Energy-Efficient Communication with Lightweight M2M in IoT Networks

CARLOS GONZALO PECES
Title: Energy-Efficient Communication with Lightweight M2M in IoT Networks

Author: Carlos Gonzalo Peces

Supervisors: Nicolas Tsiftes, Ph.D.
Joakim Eriksson, Ph.Lic.

Examiner: Prof. Ingo Sander

Affiliation 1: School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology

Affiliation 2: ETS de Ingenieros de Telecomunicación, Universidad Politécnica de Madrid

Stockholm, 12th of March, 2018
This work is supported by RISE SICS AB and developed in cooperation with the Networked Embedded Systems Group.
Acknowledgements

It is incredible how time flies. It feels like yesterday when I started my studies at the university, and now I am finished and ready to continue with my career. Now it is time to thank to all the people that helped me in this way, since the very beginning.

First of all I would like to show my gratitude to the Networked Embedded Systems group. Special mention to Nicolas and Joakim, my supervisors, for guiding me in the whole Master’s thesis and teaching me through all the project. Also thanks to Thiemo for giving me the opportunity to do the thesis in this group and for his kindness since the first time I contacted him. Thanks to all the others for the friendly environment, it was really nice to work in a place like this.

What to say about all the friends I made in Stockholm... this experience would not have been the same without you. Thanks Jorge, Patryk, Jonas and Alex for being the best corridor mates ever. Also Sebastian, Guayen, Irene, Enrique, Pablo, Vicent, Viktor, Mathilde, Alfonso... for all the dinners, trips and parties.

Thanks to all the people from my home university, la ETSIT. To Ana, Alberto, Jorge, Marta, Cris, Jose, Isa... Thank you for all those good times despite the stress from the studies. Thanks to all the people from "el labo", the B105 Electronic System Lab, for being my home for two years and introducing me in this field, and especially to Alvaro, for all the things you have taught me, not only the academic ones.

Also I do not want to forget about all the people not so linked to the university, but that also were with me all this time. Thanks to all the people from Coslada for being there since the very beginning, special mention to Mario, Diego, and of course to the whole Cachas Team, my basketball team. Also thank you people from Berrocalejo for all the laughs and funny moments that we have lived together.

Special mention to my family. Thank you mum and dad for your unconditional support, for being always beside me, and for always believe in me. Thanks to my grandmas, uncles, aunts, cousins... you are also a part of what I am nowadays.

And finally, thank you Dessy for being with me through all this seven years. You have always believed in me, since the very beginning when I was unsecure about being able to become an engineer, until now that I have finally completed it. I have no words to express how grateful I am.

Thanks to all, from my heart...

Carlos Gonzalo Peces
OMA’s Lightweight Machine to Machine (LwM2M) is an application protocol for device management in the Internet of Things (IoT) that has been recently published and widely adopted in a lot of projects. The protocol is designed to operate in sensor networks and machine-to-machine environments, where one of the main constraints is the energy consumption since the nodes are usually battery powered. Different strategies to achieve high energy efficiency in IoT networks have been developed, but there is no deep knowledge about the performance of LwM2M operating with them. Moreover, the specification of this protocol includes one strategy, called the Queue Mode, which could be more efficient than the usual ones because it has been specified for this particular protocol.

This project aims to implement this Queue Mode at both sides of the communication, and then evaluate its performance by comparing it with TSCH, which is the standard MAC protocol used in IEEE 802.15.4 that defines a way of radio duty cycling. It has been proven to achieve a high energy efficiency, and that is the main reason why it is selected. The comparison is performed according to several metrics to have a comprehensive evaluation, and in different kind of scenarios, with different numbers of IoT devices and different parameters in the communication.

The implementation was done inside the Contiki-NG OS for the client side, which is an operating systems designed for constrained devices. For the server side it has been carried out inside the Eclipse Leshan code, which is a LwM2M implementation in Java made by the Eclipse Foundation. As a result of the evaluation, it shown that both implementations operate correctly.

This thesis contributes as a guideline for making decisions about which low power strategy is better to use depending on the IoT scenario and the type of application. It shows that for many use cases Queue Mode is a better option than TSCH because it achieves a higher energy efficiency and the rest of the metrics used in the evaluation have also improved values. TSCH has a better performance only in demanding scenarios or in cases where the communication is not produced at fixed time instants.

The thesis was developed in cooperation with RISE SICS AB, Networked Embedded Systems Group.

**Keywords:** IoT; LwM2M; low power consumption; TSCH; Contiki-NG; Eclipse Leshan.
OMA:s Lightweight Machine to Machine (LwM2M) är ett applikationsprotokoll för enhetshantering i Sakernas Internet (IoT) som nyligen har publicerats och börjat användas i många projekt. Protokollet är utformat för att fungera i sensornätverk och maskin-till-maskin miljöer, där en av de viktigaste begränsningarna är energiförbrukningen eftersom moderna vanligtvis är batteridrivna. Olika strategier för att uppnå hög energieffektivitet i sensornätverk har utvecklats, men det finns ingen djup kunskap om hur LwM2M fungerar med dem. Dessutom innehåller specifikationen av LwM2M en strategi kallad Queue Mode (köläge) som kan vara effektivare än de vanliga strategierna eftersom den har utvecklats direkt för det här protokollet.

Detta examensarbete syftar till att implementera detta köläge på båda sidor av kommunikationen och sedan utvärdera prestandan genom att jämföra det med TSCH, vilket är ett MAC-protokoll specificerat i IEEE 802.15.4-standarden. Tidigare arbeten har visat att TSCH kan uppnå en låg energiförbrukning, vilket är den främsta anledningen till att detta protokoll väljs ut för att jämföra mot LwM2M:s köläge. Jämförelsen inkluderar flera olika typer av mätvärden och scenarier för att få en omfattande utvärdering, samt med flera olika antal sensor noder och parametrar.

Implementationen gjordes för Contiki-NG OS på klientsidan, vilket är ett operativsystem för resursbegränsade IoT-enheter. På serversidan har implementationen gjorts för Eclipse Leshan, vilken är en LwM2M-implementation skriven i Java och publicerad av Eclipse Foundation. Som en följd av utvärderingen har det visat sig att båda implementationerna fungerar korrekt.

Detta examensarbete bidrar med riktlinjer för att fatta beslut om vilken energibesparingsstrategi som är bättre att använda beroende på IoT-scenariot och typen av applikation. Utvärderingen visar hur Queue Mode i många användningsfall är ett bättre alternativ än TSCH eftersom det uppnår en högre energieffektivitet utan att de andra typerna av mätvärden påverkas av det. I vissa fall uppnås dessutom förbättrade resultat även i de andra typerna av mätvärden. TSCH har endast bättre prestanda i krävande scenarier eller i fall där kommunikationen inte genereras vid bestämda tillfällen.

Examensarbetet har genomförts hos Networked Embedded Systems-gruppen på RISE SICS AB.

**NYCKELORD:** IoT; LwM2M; låg energiförbrukning; TSCH; Contiki-NG; Eclipse Leshan.
Resumen

OMA’s Lightweight Machine to Machine (LwM2M) es un protocolo de aplicación para la gestión de dispositivos en el ámbito del Internet de las Cosas (IoT). Ha sido publicado recientemente y está siendo utilizado en gran cantidad de proyectos. El protocolo está diseñado para operar en redes de sensores y comunicaciones machine-to-machine, donde una de las mayores restricciones es el consumo de energía ya que los dispositivos suelen estar alimentados con baterías. Existen diferentes estrategias para conseguir alta eficiencia energética en redes IoT, pero no hay un conocimiento profundo sobre el funcionamiento de LwM2M cuando hace uso de ellas. Además este protocolo incluye en su especificación una nueva estrategia, llamada Queue Mode, la cual podría ser más eficiente que las utilizadas habitualmente.

Este proyecto tiene como objetivo implementar el Queue Mode en ambas partes de la comunicación, y posteriormente evaluar su funcionamiento comparándolo con TSCH, que es el protocolo MAC estándar utilizado por IEEE 802.15.4, el cual define una manera de reducir el ciclo de trabajo de la radio. Ha sido probado que este protocolo consigue una alta eficiencia energética, y por eso es el seleccionado para la evaluación. La comparativa se realiza con respecto a distintas métricas para ampliar el alcance de la evaluación, y en diferentes tipos de escenarios, utilizando distintos parámetros en la comunicación y redes con mayor o menor número de dispositivos.

La implementación ha sido realizada en el sistema operativo Contiki-NG para el lado cliente, el cual ha sido diseñado para operar en dispositivos IoT. Para el lado servidor se ha utilizado el código de Eclipse Leshan, el cual es una implementación de LwM2M en java realizada por la Eclipse Foundation. Como resultado de la evaluación, se muestra que el funcionamiento de estas implementaciones es correcto.

Los resultados de este trabajo contribuyen a modo de guía para decidir qué estrategia de bajo consumo es mejor utilizar dependiendo del escenario y del tipo de aplicación. La comparativa demuestra que para para muchos usos Queue Mode supone una mejor opción que TSCH porque consigue una mayor eficiencia energética y además el resto de métricas utilizadas no solo no son afectadas, sino que también mejoran. TSCH es una mejor opción únicamente para casos en los que el servidor requiere datos con una alta frecuencia o en instantes de tiempo no prefijados.

Este trabajo fue desarrollado en colaboración con RISE SICS AB, Networked Embedded Systems Group.

**Palabras clave:** IoT; LwM2M; bajo consumo; TSCH; Contiki-NG; Eclipse Leshan.
## Contents

Acknowledgements ........................................... v

Abstract .................................................... vii

Sammanfattning ............................................. ix

Resumen ..................................................... xi

Acronyms List ............................................... xxiii

1 Introduction .............................................. 1
   1.1 Problem Statement .................................... 2
   1.2 Contributions ........................................ 3
   1.3 Thesis Structure ...................................... 4

2 Background ................................................ 5
   2.1 Contiki-NG Operating System ....................... 5
      2.1.1 Protothreads, Processes and Events ............ 7
      2.1.2 Timers ........................................... 8
   2.2 Network Stack ....................................... 9
   2.3 Constrained Application Protocol (CoAP) ........ 10
      2.3.1 Messaging Model ................................ 11
      2.3.2 Message Format .................................. 11
      2.3.3 URI scheme ....................................... 12
      2.3.4 CoAP Methods and Response Codes ........... 12

xiii
2.4 Lightweight Machine to Machine (LwM2M) .............................. 13
  2.4.1 Resource Model ................................................. 13
  2.4.2 Bootstrap Interface ............................................. 15
  2.4.3 Client Registration Interface ................................. 16
  2.4.4 Device Management and Service Enablement Interface ...... 18
  2.4.5 Information Reporting Interface ............................... 19
  2.4.6 Queue Mode ...................................................... 20
2.5 Low Power MACs in Contiki ............................................. 22
2.6 Measuring Energy Consumption in Contiki ......................... 25

3 Queue Mode Implementation at the Client Side ...................... 27
  3.1 Implementation in the LwM2M RD client .......................... 28
    3.1.1 Finite State Machine operation without Queue Mode ....... 28
    3.1.2 Finite State Machine operation with Queue Mode .......... 29
    3.1.3 Design and implementation of the client awake time ....... 31
  3.2 Implementation in the LwM2M engine ............................. 32
    3.2.1 Design and implementation of the Information Interface with Queue Mode ...................................... 32
  3.3 The Queue Mode Object ............................................. 34

4 Queue Mode Implementation at the Server Side ..................... 37
  4.1 Complete implementation of the Queue Mode .................... 38
    4.1.1 Listener for registration and update messages ............. 39
    4.1.2 LwM2mQueue class for managing the Queue Mode of each client ............................................. 39
    4.1.3 Request Sender with Queue Mode operation ................ 40
    4.1.4 Complete Queue Mode Operation in the LwM2M Core ...... 42
  4.2 Reduced implementation with Presence Listeners ................. 43

5 Evaluation ................................................................. 45
  5.1 Hardware platform .................................................. 45
  5.2 Evaluation Setup .................................................... 46
5.3 Evaluation Metrics and Scenarios ........................................ 47
5.4 Measurement of the Defined Metrics ................................. 50
5.5 Results ........................................................................... 51
5.5.1 Queue Mode Scenarios .................................................. 51
5.5.1.1 Synchronized requests scenario ................................. 51
5.5.1.2 Randomized requests scenario ................................. 57
5.5.2 TSCH Scenarios .......................................................... 58
5.5.2.1 Synchronized requests scenario ................................. 58
5.5.2.2 Randomized requests scenario ................................. 62
5.6 Comparison between Queue Mode and TSCH ....................... 62
5.6.1 Energy Consumption .................................................... 63
5.6.1.1 Sending period 10 s .............................................. 63
5.6.1.2 Sending period 60 s .............................................. 64
5.6.1.3 Sending period 5 min .............................................. 64
5.6.2 Round Trip Time ........................................................ 65
5.6.3 Packet Reception Rate ............................................... 66
5.6.4 Memory footprint ..................................................... 67
5.7 Battery lifetime calculations ............................................ 68
5.8 Alternative scenarios ..................................................... 70
5.8.1 Monitoring scenario .................................................. 71
5.8.2 Scenario with default values recommended by OMA specification 72

6 Conclusions and Future Work ........................................... 74

References ............................................................................ 76
List of Figures

1.1 Typical IoT scenario with a sensor network connected to the cloud through a border router. ................................. 1

2.1 Processes execution in the Contiki-NG OS. ......................... 8

2.2 Network stack used for the communication in the developed IoT networks. 9

2.3 Resource model scheme in LwM2M. ................................. 14

2.4 Communication scheme in the Bootstrap interface of LwM2M. .... 15

2.5 Communication scheme in the Client Registration interface of LwM2M. 17

2.6 Communication scheme in the Device Management and Service Enablement interface of LwM2M. ............................... 18

2.7 Communication scheme in the Information Reporting interface of LwM2M. ..................................................... 20

2.8 Communication scheme using LwM2M protocol with Queue Mode enabled. ......................................................... 21

