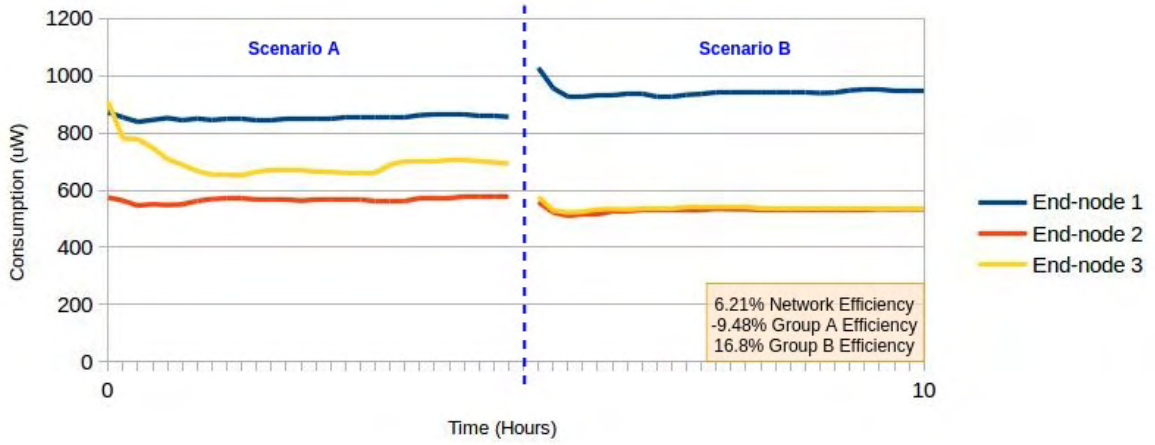

FIGURE 6.15: Container Consumption.

In the second scenario (b), the group B gains 16.8% power efficiency in an expense of increased consumption for the group A. Based on the use-case, this trade-off is tolerable because the user can replace the battery of end-node 1.

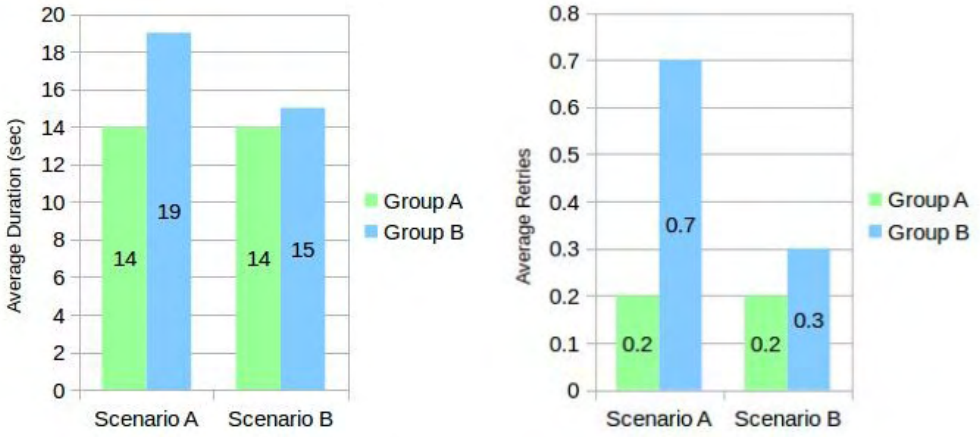

FIGURE 6.16: Container Average Communication Cycle Duration and Retries.

The decrease in the group B's average consumption is caused by the decrease of the average communication cycle duration, while the increase in the group A's average consumption is caused by its participation in the group B's communication. The duration of the cycle is closely related to the number of times that the operator must request data (retries).

6.6 Summary

The Silo and Container experiments show that the system provides the user with various ways to support their inaccessible end-nodes, either by using other end-nodes inside the same group or by enabling the system's support functionality.

At the same time, all the experiments show that the current protocol is suitable for the requested use-cases and creates networks that maintain low power consumption. However, as the network size increases or the communication period decreases, the network's lifetime drops significantly.

Finally, although the network's latency is sufficient for the current user requirements, if high throughput becomes necessary, then the communication protocol must be redesigned to achieve better results.

Chapter 7

Conclusion

Summary

In the context of this work, a system is developed that helps the user manage his sensors and collect their data. The user can interact with the system via a web server and a browser user interface. The embedded devices are creating 802.15.4 wireless sensor networks and their firmware is built using Contiki OS. Because the end-nodes are battery-powered devices their firmware is designed to optimize their power efficiency and enables them to operate for extended time periods. The wireless sensor networks create dynamic topologies that allow the ad-hoc placement of the devices and can self-cure after a critical change, e.g. an end-node that helped other end-nodes communicate, stopped working.

