UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

Faculty of Science and Technology
Department of Computer Science

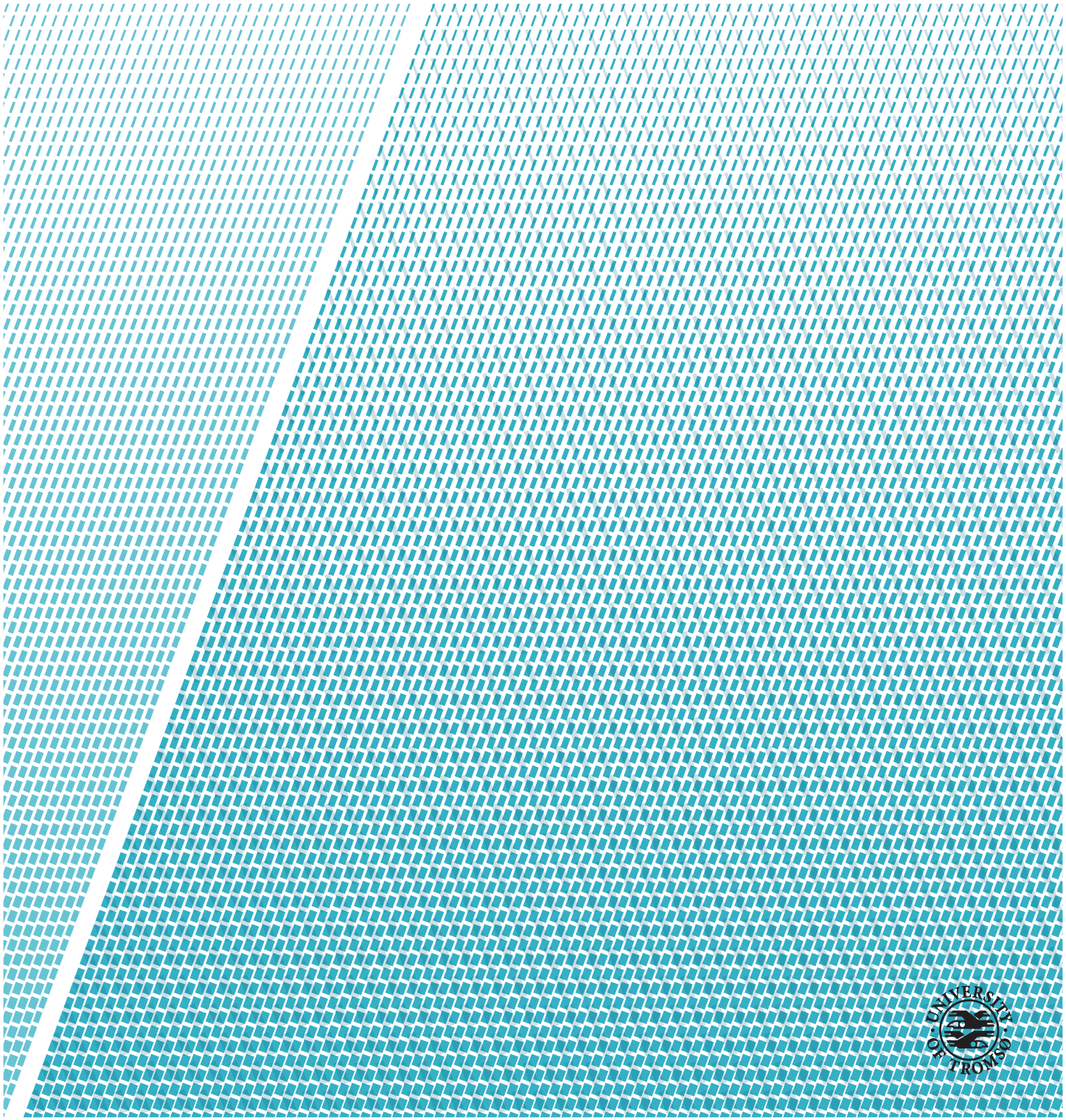
EC^3

Edge Command-Control-Communication System for Arctic Observatories

—

Łukasz Sergiusz Michalik

Master thesis in Computer Science [INF-3990] May 2017



Dedicated to those whom I can always depend upon

"We don't want to conquer the cosmos, we simply want to extend the boundaries of Earth to the frontiers of the cosmos."
–Stanislaw Lem, Solaris

"... they couldn't do it the easy way, so they cut through the problem and made another option."
–Evan Currie, Odyssey One

Abstract

This paper presents a prototype of a system for automated observations of flora and fauna in the Arctic. Currently applied methods of observation depend mostly on systems (usually consisting of a camera unit, a motion detection sensor and a memory card) that are left unattended in remote locations during extended periods of data gathering. The main problem with such approach is that no remote control or monitoring is available for those systems and manual inspection on site is not performed as often as it would be required for ensuring continuous operation. If a system fails, there is no way of detecting it, let alone fixing the issue or performing a reboot. Exposed to challenging environmental conditions of the Arctic and prone to problems such as power loss, hardware malfunction or inappropriate initial configuration, the systems have high probability of failing without it being noticed. In such cases, all several-months worth of data might be lost or never even recorded.

The solution presented in this paper intends to address the above issues by extending the functionality of an observation system with long range communication, self-monitoring and power saving capabilities. Proposed architecture allows for constant monitoring of system's health status and reporting it, together with sensor readings, via a remote gateway to the backend application. The system's designed uses IoT modules, which give it good extensibility properties if need for incorporating additional sensor types arises. The paper describes also the prototype implementation and the results of experiments performed.

The main focus of system test scenarios was on energy consumption, efficiency of data gathering and wireless communication capabilities. Currently the most serious concern identified for the system is its high energy demand. Experiments with different approaches to reducing the energy demand were conducted and presented in this paper. A satisfactory method of reducing energy demand is yet to be found, but some propositions have already been presented in the Future Work section, based on experiences with the developed prototype.

The proposed system proved capable of performing the additional functionalities intended for it. As a prototype, it still has room for refining and introducing

improvements (such as incorporating an animal recognition system into it), but already in the current state of research it is compelling, that the idea of developing an efficient and highly dedicated system for automated observations in the Arctic is sound, and the goal is achievable. We hope that this paper provides a solid base for it and sets a starting point for conducting further work on more robust approaches to environmental data collection in the Arctic regions.

Acknowledgements

I would like to thank the following people:

- My advisor, Professor Otto Anshus, for the idea of an Edge Command-Control-Communication System for Arctic Observatories, guidance and input throughout the process of writing this dissertation.

- My co-advisor, Associate Professor John Markus Bjørndalen, for many useful discussions, encouragement when I needed it and useful guidance.

- Jan Fuglesteg, for all his assistance with the practicalities involved in taking a Master's degree at the Arctic University of Tromsø.

- My parents, for all their support and endless faith in me and my ideas.

Contents

Abstract **iii**

Acknowledgements **v**

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
1.1 Problem statement 2
1.2 Contributions 3

2 Hardware Platforms **5**
2.1 Hardware selection 5
2.1.1 Support device 7
2.2 Communication 7

3 Architecture and Design **9**
3.1 Observation unit 10
3.1.1 Main device 10
3.1.2 Support device 11
3.2 Gateway 11
3.3 Backend 13

4 Implementation **15**
4.1 Observation unit 15
4.1.1 Main device 16
4.1.2 Support device 20
4.2 Gateway 21
4.3 Backend 22

5 Experiments **25**
5.1 Experimental setup 26
5.1.1 Energy-consumption reference for observation unit . 26

5.1.2 Gateway 26
5.1.3 Power consumption measurement instrument 27
5.2 Extreme load scenario 27
5.3 Medium load scenario 28
5.4 LoRa message rate scenario 28
5.5 Execution times scenario 29

6 Results **31**
6.1 Extreme load scenario 31
6.1.1 Amount of collected data 32
6.1.2 Effective LoRa message output 32
6.1.3 Data captured by Camera sensor 34
6.2 Medium load scenario 34
6.2.1 Amount of collected data 35
6.2.2 Effective LoRa message output 36
6.2.3 Data captured by Camera sensor 37
6.3 LoRa message rate scenario 37
6.4 Execution times scenario 38

7 Discussion **41**
7.1 Observation unit 41
7.1.1 Configuration 42
7.1.2 Camera module 42
7.1.3 Real Time Clock issues 43
7.2 Gateway 43
7.3 Backend 44
7.4 Prototype limitations 44
7.5 Challenges 45
7.6 Lessons Learned 45

8 Summary **47**

9 Future Work **49**

Bibliography **51**

Appendices **53**

A Observation Unit Usage **55**
A.1 Arduino Pro Mini 55
A.2 Raspberry PI 3 55

B LoRa Gateway usage **57**
B.1 AWS IoT 57

B.2 Raspberry PI 3 58

List of Figures

2.1 The LoRa star topology. 8

3.1 The EC^3 system architecture. 9
3.2 The observation unit's main components. 10
3.3 The main device's functionality. 11
3.4 Gateway architecture. 12
3.5 Gateway design. 12

4.1 The observation unit. 16
4.2 Visualization of simplified motion detection concept. 17
4.3 The LoRa module application flow. 18
4.4 Connections between Observation unit's devices. 20
4.5 Gateway applications flow. 22
4.6 Backend application flow. 23

5.1 Time measuring steps in LoRa Bandwidth scenario. 29

6.1 Energy consumption in extreme load scenario. 32
6.2 Data collected from sensors in extreme load scenario. 33
6.3 LoRa messages sent in extreme load scenario. 33
6.4 Number of captured images and occupied storage in extreme load scenario. 34
6.5 Energy consumption in medium load scenario. 35
6.6 Data collected from sensors in medium load scenario. 36
6.7 LoRa messages sent in medium load scenario. 36
6.8 Number of captured images and occupied storage in the medium load scenario. 37
6.9 Time required to send LoRa messages. 38
6.10 Tasks execution times. 39

List of Tables

2.1 Microcontrollers' features comparison. 6

4.1 Mapping of LoRa Bee pin connections. 17
4.2 Mapping of DHT11 pin connections. 19
4.3 Mapping of relay and support device pin connections. 19
4.4 Mapping of RTC DS1302 pin connections. 21

5.1 List of tasks executed during extreme load scenario. 27
5.2 List of tasks executed during medium load scenario. 28
5.3 List of tasks in execution times scenario. 29

1

Introduction

Arctic region observations made by Climate-ecological Observatory for Arctic Tundra (COAT) are usually conducted using several types of sensors (automatic cameras, microphones) to monitor wildlife and environmental factors. Each device gathers data and stores it on an internal memory for further processing. However, experiences with equipment placed in the wild showed, that several-months worth of data can be lost due to inappropriate device configuration (as stated by COAT workers). This is especially true in hardly accessible, Arctic regions.

Currently used cameras have limited functionality and, once configured, do not provide anything more than just images stored on a flash card. In order to prevent the loss of important data, observations of the Arctic tundra require a system which can report collected data and health of the devices, can provide a way to recover from most common types of failures, has self-aware configuration mechanism and allows new configurations to be remotely delivered to the devices.

Digital cameras placed in the wild are not equipped with any communication modules because, by design, those devices are closed for further extension. This problem can be addressed by using custom IoT (Internet of Things) hardware, which is extendable and allows specific customization to be made in order to achieve a more robust and configurable solution.

Several types of IoT sensor systems currently exist, such as Vicotee Njord[1],

SensorTag 2[2] or Waspmote[3]. All of these systems can monitor the environment with multiple available sensors and communicate with remote services while being low-power-consumption devices. However, some of these systems are still in a development stage and they can not be tested yet (Vicotee Njord), others cannot be extended with specific sensor types because of a closed design (SensorTag2) or due to custom module interface (Waspmote).

On the other hand, there is a variety of IoT boards designed for general-purpose tasks, for example, Raspberry PI[4] and Arduino[5]. These microcontrollers can be extended with additional modules thanks to common interfaces, such a Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C) or Camera Serial Interface (CSI). A variety of peripherals, a big community of users and open-source code and examples make them an efficient platform for prototyping a customized solution for Arctic Observatories.

1.1 Problem statement

This thesis is looking to design and build a prototype of a system for observing flora and fauna of the Arctic tundra. The main purpose of the project is to extend camera functionality beyond taking images of animals in the wild environment. The developed system needs to report various types of events to a remote server, capture images when motion is detected and collect other sensors' data. The development process can be split into several phases: defining functionality and data flow, selecting hardware for building the prototype, implementing firmware and conducting experiments according to previously specified scenarios. Those steps require performing a full development cycle, including architecture, design, implementation and experiment phases respectively.

As the purpose of the system's prototype is to extend the basic functionality beyond a camera, a compatible microcontroller module needs to be introduced into the system to bind and control additional communication and data flow. That implies examining various types of IoT hardware in order to select a set of mutually compatible components. A robust solution requires also an easy way of implementing the designed functionality using a high-level programming language like python and C++. A power consumption factor needs to be considered as well, because the device is expected to survive in the Arctic environment for as long as one year. Finally, the system proposed for COAT should report its health status and events of data collection.