2.9 TSCH slotframe example for the communication between two nodes. 23

3.1 Finite state machine implementing the RD client. ................. 29

3.2 Finite state machine implementing the RD client with Queue Mode operation. .................................................. 30

3.3 Problem (a) and solution (b) in the information interface when using Queue Mode. ............................................. 33

4.1 Architecture of the Leshan server code. Only the interesting objects for the Queue Mode implementation are shown. .......... 39

4.2 Flowchart of the Queue Mode request sender. ....................... 41

4.3 Complete operation of the Queue Mode inside the Leshan server. Only the objects added to the Leshan core for this purpose are shown. .......... 42
List of Figures

4.4 Queue Mode operation inside the Leshan server for the pull request. Only the necessary objects added to the Leshan core are shown. 44

5.1 The Zolertia Firefly hardware platform. 46
5.2 Setups used to build the IoT networks for the experiments in the evaluation part. 47
5.3 Current consumed over time by the client node in a Queue Mode scenario with sleeping time 60 s and awake time 5 s. 52
5.4 Accumulated energy consumed over time by the client node in a Queue Mode scenario with sleeping time 60 s and awake time 5 s. 53
5.5 Accumulated energy consumed over time by the client node in a Queue Mode scenario with sleeping time 60 s and awake time 5 s and 2 hops. 55
5.6 Mean round trip time in a Queue Mode scenario for different number of hops. 56
5.7 Number of retransmissions performed for each packet in a Queue Mode scenario. 56
5.8 Accumulated energy consumed over time by the client node in a Queue Mode randomized scenario with sleeping time 60 s and awake time 5 s. 57
5.9 Mean Round Trip Time in a Queue Mode randomized scenario with sleeping times of 10 s and 60 s. 58
5.10 Current consumed over time by the client node in a TSCH scenario with minimal schedule and a sending period of 60 s. 59
5.11 Total energy consumed by the client node in a TSCH scenario with the minimal schedule and a sending period of 60 s. 60
5.12 Mean Round Trip Time in TSCH scenarios with different schedules and for different numbers of hops. 61
5.13 Number of retransmissions performed for each packet in a TSCH scenario with Orchestra-71. 62
5.14 Energy comparison for sending period 10 s between Queue Mode with dynamic adaptation and TSCH. 63
5.15 Energy comparison for sending period 60 s between Queue Mode with 1 s client awake time and TSCH. 64
5.16 Energy comparison for sending period 5 min between Queue Mode with dynamic adaptation and TSCH. 65
5.17 Round trip time comparison between all Queue Mode and TSCH scenarios, with different numbers of hops. 66
5.18 Packet Reception Rate comparison between all Queue Mode and TSCH scenarios, with different number of hops. ................................................. 67

5.19 Battery lifetime comparison between all Queue Mode and TSCH scenarios, for different sending periods. ................................. 70

5.20 Accumulated energy consumed over time by the client node in a monitoring scenario using Queue Mode with dynamic adaptation and sleeping time 60s. ...................................................... 71

5.21 Accumulated energy consumed over time by the client node in a monitoring scenario using TSCH with Orchestra-17 and notification period 60s. ............................................................... 72

5.22 Accumulated energy consumed by the client node in a Queue Mode scenario with default recommended client awake time (93s) and a sleeping time of 5 min. ......................................................... 73
## List of Tables

3.1 Resources present in the Queue Mode object. .......................... 35

4.1 Necessary features to support the Queue Mode at the server side. . . 38

5.1 Metrics defined for the evaluation of the Queue Mode in comparison with TSCH. ................................................................. 48

5.2 Current consumption values of the Zolertia Firefly hardware platform. . 50

5.3 Transmission and reception duty cycles (in %) in the Queue Mode scenarios. ................................................................. 54

5.4 Transmission and reception duty cycles (in %) in the TSCH scenarios. . 60

5.5 Memory footprint of the different LwM2M software modules with Queue Mode enabled and disabled. .............................................. 68

5.6 Total system’s memory footprint when using Queue Mode or TSCH. . . 68
Acronyms List

API . . . . . . Application Programming Interface
CoAP . . . . . . Constrained Application Protocol
CRC . . . . . . Cyclic Reduncancy Check
CSMA . . . . . Carrier Sense Multiple Access
DTLS . . . . . Datagram Transport Layer Security
FSM . . . . . . Finite State Machine
HTTP . . . . . Hypertext Transfer Protocol
IFFT . . . . . Integer Fast Fourier Transform
IoT . . . . . . Internet of Things
IP . . . . . . . Internet Protocol
ISM . . . . . . Industrial, Scientific and Medical
JSON . . . . . JavaScript Object Notation
LPM . . . . . . Low Power Mode
LwM2M . . . . Lightweight Machine to Machine
M2M . . . . . . Machine-to-Machine
MAC . . . . . . Medium Access Control
OMA . . . . . . Open Mobile Alliance
OS . . . . . . . Operating System
PRR . . . . . . Packet Reception Rate
RD . . . . . . . Resource Directory
REST . . . . . . Representational State Transfer
RPL . . . . . . IPv6 Routing Protocol for Low-Power and Lossy Networks
RTOS . . . . . . Real Time Operating System
RTT . . . . . . Round Trip Time
SICS . . . . . . Swedish Institute of Computer Science
SoC . . . . . . System-on-Chip
TCP . . . . . . Transmission Control Protocol
TLV . . . . . . Type-Length-Value
TSCH . . . . . . Time-Slotted Channel Hopping
UDP . . . . . . User Datagram Protocol
URI . . . . . . Uniform Resource Identifier
WSN . . . . . . Wireless Sensor Network
Chapter 1

Introduction

The IoT is one of the most important engineering topics nowadays among academy and industry. It was created for military applications [1], but it has being applied in many different scenarios in the last years. Some examples are smart cities, smart agriculture, and smart homes. This tendency is predicted to continue increasing in the near future, and some studies show that by 2022 there will be around eighteen billion IoT devices [2]. Two main parts are typically involved in these IoT scenarios, as Figure 1.1 shows: the sensor networks and the cloud. The sensor networks often use wireless communications, forming what is called Wireless Sensor Networks (WSNs) — a research topic that contributed to a large class of IoT systems. WSNs are networks composed of different small devices, called sensor nodes, that are in charge of sensing the environment and can also act over it. These sensor nodes usually contain a microprocessor, a battery, a radio transceiver and antenna, and several sensors and actuators. The nodes collaborate and communicate with each other to send their collected data over the Internet to the cloud. This is the other part of the classical IoT scenarios, which is formed by several servers that receive the data from the sensors and process it for the purpose of the application, and also control and manage the nodes by sending messages to them. The connection between the two parts are usually the border routers, which are special nodes that act as sinks receiving all the messages from other sensor nodes in the network and forward them through an Internet connection to the servers present in the cloud.

Figure 1.1: Typical IoT scenario with a sensor network connected to the cloud through a border router.
One of the main challenges in sensor networks is to have low power consumption in the nodes in order to achieve high battery lifetimes, sometimes reaching several years. The hardware part that usually consumes the most is the radio. In order to minimize its consumption, the main strategies that have been developed over the years are usually located in the low layers of the network stack, which are the Medium Access Control (MAC) and the physical layer. These layers use radio duty cycling, which means that the radio is turned off most of the time, and is turned on only in certain instants, determined by a short period, in order to enable the communication. The time when the radio is turned off is on the order of hundreds of milliseconds, and the time when the radio is turned on is on the order of a few milliseconds, thus resulting in a low radio duty cycle that achieves a high energy efficiency.

Nonetheless, this strategy is usually not enough to achieve the desired lifetime, and also it is made independently in these two network stack layers, without taking into account the protocols that run on top of them. Moreover, the strategy is also applied internally in the sensor network without taking into account the communication with the servers present in the cloud, which is in the end what defines the purpose of the IoT scenario.

Nowadays, full standard protocols have been developed and started to be used for communication in IoT scenarios. One of these is Open Mobile Alliance (OMA) Lightweight Machine to Machine (LwM2M), an application layer protocol which is in charge of managing the sensor nodes using a normal Representational State Transfer (REST) [3, Chapter 5] interface. This interface makes it compatible with other existing and widely used Internet standards, like Hypertext Transfer Protocol (HTTP) [4]. The LwM2M protocol contains a low power mode, called Queue Mode, in which the sensor nodes can sleep for long periods of time where the server cannot communicate with them, and they wake up at some points to enable this communication. In the end, the application layer is the one that defines the communication that is made in the scenario, and therefore this strategy could be more efficient than the radio duty cycling used by the MAC and physical layers. For many applications it is not necessary that the radio is turned on so often as is done by these MAC methods, simply because there is no communication to do in many of those time instants. Examples of these applications are the monitoring scenarios where the nodes are sensing the environment, like a room or a mechanic structure, and sending the collected data to the server. If this data is not critical, it can be sent within long periods of time, or only when there has been a significant change in the values. Therefore, the periods where the radio is off can be much longer in these cases, achieving lower radio duty cycles and therefore higher energy efficiency.

1.1 Problem Statement

In general, there is little knowledge about how strategies for low power consumption affect LwM2M, as it is a protocol that has been published recently. Moreover, there is even less knowledge about the Queue Mode operation. Its specification is brief and open to interpretation, and for the moment there are not many implementations or evaluations of it.
1.2. Contributions

Therefore, the main goal of this Master’s thesis project is to evaluate the performance of LwM2M when using low power strategies at different levels of the network stack. For this purpose, a comparison between the mentioned MAC and physical layer strategies with the Queue Mode is done, for which is necessary also to implement this mode. The MAC protocol that is used for the comparison is Time-Slotted Channel Hopping (TSCH), because it has been shown that it achieves a good energy efficiency and it is also the standard protocol in IEEE 802.15.4 [5]. This evaluation is performed in the testbed present at RISE SICS ¹, inside an office environment. Hence, the project has two different parts:

- The implementation of the protocol at both communication sides: the client side inside the Contiki-NG Operating System (OS) [6] that runs in the IoT nodes, and the server side inside the Eclipse Leshan software [7]. This phase includes the corresponding testing to ensure the correct operation of the protocol, according to the specification.

- Evaluation of the LwM2M operation using low power strategies, by comparing the Queue Mode with TSCH. Several metrics are examined, such as battery lifetime, round trip times and packet losses, and also different scenarios with diverse parameters are used in the experiments in order to cover as many IoT applications as possible.

1.2 Contributions

This project has one main scientific contribution, which is to gain an understanding of how LwM2M’s performance is affected by different methods for reducing the energy consumption related to communication in different layers of the network stack. The selected methods are the MAC protocol TSCH and the Queue Mode included directly inside LwM2M. As a result of the performed evaluation, it is shown in which kind of situations and applications each protocol is a better option to adopt, and therefore the thesis can be used as a guideline for making decisions about the low power strategies in future projects. To the best of our knowledge, this is the first performance evaluation for LwM2M of this kind, and may guide future work on low-power wireless networking stacks.

In order to achieve this goal, the implementation of the Queue Mode has been done at both sides of the communication. This implementation is done in two different software stacks, which are Contiki-NG and Eclipse Leshan. The code developed for the server side has been accepted as a pull request and integrated in the master branch of the Eclipse Leshan GitHub repository [8], which shows the success of the performed implementation and makes a contribution to the IoT community. The code developed for the client side is also planned to be integrated in Contiki-NG. Therefore, all the implemented software in this Master’s project is open source and can be used in future LwM2M projects.

¹RISE SICS: https://www.sics.se/
1.3 Thesis Structure

The remaining part of the thesis is structured as follows. Chapter 2 contains all the background necessary to understand this project, including the OS that is used, and all the communication protocols involved at different layers of the network stack. It also explains how to measure power consumption in order to make the evaluation. Chapters 3 and 4 include the implementation of the Queue Mode in the client and server side respectively, explaining all the details and the design decisions that have been made. Chapter 5 contains the evaluation and results obtained, pointing out the main differences between the operation of both protocols according to the defined metrics. Finally Chapter 6 includes the conclusions drawn from the evaluation part, and makes suggestions for the future work.
Chapter 2

Background

This section explains the important background concepts that are involved in this thesis, for a good understanding of the report. First it introduces Contiki-NG, the OS that runs in the IoT devices, with all the features that make it suitable for the purpose of this project. This section also describes some of the details of its internal implementation. Then, the main protocols that are used and modified are explained in detail, which are the two application protocols Constrained Application Protocol (CoAP) and LwM2M, and also TSCH which is the MAC protocol used in the comparison. Finally, the software module used to measure the energy consumption is also described.

2.1 Contiki-NG Operating System

Contiki-NG (Next Generation) [6] is an OS designed for resource-constrained devices in the IoT, which is based on the Contiki OS [9]. Contiki was developed to run in microprocessors with constraints in terms of processing capabilities, memory, and power supply among others. The main purpose of this OS is to connect tiny devices to the Internet and therefore it has been widely used in the development of IoT networks. For this, it has several communication protocols implemented and ready to use at different levels of the network stack, such as the application protocol CoAP [10] or the transport protocols Transmission Control Protocol (TCP) [11] and User Datagram Protocol (UDP) [12]. This OS is implemented using the C language and can run in a lot different platforms, with different microprocessors, for example the Tmote Sky Board with the MSP430 or the Zolertia Firefly with an ARM Cortex-M3. Contiki-NG was released recently, and it adds several features to the old Contiki OS like a new configuration and logging system, a new lightweight and reliable IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL) [13] implementation, or a network administration shell, and also brings a massive clean up to the code base. It has also a much more active community, as its GitHub page [6] shows.

The Contiki-NG OS is a reference in the IoT field. It is widely used as the software stack for the constrained devices that conform the network, and it is competing against
other important OSs like FreeRTOS [14] and RIOT [15]. Contiki-NG has been listed in the Eclipse whitepaper about software stacks required for IoT architectures [16] as a reference of open source OS for resource-constrained devices, which shows how powerful is this OS inside the field.