The current design and implementation focuses on simplicity rather than optimizations and this leaves a lot of work to be done to make it more production-ready. Rather than designing and implementing the communication protocols from scratch, ready-to-use protocols were used to speed up the system's development. Although in some cases the latency of some end-nodes rises exponentially, the data loss was always zero, which means that the main goal of the system was achieved. At the same time the consumption, in most cases, is low enough to enable the end-nodes to operate for months.

Future work

Final Ecosystem

The system is predominantly valuable as part of a larger ecosystem that collects all the data coming from the WSN, maintains them and allows the user to monitor them, get valuable analytics about them and remotely interact with it.

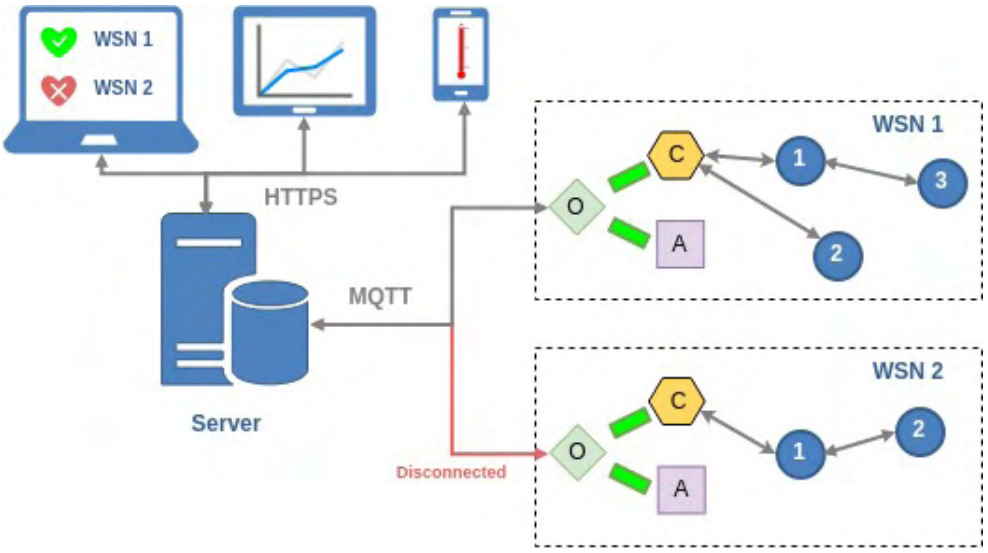

FIGURE 7.1: Final Ecosystem.

To achieve that, the operator's web server, user interface and configuration logic must be transferred from the operator to a remote server and a connection (e.g. MQTT) must be used to transfer the network state and configuration between them.

Communication Optimizations

The experiments show that the latency of the transferred data packets is aggressively affected by the user configuration and more importantly the location/role of the end-node inside the network. The main cause of such high latency is the mechanism that is used to transfer the data collection tree (DCT) upwards packets and the ContikiMAC layer that both focus on lowering the radio duty cycle instead of transmitting the packets faster. If higher latency is needed, they can both be optimized to achieve this goal.

While the routing protocol, which is used for the creation of the data collection trees, is efficient and dynamic enough to solve almost any ad-hoc topology creation, it can be further optimized by using the operation statistics of the end-nodes to distribute the traffic more efficiently. For example, if an end-node has alternate routes, it can decide which route to use by collecting the current consumption of its possible parents and use the one with the lowest consumption.

Optimized Time Sync Procedure

The system provides a time library and a time synchronization procedure that doesn't deal with the propagation delay error. The Contiki provides a **timesynch** procedure that is way more accurate but currently is not supported by the used platform. In the future the

plan is to re-create the synchronization procedure with the Contiki's **timesynch** procedure as a guideline.

Bibliography

[1] https://en.wikipedia.org/wiki/IEEE_802.15.4

[2] http://newton.ee.auth.gr/pavlidou/papers/J075.pdf

[3] http://ecee.colorado.edu/~liue/teaching/comm_standards/2015S_zigbee/802.15.4-2011.pdf

[4] http://www.contiki-os.org/

[5] https://github.com/contiki-os/contiki

[6] http://dunkels.com/adam/dunkels11contikimac.pdf

[7] http://dunkels.com/adam/dunkels11announcement.pdf

[8] http://dunkels.com/adam/dunkels07adaptive.pdf

[9] http://dunkels.com/adam/dunkels06protothreads.pdf

[10] http://dunkels.com/adam/dunkels07softwarebased.pdf

[11] http://www.ti.com/lit/ds/symlink/cc1350.pdf

[12] http://www.ti.com/lit/ds/symlink/cc2650.pdf

[13] http://dev.ti.com/tirex/content/simplelink_cc13x0_sdk_1_00_00_13/docs/proprietary-rf/html/rf-proprietary/packet-format.html

[14] https://www.wireshark.org/

[15] https://github.com/g-oikonomou/sensniff