1.2 Contributions

This thesis contributes to the Arctic observations research with the following:

- architecture and design of the sensors system prototype to monitor the Arctic environment, with the ability to store and report different types of data and operate even if there is no connection to the remote services,
- working EC^3 System prototype,
- an analysis of how the prototype performs during weeks of tests and how much energy it consumes,
- definition of most critical considerations to be made when developing a system for Arctic observations (such as determining a balanced system configuration for effective data collection, communication and power consumption) and providing preliminary solutions as basis for further development,
- thoughts on future work and further improvements to the current prototype.

2

Hardware Platforms

This chapter presents a research on hardware components and why particular models were chosen for EC^3 System implementation. Additionally, communication considerations were presented, to provide for an effective exchange of information between the devices.

2.1 Hardware selection

It was difficult to choose the hardware for this project because of a large variety of IoT boards available on the market, all differing greatly between one another. In practice, it is necessary to have them physically available in order to be able to compare their features. However, not many of the boards were actually at our disposal beforehand. This led to a selection of the boards based only on their specification. The conducted hardware examination focused on several characteristics, such as possibility of a dynamic configuration, power consumption, available storage, memory size, and types and numbers of connections for peripherals. The following devices were examined: Nordic Semiconductor NRF52[6][1], STM Nucleo L476RG[7], Arduino UNO and Pro Mini, and Raspberry PI 3. The comparison of the boards and the features required in the project is presented in Table 2.1. It was determined that only Raspberry PI 3 could fulfill the basic functionality of the system (image capture, SD card

1. Thanks to Telenor ASA, which delivered the board

storage) planned in the project. That is because it is the only microcontroller in this comparison with a built-in camera connection socket and an SD card slot, that are both needed for capturing and saving images.

		Microcontroller						
		Raspberry PI 3	NRF52	STM_{32} L476RG	Arduino UNO	Arduino Pro Mini		
Feature	SD Card slot	Yes	No*	No*	No*	No*	**Design requirements**	Yes
	Camera interface	Yes	No*	No*	No*	No*		Yes
	Sleep mode	No	Yes	Yes	Yes	Yes		Yes
	Measured power consumption	200 mA	4 mA	40 mA	47 mA	12 mA		***
	CPU speed	1,2 GHz	64 MHz	80 MHz	16 MHz	16 MHz		-
	Number of cores	4	1	1	1	1		-
	RAM size	1 GB	64 kB	-	-	-		-
	SRAM size	-	-	128 kB	2 kB	2 kB		-
	FLASH size	-	512 kB	1 MB	32 kB	32 kB		-
	SPI	Yes	Yes	Yes	Yes	Yes		Yes
	I2C	Yes	Yes	Yes	Yes	Yes		Yes
	Number of I/O pins	40	32	51	20	20		****
	Dynamic configuration**	Yes	No	No	No	No		Yes

* extension board required
** or remote configuration upload
*** less is better
**** more allows to connect more peripherals

Table 2.1: Microcontrollers' features comparison.

However, since Raspberry PI 3 does not have a sleep mode[2], and therefore drains around 200 mA when being idle (as presented in Figure 6.1), there was a need to turn it off to save power. It introduced an additional problem of turning the device back on, which had to be done by an auxiliary board (called a ***support device*** in this work). That kind of device allowed to build a more flexible solution, because it provided an additional source of information for

2. A state, in which a device is not functional but is able to recover within short time, similar to the shutdown. The difference, however is, that device saves its state and could be fully operational in a quick time, without full booting

the Raspberry PI 3 (***main device***). In this project, the ***main device*** and the ***support device*** form the ***observation unit*** (*OU*) embedded computer.

A similar purpose hardware, with sleep mode, already exists and is available on the market as Sleepy PI 2[8]. It could be used as an external power supply for Raspberry PI and allow to wake it up. Unfortunately, Sleepy PI was not available to us in the frame of this project and it was not used in the presented system prototype.

2.1.1 Support device

A *support device* must be an extremely low-powered element because it is turned on all the time. It could be used with a connection to a motion sensor to quickly turn on the ***main device*** when a motion is detected. Therefore, Nucleo L476RG does not meet these requirements, since it consumes 40mA. For NRF52 and Arduino Pro Mini it is 5mA and 12mA respectively. All devices are able to communicate with the ***main device*** via I2C bus.

Arduino Pro Mini was selected for this project because it has an Atmel ATmega328P [9] microcontroller, which can be used on a custom PCB (as detailed in chapter 9) designed for Raspberry PI. A Sleepy PI 2[8] extension board uses the same [9] chipset, which gives good prognosis for the likelihood of developing a working solution. Furthermore, Arduino is a low-power-consumption board, which makes it suitable for the prototype. It is possible to reduce its demand for energy even more, to less than 1 mA[10], but since it requires introducing physical modifications to the board itself, it was not introduced in this project. Finally, there are many examples of source code and designed circuits available for Arduino boards, which was of significant importance in this project.

2.2 Communication

The communication between the sensors system and a remote server does not need to be permanent (like in TCP for example). It means that the sensor system should be able to reach the other side of the connection at least from time to time. As the other parts of the prototype, the communication module should be low-powered and offer long range coverage, even at the expanse of low bandwidth. It is crucial for COAT to receive any kind of information from sensors, even meta-data only, therefore long range is more important characteristic than the bandwidth.

The solution which meets the described characteristics was LoRa[11] technology. It uses a spread-spectrum modulation technique to send data at low-rate on a long range, using both 868 and 900 MHz ISM bands. It uses a star topology (presented in Figure 2.1), in which IoT devices are sending information to the gateway connected to the Internet and then such data is forwarded to the remote server (called ***backend***). It consumes only 10 mA when receiving and 40 mA when transmitting data. Moreover the LoRa ISM bands are opened for private use (license-free) in Europe without any cost. Taking into account the low power consumption and long range coverage, the LoRa communication modules were selected for the EC^3 System. Some documents, like a *LoRaWAN Range Testing*[12] prepared by Laird company, claim LoRa range to be around 15-20 km following a line of sight. It could be extended even more if the gateway is installed in a high location above the ground.

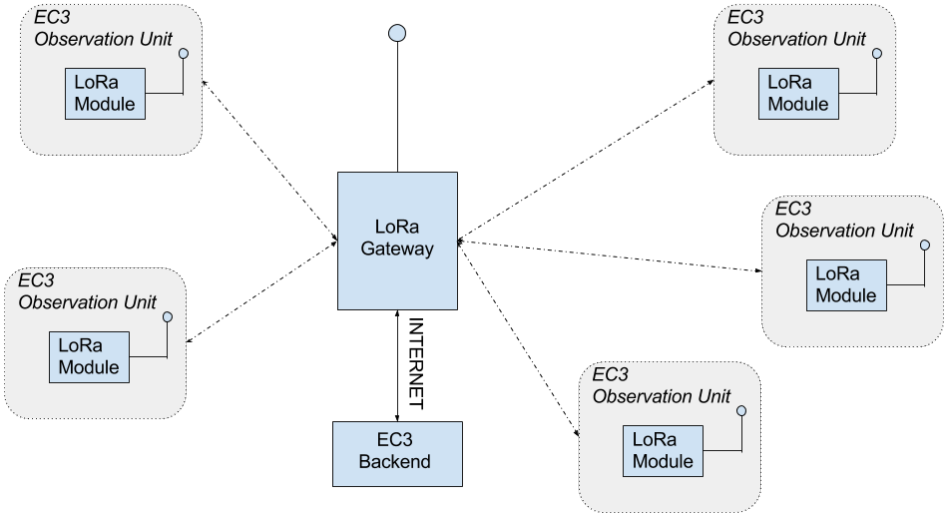


Figure 2.1: The LoRa star topology.

3

Architecture and Design

The EC^3 is a distributed observations system for the Arctic environment, composed of an *observation unit*, a *gateway* and *backend* services. The *OU* collects information about the surrounding environment and sends meta-data to the *gateway* using a long-range link. The *gateway* forwards the meta-data to the *backend* services for further processing (decryption, storage). The overview of the EC^3 system architecture is presented in Figure 3.1.

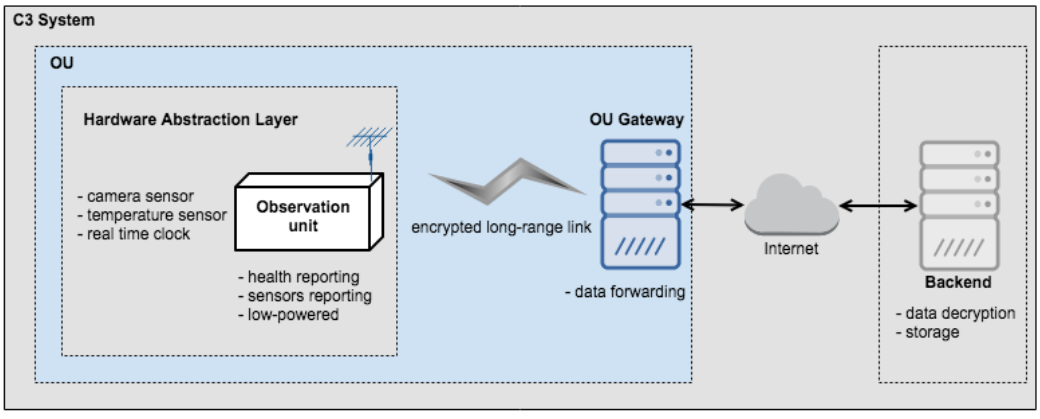


Figure 3.1: The EC^3 system architecture.

3.1 Observation unit

The observation unit is composed of two boards (the *main device* and the *support device*) connected via I2C bus. The *main device* is the heart of the system and its main purpose is to collect and send information such as sensor readings and own health status. The *support device* is used to power on the *main device* on scheduled time and supply it with the current time value from Real Time Clock (RTC) because the *main device* does not have a hardware clock. The COAT needs the sensor readings to be marked with the current time, which is later used in scientific analysis. It means, that every time the device is off, its software clock is stopped and it needs to be updated to the current time value on the next start-up. The *support device* works as a slave and executes operations only when requested. The overview of the *OU* design is depicted in Figure 3.2.

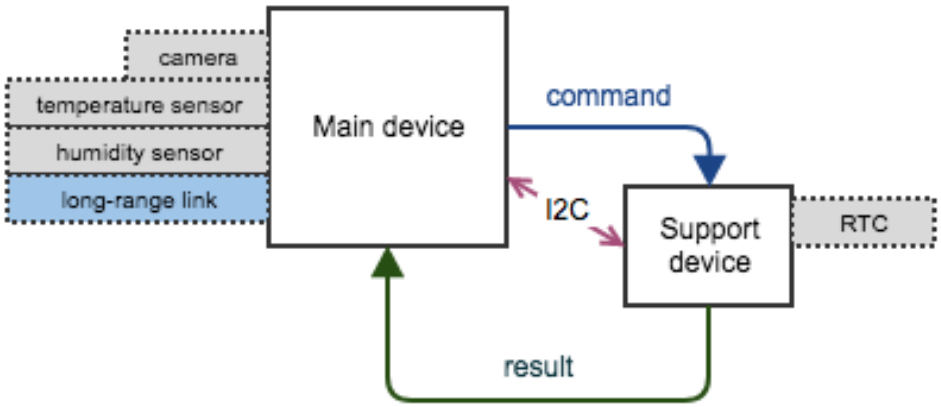


Figure 3.2: The observation unit's main components.

3.1.1 Main device

This device is designed to work in a periodic fashion. Figure 3.3 presents the *main device*'s functionality. Once turned on, the board is collecting data (sensor readings, health reports), capturing images if motion is detected, encrypting and sending information to the *gateway*. Then, it is turned off for a specified amount of time. The camera sensor is responsible for motion detection but only when the device is turned on. The data is delivered to the *gateway* using the LoRa module.

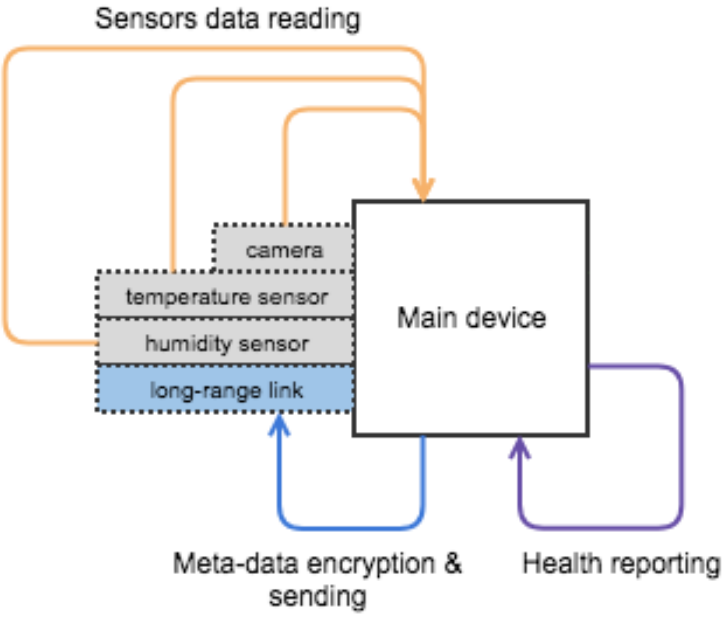


Figure 3.3: The main device's functionality.

3.1.2 Support device

The *support device* operates continuously and conducts tasks requested by the *main device*. The following list of operations was designed:

set_wakeup_time defines when the *support device* turns on the *main device*,

set_RTC_clock initializes the RTC module with the current time requested from API[28],

get_RTC_clock returns current time from the RTC module.