All these reasons lead to the decision of using this software in the Master’s thesis. The original plan was to use the normal Contiki OS, but Contiki-NG was released while the project was going on, and it was decided to switch to it in order to be up to date.

There are several interesting features that make this OS suitable for its goal. Some of these features are:

- **Full IP [17] Networking:** one of the main parts of Contiki-NG is the communication stack uIP. This is a complete certified IP stack, which nowadays is focused on IPv6 [18]. It can work with both transport protocols TCP and UDP. In this thesis UDP is used because it is the transport protocol that runs with CoAP. Contiki-NG provides a full Application Programming Interface (API) with all the functions used to open, connect, send and receive UDP packets.

- **Memory Management:** Contiki-NG provides a library to manage the RAM memory by allocating and freeing blocks, which constitutes an efficient way to manage the memory in constrained devices. It is recommended to use Contiki’s functions rather than the standard malloc. Also the OS code footprint is small, which is necessary because devices have also restrictions in the amount of ROM.

- **Power Efficiency:** this OS is intended to be used in devices that are usually operated with batteries, and therefore their power consumption is a critical aspect. For this reason, Contiki-NG has been developed with low power consumption in mind. Some examples of it are the different low power MAC protocols for reducing the power consumption of the radio, which is one of the main parts in this aspect (Section 2.5). Contiki-NG also provides ways to measure the power consumption during run time, which can help the device to take decisions or operate in different ways according to it (Section 2.6).

- **Implementation of other Communication Protocols:** apart from uIP, Contiki-NG contains the implementation of different communication protocols at different levels of the network stack. An example of this can be 6LowPan [19], which is used to adapt IPv6 packets to the IEEE 802.15.4 protocol used for the MAC and Physical layers [5]. Also Contiki implements RPL protocol, which is in charge of building the tree topology and establish the routes inside the sensor network. Contiki-NG provides a new more lightweight and reliable implementation of RPL than the old one present in Contiki. But the most important protocols that are going to be used in this thesis are CoAP and LwM2M (Sections 2.3 and 2.4). LwM2M works on top of CoAP and it is the protocol that is going to be modified in order to implement the Queue Mode.

- **Useful Developing Tools:** Contiki-NG provides a set of tools to simplify the development of applications. It has a user-friendly build system, which makes it
2.1. Contiki-NG Operating System

easy to compile and upload programs to different hardware platforms, and also modify different aspects of the software stack, like the MAC protocol to use. It also provides a logging tool for debug purposes, and a shell to check and change different parameters during runtime. It also includes a network simulator called Cooja [20], with which a developer can test the applications before they run on actual hardware.

- **Libraries**: the OS contains a diverse set of libraries that helps to create Contiki-NG applications. These libraries include the management of linked lists and ring buffers, the generation of random numbers, and the calculation of Cyclic Redundancy Check (CRC) and Integer Fast Fourier Transform (IFFT).

After explaining all the different features of this OS, further details regarding how it is internally implemented are going to be explained, focusing on processes and timers as they can be considered a key part of any Contiki-NG application.

### 2.1.1 Protothreads, Processes and Events

Before going into detail about the processes, it is necessary to understand what is a protothread and how it works [21]. Protothreads are a programming abstraction which aims to reduce the memory overhead of usual threads. In order to do that, they do not have their own memory stack, all of them share the same one. Every time a context switch occurs, which means that the actual thread is stopped and a new thread is executed, this memory stack for the protothreads is rewound. Therefore the developer has to declare as static variables the context that wants to be saved or it will be overwritten. The main advantage of this is the amount of memory that is saved compared to normal threads with their own stack. On the other hand, the main disadvantage is that the developer needs to manually block or yield the threads by calling defined macros, and save the context. The scheduler cannot perform the context switches itself as with normal threads, due to the fact that they do not have assigned memory stack.

Processes constitute the most important part of the Contiki-NG OS. Every program in Contiki-NG is a process or a set of them running together, and they have two main parts:

- **Process control block**: it is a structure that contains information about the process, like the name, a pointer to the process thread (the body of the process), and the state of it. This structure is just used by the Contiki-NG kernel to manage the process, but it is never accessed by user’s code.

- **Process thread**: it is a protothread which contains the code that runs when the process is executed. They always begin with the macro `PROCESS_BEGIN()` and finish with the macro `PROCESS_END()`.

In Contiki-NG there are two different execution contexts: the cooperative and the preemptive, as Figure 2.1 illustrates. The cooperative is where all the processes run and
are executed in a sequential way until they end or until they have to wait for an event, which is done by calling the macros \texttt{PROCESS\_WAIT\_EVENT()} or \texttt{PROCESS\_YIELD()}, among others. So there is no possibility to assign different priorities to different processes and preempt each other, as in Real Time Operating Systems (RTOSs). The kernel entity in charge of executing the processes is called the process scheduler. At any time, it invokes the process that has to be executed, by calling the function that implements the process thread. On the other hand, the preemptive context is where interrupts and callbacks from the real-time timer run. They preempt the processes in the cooperative context at any point, as soon as they arrive.

![Processes execution in the Contiki-NG OS.](image)

Contiki-NG can be seen as an event driven OS. Every action in Contiki-NG is caused by events, since processes are executed when a specific event happens. There are two different types of events: asynchronous, which are first put into a queue and then delivered to the receiving process, and synchronous, which are instantly delivered to the receiving process. Processes can communicate with each other through events or get events from other parts of the OS, like timers. The process scheduler mentioned before is the one in charge to run the processes as a response to an event and deliver this event to it. There are different types of events, each one with a different identifier. Some examples are the \texttt{PROCESS\_EVENT\_INIT}, which is sent to a process when it is initiated, and the \texttt{PROCESS\_EVENT\_TIMER}, which is sent to a process when the event timer (Section 2.1.2) has expired.

### 2.1.2 Timers

Contiki-NG contains a set of timer libraries that can be used to manage time inside programs. It has one clock module, which handles the system time, and a set of timer modules, which are:

- **Timer Module**: it provides several functions to set, reset, and restart timers, as well as a function to check if the timer has expired. The application should check manually whether it has expired. It uses system clock ticks for the count, which means that the count variable is overflowed quickly, and therefore is not suitable for counting large periods of time.

- **STimer Module**: the functionality is the same as the previous one, but it uses seconds for the count, allowing much larger periods of time without overflow.
• **ETimer Module**: this timer generates an event with the identifier \texttt{PROCESS\_EVENT\_TIMER} when it has expired. Therefore, a process can wait for this event to do something with a concrete period of time. The etimer has to be started before the macro \texttt{PROCESS\_WAIT\_EVENT()} is called.

• **CTimer Module**: in this timer module a callback function is executed when the timer expires. This function is executed in the context of the process that started the ctimer, therefore the callback function is executed in the cooperative context, and may need to wait for the execution of another process in this context.

• **RTimer Module**: this timer also executes a callback function when it has expired, but this function is executed in the preemptive context, and therefore it can preempt any other running processes. This offers a reduced way to schedule real time tasks, which is not necessary in this project.

### 2.2 Network Stack

This section shows the network stack that is used in this thesis, and explains briefly the protocols that conform it. The most important protocols for the development of the Queue Mode are explained in detail in the next sections. Figure 2.2 shows this network stack.

![Network Stack Diagram](image)

**Figure 2.2**: Network stack used for the communication in the developed IoT networks.

The different protocols present in the network stack are:

• Lightweight Machine to Machine (LwM2M): it is one of the two protocols that conform the application layer in this network stack. It is used for device management, and it is explained in detail in Section 2.4.

• Constrained Application Protocol (CoAP): it is the other protocol that conforms the application layer, and it is used as a reduced web transfer protocol for constrained devices. It is explained in more detail in Section 2.3.
• User Datagram Protocol (UDP) [12]: it is the transport layer protocol. It is used to send datagrams between two different hosts on an Internet Protocol (IP) network, in a connection-less way.

• Internet Protocol version 6 (IPv6) [18]: it is the network protocol that provides an identification and location system for computers inside networks and routes traffic across the Internet.

• 6LoWPAN (IPv6 over Low-Power Wireless Personal Area Networks) [19]: this layer acts as an intermediary between IPv6 and 802.15.4. Its purpose is to allow constrained devices in sensor networks to communicate through internet using the IP protocol. For this purpose, it defines methods of encapsulation and header compression to adapt IP packets to the 802.15.4 protocol.

• Carrier Sense Multiple Access (CSMA) [22]: it is the MAC protocol that is used when Queue Mode is enabled. In this protocol the radio is turned on all the time and every node verifies the absence of any other traffic in the shared medium before transmitting, to avoid collisions.

• TSCH [10]: this is the MAC protocol used in the comparison, when Queue Mode is disabled. It is the standard in IEEE 802.15.4, and its main purpose is to reduce the radio duty cycle by turning off the radio. In order to achieve that, it divides the timeline in different slots that are assigned to transmit, receive, or sleep (Section 2.5).

• IEEE 802.15.4 [5]: it is used as the physical layer. It provides frequency channels in different ISM (Industrial, Scientific and Medical) bands. There are 16 channels in 2.4GHz, 10 channels in 915MHz and 1 channel in 868MHz. It has data rates of 250, 40 and 20 Kbps, depending on the configuration.

2.3 Constrained Application Protocol (CoAP)

CoAP is a web transfer protocol designed to be used in constrained devices and networks, for Machine-to-Machine (M2M) communication. It was designed in order to be compatible with Hypertext Transfer Protocol (HTTP) [4] and it consists basically on a smaller set of HTTP methods with some other features added to operate in this kind of networks, like multicast support. This protocol is being specified in the RFC 7252 [10].

The protocol is based on the Representational State Transfer (REST) model [3, Chapter 5], where servers have different resources available under their corresponding Uniform Resource Identifiers (URIs) and clients can interact with these resources using the methods GET, PUT, POST and DELETE. By implementing this well known model, it can be fully compatible with existing HTTP solutions. Only a proxy is needed to translate the requests and responses from one protocol to the other, in order to be compatible, as the message format of CoAP is slightly different, as Section 2.3.2 explains.
CoAP is located in the application layer of the network stack, and it works on top of UDP and IPv6. This is another difference with respect to HTTP, which usually works on top of TCP. Therefore CoAP has to deal with some of the TCP features that UDP lacks, like the acknowledgement of messages in order to confirm the reception of them. It can also use Datagram Transport Layer Security (DTLS) [23] on top of UDP for communicating in a secure way.

The protocol is implemented inside Contiki-NG in an efficient way, and it has also been proven to be energy efficient by using radio duty cycling [24]. This protocol is not going to be modified in this project, but it has to be understood because LwM2M works on top of it.

### 2.3.1 Messaging Model

CoAP bases its communication on a request/response model between two endpoints, a server and a client, like HTTP. The server has internal resources that are identified by an URI, and the client makes different requests in order to interact with them. The server is in charge of processing the request, perform the corresponding action, and generate the response to inform the client what has happened with its request.

CoAP packets can be transmitted in two different ways:

- **Confirmable**: it consists on a reliable transmission. The receiver of the message has to send an acknowledgement message to the sender to confirm the proper reception. If the sender does not get any ACK, it retransmits the packet with an exponential timeout until it gets the confirmation from the receiver or it reaches the maximum number of possible retransmissions. The receiver can send two different messages: an ACK, which means that the message has been received and can be processed correctly, or a RESET, which means that the message has been received but there are some missing context to process it.

- **Non-confirmable**: this is the non-reliable way. The sender transmits the message just once and does not wait for any acknowledgement.

A request packet can be carried in a confirmable or non-confirmable message. In the former case, the server has two different options. The first one is to send the response directly in the ACK message, which is called piggybacked response. The second one is to first send an empty ACK to tell the client that the message has arrived properly, and then generate what is called a separate response in a different message, which can be again confirmable or non-confirmable. In order to match the request with its corresponding separate response, there is a field in the header called token (Section 2.3.2) that has the same value in both messages.

### 2.3.2 Message Format

CoAP messages are formed by two different parts: a header and a payload. The header consists on 4 fixed bytes at the beginning that include: CoAP version, type of
the message (Confirmable, ACK, etc.), token length, the code of the message indicating the method of the request or the response code, and the message ID. This fixed header is followed by a variable-length token, used to bind a request to a response as Section 2.3.1 explains. Then there is a set of options that can be included, for many purposes. Finally, if the message contains a payload, it is included in the packet starting with a fixed 0xFF byte to indicate the start point of it.

The options part of the header can be used for different purposes. One of the main ones is to carry the different URI parts, which Section 2.3.3 explains. Also these options can be used to carry out block wise transfers. When a request or a response does not fit in just one CoAP message, it needs to be sent in different messages. For doing this, the options Block1 and Block2 can be used, to indicate the position of the current sent block in the total message and the amount of blocks that are left.

The message payload is used to send the concrete information of the specified resource. For example, in the response to a GET request, the payload contains the information read from the resource. In a PUT request, the payload contains the information to be written. There are multiple formats for this payload, such as plain text, Type-Length-Value (TLV) or JavaScript Object Notation (JSON).

2.3.3 URI scheme

One of the main parts in any web transfer protocol is the URI. It identifies a concrete resource present in the server, so that the clients can interact with it. In CoAP the URI has the following main parts, which are carried in the options field of the CoAP header:

- URI-Host: identifies the host where the resource is located.
- URI-Port: identifies the transport port that can be used to access the host.
- URI-Path: indicates the specific location of the resource inside the host.
- URI-Query: can be used to include different information inside the URI, for example the name of the CoAP endpoint.

The CoAP URI is important in order to understand the LwM2M protocol. The messages that this protocol uses are mapped to normal CoAP messages using the different parts of the URI (Section 2.4).

2.3.4 CoAP Methods and Response Codes

CoAP uses a small set of the standard request methods and response codes from the REST model. The four available request methods are:

- GET: used to read the representation of a concrete resource.
2.4 Lightweight Machine to Machine (LwM2M)

LwM2M is an application layer protocol designed and specified by OMA for device management in sensor networks and M2M environments [25]. This protocol runs on top of CoAP (Section 2.3) and uses its REST interface for the communication. LwM2M extends the resource model to enlarge the features of CoAP, but the use of the REST model makes possible to still integrate it with HTTP.