Communication between the *main device* and the *support device* is designed to use I2C bus.

3.2 Gateway

The *gateway* architecture depicted in Figure 3.4 consists of a communication module, data collector and data forwarder. The first is used to communicate with the *OU* and pass all data to the data collector process, which saves the data in a database. The data forwarder process selects records from the database

and sends them to the *backend* services.

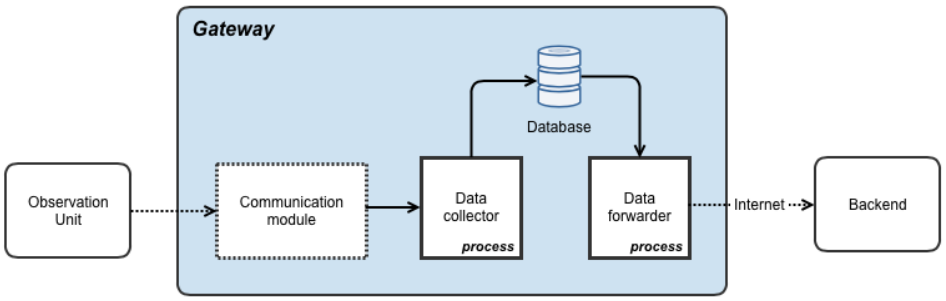


Figure 3.4: Gateway architecture.

The *gateway* is designed to be a transparent component in the EC^3 system. It means that this device receives an encrypted message but is not supposed to decrypt it. Instead, the encrypted meta-data is forwarded to the *backend* services. In that way, any other public LoRa gateway might be used for delivering data, without a risk of information leak. It allows mixing the EC^3 system infrastructure with another operator's LoRa network. The following list of functionalities was designed for the *gateway*:

- receiving and acknowledging messages from the *OU*,
- storing the messages in a database,
- pushing the messages to cloud services.

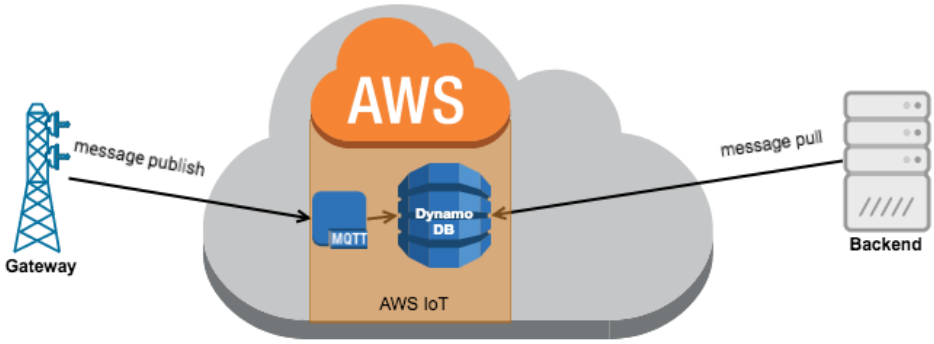


Figure 3.5: Gateway design.

In this project, the *gateway* is designed to use MQTT[14] protocol to deliver data to AWS IoT[15] service, where the received message in stored in DynamoDB[16] database, as depicted in Figure 3.5. It prevents duplication of messages and ensures that the received data is stored even if the *backend* services are not available at the time of information forwarding.

3.3 Backend

The EC[3] *backend* service receives and decrypts data from the *observation unit*. In the proposed design the *backend* downloads data from AWS IoT service, requesting LoRa messages stored in DynamoDB. However, neither GUI nor any other interface to browse and analyse data was not designed, since it is not a part of this thesis.

4

Implementation

The EC^3 system prototype is build on Raspberry PI and Arduino hardware. The software implementation was done using C++ and Python, and it works on Raspbian Jessie[17] (for the *OU* and the *gateway*) and on any linux distribution with support of Python >= 2.7 (for the *backend* services).

4.1 Observation unit

The *observation unit* is based on Raspberry Pi 3B (the *main device*) and Arduino Pro Mini (the *support device*). Both operate on 5 volts of input voltage, which allows to use the same power source for them without additional logic-level converters[18]. Detailed unit circuit is presented in Figure 4.1, which corresponds to the real prototype built on a breadboard.

The following subsections describe all parts of the *OU* prototype, including both hardware and software implementations.

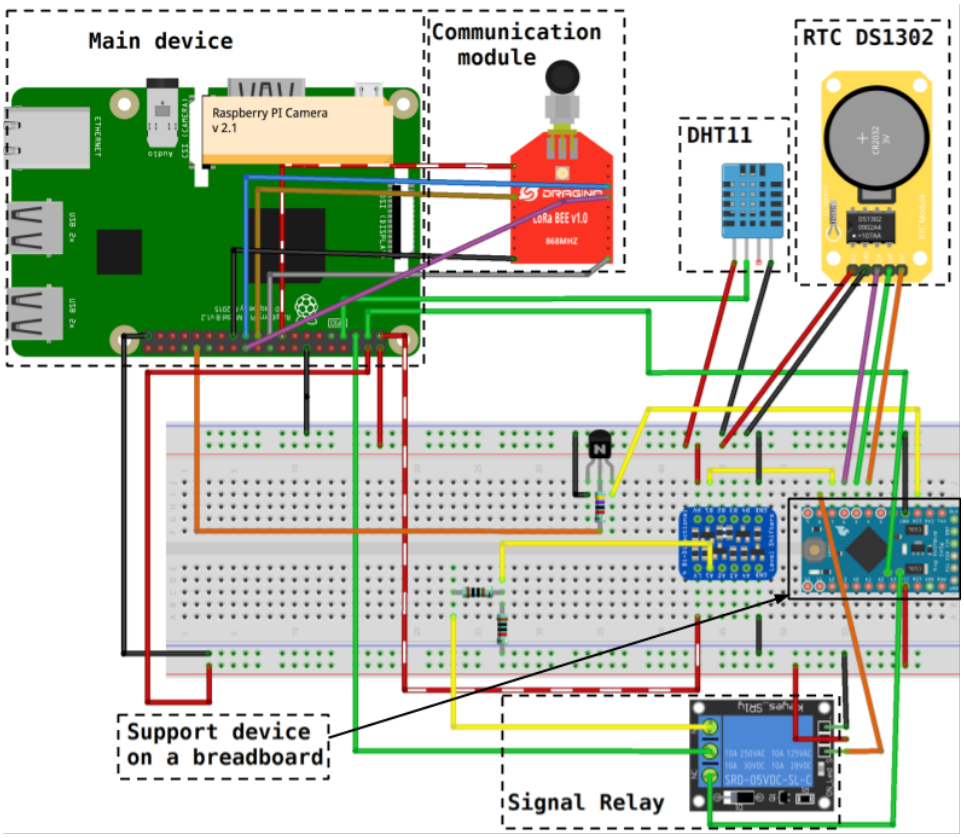


Figure 4.1: The observation unit.

4.1.1 Main device

Camera sensor

The camera module used in the prototype is *Raspberry Pi Camera Module v2*[19], which comprises an 8-megapixel sensor. The Python application written for this purpose is a modified version of *pi-timolo*[20] (written by Claude Pageau). It is a *picamera*[21] module capable, among others, of detecting motion and taking images even in a low-light environment. Its role is to detect motion and take a picture when it occurs. The detection algorithm compares the area seen by the camera in real time with the past frame and if the compared fragments of the view (small groups of pixels) differ between each other, it takes a picture. A simplified concept of the algorithm is depicted in Figure 4.2.

The camera module is configured to capture images in resolution of 1024 x 768 pixels to save device's storage space. The captured images are not sent by the LoRa module. Instead, when photos are taken, only an information is send to

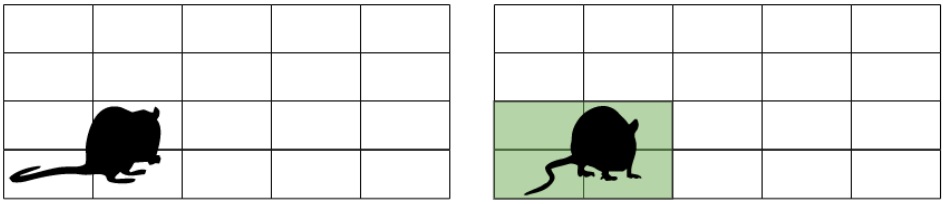


Figure 4.2: Visualization of simplified motion detection concept.

the *gateway* about detecting a motion.

Communication module

As mentioned previously in chapter 2.2, the communication module selected for this project is LoRa module, specifically Dragino LoRa Bee[22], which is based on the Semtech SX1276[23] transceiver chip. The module is connected via SPI to the *main device*, using the pin mapping presented in Table 4.1.

Main device		LoRa Bee	
Function	Pin	Pin	Function
+3.3V	17	1	+3.3V
MOSI (SPI)	19	11	MOSI (SPI)
MISO (SPI)	21	4	MISO (SPI)
SCLK (SPI)	23	18	SCK (SPI)
CEO_N (SPI)	24	17	NSS (SPI)
GND	25	10	GND

Table 4.1: Mapping of LoRa Bee pin connections.

A C++ code for sending data from the *observation unit* to the *gateway* is based on *LowCostLoRaGw*[24] implementation made by Congduc Pham from the University of Pau in France. The code selects information from sqlite3 database stored on the *main device*, where the *OU* saves all sensors' readings and its health status. The data is then sent to the *gateway*. The program's flow is depicted in Figure 4.3.

The LoRa module operates in 868 MHz band frequency and it is configured to use 125 kHz Bandwidth (BW), Spreading Factor (SF) 12 and Command Rate (CR) 4/5 for the maximum range. This configuration is optimal in the terms of long range communication, as specified in *Semtech LoRa Modem Design Guide*[13].

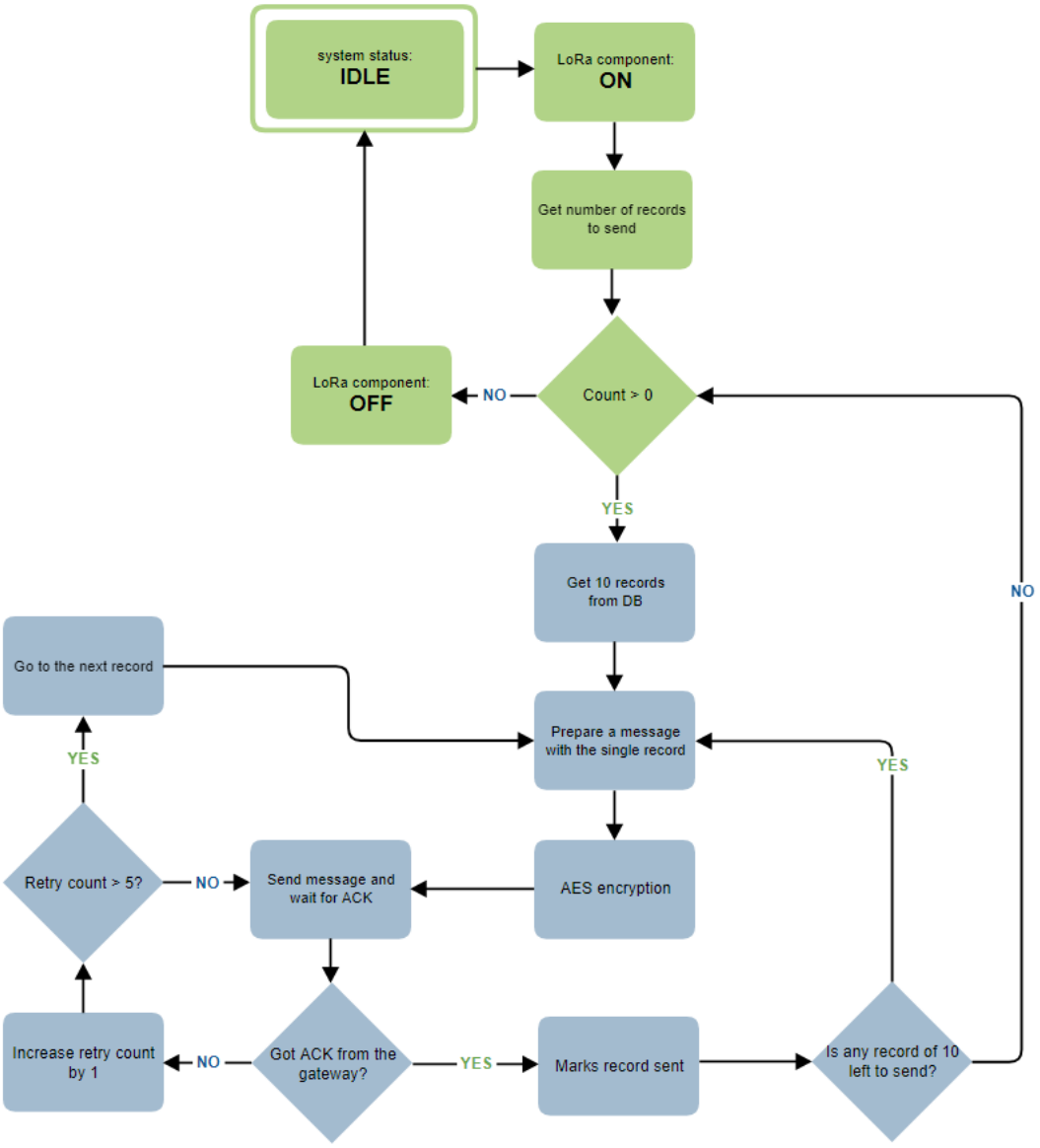


Figure 4.3: The LoRa module application flow.

Humidity and temperature sensor

The prototype is equipped with DHT11[25] humidity and temperature sensor. It is connected with a single bus (Single-Wire Two-Way) to the *main device*, which means that a single connection is used to request and receive 40-bits of data from the sensor. Connections between the device and the module are presented in Table 4.2. The implementation is written in Python and uses *Adafruit Python DHT Sensor Library*[26] to read the sensor's output. The value

returned by the sensor is stored in sqlite3 database. The newest value is sent by the communication module to the *gateway*, when possible.

Main device		DHT11	
Function	Pin	Pin	Function
+5V	2	1	+5V
GPIO_GCLOCK	7	2	Signal
		3	Not connected
GND	14	4	GND

Table 4.2: Mapping of DHT11 pin connections.

Connection to the Support device

The *main device* is connected to the *support device* via I2C bus. However, since one of the Raspberry PI 3B I2C pins is also used to turn this device on, a signal switch (Keys SRD-05VDC-SL-C relay in this case) is required. It allows to share the same pin by two separate circuits and switch the pin to the circuit which is used at the moment. The default circuit (normally closed) is the one which uses I2C bus for communication between devices. When the *main device* is turned off, after a specified amount of time, the *support device* turns it on by switching circuits for 100 milliseconds. The operation of turning on the device requires a short circuit. Therefore, this short signal is enough, and it should not be longer in order not to damage the device. Table 4.3 presents connections between the *main device* and the *support device*, including signal switch.

Main device		relay	Support device	
Function	Pin	Pin name	Pin	Function
SDA1 (I2C)	5		A4	SDA (I2C)
SCL1 (I2C)	3	C		
		NO	D7	*turn on*
		NC	A5	SCL (I2C)
		VCC	VCC	+5V
		GND	GND	GND
		In	D8	*switch signal*

Table 4.3: Mapping of relay and support device pin connections.

An additional element used in the prototype to make connections between devices is a logic-level converter. The reason it is needed is because Raspberry PI 3 supplies voltage of 3V for every pin, while for Arduino Pro Mini it is 5V. This component prevents damage of pins when there is a difference in voltage between connected devices. Figure 4.4 presents both circuits used in the project.

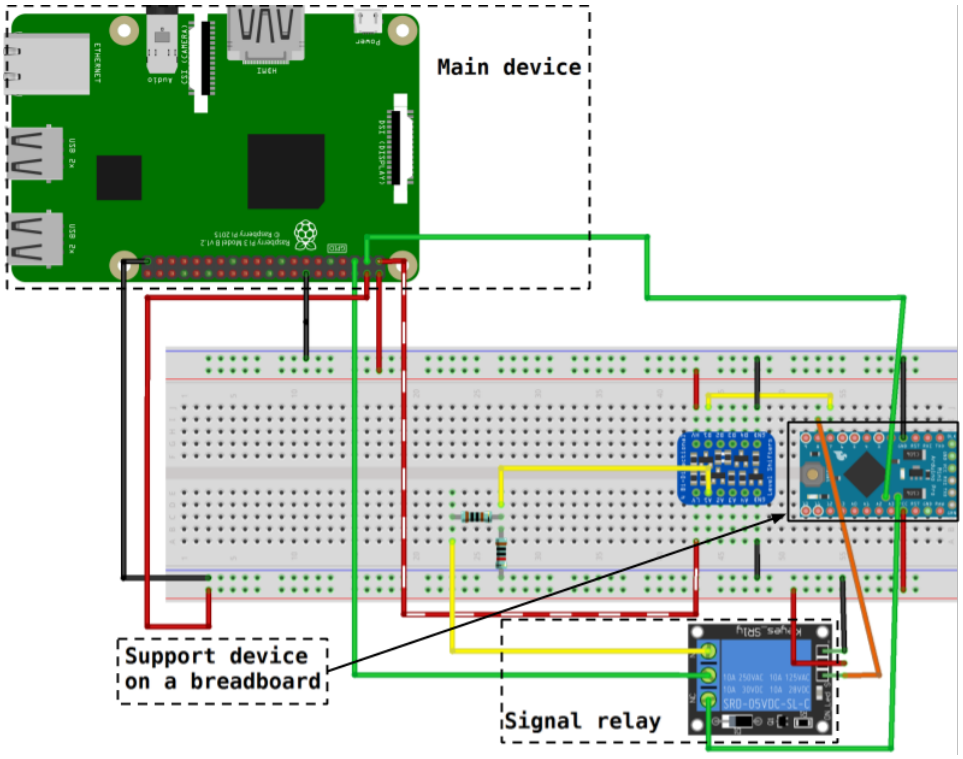


Figure 4.4: Connections between Observation unit's devices.

The I2C bus communication between devices works using the following schema: if the *main device* requests data, it sends an operation code to the *support device*, and in return it receives the same code as the confirmation. Then, it requests and receives the size of the requested data (in bytes) and finally, it receives the data itself, byte by byte. It works similarly in the other direction, when the *main device* sends data to the *support device*. When an error occurs during the communication, since it is unknown when the connection was broken, the *support device* is restarted by the *main device*. This solution prevents conflicting situations, in which one of the devices, Arduino or Raspberry PI, would be waiting for data, while the other would be in the middle of another operation.

4.1.2 Support device

Arduino Pro Mini 5V was selected as the *support device* for the EC³ system prototype. It runs on only 12 mA of current and its firmware is written in C++. Therefore, every time the device's code is changed, it requires to be

compiled on an external host and flashed to the board using an FTDI USB serial cable.

Real Time Clock

Since none of the boards used in the project have a build-in hardware clock, the *support device* was equipped with the RTC module, specifically DS1302. It has an additional battery power source, which starts its *ticking* as soon as the clock is initialized. In the current implementation of the EC^3 system, the RTC module is used by the *support device* to turn on the *main device* at a specified time and to pass the current time for it. The first initialization is done by the *main device*, using I2C communication with the *support device* and *timezonedb.com* API. The connections between the *support device* and RTC clock are presented in Table 4.4.

Support device		DS1302	
Function	Pin	Pin	Function
GND	GND	1	GND
+5V	VCC	2	VCC
	D6	3	CLK
	D5	4	DAT
	D4	5	RST

Table 4.4: Mapping of RTC DS1302 pin connections.

4.2 Gateway

Raspberry PI 3B is used as the *gateway* and uses the same LoRa module with the same device's pins (Table 4.1) as the *observation unit*. To receive LoRa messages, a modified *LowCostLoRaGw*[24] library is used. It is executed as soon as the device's operating system is up and running and it remains in the background until the system's shutdown. All received messages are stored in an sqlite3 database and published in AWS MQTT using a separate python application. The *gateway* applications flow is depicted in Figure 4.5.

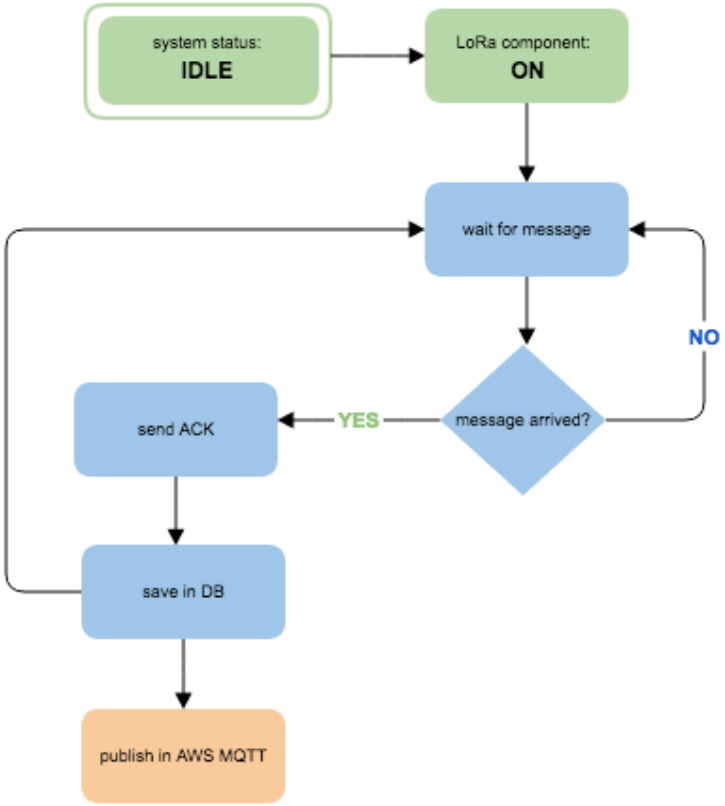


Figure 4.5: Gateway applications flow.

4.3 Backend

In the current implementation, the *backend* might be any host capable of running a python script in which the data is requested from AWS DynamoDB database. The code marks every received record as downloaded and it is trying to decrypt it. If the data was sent by the *OU*, then the *backend* examines Message Integrity Check (MIC) for the received record (the last 4 bytes) and decrypts the data using AES keys associated to the device. The full application flow is depicted in Figure 4.6.

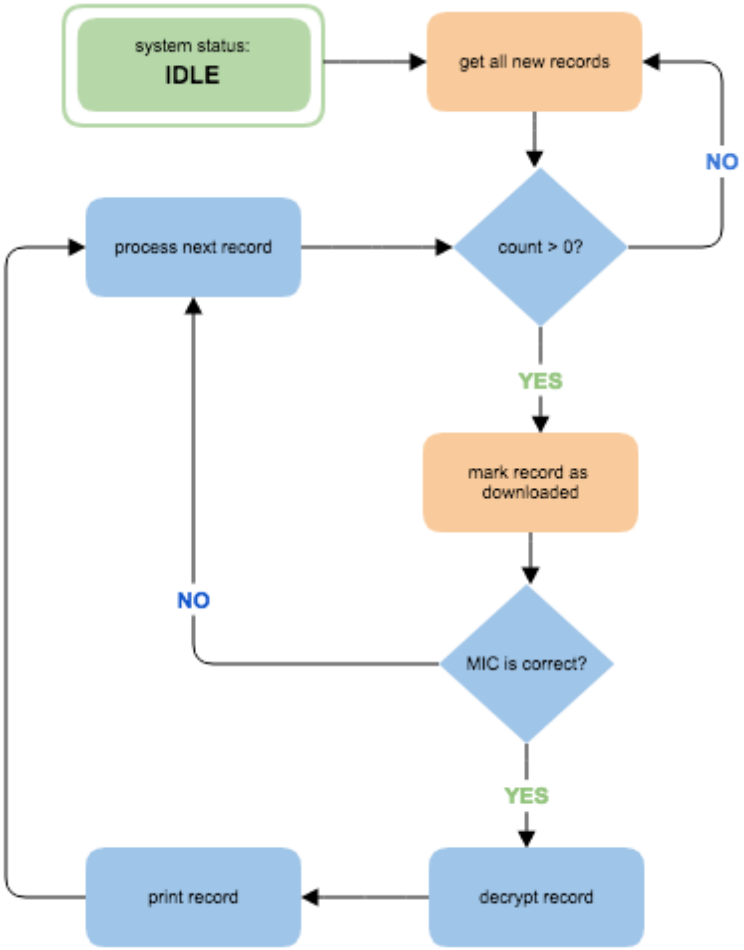


Figure 4.6: Backend application flow.

5

Experiments

All experiments were conducted on the *observation unit* as the element in the EC^3 System which will be placed in the Arctic environment, where the access to the device is limited and the possibility of failure is highest. In order to determine how well the *OU* performed in terms of power-consumption and task-execution times, several experiment scenarios were designed and performed, revealing some of the device's capabilities and limitations. Moreover, the tests and the received results have shed some light on the configuration and components that need to be adjusted in order to keep the device operational. Several software fixes were made in order to obtain the prototype stability during these tests.

Other parts of the system, such as the *gateway* and the *backend*, are powered up by regular power grid and constantly connected to the Internet, so their failure, if it occurs, could be easily detected virtually instantaneously. Since those parts of the system are not deployed in remote locations and are available for inspection and maintenance, unlike the *OU*, ensuring their reliability through extensive testing was considered out of the scope of this project.

Each of the following sections contains a description of a test scenario and the conditions in which every test was conducted. The obtained results and the conclusions are presented in chapter 6.

5.1 Experimental setup

The configuration of the *observation unit* differs for each experiment and is detailed in the following sections. However, the configuration of some system components, such as the *gateway*, is always the same and thus it is specified only once. This section includes the setup description for all common parts and tools used in the EC^3 System examination.

In the current implementation, the system does not allow for setting its parameters after being deployed. Changing parameter values for different scenarios is currently done by hardcoding them. That setup includes the following variables:

- how often data from the sensors is gathered,
- how often the device's health status is reported and written to database,
- how many messages are sent before the device is powered off.

For the purpose of conducting the tests, the camera unit was artificially forced to take photos every fixed amount of time specified in the scenario. In actual operations the image capture would be triggered by detecting a motion.