This protocol defines the communication between a LwM2M server and a LwM2M client. The client is usually a sensor node inside a network, and the server can be any machine located on the Internet. This naming is confusing because both a LwM2M client and server can act as a CoAP client or server, depending on the type of communication that they are having in each moment. This means that a LwM2M client can act as a CoAP client by making requests and receiving responses from the server, but also as a CoAP server receiving requests from the CoAP client, which in this case would be the LwM2M server. Moreover, the client has the resources that the server interacts with, which is also a confusing fact from this naming. So the LwM2M server must be understood as the coordinator and the LwM2M client as the node, and not as the normal client and server naming used in the REST model.

The specification defines four different types of interfaces, which conform the four different communication scenarios that the client and the server can establish between them, for different purposes. This is explained in the next sections, which also show how the messages are mapped to the CoAP protocol.

In the following sections, when client or server are mentioned they should be understood as the LwM2M ones. When the report refers to a CoAP client or server, the protocol name is specified.

2.4.1 Resource Model

OMA LwM2M defines a simple resource model where all the useful information that the client holds and the server interacts with consist of resources inside object instances. Figure 2.3 shows an scheme of how this resource model is structured. The client has different kinds of objects, which are the definition of a set of resources that have something in common. Each of these objects has an identifier. An object example could be a temperature sensor object or the device object containing all the

- **PUT**: request the server to update or replace a resource.
- **DELETE**: for erasing a resource from the server.
- **POST**: request a server to create a new resource.

In response to this methods, the server sends a code to indicate the client what has happened with the request. Some examples of these codes are: 2.01 Created, 2.05 Content, 4.01 Unauthorized or 5.00 Internal Server Error.
important information of the node (manufacturer, version, etc.). These objects are just a definition, and therefore they need to be instantiated so that the client and the server can interact with them. This is done by the client, on its own or by a request from the server, creating a new instance with a concrete identifier, and with concrete values for the resources. It is in this moment when the server can start to interact with the object instance. One client can have several instances for the same object, for example if the node contains several temperature sensors.

LwM2M runs on top of CoAP and uses the URI path of the request in order to interact with the different objects, instances and resources that the client contains. To interact with a concrete resource, the server request should contain the three identifiers in the way <Object_ID/Instance_ID/Resource_ID>. If only an object instance with all its resources wants to be requested, then the path should be <Object_ID/Instance_ID>.

It is also important to remark that OMA defines eight default objects, with the identifiers 0-8, which cannot be used for other purposes. These objects are, among others, the Security object, with all the security information for the communication; the Device object, with all the information about the node platform; and the Firmware Update object to update the client’s firmware. By using these objects, the server can manage different aspects of the client node in an easy way, and this is the reason why LwM2M is called a device management protocol. By interacting with these resources the server can change several aspects in the device during run time, like the security key used in the communication, or the firmware that the node should run.

There are other predefined objects but they are external to OMA. These are called the IPSO objects, created by the IPSO Alliance [26]. They define objects that are commonly used in IoT environments, for example sensors, actuators, and light controls.

With this simple resource model, LwM2M extends the CoAP one to add more functionality and adapt it to M2M communications, and also to make it suitable for device management.
2.4.2 Bootstrap Interface

Communication in LwM2M is done in four different interfaces, which can be seen as different scenarios, each one with a different purpose. The first one is the bootstrap interface. In this interface, the client communicates with a bootstrap server in order to get the essential information to start the registration in the LwM2M server (Section 2.4.3), for example the server address or the security protocol. This interface is optional, the information to initiate the registration can be configured in the client before the deployment of the network and then the registration can be made directly without previous communication with a bootstrap server. This can be directly programmed in the node’s firmware, which is called factory bootstrap, or the information can be present in an card that the node reads, which is called bootstrap from smart card.

In case a bootstrap communication is necessary, there are two options: client-initiated bootstrap and server-initiated bootstrap. In the first one, a request from the client is necessary to start the communication, and in the second one is the bootstrap server the one that initiates. Figure 2.4 shows this communication scheme. In this case, a client-initiated bootstrap is done. In case the server initiates, the first message (bootstrap request) disappears.

![Communication scheme in the Bootstrap interface of LwM2M.](image)

This interface consist on five steps:

1. First the Client sends a bootstrap request to initiate the communication. This
is done by sending a POST CoAP message with the URI path /bs and in the URI query the name of the endpoint.

2. After receiving the request, the bootstrap server checks if it has bootstrap information related to that endpoint name, and if true it sends a response code 2.04.

3. Then the bootstrap from the server starts. The first action that the bootstrap server takes is to send a DELETE message to delete previous objects that the client may have.

4. After that, the bootstrap server starts creating the necessary objects for doing the registration to the correspondent server. It sends several PUT or POST messages in order to do it. The main objects that the bootstrap server needs to create are the security object, with all the security information that is necessary (type, keys, certificates, etc.) and the server object, which contains the important information about the server (address, port, etc.). All these messages are responded by the client with the response codes that tell if the creation has been successful or not.

5. When the bootstrap server has finished creating the objects, it sends a bootstrap finished message. This message consists of a POST message with the URI path /bs.

After doing this exchange of messages, the client has all the necessary information to do the registration in its corresponding server.

2.4.3 Client Registration Interface

This interface is used by a client to register in a server, de-register or update the registration information. This step is mandatory to start the communication using the LwM2M protocol. By this registration, the client provides the server with the necessary information to communicate with it, like the endpoint name, and also specifies the objects that it supports and the instances that it has, in a directory format (<Object_ID/Instance_ID>). So after doing this, the server knows the resources that the client contains and is able to start interacting with them. A server can have different clients registered on it at the same time, and a client can be registered in different servers.

In this interface the client is always the one who starts the communication with a register message, which contains all the necessary information already mentioned. This register message contains four main parameters in the query:

- The endpoint client name.
- The lifetime, which determines how long the server will keep the client registered. If after this time the client doesn’t send an update message, it is removed from the server’s registered clients list, as it interprets that the client is no longer available.
- The LwM2M version.

- The binding mode. This parameter is not mandatory, but it’s really important for this project because it determines if the Queue Mode is enabled or not (Section 2.4.6). This parameter is also used to indicate the transport protocol that runs under CoAP (U for UDP and S for SMS). In case the Queue Mode is used, a Q is added to the binding mode.

Figure 2.5 shows the message exchange in this interface. First, the client sends a Register message with all the parameters mentioned above. This message consists of a POST with the URI path `/rd`, the query with all these parameters and the payload with the object instances. The server responds with the corresponding response code including the location, which is an identifier for this client, and then it saves the client information in the registered clients list. This location must be used by the client when doing an update or a de-registration. Then, the next interface can start (Section 2.4.4).

![Communication scheme in the Client Registration interface of LwM2M.](image)

Before the lifetime expires, if the client wants to keep registered in the server, it needs to send an Update message with a new lifetime. This message is again a POST with the URI path `/rd/"location"` and a query with the new lifetime. If the client
does not send this message, the server de-registers it. Finally, if the client wants to
de-register itself manually, it sends a de-register message, consisting of a **DELETE** with
the path `/rd/"location"`.

## 2.4.4 Device Management and Service Enablement Interface

This is the interface where, after knowing all the client necessary information, the
server can start submitting requests to the client to interact with its resources. There
are six different types of operations that a server can do. Figure 2.6 shows some
examples of these operations.

![Communication scheme in the Device Management and Service Enablement interface of LwM2M.](image)

These operations can be listed in:

- **Read**: this operation is used to read the value of a full object, a concrete object
  instance or a concrete resource from the client. This is performed by sending a **GET** message to the specific path of the resource, instance, or object. The client
  sends a response with the content requested in the payload. This content can be
  sent in different formats, for example text plain, TLV, or JSON.

- **Discover**: this operation is used by the server to know all the objects that the
  client has, or all the attributes that a concrete object has. This is also done with
  a **GET** message.

- **Create**: with this operation, the server can create a new object in the client. It
  sends a **PUT** or a **POST** message with the object id to be created as the URI path.
• **Write**: it is used to change the value of one or more resources in an object. It consists of a PUT or POST with the path to the concrete resource and the content to be written in the payload.

• **Execute**: there are some resources that are used to perform actions inside the client. To execute these actions, the server can send an execute operation to the concrete object, which consists of a POST message to the resource path.

• **Delete**: used to remove an object instance in the client. It consists of a DELETE message to the object instance path.

With these operations, the server can manage the different client nodes. It can get some useful information from the client, like a read operation to the sensor temperature object, or execute a concrete action in the client, like activate an actuator. Also, using the defined objects, it can also update the firmware of the client by creating a new firmware object, or change the security parameters of the communication by writing different values in these resources. This is what makes LwM2M a device management protocol.

### 2.4.5 Information Reporting Interface

This interface is used by the server to observe a concrete resource in a client’s object. The client sends the new value of the resource whenever it changes, or when the period expires if it is a periodic resource. So this interface is useful for the server to keep track of a concrete resource. Figure 2.7 presents the communication scheme.

It consists of three different parts:

• **Observe**: with this message the server tells the client which resource it wants to observe. It is a GET message with the observe option set to 0 and the path to the corresponding resource. In this case the token part of the CoAP header needs to be used in order to link the future notifications with this observation. The client sends a response with the content of the resource at that moment, and gets ready to send notifications every time the resource changes in the future.

• **Notify**: in this part, the client sends notifications with new values of the resource to the server. A notification consists of a response message with the code 2.05 Content, the same token as the observe message, the observe option set to 0, and the value of the resource in the payload. To perform a notification there are different conditions. There is a minimum period in which if the resource’s value changes, the notification is not sent. There is also a maximum period when, if the value has not changed yet, a notification is sent anyway. There are also some possible conditions of the type greater than, lower than, or step, which determines if the value’s change is enough to be notified. All these parameters can be set by the server in what is called a write attributes operation, which consists of a PUT message from the server with all this parameters in the query. The client saves them to use in the information reporting interface.
Cancel observation: with this message the server tells the client that it wants to stop observing a concrete resource. It consists of a GET message with the observe option set to 1 and the path to the resource.

This interface can be useful in the context of IoT. A typical example is a monitoring scenario where the server wants to keep track of specific parameters, like the temperature of a room or the state of a mechanical structure. The server can send an observe message to the object inside the node, and then when the parameter changes (and this change fulfils the conditions set for a notification) or the period expires, the client sends a notification.

### 2.4.6 Queue Mode

This is the main topic of this Thesis. Queue Mode is a part of the LwM2M protocol created with the goal of saving energy at the client side. Its purpose is to allow the client to sleep for long periods of time, when the server cannot make any requests to it. The specification of this low power mode is quite brief and there are not many implementations and evaluations of it.

In Queue Mode, when the client registers with the server, it includes the character
'Q' in the binding mode inside the query. This has to be interpreted by the server as a registration with the Queue Mode enabled, meaning that it would not be able to send requests to the client at any time. Instead, it has to wait for a message from the client telling that it is awake again, and then it can make these requests. In this time where the client is sleeping it is up to the server what to do with the requests that can be generated. One option would be that these requests can be saved in a queue, which is the reason for the naming of this mode, and then sent them when the client is awake again. But this is not a part of the specification, this queue is not mandatory and the application running in the server can manage it in the way it wants.

This mode is implemented by making use of the registration interface, changing two aspects of it. The first thing is to include the Queue Mode binding mode in the registration message. The second is to send an update message when the client is awake and it is ready to process incoming requests. This update message can be also used to update the life time or other aspects of the session as in normal mode, but has to be interpreted by the server as a change in the client state, from sleeping to awake.

![Communication scheme using LwM2M protocol with Queue Mode enabled.](image)

Queue Mode can operate in both interfaces, Device Management and Service Enablement, and Information Interface. The first one can be divided in three steps,
as Figure 2.8 shows:

- The client sends the registration message with 'Q' as the binding mode ('UQ' if it uses UDP).

- The client can go to sleep, but first it has to wait some time in order to receive new server requests. This time is going to be called client awake time in the rest of the report. The specification recommends to wait for the constant time MAX_TRANSMIT_WAIT present in the CoAP protocol. By default this time is 93 seconds, which is not efficient for low power consumption in the client nodes, as the radio has to be turned on all this time. This time has been the object of an open discussion in the OMA LwM2M GitHub page \(^1\). Some people suggested to send a message from the server telling that it has no more requests. With this, the 93 seconds would be only a worst case when this message is lost, and therefore the client saves energy in most of the cases. This is not included in the specification yet, so there can be different approaches about how much time the client should stay awake, as the MAX_TRANSMIT_WAIT is only a recommendation. After this time, the client can go to sleep, put the CPU in low power mode and turn off the radio.

- At some point, the clients wakes up and sends an update message to the server, informing that it can receive and process some requests. Then the server can communicate. After processing these requests, the client can go to sleep again, after waiting for the client awake time, as in step 2. There is no specification about how long the sleeping time should be, or who determines it: the client, the server, or an agreement between them.

For the information interface, the operation is very similar. The only thing that changes is that the client is woken up by a notification event, then it sends the update message to tell the server that it is awake, gets the ACK response, sends the notifications, and then it can receive requests from the server.

With this mode a large amount of energy can be saved by allowing the client to sleep for long periods of time. This mode can be useful when there are no critical resources that the server needs to know very often, for example the temperature and humidity of a room. If there are critical resources that change often, this mode would also work if the server is observing them, but it does not save too much energy if the client is woken up all the time. Moreover, it would consume more energy because the client has to send the update messages.

### 2.5 Low Power MACs in Contiki

The MAC layer in the communication protocol stack is the layer between the network and the physical ones. It provides addressing and channel access control

\(^1\)OMA LwM2M GitHub Page. Issue "Queue mode and timeout for offline": https://github.com/OpenMobileAlliance/OMA_LwM2M_for_Developers/issues/98. Visited: 01/11/2017
In environments where there is a shared medium, like the radio spectrum in a WSN, in Contiki-NG there are different implemented MAC protocols, for example CSMA or TSCH. The first one constitutes an always on protocol where the radio is in listening mode all the time, and it checks if the shared medium is occupied when it tries to send a new message. As the radio is turned on all the time, its power consumption is high. It is the MAC protocol used when the Queue Mode is enabled, in which case the radio is turned off manually by the application layer. On the other hand, Contiki-NG contains TSCH, which constitutes a way of radio duty cycling as a low power strategy at the MAC level of the network stack. The old Contiki OS included another radio duty cycling protocol called ContikiMAC [27], but it has not been ported to Contiki-NG, as it is not a standard protocol. Therefore, TSCH is the protocol chosen to make the comparison with Queue Mode, also because it has been already proven that its energy efficiency is higher than CSMA and ContikiMAC [28], and it is the standard MAC layer in IEEE 802.15.4. Then, the goal of the Master’s thesis is to compare and see if the LwM2M protocol using CSMA with Queue Mode enabled can achieve better energy efficiency than TSCH. Therefore, this protocol is explained in this section for the reader to have a good understanding of it.

TSCH is a MAC layer which offers a globally synchronized network where nodes can turn off the radio for fixed periods of time. The time is divided into different timeslots which are usually 10 ms long, and each slot is dedicated for transmitting, receiving, or sleeping. There can be several slots where a node is transmitting different kinds of traffic. With this strategy, the nodes can sleep and turn off their radios a large amount of the time. This timeslots are grouped into slotframes with a specific length, and each slotframe is repeated periodically. Also when a node wants to communicate, a channel hopping is done, which means that the communication is produced in a different frequency channel for different occurrences of the same slot. Then a time slot has two main parameters: the time offset indicating the position inside the slotframe in which the slot is located, and the channel offset, indicating the frequency of the communication. Figure 2.9 shows a simple example of two nodes communicating with each other using TSCH. Each time the slotframe is repeated, the communication (TX and RX) happens at a different frequency channel.

![Figure 2.9: TSCH slotframe example for the communication between two nodes.](image)

This layer is successfully implemented in Contiki-NG, using timer interrupts and doing all the communication in the callback functions. To communicate with upper layers, a ring buffer is used to queue the received packets or the packets to transmit. Then, the upper layer, which is usually IPv6, puts the packets that wants to transmit...
in the outgoing queue, and read the packets from the incoming queue. This queue-
based implementation favours the portability of the TSCH implementation to other
OSs.

In TSCH the time synchronization is important because nodes have to communicate
with each other at a concrete time in a synchronous way, and therefore there cannot
be a clock drift between them. The time precision needs to be at the micro-second
level. Clock drift is implicit in the crystal oscillators and cannot be avoided, so a
way of re-synchronization is necessary. This time re-synchronization is done when
a frame or an ACK is received from a timer source, which is usually a parent node
in the network architecture. The receiving node compares the reception time with
the expected time and adjusts its clock to it. The Contiki-NG implementation also
includes an adaptive way, where the nodes learn from previous time drifts and adjust
the clock automatically when performing a wake up.

Scheduling is also a main part of this MAC protocol, which determines how the
slots are configured and distributed in time on any node. This has to be done in
an intelligent way so that the nodes can communicate with each other in the same
slot without interfering with other ones. In the Contiki-NG implementation there are
basically two types of scheduling:

- Minimal configuration is a simple schedule. All the nodes have a single
timeslot inside a single slotframe where all the communication happens, based on
contention. Therefore this is not a very efficient way of communicating, because
if there are many nodes in the network, there can be high interferences. That is
the reason why Orchestra was created.

- Orchestra [29] works together with RPL, making it possible for nodes to compute
their own schedules locally based on the structure of the network. The Orchestra
schedule gives one slot for each kind of communication that a node needs in the
RPL architecture, which are: a common slot for all nodes in the network for
RPL signalling, a unicast slot from every node to its parent, several slots from
a node to its children, and a dedicated broadcast slot from every node to send
TSCH beacons. With these four types of slots, all the communication can take
place in the network. In order to compute the schedules locally on every node,
the coordinates (time and frequency) of each slot are based on the sender or
receiving identifier, therefore no third party is needed. Orchestra also makes it
possible to maintain several slotframes for each kind of traffic that the nodes
are sharing. For example there can be one slotframe for application traffic like
LwM2M and another for RPL traffic.

In relation to power consumption, TSCH allows the nodes to use an efficient way
of radio duty cycling. This means that the radio can be turned off in the slots that are
not used to communicate, and it only needs to be turned on when a slot with some
kind of communication arrives, like to send a packet or to listen for possible incoming
messages. Also inside every slot, TSCH does not turn on the radio all the duration of
it to reduce even more the energy consumption. In the transmitting slots, the radio
is turned on only if there is a packet placed in the queue that needs to be sent. In
the receiving slots, the radio is turned on with a guard time of ± 1.2 ms as default. This ± indicates that it turns on the radio 1.2 ms before the starting point of the slot, until 1.2 ms after this point. If there is a packet reception, then the radio remains on until the end of this packet, but if there is no packet reception the radio is turned off immediately. So this strategy allows to reach a much lower radio duty cycle.

TSCH is a more efficient protocol than for example ContikiMAC, in which the radio is also duty cycled, but without any schedule. With ContikiMAC, a node that wants to send a message retransmits this message several times until the receiver tells that it’s awake with an ACK message. On the contrary, in TSCH there is a schedule, specially efficient if Orchestra is used, when the nodes know exactly when they can communicate, so they can wake up only on the dedicated slots, and they only need to send a packet once.

2.6 Measuring Energy Consumption in Contiki

For this project, a tool for measuring the energy consumption of the hardware nodes is needed in order to perform the evaluation of LwM2M using different low power methods, by comparing them. The most accurate way to determine the power consumption of a node is to measure the electric current at its input and the battery voltage. This can be done by adding additional hardware in the node board. But it has been proved [30] that the cost of this module would be similar to the cost of the node itself, and therefore it is not an efficient way to do it. Also it makes impossible to use standard hardware boards directly, and some hardware should be developed, which is out of the scope of this project.

Therefore, a software tool has been developed for Contiki-NG OS, to avoid all the difficulties of measuring the power consumption with dedicated hardware [31]. This tool has been proved to be efficient and accurate. It does not add any additional cost neither much memory footprint. And it is also easy to use by just adding a few lines of code in the hardware drivers. This tool is also an advantage with respect to the mentioned hardware tools, which is that it is possible to measure the consumption of hardware parts in isolation, like the radio or the CPU, and not only the total energy consumption of the whole device.

The method is based on measuring the time in which a concrete hardware component is on, and then take its current consumption from the datasheet, and multiply. The energy model used is:

$$E = I_m t_m + I_l t_l + I_t t_t + I_r t_r + \sum_{i=1}^{N} I_{ci} t_{ci}$$ (2.1)

The symbol I stands for the electric current, and t for the time that a concrete hardware component is on. Then the subindex m stands for the microprocessor in active mode, l for the microprocessor in low power mode, t for the radio transmit mode, r for the radio in reception mode, and ci for other additional components such
as sensors or LEDs. So with this model, all the components are taken into account and contribute to the total energy consumption.

This software method is implemented inside Contiki-NG OS in the files "energest.[hc]". This module maintains a table with all the times that the different components are turned on. Then, in the device driver, before turning on a specific component, the macro \texttt{ENERGEST\_ON(type)} is called, which saves a timestamp using the function \texttt{RTIMER\_NOW()} that returns the system time in ticks. This macro gets the type of component that is turned on. When the same component is turned off again, the macro \texttt{ENERGEST\_OFF(type)} is called, the difference with the previous timestamp is calculated, and the time is added to the table in the entry of the hardware component. Hence, this table contains all the time that all the devices have been turned on, consuming energy.

All this code is already included in the device drivers of the different platforms supported by Contiki-NG, so the developer just has to enable the module by defining the constant \texttt{ENERGEST\_CONF\_ON} as 1. Then, in the main application, some functions can be called to obtain these times, multiply them by the known current consumption, and determine the energy consumption.

This is the tool that is used in this project, as it has been proven to be accurate enough [31], and it is suitable to use. With it, the energy consumption of the nodes is going to be measured in several scenarios using the different low power communication protocols.
Chapter 3

Queue Mode Implementation at the Client Side

The first part of the project is to implement the Queue Mode at the client side of the LwM2M protocol, inside the code that will run in the nodes. For the implementation, the existing Swedish Institute of Computer Science (SICS)’s LwM2M client software is used and modified to add the Queue Mode operation. This code is available in GitHub with an open-source BSD licence [32]. It was firstly designed to run in Contiki, but it has also been rewritten to be portable to other operating systems, and even to be runnable in a normal Linux machine, which facilitates the development. This portability makes it easy to use this software inside Contiki-NG, with just small changes in order to build it. This code needs to be changed as a part of this project to support the Queue Mode.

All the sources files are included inside the directory `contiki-ng/os/services/oma-lwm2m`. There are two main parts in this LwM2M client software. The first one is the LwM2M Resource Directory (RD) client, which manages the bootstrap and the registration interfaces. It is called resource directory because when it registers with the server, it specifies the resources that it has in a directory way, with the path `<Object_ID/Instance_ID/Resource_ID>`. It is implemented in the files `lwm2m-rd-client.h/c` and all the functionality is done with a Finite State Machine (FSM) that executes periodically using a timer. The second important part is the LwM2M engine, which manages the reception and processing of requests from the server, and also is in charge of sending the responses. This means that the engine mainly performs the other two interfaces, device management and service enablement interface, and information interface. But it does not only manage this two interfaces, it is always running in parallel with the RD client, processing also the requests that the server does in the bootstrap phase, for example. It is implemented in the files `lwm2m-engine.h/c`. All the logic is done inside a handler that is executed every time a new CoAP packet arrives, by the CoAP engine that runs bellow. Apart from these two, there are other source files for different purposes inside the LwM2M directory. Some files implement the standard LwM2M objects (server, security, firmware, etc) and other files implement the parsing of the different payload formats (plain-text, TLV, and JSON).
Chapter 3. Queue Mode Implementation at the Client Side

For implementing the Queue Mode, the main parts that need a change to support it are these two: the RD client and the engine, which are the ones that manage the four communication interfaces present in LwM2M.

3.1 Implementation in the LwM2M RD client

The RD client is the part of the software in charge of the bootstrap and the registration interfaces, which have been explained in Section 2.4. Therefore it controls the exchange of messages with the server to perform the corresponding actions defined in these interfaces. Queue Mode does not use the bootstrap interface, and therefore this part of the software does not need to be changed. For the registration interface, two main changes are necessary:

- Send the 'Q' binding mode in the registration message.
- Send a registration update message every time the client is awake and can receive and process new requests. After sending this, the client needs to wait for the client awake time to be sure that there are no more requests. When this time expires, it can go to sleep. As the specification only gives a recommendation for this waiting time, which is 93 seconds, a design decision must be taken in order to set this time because it is too long for energy efficiency purposes. As this section explains in detail later, a dynamic adaptation of this time has been implemented, based on measuring the time between two consecutive server requests.

The RD client is implemented as an FSM, which Figure 3.1 shows. This FSM is executed in a periodic way using the network timer (ntimer). This timer is simply an abstraction for not using Contiki’s timers as default, and therefore not restricting this software to run on top of it. This abstraction makes it possible to port the LwM2M software to other OSs. The network timer has to be implemented for every OS that will run the LwM2M software. If this software is run inside Contiki-NG, as it is the case of this thesis, the ntimer is implemented just by using the event timer (Section 2.1.2).

3.1.1 Finite State Machine operation without Queue Mode

Figure 3.1 illustrates how the FSM without the Queue Mode implementation works.

The first two states of the FSM (INIT and WAIT_NETWORK) are used just to connect the node to the internet, and wait until it has network access, through a border router. After this, if the bootstrap phase is necessary, it moves to the state DO_BOOTSTRAP for sending the corresponding message to the bootstrap server (Section 2.4.2). Then it moves to an intermediate state just for waiting for the server to do the bootstrap phase, and send the response. When the response from the server arrives, a callback is executed by the CoAP engine, and the FSM is moved to the BOOTSTRAP_DONE state. In this state, the client checks if all the necessary objects have been created correctly, and moves the FSM to the DO_REGISTRATION state to start the registration phase. Here the client sends the registration message to the server and then waits for the
response. Once this response arrives, the FSM is moved to the `REGISTRATION_DONE` state. It is at this moment when the client is registered in the server and it is ready to receive requests, according to the device management and service enablement or the information interfaces, which are processed by the LwM2M engine. The client will remain in this state, sending a registration update message every time it detects that the lifetime will expire. Actually, it sends one update message when it detects that half of the lifetime is consumed, just to be sure that it is not deregistered from the server. Therefore, the period of the FSM has to be lower than the lifetime in order to send the update message at a proper time. If the client wants to force a de-registration, it is moved to the state `DEREGISTER` to send the corresponding message.

If at any point the server responds with an error message, the client is driven to the `INIT` state again for trying to restart the registration process. For simplicity these arrows have not been included in the figure.

### 3.1.2 Finite State Machine operation with Queue Mode

For implementing the Queue Mode inside this software as part of this Master’s thesis, the main change in the RD client is the addition of two new states in the FSM to handle it, as Figure 3.2 shows.

In particular these states are `Q_MODE_WAITING` and `Q_MODE_SEND_UPDATE`, which are highlighted in grey. After sending the registration message with the binding mode set as 'Q' in the `DO_REGISTRATION` state and receiving the corresponding response, the machine is moved to the `Q_MODE_WAITING` state. In this state the client waits for the *client awake time* to receive new server requests, and when this time ends, it goes to sleep. When the client enters this state, it stops the periodic ntimer so that the FSM
is not triggered in a periodic way, and starts another ntimer with the client awake time as expiration time. This timer is restarted by the LwM2M engine every time a new request arrives, since the client needs to wait for this time after the last received request. When this timer expires, meaning that there are no new requests, a callback function is called and the client is put in sleep mode, turning off the radio. Therefore, in this implementation the control of the client’s state (awake or sleeping) is done by this part of the software.