5.1.1 Energy-consumption reference for observation unit

Two reference energy consumption values were obtained for the *observation unit*. First, the measurement of the energy consumed was done for 24-hour period while the system was left in idle state. Then, the system was powered off completely for another 24 hours (since the prototype drains power even if the *main device* is turned off) and the corresponding energy consumption was measured again. For the idle state, the device's operating system was running in minimal mode, which means that only the default system tasks were occupying the CPU and the memory. It is a nominal state of the device when nothing from the user's space is running.

5.1.2 Gateway

The *gateway* was running in a continuous[1] mode, receiving LoRa messages and passing them to the AWS MQTT service. The device was placed in the

1. Once turned on, the *gateway* was waiting and receiving messages until this process was manually stopped.

same room as the *OU* with 1 meter of distance between them. The ambient temperature of the room varied between 18 to 23 Celsius degrees.

5.1.3 Power consumption measurement instrument

In the experiments where power consumption needed to be measured, *Mini USB Charger Doctor*[27] was used. This device is capable of detecting electric current consumption and voltage levels in the circuit connected to it via USB. Its specification states, that this *charger doctor* is able to measure a current within the range 0 A to 3 A (with +- 1% of error and minimum resolution of 10 mA) and time (only when a device is draining power) within range 0 to 99 hours. Therefore, it was a good choice for the long power-consumption experiments conducted in this project.

5.2 Extreme load scenario

The main purpose of this scenario was to determinate how much energy the *observation unit* consumes when all tasks defined for this scenario are executed in predefined intervals. The device was examined during 24 hours of testing for every interval case (the same for every task during the whole test) which gives 4 days of testing in total. In order to obtain the reference value for power consumption, the device was left in idle state for another 24 hours. Each task was running independently in endless loop with a sleeping time specified by interval length. The list of tasks and used intervals included in this scenario are presented in Table 5.1.

Task name	Idle-state Interval				System idle
	1s	10s	30s	60s	
1 x LoRa message sending	x	x	x	x	
1 x RTC value reading	x	x	x	x	
1 x DHT11 sensor's value reading	x	x	x	x	
1 x Camera image capture	x	x	x	x	
Test duration	24h	24h	24h	24h	24h

Table 5.1: List of tasks executed during extreme load scenario.

Taking advantage of executing the extreme load scenario, additional quantities were measured for logging purposes. Those were not connected to the energy consumption and included the number of data samples collected for all sensors and the number of LoRa messages sent.

5.3 Medium load scenario

Medium load scenario was a second power-consumption test for the *observation unit*. In this scenario the device was being turned on and off. As soon as the device was turned on and finished its booting sequence, it was executing the same tasks as described in section 5.2 and then it was turned off for specified time. This cycle was repeated in a loop of 24 hours in one test, and four such tests were performed – one for every interval. Tasks performed during this scenario and the lengths of power-off intervals are specified in Table 5.2.

Task name	Power off Interval				System idle
	1s	60s	30m	60m	
1 x LoRa message sending	x	x	x	x	
1 x RTC value reading	x	x	x	x	
1 x DHT11 sensor's value reading	x	x	x	x	
1 x Camera image capture	x	x	x	x	
Test duration	24h	24h	24h	24h	24h

Table 5.2: List of tasks executed during medium load scenario.

The same additional parameters as in the case of extreme load scenario were also measured during the execution of medium load scenario.

5.4 LoRa message rate scenario

The purpose of this scenario was to determine how many messages (with ACK) of size 130 bytes could be sent by the *observation unit* during a specified time period (1s, 10s, 30s, 60s). This test was conducted 5 times for every time period, in order to obtain the average values, and the time itself was measured in three steps. The first step was focused on the internal application time for every message sent separately, the second step took into account the number of seconds from the beginning of the application's execution until the end. The last step measured the external time of application's execution and was conducted using the linux commandline function called *time*. All three steps, presented in Figure 5.1, helped to determine how much time the *OU* needs to send a message, how long time the message preparation takes, and finally, what the total time of application execution is.

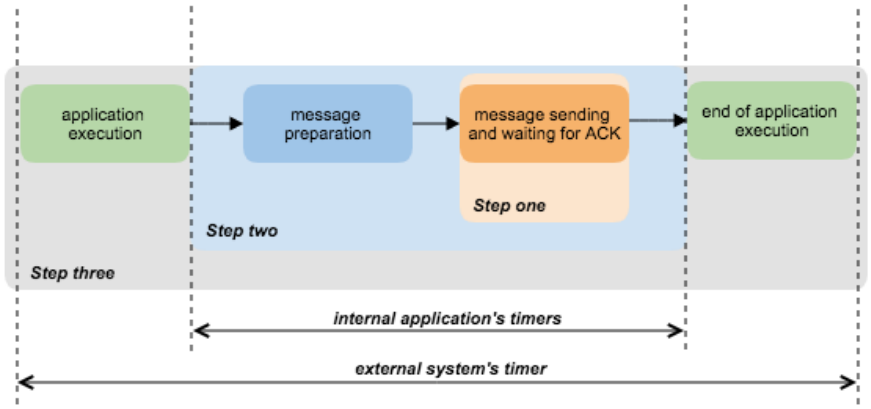


Figure 5.1: Time measuring steps in LoRa Bandwidth scenario.

5.5 Execution times scenario

The last scenario was focused on the time of a specific task execution. The time was measured using the linux commandline function *time*. In this test, to obtain the average value, every single task was repeated 99 times.

Task name	Number of repetitions	Time measurement method
Lora Message sending RTC value reading DHT11 sensor's value reading Camera image capture	99	linux *time* function
The device shutdown The device booting	5	*digital stopwatch*

Table 5.3: List of tasks in execution times scenario.

An exception to this rule was the measurement of the *observation unit* shutdown and booting times, which was done using a digital stopwatch (in case of booting, the timer was stopped when the device was ready to conduct user's space tasks). The list of tasks from this experiment is presented in Table 5.3.

6

Results

This chapter presents the results of the experiments conducted to measure the device's power consumption in various scenarios. Moreover, in several cases, additional metrics were collected and found useful for the analysis of the *observation unit*'s behavior. Those include: an analysis of the amount of sensors' data collected, an effective LoRa message output, and an influence of the captured images on the device's storage capacity. The additional metrics are described in sections 5.2 and 5.3.

6.1 Extreme load scenario

The main purpose of this scenario was to determine how much energy the *observation unit* consumes when tasks execution are interleaved with idle states. The results presented in Figure 6.1 show that the device's power demand proportional to the frequency of tasks executions. In the most intense test case the power consumption was 1,7 times bigger than in the idle state. The tasks execution frequency for which the power consumption was close to the one in the idle state resulted to be around once every minute. It suggests, that if the measurements are less frequent, then they do not have any significant influence on the power drainage by the device.

What is most interesting here, is the fact that even in the idle state the device consumes around 5 Ah a day. If it was to be left alone for a year, it would

require approximately 1825 Ah for the device to remain operative. This is a huge demand on energy, and to visualize this, a decent smartphone battery, which has a capacity around 2,5 Ah, would be exhausted by our device in half a day if it was just in its idle state doing nothing. Even if it was acceptable, the issue with the *OU* is that, unlike with a mobile phone, it is not possible to recharge the device's battery yet.

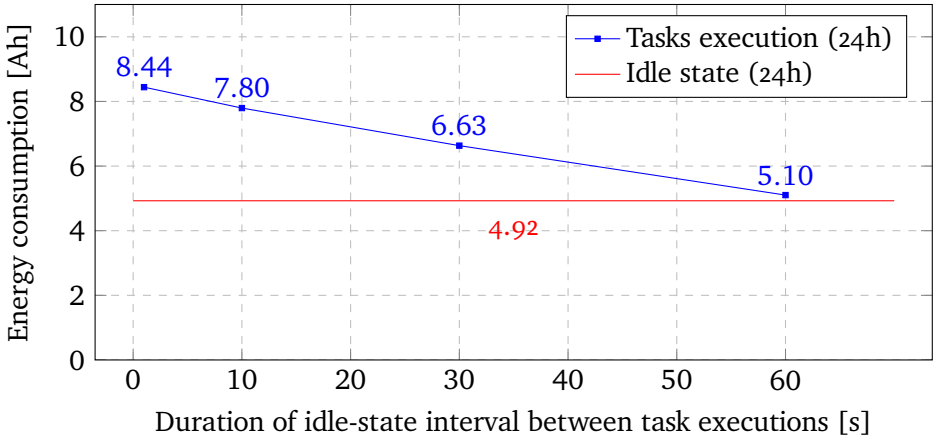


Figure 6.1: Energy consumption in extreme load scenario.

It is clear that, due to the energy demand, the device could not sustain collecting data as often as every second for prolonged period of time. In fact, the energy consumption is so high, that even keeping the device just powered on all the time, without executing any tasks, would pose a demand for energy that is difficult to satisfy with regular batteries.

6.1.1 Amount of collected data

Figure 6.2 depicts the numbers of sensors readings performed in the extreme load scenario as a function of idle-state intervals length. For the highest intensity 24-hour test it almost reached a hundred thousands records. It was not analyzed, however, which sensor collect the most of the data. The purpose of this metric is to present the *observation unit*'s performance in collecting data, which will be useful for adjusting the device's configuration.

6.1.2 Effective LoRa message output

One of the repeatedly executed tasks was sending the collected sensor's data to the *gateway*. Figure 6.3 presents how many LoRa messages were sent during 24 hours for each of four specified lengths of idle-state intervals. These metrics

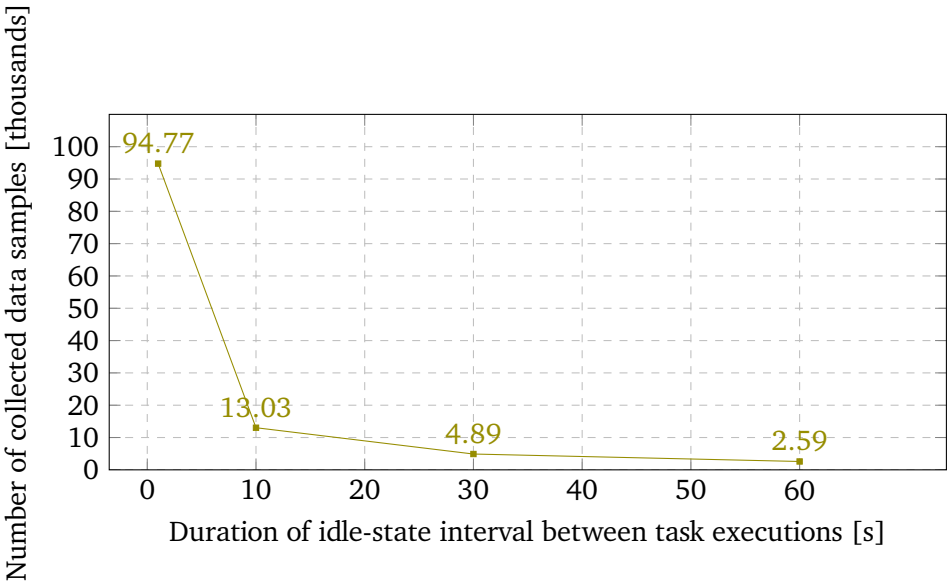


Figure 6.2: Data collected from sensors in extreme load scenario.

allow to estimate that a single LoRa message requires in total around 10 seconds to reach its destination and to get an acknowledgment, which was calculated by dividing the number of seconds in a whole day (24h) by the number of messages and subtracting the length of idle time between each message. However, these numbers are only an overview of the LoRa message sending time. Since the message preparation time and the LoRa module initialization time are unknown, a more accurate numbers collected from *LoRa message rate scenario* are presented in section 6.3.

Figure 6.3: LoRa messages sent in extreme load scenario.

6.1.3 Data captured by Camera sensor

As the device's storage is limited, it is important to know how many images in resolution 1024 x 768 can be captured by the camera sensor without overflowing the SD card and causing the unit's failure. Figure 6.4 presents the number of images and the corresponding size of the occupied storage for different time intervals between tasks execution. Based on the metrics, an average image size is 75 KB. In the experiment, the free storage capacity was limited to 13 GB, which would be enough to save almost 182 thousands of images. In a scenario where images are taken every 60 seconds, such storage would be sufficient for almost half a year of image recording.

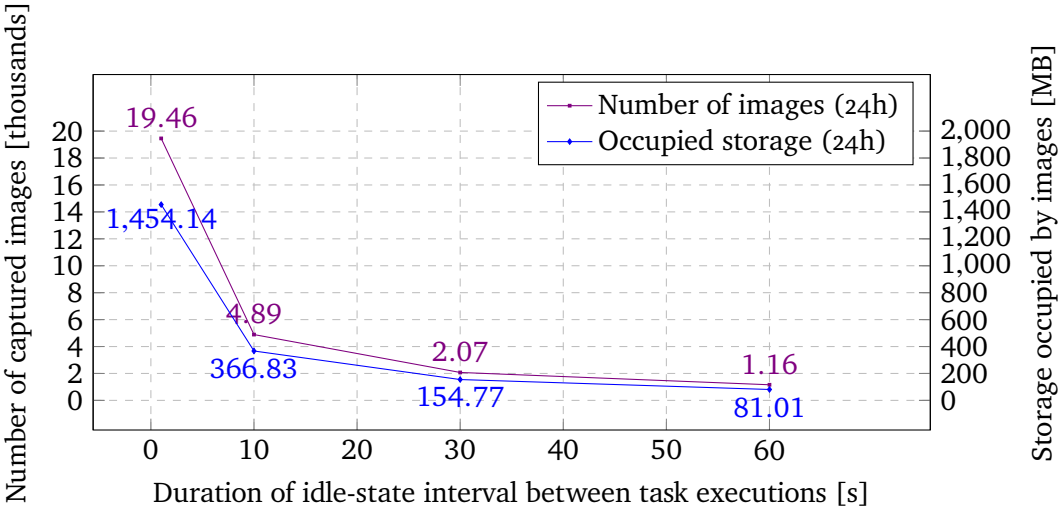


Figure 6.4: Number of captured images and occupied storage in extreme load scenario.