When Queue Mode is enabled, no radio duty cycling at MAC level is used. The radio is turned on and off according to the Queue Mode operation, at the same time as the CPU is changed between active and Low Power Mode (LPM). For entering the sleep mode, only the radio is necessary to be turned off, because Contiki-NG automatically puts the CPU in LPM when there are no processes to run. The network stack process run at some time instants, trying to send some packets, for example for the RPL protocol, and therefore putting the CPU in active mode. Also some other process can consume CPU active time, like the Energest module to measure energy consumption. But these processes only consume ta few milliseconds of CPU, so it is considered as a valid option to keep this operation in relation to CPU. The problem that the network stack executions create is that, as how the MAC layers and radio drivers present in Contiki-NG are implemented, when the radio is turned off manually by the RD client, it is turned on again by the network stack when a new packet wants to be sent, and it is not turned off again after sending it. The result of this is that the radio remains listening all the time, consuming a lot of energy. To avoid this, a simple flag is added in the CSMA driver, which is enabled when the `off()` function is called, and then the functions for sending packets do not execute if this flag is active. For turning the radio on again, the `on()` function needs to be called by the RD client when waking up.
So when the client awake time has expired, the radio is turned off and the ntimer in charge of triggering the periodic FSM is reset with the time to sleep as an expiration time, to fire the FSM again when it is time to wake up. When this timer expires, the FSM is moved into the Q_MODE_SEND_UPDATE for sending the update message to the server informing that it can receive requests. When the server responds, current state is changed again to Q_MODE_WAITING and the process starts again. Therefore, by moving between this two states the registration interface with the Queue Mode is handled, and also the action of driving of the node into sleep mode.

3.1.3 Design and implementation of the client awake time

As it has been mentioned before, the client awake time, which is the time that the client stays awake since the last received request, is not specified and only recommended by the OMA LwM2M specification. The problem is that the recommended time is too large for low energy consumption purposes, so something different needs to be designed. What has been implemented in this thesis is an adaptive way of setting this time, in order to be efficient and to not wait for a longer time than needed. This has a high impact on the power consumption, because all the time that the node is waiting for new requests, the radio is turned on in listening mode. So this time should be reduced as much as possible.

The design that has been developed is based on choosing this time in an adaptive way. The idea is to keep track of the time that the server needs between the transmission of two consecutive requests, which means since it sends one request until it sends the next one, after the client’s response. If this time is determined, then the client can go to sleep after it, assuming that there are no new requests because the server would have already sent them after this time. For implementing this algorithm, the LwM2M engine measures the time between two consecutive received requests from the server, saves this times in a fixed size list, to have a window of the last times, and finally adapts the awake time accordingly. The size of this list is decided to be set to ten which is enough for not increasing the memory usage too much, and therefore the last ten times are taken into account. The client awake time is determined taking the maximum value of these times, and adding a margin to it as a percentage.

This margin needs to be set in a safe enough way, because it determines how the client would adapt if the network conditions change. If at some point of the communication the time between two consecutive requests increases more than this margin, for example due to a more congested network or a more busy server, the client would go to sleep before the requests can reach it. As the client does not get any request, no time between two consecutive requests can be measured, then window of times is not updated and the client awake time is not adapted. This causes that no request can reach the client before it goes to sleep, so it becomes unreachable unless the conditions of the network change back to the initial ones.

In the Evaluation chapter, this dynamic adaptation is also tested to see if it is safe enough, or if it is better to have a fixed time, low enough to save energy, but much safer than the dynamic one.


3.2 Implementation in the LwM2M engine

The LwM2M engine is the part of the software in charge to process the requests from the server and send the corresponding response. This is done by using a callback function that is executed every time a new CoAP packet arrives, by the CoAP engine that runs in the network stack. In this callback, the method and the path of the request are checked, and then an action is performed according to it. These actions could be any of the ones that Section 2.4 explains.

This part of the software does not need many changes for implementing the Queue Mode. One minor needed feature is to restart the timer that holds the client awake time when a new request is received if the RD client is in the Q_MODE_WAITING state, which is known by a flag set by the RD client code. This is programmed by just calling the restart function of the network timer.

3.2.1 Design and implementation of the Information Interface with Queue Mode

The LwM2M engine is also in charge of the information interface, which means sending the notifications to the server when it is observing any resources. This engine acts as an intermediary between the CoAP engine and the object implementation. This means that it is the object software that should know when to send a notification and call the LwM2M engine for it, which creates the path according to the instance and resource, and gives it to the CoAP engine to perform the notification. The CoAP engine checks in the saved list of observed resources if this resource has any observers, and in that case it sends the notification. So it is the CoAP engine that checks if there are observers or not, whereas the object implementations and the LwM2M do not care about it.

For implementing the information interface together with Queue Mode, the only thing that is different is that if an event is created while the client is sleeping, then the client should wake up, send the update message, and then send the notification (see Section 2.4.6). For implementing this, whenever an object tries to send a notification, first the LwM2M engine executes manually (not by a timer callback) the RD client FSM and put its state into Q_MODE_SEND_UPDATE for sending the update message, and then it sends the notification. But before waking up the client and executing the FSM, the engine needs to check if there are any observers for the resource, otherwise it would be pointless and energy would be wasted. For doing that, the list of observed resources from the CoAP engine is checked in order to find if the concrete resource is inside it.

When implementing this, a particular problem was detected. When the client sends the update message after waking up by a notification event, it has to wait for the server response before sending any notifications. But in this particular moment when the client is waiting for the response, other resources can create a notification event, and as the client is woken up, the engine sends these notifications directly to the server before receiving the response. Figure 3.3 (a) illustrates this problem. The solution implemented in this thesis (Figure 3.3 (b)) is to save in a queue, implemented as a linked list, all the notifications that are created at this point, and send all of them...
once the response from the server has arrived. With this, the protocol specification is respected.

![Diagram](image)

(a) Problem.

![Diagram](image)

(b) Solution.

Figure 3.3: Problem (a) and solution (b) in the information interface when using Queue Mode.

In the original implementation of this LwM2M software present in the SICS’ software [32], every observed resource is identified by a constant array of twenty characters, to allow having 6 digits for object, instance, and resource identifiers, separated by two slashes (`object_id/instance_id/resource_id`). The problem with these constant arrays is that they contain 20 bytes, so only a few of them can be stored in the linked list inside the RAM memory. This translates in the fact that many notification events can be lost in this short period of time when the client is waiting for the response, because they do not fit in the memory. The implemented solution to this problem is to translate the character array paths to an array of three 16 bit integers, one for each identifier, before storing them. With this, every generated event is just 6 bytes long.
Even with this reduction in the size, not too many events can be saved, just around ten, of course depending on the hardware platform. Therefore, a policy is applied when the queue is full and a new event is generated. The rules that have been designed for this policy, with the goal of making it as fair as possible, are, applied in order:

1. If there are any other events from the same resource, delete the oldest one and insert the new event.
2. If there are more than one event from other resource saved, delete the oldest one and insert the new event.
3. If the list is full with events from different resources, delete the oldest one and insert the new event.

With these rules applied in the correct order, the queue is managed in a fair way, and the newest events from all resources are always kept.

All the implementation of this queue is done in two separate files, lwm2m-notification-list.[hc], to make it easier for future changes and to not include all this code inside the LwM2M engine code.

At the moment when this report is done, the notifications management in the LwM2M implementation was not finished, and only a limited version is implemented, where all the logic resides in the CoAP engine and the object’s software, and the LwM2M is just an intermediary between both of them. Also, the current implementation uses the large character arrays to identify the resource events. This is presumed to change in the near future, and all the logic will be implemented in the LwM2M engine, making it more easy to develop new objects, and making it more efficient than the CoAP engine does. For this implementation, the translation to 16-bit integer arrays and the handling of the notification’s queue can be used.

### 3.3 The Queue Mode Object

The last part of the client implementation done in this thesis is to create an object inside the LwM2M’s resource model for managing the Queue Mode. This object has two different purposes: select the time to sleep and the client awake time, and report some general statistics about power consumption. For that purposes, it has several resources that Table 3.1 shows.

As it has been explained, there are no specifications about the time the client should sleep and about the time it should stay awake. This could be done by some exchange of messages between the client and the server to reach an agreement, because for many applications it would not make sense that these times are directly programmed inside the client’s software. It is necessary that the server can take part of this decision, for the good performance of the whole IoT application. For example for some application it could be fine for the server that the client sleeps for a minute and that it uses the dynamic adaptation for the client awake time (Section 3.1), but in other applications
3.3. The Queue Mode Object

Table 3.1: Resources present in the Queue Mode object.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Operations</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sleeping time</td>
<td>Read/Write</td>
<td>Set the client’s sleep time.</td>
</tr>
<tr>
<td>Client Awake Time</td>
<td>Read/Write</td>
<td>Set the client awake time.</td>
</tr>
<tr>
<td>Dynamic Adaptation Flag</td>
<td>Read/Write</td>
<td>Enable and disable the dynamic adaptation.</td>
</tr>
<tr>
<td>Cycle CPU time</td>
<td>Read</td>
<td>CPU in active mode in the last cycle.</td>
</tr>
<tr>
<td>Cycle LPM time</td>
<td>Read</td>
<td>CPU in low power mode in the last cycle.</td>
</tr>
<tr>
<td>Cycle TX time</td>
<td>Read</td>
<td>Radio transmitting in the last cycle.</td>
</tr>
<tr>
<td>Cycle RX time</td>
<td>Read</td>
<td>Radio listening in the last cycle.</td>
</tr>
<tr>
<td>Total energy</td>
<td>Read</td>
<td>Total energy consumption since booting.</td>
</tr>
</tbody>
</table>

the server might need the client awake every ten seconds and it needs it to stay up for five seconds because it is a more demanding application.

Three basic resources are included in the Queue Mode object, that are readable and writeable. The purpose of these resources is to give the server the capability to decide the sleeping and the waiting times. The first one is precisely the time to sleep, for the server to decide it. Of course the client can also change this value, so both have power in the decision. The second one is a flag to enable or disable the dynamic adaptation of the client awake time, for the server to decide if this should be applied or if a fixed time should be used. The third one is precisely the client awake time, which can be fixed or dynamically changed, depending on the value of the previously mentioned flag.

Apart from these resources, the client can also report some statistics about the power consumption. Five resources are present, which are the times where the four main hardware parts are turned on in the last cycle, and the total energy consumed from the booting. The first four contain the time where the CPU has been in active mode and in low power mode, and the time that the radio has been in listening and transmitting mode. These four actions are the ones that consume most energy in the node, but if some sensors, actuators or other hardware devices are added to it, they can be included easily in the queue object. The times are computed for every cycle, which is the time between two consecutive client waking up instants. Therefore, this time includes the two main parts of the Queue Mode. First, the moment when the client is receiving and processing requests from the server, with the radio in listening mode all the time and transmitting for short times to send the responses, and the CPU is mainly in active mode. And second, the time when the client is sleeping, when the radio is completely off and the CPU in low power mode. A good picture of the Queue Mode functionality can be obtained with these four resources. The fifth resource is the total energy consumed since the node was turned on. It is computed on every cycle by multiplying the obtained times by the current consumption, and accumulating the result in the resource value. Then, this resource represents the total energy/voltage since booting the node.

These statistics have two different purposes. The first one is just for the debugging of the Queue Mode implementation inside this project. The second one is to take
some decisions about the *client awake time* or the time to sleep according to the energy that has been consumed. For example if the client or the server see that the energy consumption is increasing too fast, they can change the sleeping time and put a higher one, in order to save more energy. In this moment nothing has been done in this sense, because the focus for now is to have a basic implementation according to the specification, but maybe in the future something can be done in this direction to have an intelligent sleeping node.
Chapter 4

Queue Mode Implementation at the Server Side

In this section, the implementation of the Queue Mode at the server side done as a part of the thesis is explained in detail. After implementing the Queue Mode in the client side, the next part of the project is to implement it in the server side. This is necessary in order to perform the main goal of this thesis, which is the evaluation of the Queue Mode in comparison with other low power strategies. For this implementation, Eclipse Leshan [7] code is used and modified to achieve the desired performance of this Queue Mode. Leshan is an OMA LwM2M client and server implementation written in Java code. It is an open source project created by the Eclipse foundation in 2014 and provides a set of libraries in order to make it easy for a future developer to make its own client, server, or bootstrap server. It also gives example code of how to implement them, for demonstrations and testing. The Leshan code runs on top of Eclipse Californium [33], which is an implementation of the CoAP protocol that is also written in Java. Leshan uses Californium in order to send and receive CoAP packets.

The server code can be divided in three different layers: the Californium core running as the bottom layer, the LwM2M core running on top of Californium, and a web application running as the upper layer. The web application is used just as an example and contains different buttons for making requests to the registered client, using the functions present in the LwM2M core. This core provides different libraries for supporting the different features that the protocol has. These features are, among others, the client registration, the device management with the corresponding operations (see 2.4.4), and the information reporting with notifications. Some features present in the LwM2M standard are not supported yet by Leshan, like the Queue Mode, so that is the reason why it needs to be implemented for this project.

For the correct operation of the Queue Mode, there are three main features that should be supported by the Leshan server code, and therefore need to be implemented. Table 4.1 presents these features.
Table 4.1: Necessary features to support the Queue Mode at the server side.

<table>
<thead>
<tr>
<th>Feature name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update message detection</td>
<td>Detect when a new update message arrives from the client, which means that it is awake and can receive requests.</td>
</tr>
<tr>
<td>Client awake time expiration</td>
<td>Consider that the client is sleeping when its awake time has expired after the last sent request. For the sleeping state, the client does not send any message, so it is the server that needs to handle this internally.</td>
</tr>
<tr>
<td>Request queuing</td>
<td>Put in a queue the requests that are generated when the client is sleeping, and send them when the client is awake again.</td>
</tr>
</tbody>
</table>

Two different implementations of the Queue Mode were done in this thesis. For the main goal of the project, which is to perform the comparison between Queue Mode and TSCH, a full implementation is done inside the Leshan core, which contains the actual queue for storing the requests, apart from the other necessary features. After proving the good performance, this code was proposed as a pull request inside the Leshan’s GitHub repository [8] as an addition to the thesis work, which resulted in a reduced implementation with more basic functionality, as Section 4.2 explains.