6.2 Medium load scenario

The results of this experiment could contribute to the system's configuration adjustments that are necessary for improving the *observation unit* power efficiency. Figure 6.5 shows the energy consumption levels for 24-hour-long periods of task executing, interrupted with shutdown states of various lengths (the lengths of shutdown states are marked on the horizontal axis of the figure). The highest energy consumption measured in this scenario (5.43 Ah obtained for 1-second long shutdown intervals) is comparable with the lowest energy consumption measured in the extreme load scenario (5.10 Ah obtained for 60-seconds long idle intervals), as presented in Figure 6.1. What needs to be taken into account in this scenario, is that there are additional delays connected to the device's shutdown (around 8 seconds) and the device's booting (ranging between 25

and 45 seconds), so the effective time intervals between tasks executions (during which the device is not operational) are longer than the shutdown intervals presented on the figures in this section. Taking into account the shutdown and booting times reported above, it can be estimated that for a 1-second shutdown interval, the actual effective non-operational time is around 34 - 54 seconds. Considering this, both the energy consumption levels (5.43 Ah versus 5.10 Ah) and the effective non-operational times (up to 54 seconds versus 1 minute) are comparable.

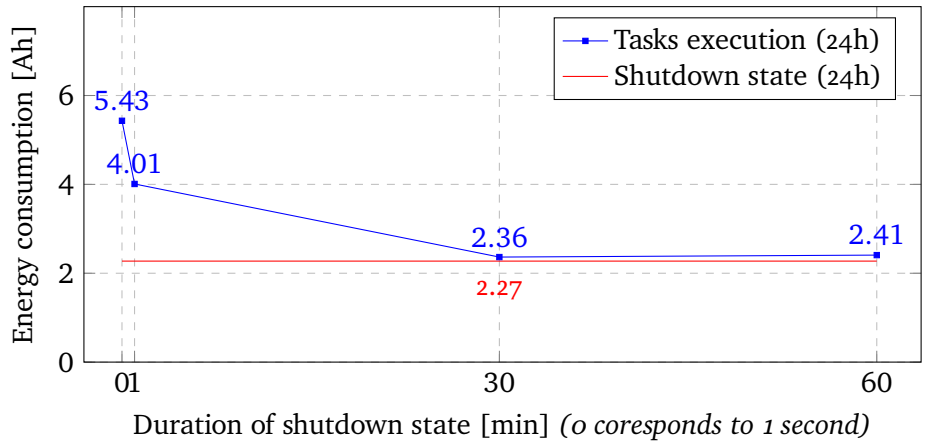


Figure 6.5: Energy consumption in medium load scenario.

The important result from this scenario is the energy consumption in the shutdown state, in which the unit still drains a lot of power. Since the *support device*'s power efficiency is estimated to be around 20 mA, it means that powered-off Raspberry PI 3 *requires* around 75 mA of energy. The value was calculated by dividing the reference (shutdown state) value by 24h and subtracting the *support device*'s power consumption. This is an issue which could be addressed by reducing the power supply to the *main device*, as described in chapter 9.

The power consumption for the 60-minute intervals case is slightly above the value for the 30-minute intervals case. Most probably it was caused by the variations in image capture times. A more detailed explanation to this is provided in section 6.4.

6.2.1 Amount of collected data

The amount of data collected by the *observation unit* decreased significantly in comparison with the previous scenario. It is depicted in Figure 6.6, which shows how many data records were produced for various system configurations. The difference is connected to the effective delay in tasks execution as mentioned

at the beginning of this section.

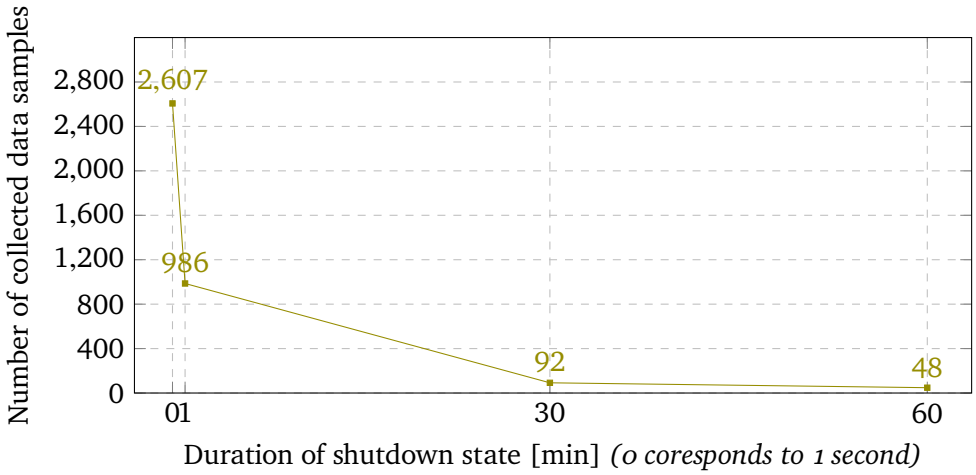


Figure 6.6: Data collected from sensors in medium load scenario.

6.2.2 Effective LoRa message output

Figure 6.7 presents the communication metrics for the medium load scenario. It shows how many LoRa messages are sent in each test case in order to deliver all sensors' readings collected in that case.

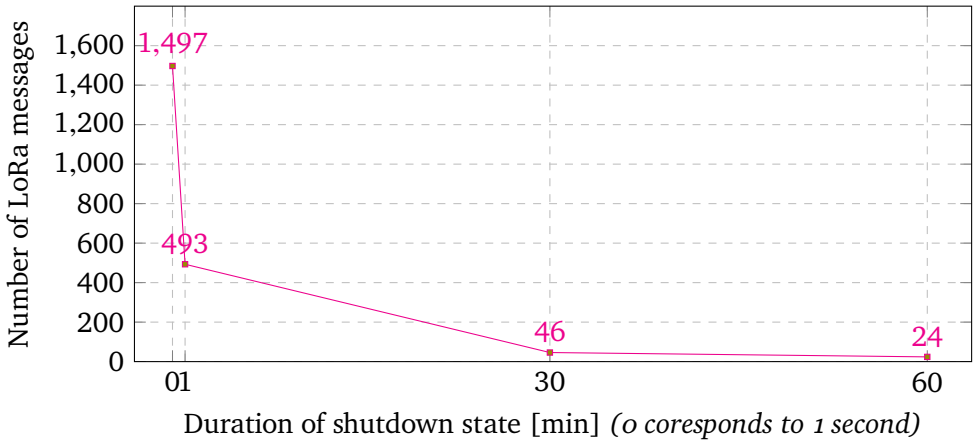


Figure 6.7: LoRa messages sent in medium load scenario.

As it resulted from the power consumption experiments, also here the measured values are similar between two cases: the 60-second intervals case in extreme load scenario (with results presented if Figure 6.3) and the 1-second intervals case in medium load scenario. The corresponding numbers of sent LoRa messages are 1241 and 1497 respectively. This results are in accordance

with the previously presented conclusions about the effective non-operational times in medium load scenario (which are longer than the intervals presented on the horizontal axis of Figure 6.7 due to device booting and shutting down times).

6.2.3 Data captured by Camera sensor

Figure 6.8 presents how the system configuration impacts the number of captured images. It is worth noticing that the results obtained here for the most intense case (1 s intervals) are roughly corresponding to the results from the extreme load scenario for the least intense case (60 s intervals). It can be explained by the overhead cost of the booting and shutting down times that prolong the effective time interval between task executions. Obtaining similar numbers for both aforementioned cases supports the earlier conclusions that the total extra overhead in medium load scenario sums up to around 1 minute.

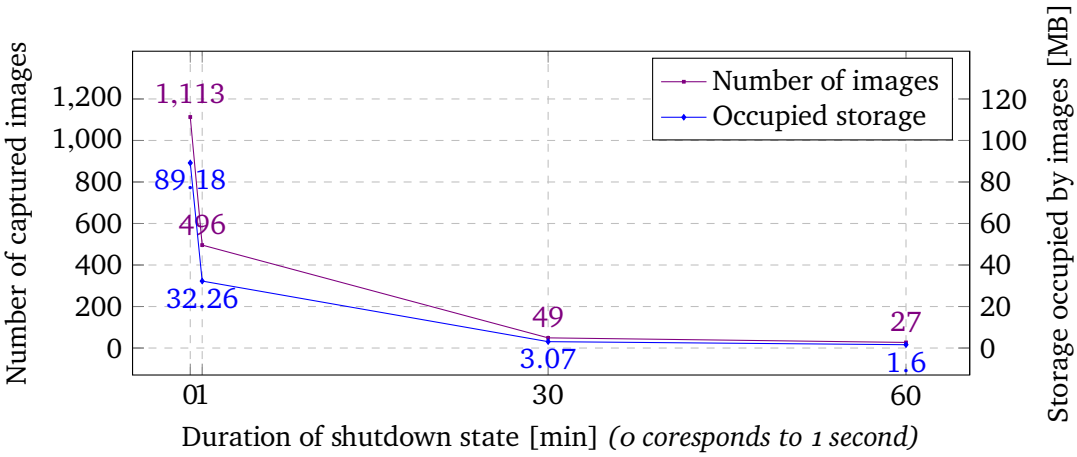


Figure 6.8: Number of captured images and occupied storage in the medium load scenario.

6.3 LoRa message rate scenario

Figure 6.9 presents measurements of the number of LoRa messages sent by the *observation unit*. Since the purpose of this experiment was to determine how many messages could be send for a given time period, the chart shows data in range from 0 to 65 seconds. One message requires slightly above 5 seconds to be delivered and acknowledged, if the time of module initialization and message preparation is not taken into account. Those application and

module preparation times are roughly constant, independent of the number of messages sent and oscillate around 4.35 second (+- 0.02s).

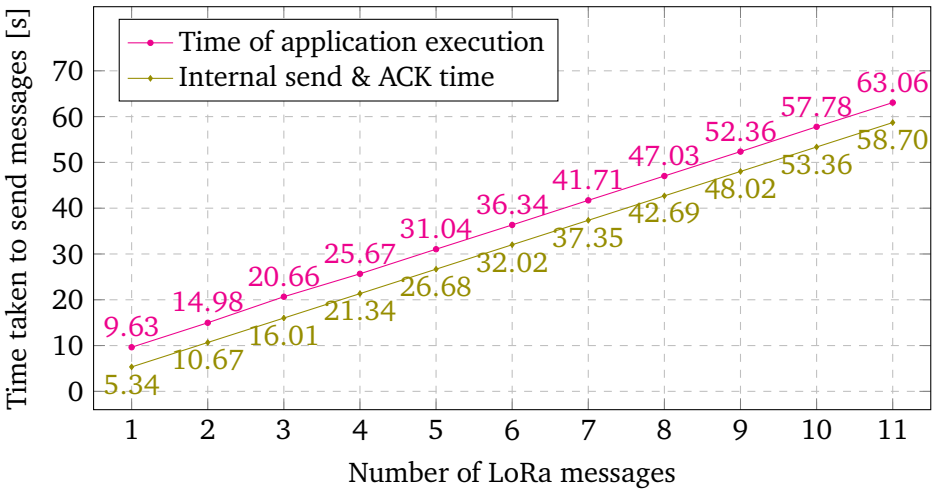


Figure 6.9: Time required to send LoRa messages.

6.4 Execution times scenario

Execution times of specified tasks were measured in this experiment. The results are presented in Figure 6.10, including the standard deviation mark for each task type. The camera image capture times are covered here as well, to elaborate on the inconclusive power consumption results described in section 6.2.

Lets consider the task types with considerably high standard deviation from the mean execution times. One such task is sending LoRa messages, which on average took around 10 seconds. In some cases, however, the acknowledgment was not received within a period of time specified in the source code, called *acknowledgment_timeout*. Even if the message was delivered, it was unconfirmed, so the *OU* tried to resend it up to five times, hence the variation. The experiment was conducted in a laboratory, where conditions for communication were close to optimal, so it should be expected that in the field tests the message would be sent repeatedly, several times before the *observation unit* finally receives the confirmation.

Another interesting task is *the observation unit*'s booting time. The shortest registered time was 25 seconds, while the longest reached 45 seconds. This variation is directly connected to the services enabled in system's start-up configuration, especially *network dhcpd* service, which was not disabled during tests, because it would not be possible to collect the measured system metrics

without it. Therefore, the list of start-up services enabled for Raspberry PI 3 should be carefully reviewed in order to disable the services that are not needed for environmental observations, especially those negatively impacting the unit's booting time.

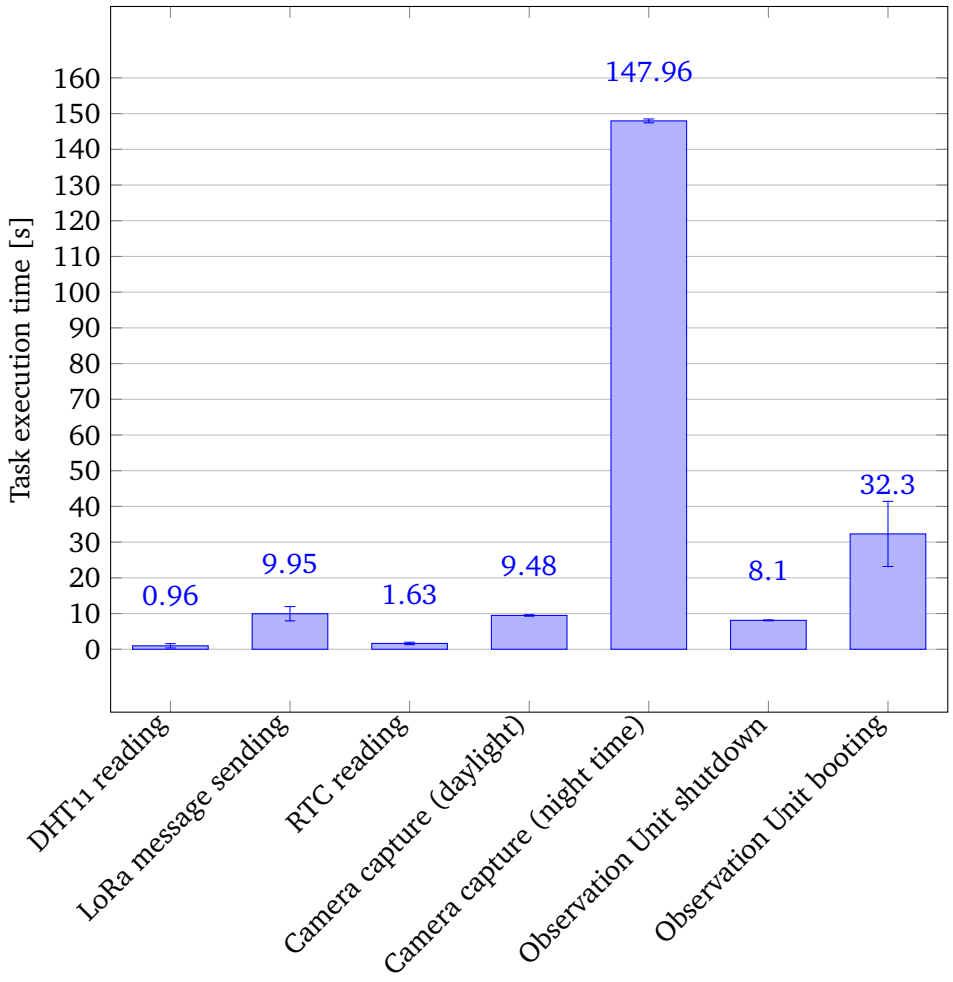


Figure 6.10: Tasks execution times.

Analyzing the medium load scenario test results, an interesting fluctuation in power consumption levels was noticed. The camera image capture times were believed to be directly connected to those changes in power demand, so an additional experiment was performed to check the influence of light intensity on the capture times. Not surprisingly, the length of capture time resulted to be inversely proportional to the level of ambient light. A significantly longer time of capture was observed in dark conditions. The camera sensor captured images 15.5 times faster during the day than at night. This is an important consideration for the power consumption, given that in the Arctic day and

night time can last much longer then in lower latitudes. Depending on time of the year, the *observation unit* might stay turned on far longer or shorter than expected, consuming different amounts of energy independently from the unit shutdown times. The OU might also be completely unable to take photos of an animal at night because of long capture times in dark conditions.

7

Discussion

7.1 Observation unit

The *observation unit* implementation is quite complex and its operations can be rather resource-intensive. We believe, however, that as the first attempt to automated Arctic observations, the presented solution performed promisingly enough to invest further work into it. The power efficiency of the unit needs to be improved, since Raspberry PI 3 chosen for the *main device* is a powerful component and it requires considerable amounts of energy. This could be alleviated by introducing a power management circuit into the system to decrease its energy demand. Such solution would have non-negligible additional complexity and would depend on incorporating modules that were not easily obtainable for the project, so it was decided that such power management was out of the scope of this work.

When choosing an approach for reducing energy demand there are two aspects to consider. Shutting the device down seems to be saving slightly more energy then having it idle for a corresponding amount of time. However, having the device powered-off and unavailable for instantaneous data capture when motion is detected (the booting time is at least half a minute) might actually be a too big of a cost to be balanced by the little energy gain. If the acceptable non-operational times are longer (around 30 minutes and more) then the approach of powering the device off becomes more justifiable.

It also needs to be noted, that power consumption was measured in a room

temperature. It should be expected that the batteries exposed to the cold temperatures of the Arctic region would have much shorter lifespan. The same consideration would need to be made for particular electronic elements of the system, as their behavior under low temperatures might also vary.

7.1.1 Configuration

The results show, that it might be difficult to find the optimal configuration for the device when the demand for the sensors' data is unknown. Therefore, testing system load for various configurations was considered to be most practical and conclusive approach to that problem. As learned, the system has its constraints and it is crucial to choose an appropriate configuration in order not to violate them. The *OU* itself would need to be aware of such limitations in order to prevent hard failures. It could adjusts its configuration based on its health state, for example to decrease the power consumption or when the available storage is almost full.

Another aspect of the system's configuration is the amount of collected data. The device is configured to send the sensors readings in FIFO manner, the oldest first. Therefore, the amount of data gathered and queued for sending should not exceed the capacity of the communication module's message rate in order to prevent long delays in the data collection at the *backend* side. Alternatively, similar results might be grouped together to decrease the amount of stored data. In the case of more than two sensors connected to the device it is more than desired to select at least one of proposed solutions.

7.1.2 Camera module

The camera module does not work exactly as expected. It was known that taking an image at night would increase the capture time, but the current results are far beyond the acceptable times for catching an image of an animal in front of the *observation unit*. An infrared camera module was, unfortunately, not part of the tests, but having it available would not solve the problem directly either. It is not possible to have two camera modules connected simultaneously to the same Rasppbery PI 3 device (most probably not even to any other device from Raspberry PI family). It would be reasonable, however, to choose the infrared module instead of the current one, and to stop using the camera itself to detect motion. A dedicated motion detection sensor wired to the *support device* could be used as a trigger to wake up the *main device*, which in turn would be taking images with an infrared camera, regardless of the time of day. Such solution is currently not possible with the motion detection code used in the *observation unit*'s implementation.

Current EC^3 System prototype does not enable sending captured pictures over LoRa messages because of their size limitation. An attempt was made to limit the image size from 450 kB (raw picture) to 75-85 kB (at the current level of post-processing). Unfortunately, even images of this small size are still too big to send them with the LoRa module. A single file has a size of roughly 80 kB, while a single message length can not exceed 256 B. To send such image, it would require 320 messages (withoud including AES encryption and the CRC overhead). Taking into account single message sending time (presented in section 6.3) it would take at least half an hour to send one photo, not nearly enough to send images in real time at high capture rates. A solution for that issue could be to use the capabilities of Raspberry PI 3 based *OU* and implement an animal recognition software into it, in order to be able to later send only a meta-information about spotted animals.

7.1.3 Real Time Clock issues

Some issues with DS1302 RTC module encountered during the medium load tests should be mentioned. Several phases of the experiment had to be repeated, because the *main device* was not always woken up by the *support device* on a specified time. The logs collected show, that in some cases the time value returned by the RTC module was too far in the future (several hours) and it caused the experiments to fail. In an attempt to eliminate the malfunction, all the wired connections of the module and its backup battery were replaced, but the problem reemerged. At this point, it is assumed that the DS1302 module or the library used to read its value is unreliable, but without access to another DS1302 module it is not possible to confirm the real source of encountered problems.

7.2 Gateway

LoRa gateway implementation used in the EC^3 System worked as expected during all conducted tests. It was experienced, as stated in the LoRa specification, that the *gateway* receives all the LoRa messages in its range, also those not sent by the *observation unit*. Such messages were also stored in the AWS IoT cloud and later downloaded by the *backend* application. The *gateway* could be able to filter those messages out based on the sender's address, which is an unencrypted part of the message. Even if the sender's address matched the address of *observation unit*, which could happen as the addresses are arbitrarily chosen and hardcoded, it would be possible to filter those message later on the *backend* side, since it would not be possible to decrypt the wrong message without AES keys associated to the source device.

What was not examined in the presented solution was the effective range of the communication module. It is advisable to compare the *gateway* range with the professional solutions and decide which one should be selected for the EC^3 System.

7.3 Backend

The implemented backend part of the EC^3 System was found useful in monitoring the sate and results of the conducted experiments in real time. The application was downloading the LoRa messages from the AWS IoT cloud and decrypting the data contained. It was possible to check the content of the messages and the time when a single message was received by the *gateway*. This allowed to match the received information with the expected experiment results and thus detect and fix several bugs in the *observation unit*'s implementation.

The *backend* itself does not store the downloaded data, so additional work is required to present the received information in a more user friendly way. The main focus should be on facilitating the analysis of the collected sensors readings and checking the *OU* health status.

7.4 Prototype limitations

The following simplifications have been assumed in the current implementation of the EC^3 System:

- The delays in sensor readings, the device sleeping time and LoRa physical address are not configurable and their values are permanently set in the source code. The last parameter needs to be changed for every *OU* because without it the data collected from more than one device will not be distinguished. A good solution that addresses this limitation is to create a separate configuration file to store these parameters and then rewrite the code to read the file.

- The system prototype provides security only in communications between the *gateway* and AWS IoT services, AWS DynamoDB and the *backend* services, and between the *OU* and the *gateway*. AES encryption and decryption keys for the *OU* and the *backend* services are set in the source code. The configuration file would solve also this limitation.

7.5 Challenges

The challenges that one can expect taking up a project are often not as demanding as the ones that result from unforeseen development.

The biggest initial challenge was to choose the components for the final system basing just on their technical specifications. Several times, starting to work with a component, or putting it under tests in combination with other system elements, caused it to exhibit an unexpected behavior. In some cases, only after including a component in operations was it discovered that it was unsuitable for the required task. The problem with that late discovery was that obtaining a different, replacement device was often out of question due to timing constraints of this work.

Familiarizing with IoT field, reading and modifying electronic schematics in order to select and properly combine basic building blocks into a more complex system, programming Arduino boards – it was all an anticipated challenge. Somewhat underestimated part was the actual, physical building of the system. Many of the elements required to be soldered, which is quite a meticulous task, or connected through a breadbord with numerous wired connections into one big mesh. Using the breadbord in a systematic way and keeping track of the wired connections proved to be particularly important in cases where hardware problem needed to be traced and a component replaced.

Probably the least expected challenge was the necessity of inspecting the system at the lowest, electronic level. Some of the problems could not have been traced without determining the values of electrical current and voltage between specified parts of the circuit. The resulting conclusion was often that the two neighboring components are not mutually compatible and require and additional middleman part. That in turn required calculating the accurate values of the parameters of the electronic components to be introduced (e.g. resistors and diodes), and finally physically incorporating them into the system.

7.6 Lessons Learned

Many issues were detected in the process of developing the EC^3 System presented in this document. Starting to conduct experiments in the early stages of the development is crucial for confirming system's stability at every step, before further expanding the implementation. The workload devoted to early testing pays off in the later stages, when due to growing system's complexity detecting problems consumes significantly longer time and effort.

It was important to select hardware's components for the prototype, which are widely supported by the IoT community in order to ensure multi-platform (Raspberry, Arduino) support. Availability of well documented libraries was crucial for meeting time constraints of the project.

Before the final devices for the prototype were selected, much time was spent on selecting the candidates from a much bigger set of available modules. An example was a Nordic Semiconductors NRF52 microcontroller that was difficult to integrate with the LoRa module due to limited debugging capabilities and lack of libraries. The same applied to two other devices with flashed firmware that were, in consequence, dismissed in the design phase of the project. It is good to start with a more intuitive, linux-based board, which behaves in a more predictable way, rather then assuming a complicated and apt system and achieving little functionality with it.

Perhaps the most important lesson is that the stability of a distributed system, like the EC^3 System presented in this paper, depends on many components, both the software and the hardware implementations. Often the hardware components prove insufficient or defective and require replacement. Having an easy access to spare parts is essential in confirming the source of emerging issues and fixing them. As an example, much time was devoted to pinpoint troubles with the *observation unit*'s communication module. The issue could not be traced neither to the hardware nor to the software implementation, until a broken wired connection proved to be the source of the problem.

8

Summary

The goal of this Master's thesis was to develop a system capable of observing the Arctic environment, detecting motion in front of the installed camera, reporting collected data and the state of system's sensors, and recovering from several types of failures. Such system, named a EC^3 system, was designed, built, tested and described in this paper.

The EC^3 system is a first attempt towards monitoring the wildlife in a more robust way than just through a simple photo camera capturing images when the motion is detected. The research brought much positive outcome. The *observation unit* provides a way of delivering information on a long distance using a low-powered communication module. The infrastructure for data receiving could be build using the EC^3 System's gateway implementation or using the professional LoRa gateways with bigger antennas and more sensitive receiving modules available on the market. The data could be also delivered via an existing infrastructure by cooperating with a third-party operator without a risk of data leakage. The device can be extended with additional sensor peripherals to collect even more types of data for climate change analysis.