### 4.1 Complete implementation of the Queue Mode

The Leshan’s LwM2M core consists of a set of files that constitute the necessary libraries for building the Leshan server. As an example, there are files to manage the sending and reception of packets, the registration of a client, and the security. In this chapter only the relevant parts that need to be changed are mentioned. Figure 4.1 shows the architecture of the Leshan server code.

The server is implemented as a Java class called **LeshanServer** that wraps all the logic and contains all the necessary elements, which are objects from other implemented classes. These elements are:

- A request sender that uses Californium to send CoAP requests.
- A list of registration objects. Each client that is registered in the server is identified by an object from the class called **Registration**. This object contains the relevant information of the registered client, like the IPv6 address, the endpoint name, the lifetime and the binding mode.
- Registration listeners that are notified every time a new registration, de-registration or update message from a client arrives. This is an API that the developer can use in order to perform specific actions when these messages arrive, by just implementing the methods **registered()**, **updated()** and **unregistered()**. But it is not responsibility of the developer to manage the registration interfaecer of LwM2M. The server core itself is in charge of adding a new **Registration** object to the list when a registration message arrives,
4.1. Complete implementation of the Queue Mode

**Figure 4.1**: Architecture of the Leshan server code. Only the interesting objects for the Queue Mode implementation are shown.

- updating the parameters when an update message arrives or removing the Registration object from the list when a de-registration message is received. So this API is useful to perform specific actions.

- Other elements that are not relevant for the Queue Mode implementation, for example to manage the resource observations or the security.

In order to implement the Queue Mode, some of this classes are modified, and some new ones are added. This is explained in the following sections.

4.1.1 Listener for registration and update messages

The registration listeners that the Leshan code provides can be used to implement the *update message detection* feature (see Table 4.1). These registration listeners are notified when a client registers, updates it registration, and deregisters. Therefore, for implementing this feature, a registration listener is included in the Leshan Server, which is called **LwM2mClientStateListener**. Every time a registration message or an update message arrives, the listener checks if the client uses the queue mode, and in that case it sets the state of the client to awake and sends all the requests that are stored in the queue. It also starts a timer to manage the *client awake time*, but this is explained in more detail later in this section.

4.1.2 LwM2mQueue class for managing the Queue Mode of each client

In order to achieve the other two features (Table 4.1), which are *client awake time expiration* and *request queuing*, and also to manage the Queue Mode of each client in
an independent way, the approach taken in this thesis is to create a new Java class called \texttt{LwM2mQueue} to include all elements of the Queue Mode that each registered client needs. Every time that a new client registers specifying the Queue Mode as its binding mode, one object from this class is created and added to a list that the server maintains, linked to the \texttt{Registration} object of the same client. The endpoint name is used as the linker, so both the \texttt{Registration} object and the \texttt{LwM2mQueue} object contains the same endpoint name. With this implementation, the server can control each device in a separate way. The \texttt{LwM2mQueue} class contains the following elements:

- The state of the client, which is defined as an enumeration type with two different values: \texttt{AWAKE} and \texttt{SLEEPING}.
- A timer to manage the \textit{client awake time}, in order to achieve the \textit{client awake time expiration} feature. As it is explained in Chapter 3, the client stays awake for a specific time before going to sleep in order to receive new requests from the server. This time is controlled by an internal timer, and it can be dynamically adapted (Section 3.1.3) or set with a fixed value. So the idea is that the server has also a timer that controls the \textit{client awake time}, that is started on every update message, and restarted on every request sent. When this timers expires, the state of the client is set to \texttt{SLEEPING}, and from that moment the generated requests need to be placed in a the queue. For setting the time, a read request to the client’s queue object (Section 3.3) is sent.
- A queue to store requests when the client is sleeping. This is used for implementing the \textit{request queuing} feature. This queue has a policy in order to optimise the number of requests. Before adding a new request, the server first checks if a request of the same type and to the same resource is already present. In that case, it removes the old request and adds the new one.

### 4.1.3 Request Sender with Queue Mode operation

The last step in the server implementation is to achieve the \textit{request queuing} necessary feature (Table 4.1). This means placing in a queue all the requests that are generated when the client is sleeping, and send them when the client is awake again. In order to do this, the implementation of a new request sender class is added. This class is called \texttt{LwM2mQueueModeRequestSender}, and it is based on the code of the request sender that Leshan already includes, which is called \texttt{CaliforniumLwM2mRequestSender}. This class has several methods for sending requests that require two main parameters: a \texttt{Registration} object identifying the destination client, and the request which is represented as a \texttt{LwM2mRequest} object. The sending can be done in a synchronous way, which means that the request sender waits after sending a request until it gets a response or until a timeout expires, or in an asynchronous way where the request sender adds a callback function that is executed when a response or a timeout occurs, so it does not block the execution.

For sending messages, the \texttt{CaliforniumLwM2mRequestSender} uses the Californium libraries that implement the CoAP protocol. Two main functionalities are used from
4.1. Complete implementation of the Queue Mode

this module: the send() function that is used to send a CoAP packet, and the possibility to add a message observer to a request. This message observer contains callback functions that are executed by the Californium core when some events happen, such as when a response is received, a request is retransmitted or the timeout has expired. So the CaliforniumLwM2mRequestSender operation can be divided in three steps:

1. Create a message observer and add it to the request. If the request is sent in a synchronous way, this observer has the SyncRequestObserver type, which contains a method that waits until a response is received or the timeout expires. On the other hand, if the request is sent in an asynchronous way, this observer has the AsyncRequestObserver type and offers the possibility of including response and timeout callbacks.

2. Send the request using the Californium Core.

3. If the request is sent in a synchronous way, call the method waitForResponse() that blocks until the response is received or the timeout expires.

Figure 4.2 shows a flowchart of the implemented request sender. This new request sender class implemented (LwM2mQueueModeRequestSender) checks first if the client uses Queue Mode. If it does not use it, then the request is sent in the same way as in the CaliforniumRequestSender. Otherwise, if the client uses Queue Mode the sender checks its state by getting the LwM2mQueue object from the registration object that represents this client. If the client is awake, it simply sends the request as in the CaliforniumRequestSender and restarts the client awake timer. But if the client is sleeping, the request is put in the queue inside the LwM2mQueue object, which is implemented as a Java list. For synchronous requests, a new message observer is created. This observer blocks the timer that controls the timeout of the request while it is located in the queue, because otherwise it will execute the timeout callback and warn the user about the impossibility of sending the request, when the request has not been sent yet. In order to do this, a flag is set to true and the timeout timer is blocked using a lock. When the request is sent out of the queue, this flag is set to false and then the timeout timer continues normally.

![Flowchart of the Queue Mode request sender.](image)

Figure 4.2: Flowchart of the Queue Mode request sender.

Finally, to complete the request queuing feature, when the LwM2mClientStateListener detects an update message and sets the client’s state
to awake, it also iterates over the queue and sends all the queued requests to the client, using this request sender.

### 4.1.4 Complete Queue Mode Operation in the LwM2M Core

In the previous subsections all the new elements added in the Leshan core have been presented, and now it is shown how this elements are connected to each other in order to offer a full Queue Mode operation. Figure 4.3 illustrates this interconnection, showing the new classes added to the core. The `LeshanServer` object contains objects from these classes together with the ones shown in Figure 4.1, which are not included here for simplicity.

![Figure 4.3](image-url)  
**Figure 4.3:** Complete operation of the Queue Mode inside the Leshan server. Only the objects added to the Leshan core for this purpose are shown.

The client operating with Queue Mode sends an update message every time it wakes up. This is detected by the `LwM2mClientStateListener`, which is in charge of setting the state of the corresponding client to `AWAKE`, start the client awake timer, and send the queued requests. In order to perform all those actions, the `LwM2mQueue` class offers several methods. First the method `setAwake()` is called for setting the state of the client to `AWAKE`, which informs the request sender that the client can receive requests. This method itself calls another method of this class called `sendQueuedRequests()` that sends the requests that have been put in the queue when the client was sleeping. This method iterates over the Java list that contains the requests, and send them using the `QueueModeRequestSender`.

The `startClientTimer()` is also executed by the `LwM2mClientStateListener` when the update message is detected. Each `LwM2mQueue` object contains a timer that controls the `client awake time`, so that the server can know when the client is sleeping again, since it does not send any message for this purpose. For setting the expiration
time of this timer, a read request to the queue object present in the client is sent. This is done every time the client wakes up because it may change from one iteration to the next one, if the client uses the dynamic adaptation explained in Section 3.1.3. If the client has a fixed awake time, then this read request can be done only once at the beginning. When this timer expires, the method `setSleeping()` present in the `LwM2mQueue` class is executed to set that state in the client. This timer is restarted by the `QueueModeRequestSender` every time it sends a new request, because the client stays awake for this time since the last received request, and not from the update message.

It is also important to remark that when the client sends the registration message at the beginning, the server checks if the Queue Mode is used as binding mode, and in that case it creates a new `LwM2mQueue` object linked to the corresponding `Registration` object that identifies the client, and adds it to the list present in the `LeshanServer` object where all these objects are stored. Then, it does the same actions explained before when an update message arrives in order to set the state as awake and start the timer, but it does not send any queued request because there can not be requests for a device before it is registered.

### 4.2 Reduced implementation with Presence Listeners

Once the implementation of the Queue Mode inside the Leshan server was tested and proved that it was working properly, a pull request was made to the Leshan’s GitHub repository [8], as an addition to the thesis work. The goal was to open this implementation to the community so that the Queue Mode can be used in their own open source projects. As a result of a discussion with the main contributors, it was decided to create a first version of the implementation slightly different and more simple that the one done for the thesis, and then add more features from what has been implemented in this project.

This implementation is based on having the possibility to create a new type of listeners that are notified when the client state changes to awake or sleeping, in order to perform some actions that the future developer decides. This is useful to have more flexibility to run different types of applications on top of the core that want to perform several actions depending on the state of the client. In this implementation, no actual queues where the requests can be stored are present, only this listeners feature, so the responsibility of not sending requests when the client is sleeping relies on the application. This is the reason why this implementation is called reduced.

In order to do this, only a few modifications to the previous implementation are required. First of all was creating an interface called `PresenceListener` which has two methods: `onAwake()` and `onSleeping()`. These methods are called when the state of the client changes, and future developers can just implement this interface and include inside those methods the code that they want to execute.

The second main change was to create the class `PresenceService` that contains all the necessary logic for managing the Queue Mode. This class has a list with all the
PresenceListeners that have been created, and another list with all the LwM2mQueue objects of the registered clients. This class has been renamed PresenceStatus, and contains the same elements except the request queue.

The methods for managing the timer have also been moved from PresenceStatus to the new class PresenceService for a better implementation. Therefore, in this pull request the PresenceStatus object contains a timer that the PresenceService gets whenever it wants to start, restart or stop it.

Figure 4.4 shows the operation of this Queue Mode code, which is similar to the previous one that was implemented for the thesis project. The LwM2mClientStateListener (now renamed PresenceStateListener) is notified when a new update message arrives. Then, it calls the method setAwake from the PresenceService class. This method extracts the PresenceStatus object that corresponds to the client, set its state to AWAKE, gets the timer and starts it. Then it iterates over the list of listeners and notifies them by executing their onAwake() method. When the timer expires, the state is changed back to sleeping, and the listeners are again notified, calling the onSleeping() method. In this implementation, the QueueModeRequestSender operation is the same as before, and the timer is also restarted on every request sent.

![Figure 4.4: Queue Mode operation inside the Leshan server for the pull request. Only the necessary objects added to the Leshan core are shown.](image)
Chapter 5

Evaluation

The last step of the thesis, and the one that achieves the scientific goal, is the evaluation of the Queue Mode comparing it against another low power strategy. This strategy is the TSCH MAC protocol (Section 2.5). Therefore, the comparison is made between two power saving strategies at different levels of the network stack. The Queue Mode is performed at the application level, where LwM2M decides when to sleep and when to wake up (turn on and off the radio), and TSCH is done at the MAC layer as a way of radio duty cycling. The purposes of the evaluation are:

- Determine which protocol has a better performance according to different metrics and depending on the IoT scenario. This scenario is defined based on different aspects like the frequency in which the server requires data from the nodes, or the immediacy in which this data is required. As a result of this evaluation, the thesis could be used as a guideline for future developers to decide which protocol to use.

- Test the performance of the Queue Mode itself and see if there are some issues in the implementation that were not noticed before.

5.1 Hardware platform

The hardware platform Zolertia Firefly [34] is used for the evaluation experiments of the thesis. Figure 5.1 shows this platform and its components. It is an electronic board designed specifically for IoT applications, and Contiki-NG supports it with all the necessary drivers. The core of this board is the Zoul module [35], developed also by Zolertia. It is based on the Texas Instruments CC2538 [36] which is a System-on-Chip (SoC) that includes an ARM Cortex-M3 microprocessor with 512 KB of flash memory and 32 Kb RAM memory, and a transceiver for 2.4 GHz IEEE 802.15.4. The Zoul module adds also the transceiver CC1200 [37] to enable the communication in the Industrial, Scientific and Medical (ISM) bands of 868, 915, 920 and 950 MHz, in order to support all possible channels present in IEEE 802.15.4. Therefore, this board is designed to build networks using this protocol as the physical layer. Apart from this
core module, the Firefly platform includes also a on-board printed antenna for sub-1 GHz and a ceramic antenna for 2.4 GHz. It also has a reset and a general purpose button, an RGB LED and a USB interface for programming and debugging.