On the other hand, the *OU* consumes too much power to be placed in the field in its current form and it shows some signs of instability. The increased functionality of the prototype definitely did not come without a price, but with more extensive examinations and the addition of several missing parts it could be polished to the state when it proves useful in the real environment.

The *observation unit*'s ability to recover from failures is limited to *retry and reboot*. It is the most obvious way of dealing with system's internal issues and it requires only a few seconds of device's operational time to recover. The unit reports the state of observations, and even in case of a hard failure the lack of information itself is a sufficient sign that the device *requests* a human intervention.

Since the *main device* is under the control of a linux operating system, it allows to use much more existing software solutions and programming languages to deliver the desired functionality, which could not be provided by the embedded systems only. It opens a door for introducing further improvements without the necessity of being familiar with IoT development as a prerequisite. As an example, an animal-recognition system could be written in a software programming language and integrated into the EC^3 system, thanks to its linux compatibility. The *OU* would first serve as a test data generator for the animal-recognition system and then could be integrated with it for more efficient processing of the camera-captured images. This would not only speed up the data analysis but it could also eliminate the need to collect images in a traditional way of copying them from an SD memory card. The information about recognized animals could be send directly to the *gateway* via LoRa message.

The implementation of the *observation unit* prototype showed how many different aspects need to be taken into account beyond software implementation. The system's stability, for example, does not depend only on the quality of the software part, but also on the selected hardware components, which sometimes are less reliable than others. Testing several models of the same type of peripheral is highly advisable when choosing a candidate for the final solution to be deployed in the field.

9

Future Work

Right now the *observation unit* consumes too much energy to be placed in the Arctic environment and survive on batteries for several months. Raspberry PI 3 used as the *main device* drains power even in the shutdown state, which requires an external power circuit controlled by the *support device* to cut off such *power-leakage*.

Another approach to limit the *OU*'s power demand would be to replace Raspberry PI 3 with a much more power-efficient Raspberry PI Zero (a less powerful version of Raspberry PI family boards with a single core chip). More experiments need to be conducted in order to determine the accurate numbers, but just by looking through specification, the board seems promising. Moreover, it would not require a lot of software refactoring, except for adjusting the SX1276 chip library, which is written for more powerful versions of Raspberry PI.

The *observation unit* prototype is build on a breadboard with a lot of wired connections between elements of the unit, which in many cases was the source of failure and caused significant delays in conducting experiments. The solution to such problem would be a customarily designed PCB mounted with screws at the top of the *main device*. Designing the PCB circuit would entail more work and soldering of the elements would require more advanced tools, but the resulting hardware would constitute a much more physically-robust solution.

Real Time Clock (DS1302) used in the system's prototype gives signs of being unstable, which caused problems in *the medium load scenario* experiment,

leading to different than expected results. In order to stabilize the *OU* prototype, a new, more reliable RTC module should be used.

The EC^3 System, which states for Command-Control-Communication System, is actually an EC^2 System (Control-Communication) with Command part missing. It is the consequence of using the LoRa module in the *observation unit* for sending messages, which does not allow for establishing a connection between devices. In the current software implementation, the transceiver peripheral wired to the *main device* works as a transmitter only, the data exchange is unidirectional, from the sensors to the *gateway*. The solution to this matter would be to change the *OU*'s behavior so that it plays a role of the *gateway* for a specified amount of time. It could send a message to the *gateway*, stating that it is awaiting incoming data for a specified amount of time, thus allowing the Command part to be added to the current EC^3 System implementation.

Security is one of those important real-life factors that was out of the scope of this project. The data send from the *observation unit* is encrypted by the 128 bits AES mechanism, but the device itself is not secured. Anyone with direct access to it would be able to take over it. The device's storage could be encrypted as well, in order to prevent the data leakage, but a physical security layer should be also considered.

As the results of the test scenario described in Section 6.4 showed, the camera module requires different amounts of time to take images, depending on ambient brightness. In daylight, the average capture time is around 10 seconds, while at night the same operation takes more than 2 minutes. It is an issue with the *pi-camera* library, and probably with the camera module itself as well. The solution to this matter would be to have two camera modules, one for daylight photos (e.g. the current one) and another one handling better the night-time light levels (for example an infrared camera module).

Finally, the EC^3 System should be examined in the field, in conditions similar to the ones of the Arctic environment. Without it, it is difficult to determine how many more new issues would emerge as a result of exposing the system to actual environmental factors. However, since the *observation unit* is missing the power circuit in its current implementation, it is not possible to power it from a battery source and thus leave unattended for extended periods of time. Providing for that would be the necessary first step in taking the system's tests out of the lab into the outdoors.

Bibliography

[1] "Vicotee Njord", http://www.vicotee.com/. Accessed: 18.03.2017.

[2] "SensorTag 2", http://www.ti.com/tool/cc2650stk. Accessed: 18.03.2017.

[3] "Waspmote", https://goo.gl/MSKObw. Accessed: 18.03.2017.

[4] "Raspberry PI", https://www.raspberrypi.org/. Accessed: 18.03.2017.

[5] "Arduino", https://www.arduino.cc/. Accessed: 18.03.2017.

[6] "Nordic Semiconductor NRF52", https://goo.gl/gyvshb.
Accessed: 27.03.2017.

[7] "STM Nucleo L476RG", https://goo.gl/dIXSdB. Accessed: 27.03.2017.

[8] "Spell Foundry Sleepy PI", https://goo.gl/5V7KyH. Accessed: 27.03.2017.

[9] "Atmel ATmega328P", https://goo.gl/q7jiOp. Accessed 27.03.2017.

[10] "Arduino Low Power", https://goo.gl/5YDFVq. Accessed 29.03.2017.

[11] "LoRa Alliance™ Technology ", https://goo.gl/9Nebiq.
Accessed 29.03.2017.

[12] "LoRaWAN Range Testing", https://goo.gl/3Mp71h.
Accessed 29.03.2017.

[13] "Semtech LoRa Modem Design Guide", https://goo.gl/U1ZMvK.
Accessed 17.04.2017.

[14] "AWS IoT MQTT protocol", https://goo.gl/V41mLT.
Accessed 09.04.2017.

[15] "AWS IoT", https://aws.amazon.com/iot/. Accessed 09.04.2017.

[16] "AWS DynamoDB", https://aws.amazon.com/dynamodb/. Accessed 09.04.2017.

[17] "Raspbian Jessie", https://www.raspberrypi.org/downloads/raspbian/. Accessed 16.04.2017.

[18] "Overview for Voltage Level Translation", https://goo.gl/xyXVZo. Accessed 16.04.2017.

[19] "Raspberry PI Camera Module V2", https://goo.gl/vv382W. Accessed 16.04.2017.

[20] "pi-timolo", https://github.com/pageauc/pi-timolo. Accessed 16.04.2017.

[21] "picamera", https://picamera.readthedocs.io/en/release-1.13/. Accessed 16.04.2017.

[22] "Dragino LoRa BEE", https://goo.gl/oMi67i. Accessed 17.04.2017.

[23] "Semtech SX1276", https://goo.gl/OqiuBt. Accessed 17.04.2017.

[24] "Low-cost LoRa IoT & gateway with SX1272/76, Raspberry and Arduino", https://goo.gl/aEwHT3. Accessed 17.04.2017.

[25] "DHT11 - Humidity & Temperature sensor", http://www.micropik.com/PDF/dht11.pdf. Accessed 18.04.2017.

[26] "Adafruit Python DHT Sensor Library", https://goo.gl/8sRbul. Accessed 18.04.2017.

[27] "USB Charger Doctor", https://www.adafruit.com/product/1852. Accessed 1.05.2017.

[28] "timezonedb", https://timezonedb.com/. Accessed 10.05.2017.

Appendices

A

Observation Unit Usage

A.1 Arduino Pro Mini

Arduino Pro Mini needs to be flashed first using the user's host. The code for the device is placed inside the EC^3 System repository. The full path after extraction is: C3/device/arduino/.

The recommended software for flashing the Arduino is PlatformioIO IDE (http://platformio.org/).

A.2 Raspberry PI 3

In order to run the system, Raspbian Jessie Lite must be written on the SD card, then the EC^3 System repository must be extracted in /home/pi directory, so that the full path to the files is /home/pi/C3.

The system requires *wiringpi* library to be installed, which is done with the following commands in Raspbian Jessie:

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get install wiringpi
```

System initialization is done with the following command executed in /home-/pi/C3/device/ directory:

```
$ sh setup.sh
```

After a successful execution of this command, Raspbian Jessie needs to be rebooted. This will start the *observation unit* tasks after system booting. It means, that the device is going read values of:
- DHT11 sensor,
- system's free space,
- number of captured photos.

Moreover, it will run:
- motion detection script,
- LoRa messages sending application.

When LoRa application finishes its execution, the device will request Arduino to wake it up after a period of 1 minute and will then shutdown on its own.

The whole cycle will be repeated in an endless loop.

B

LoRa Gateway usage

B.1 AWS IoT

AWS account needs to be created first as described in AWS Documentation: https://goo.gl/DhQ571

When AWS account is active, in order to access AWS Command Line Interface (AWS CLI) the AWS IAM user named *Gateway* needs to be created with the following settings:

AWS access type Programmatic access

Permissions Policies AWSIoTConfigAccess, AmazonDynamoDBFullAccess

Access key ID and *Secret access key* from AWS IAM user summary page need to be saved in a safe place, because these credentials are required to configure AWS CLI.

Next, a new IAM policy named *RolePolicy* needs to be created in AWS IAM console with the following content:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
```

```
            "Sid": "Stmt1482712489000",
            "Effect": "Allow",
            "Action": [
                "iam:CreateRole",
                "iam:PutRolePolicy",
                "iam:PassRole"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

When *RolePolicy* policy is successfully created, it needs to be assigned to the IAM User *Gateway*.

Next steps related to AWS IoT are described in section B.2.

B.2 Raspberry PI 3

In order to run the *gateway*, Raspbian Jessie Lite must be written on the SD card, then EC[3] System repository must be extracted in /home/pi directory, so that the full path to the files is /home/pi/C3.

The system requires *wiringpi* library to be installed, which is done with the following commands in Raspbian Jessie:

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get install wiringpi
```

User needs to install python-2.7 and pip-2.7 if these packages are not present in Raspbian Jessie.

The next step is to install and configure AWS CLI using the following commands:

```
# Acces key ID and Secret access key is required in this step
  $ sudo pip install --upgrade awscli
  $ aws configure
```

When the AWS CLI configuration is done, the AWS IoT Cloud needs to be configured via the following commands:

```
$ aws iot create-thing --thing-name ou
$ aws iot create-policy --policy-name gateway-policy \
--policy-document file://C3/gateway/aws/policy/gateway.json
$ aws iot create-keys-and-certificate --set-as-active \
--certificate-pem-outfile C3/gateway/aws/keys/gateway.cert.pem \
--public-key-outfile C3/gateway/aws/keys/gateway.public.pem \
--private-key-outfile C3/gateway/aws/keys/gateway.private.pem
```

The next command will return a JSON output, from which *certificateArn* field's value needs to be passed to the next commands:

```
$ aws iot attach-principal-policy --policy-name gateway-policy \
--principal value_of_certificateArn
$ aws iot attach-thing-principal --thing-name ou \
--principal value_of_certificateArn
```

In order to store LoRa messages, DynamoDB table named *package* needs to be created as fallows:

```
$ aws dynamodb create-table --table-name package2 \
--attribute-definitions AttributeName=address,AttributeType=N \
AttributeName=timestamp,AttributeType=N \
--key-schema AttributeName=address,KeyType=HASH \
AttributeName=timestamp,KeyType=RANGE \
--provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5
```

It will return a JSON output, from which *TableArn* field's value needs to be written to C3/gateway/aws/policy/dynamodb.json file, in the *Resource* field.

The last step in AWS IoT CLI configuration is to assign a rule in which all LoRa messages from MQTT protocol will be saved in DynamoDB table. It needs to be done via the following commands:

```
$ aws iam create-role --role-name Gateway-Role \
--assume-role-policy-document \
file://C3/gateway/aws/role/gateway.json

# above command will return JSON from which ARN field value
# needs to be written to file C3/gateway/aws/rule/
# mqtt-to-dynamodb.json in field roleArn before the next
# two commands can be executed:
```

```
$ aws iam put-role-policy --role-name Gateway-Role \
--policy-name Permissions-Policy-For-Gateway \
--policy-document \
file://C3/gateway/aws/policy/dynamodb.json

$ aws iot create-topic-rule --rule-name MQTTToDynamoDB \
--topic-rule-payload \
file://C3/gateway/aws/rule/mqtt-to-dynamodb.json
```

When the above steps are successfully executed, the file C3/gateway/postprocessing.py needs to be edited to replace an argument of function *myMQTTClient.configureEndpoint* with the value obtained from *Settings* page from AWS IoT web interface.

To finalize the *gateway* setup, the last two commands need to be executed:

```
$ cd /home/pi/C3/gateway/ && sh setup.sh
$ sudo reboot
```

Performing the above commands ensures that the *gateway* application is correctly installed and will be running at every system booting.