![Figure 5.1: The Zolertia Firefly hardware platform.](image)

SICS has developed an IoT testbed that contains 25 Firefly platforms distributed in an office environment, connected to a central server. These platforms can be programmed independently to test IoT applications with a larger number of nodes. The testbed is used to run experiments in which the LwM2M client is located inside a network with different number of hops between the server and itself, where other nodes act as routers in the network.

### 5.2 Evaluation Setup

To perform the evaluation, it is necessary to build a network that connects the LwM2M server with the LwM2M client. Three different elements are part of this network: the LwM2M server implemented using the Eclipse Leshan code (Chapter 4), a border router, and the LwM2M client implemented that runs in Contiki-NG (Chapter 3). The border router acts as a gateway between the WSN nodes and the Internet, in which the server is located. Contiki-NG provides a software that implements a border router, which acts as the RPL coordinator of the network, and tunnels the internet traffic through its USB interface to the host it is connected, which in this case is a computer running Linux. The border router gives a public IPv6 key to every node in the network, with a fixed prefix. In this thesis, for simplicity the Leshan server runs also in the host computer and not in the cloud inside the Internet. Running it in the cloud would just add more transmission delay to the packets, but for the purpose of this project it is considered enough to run it locally in the host. So in all the experiments, the Leshan server runs in the host computer and the border router is connected to it through USB, tunnelling the Internet traffic. Both of them are placed on an office desk.

The LwM2M client is another node present in the IoT network in which the border router is the coordinator. For the evaluation, we have run experiments with different
numbers of hops between this border router and the client. For the experiments with one hop, Figure 5.2 (a) shows the setup. The client is placed in the same desk and connected to the same host computer to collect the relevant data through the USB interface. But for scenarios with more hops, the setup is the one illustrated in 5.2 (b). In this case the SICS testbed is used, where one node is programmed to be the LwM2M client, and the rest act simply as routers in charge of forwarding packets inside the network until they reach the border router. In order to achieve the desired number of hops, only some of the nodes are programmed and the rest are turned off. We have run scenarios with 2 and 3 hops.

![Diagram of setup with one hop and several hops.](image)

Figure 5.2: Setups used to build the IoT networks for the experiments in the evaluation part.

5.3 Evaluation Metrics and Scenarios

The evaluation part of this project is mainly focused on low power consumption, because that is the purpose of the Queue Mode. But other metrics also need to be taken into account in the comparison, in order to make a comprehensive evaluation of the performance of each protocol. It is necessary to compare both protocols not only in terms of energy efficiency, but also in relation to other aspects of the network that can be affected by the low power strategies. Table 5.1 shows the metrics that are defined for the evaluation, based on the usual parameters that are used among academia in the evaluation of IoT protocols [24] [29] [38].
Table 5.1: Metrics defined for the evaluation of the Queue Mode in comparison with TSCH.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy consumption</td>
<td>The total energy consumed by the client node.</td>
</tr>
<tr>
<td>Round Trip Time (RTT)</td>
<td>The time since the server attempted to make a request until the response to that request, sent by the client, arrives. Retransmissions due to packet losses are not considered inside this time.</td>
</tr>
<tr>
<td>Packet Reception Rate (PRR)</td>
<td>Ratio between the number of packets that are received correctly by the client, and the total number of packets sent. If either the request or the response is not received, one packet is considered as lost.</td>
</tr>
<tr>
<td>Memory footprint</td>
<td>The memory usage that the implementation of the each protocol adds to the whole OS.</td>
</tr>
</tbody>
</table>

Once the metrics are defined, experimental scenarios also need to be defined in order to perform this part of the project. These scenarios determine the different parameters used for each protocol, and which kind of communication is established between the server and the client. We decide to build two different types of scenarios to cover as many different situations as possible. Both scenarios are evaluated using Queue Mode and TSCH. These scenarios are:

1. **Synchronized requests scenario**: In this scenario, the server sends five requests to different resources present in the client with a fixed period of time. For the Queue Mode, this period is synchronized with the client sleeping time, which means that when the client informs the server with an update message that it has woken up, the server sends the five requests and then waits until the next update message from the client. In scenarios where the nodes use the TSCH protocol, the server has a fixed period to send the five requests.

2. **Randomized requests scenario**: The server sends the five requests at a random point inside the defined period. This means that on every periodic iteration, the server calculates a random time offset, and after that time it sends the five requests. The time offset is taken using a uniform distribution from 0 to the period value. In the Queue Mode case, if the client is sleeping after this time offset, the server puts the requests in the queue and sends them when the client is awake again. For scenarios running TSCH, the server just sends the requests at the random time instant.

The scenarios are run with different configuration parameters of both protocols. For Queue Mode, three different sleep and awake times are used, giving nine possible combinations. The selected *client awake times* are: 1 second, 5 second and the dynamic adaptation enabled (Section 3.1.3). For this dynamic adaptation, a margin of 100% is selected, because the time between two consecutive requests in these scenarios is just on the order of tens of milliseconds and therefore it is safer to have a high percentage margin. This margin gives enough time time for the server to send the desired requests,
and makes *client awake time* still appropriate for low power purposes. Section 5.5.1 shows results that proof this fact. In a different network with larger time between to consecutive requests, this margin should be reduced to achieve a good energy efficiency, otherwise the awake time would be too long. The selected sleep times are 10 seconds, 60 seconds, and 5 minutes. These are also the sending periods used by the server in the scenarios running TSCH. With these three values, a large amount of use cases is covered. Ten seconds simulates a very demanding scenario where the server requests critical data very often. Sixty seconds is used as a middle point between a demanding scenario and a low demanding one, which is simulated with the 5 minutes. This is a scenario where power saving is the main goal and where the data retrieved by the nodes is not critical, so the server does not need it very often. Later in the evaluation, the notation $QMode$-<sleeptime>-<awaketime> is used to refer to these different configurations.

For the TSCH MAC protocol, three different schedules are tested:

- **6TiSCH minimal configuration** [28]: all the nodes share the same time slot to communicate inside a slotframe. The length of the slotframe is 7, which the default one defined in Contiki-NG. This gives every node one TX/RX slot every 70 ms, and a contention mechanism is used in order to reduce the collisions that can occur because all the nodes share the same time slot.

- **Orchestra default configuration**: every node has a slot for TSCH beacons in a slotframe with a length of 397 (one slot every 3970 ms), another slot for RPL multicast traffic in a slotframe with a length of 31, and finally one unicast slot for communicating with its parent and several to communicate with its children, both in a slotframe with a length of 17. The last slotframe is the one used for the CoAP packets, and its length determines the radio duty cycle that is interesting for the comparison.

- **Orchestra with a larger slotframe length**: to save more energy, the length of the slotframe dedicated to CoAP traffic is increased to 71, in order to reduce the radio duty cycle. This means that each node will have one slot every 710 ms for communicating with its parent or children inside the network.

There are also nine possible scenarios using TSCH, as each schedule is used with different server periods for sending the requests, which are the same as the sleeping times with Queue Mode: 10 s, 60 s and 5 min. The notations used to identify these scenarios is $TSCH$-<sending period>-<schedule>, where schedule can be: minimal, Orchestra-17 or Orchestra-71.

Apart from these two main scenarios, we have run additional ones to see if there are any remarkable differences. One of these is a scenario in which the server does an observation request at the beginning to two different client resources, and then the client sends notifications either in a periodic way, to evaluate periodic notifications, or at random instants, to evaluate a typical case in which a notification is sent every time a resource changes, for example in a temperature monitoring scenario. Also we have performed a scenario using Queue Mode with 93 seconds of *client awake time* as
recommended by the LwM2M specification, to show the consequences of using such a long time.

5.4 Measurement of the Defined Metrics

In order to extract the desired metrics from the experiments, the implementation code is changed to take the necessary measurements, save them to files, and then process the data using Matlab scripts. For each metric, a different way of measuring is used:

- **Energy consumption:** in order to measure the energy consumed in the client, the Energest module of Contiki-NG (Section 2.6) is used. With this software module, the times that the different hardware parts are turned on are measured and printed through the USB interface to a file in the host, in a table format so that they can be imported to Matlab easily. The times extracted are CPU in active mode, CPU in low power mode, radio transmitting, and radio listening. Then, multiplying these values with the corresponding current that each hardware part requires, the total energy consumption can be extracted. Table 5.2 shows the current consumptions of these hardware components in the Firefly platform. The Zoul module has two different low power modes. In this project, as the purpose is to perform an evaluation, only the Low Power Mode 1 is used. But when developing a real product, this low power mode is not enough to achieve large battery lifetimes, because its current consumption is still too high. Therefore, for the calculation of the total energy consumption, the current consumption of the Low Power Mode 2 is used, as it is more realistic to use when calculating the defined metric. A remarkable aspect of performing these measurements is that the node has a higher energy consumption that it would have in a real product, because it needs to print periodically the times through the USB interface, for which it is necessary to put the CPU in active mode even when the client is in the sleeping state. The selected period to calculate and print these times is 500 ms, which is enough to have a good sampling rate in order to extract results and plot graphs, but not affect too much in the total consumption. Even though, this have consequences in the total energy consumption, as Section 5.7 where battery lifetimes are calculated shows. But as the purpose of the thesis is to perform an evaluation, this consequence is considered valid because it affects both protocols in the same way.

Table 5.2: Current consumption values of the Zolertia Firefly hardware platform.

<table>
<thead>
<tr>
<th>Component</th>
<th>Current consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Active Mode</td>
<td>20mA</td>
</tr>
<tr>
<td>CPU Low Power Mode 1</td>
<td>0.6 mA</td>
</tr>
<tr>
<td>CPU Low Power Mode 2</td>
<td>1.3 μA</td>
</tr>
<tr>
<td>Radio transmitting</td>
<td>24 mA</td>
</tr>
<tr>
<td>Radio listening</td>
<td>20 mA</td>
</tr>
</tbody>
</table>
5.5. Results

• Round Trip Time (RTT): this metric is measured by the server. When the method for sending a request is called, it saves a timestamp. Then when the response to that request arrives, another timestamp is taken. Finally, the difference between these two timestamps is calculated and saved in a file in a table format. Then this can be easily processed by the Matlab scripts, which take the mean and standard deviation of these values, without considering the retransmissions.

• Packet Reception Rate (PRR): this metric is also measured by the server. Every time a CoAP retransmission occurs, the message observer attached to the request increments a variable that contains the number of retransmissions for that concrete request. Once the request is acknowledged or cancelled, the server saves to a file the total number of retransmissions made. Then with a Matlab script, the total number of packets lost and sent is determined, and with them the PRR.

• Memory footprint: for this metric the tool for measuring the size of a .elf file present in the ARM compiler is used. This tool can be called with the command `arm-none-eabi-size <filename>`. With it, the total RAM and ROM memory used by a concrete object module, for example the LwM2M engine, can be measured. Therefore, the memory footprint can be calculated with and without Queue Mode to see how much memory is used by it.

5.5 Results

This section explains the results that are derived from the execution of the different scenarios independently, without comparing both protocols. It shows the obtained metrics and how they vary depending on the parameters chosen for the scenario, and also the important aspects that are derived from the analysis of these metrics. The following sections show the actual comparison between Queue Mode and TSCH.

5.5.1 Queue Mode Scenarios

This sections shows the results obtained for the two types of scenarios made with Queue Mode enabled: synchronized and randomized. It explains the most important results from each scenario are explained and also highlights the differences between them.

5.5.1.1 Synchronized requests scenario

In the synchronized scenarios, the LwM2M client using Queue Mode, has two different states that are repeated periodically. One state where the client is awake and the radio is turned on, and one state where the client is sleeping and the radio is turned off. Figure 5.3 shows the current consumed over time by the client node, in an scenario with 60s for sleeping time and 5s for client awake time, as an example. For plotting the current consumption of the device, it can not be directly sampled, only the time
that each component has been turned on in each 500 ms slot is determined with the Energest module. Therefore, the total current consumption is adapted to this period, which means that for every slot, the duty cycle of each component is calculated, and multiplied by the current consumption. As an example, if the CPU has been active half of this time, it would contribute with 20 mA during 250 ms, but in this graph it contributes with 10 mA during 500 ms, resulting in the same total consumption. This is useful to illustrate the current profile of the time in the node, even though it does not adjust completely to the real one. The vertical lines in the figure mark the instants when the client goes to sleep and wakes up. When the client is awake, a high baseline is produced due to the current consumed by the listening mode. Then, there are small peaks due to the transmission of packets, because this action needs additional current. In the sleeping state, only a small constant energy is consumed by the CPU in low power mode.

At the beginning of the scenario there is a high energy consumption state because the client is trying to connect to the Internet through the border router, with the radio always on. Once it is connected, the periodic cycles start.

![Figure 5.3: Current consumed over time by the client node in a Queue Mode scenario with sleeping time 60 s and awake time 5 s.](image)

If this energy is integrated, the total energy consumed by the node can be calculated. Figure 5.4 shows the accumulated energy consumed by the client over time in the same scenario as Figure 5.3. Here again the two states that the client has can be clearly seen. When the client is awake, the energy consumption curve has a high slope, with a value slightly higher than 21 mA. This includes the current consumed by the radio in listening mode (20 mA) plus the small contributions of the CPU and the radio in transmitting mode. This indicates that even when the client is awake, the CPU is not all the time in active mode, because the OS itself puts it in low power mode if there are no processes running, which for this scenario basically means there are no requests
5.5. Results

to process. Moreover, the duty cycles of each component calculated using the Energest measurements (Section 2.6) are 8.3% for listening mode, 0.03% for transmitting mode, 1.47% for CPU in active mode, and 98.52% for low power mode. So this also shows that the active mode duty cycle is much smaller than the radio in listening mode, and the transmitting mode is almost insignificant. When the client is in the sleeping state, the slope of the energy curve is around 0.2 mA. This is higher than just the current consumption in low power mode because the CPU is also in active mode at some time instants in this state, with a duty cycle of around 1%. These active instants are used to calculate the consumption times with Energest and print them through the USB port, among other tasks. This adds an extra energy consumption that would not exist in a final product would. The node does not need to do such tasks, so the active CPU duty cycle is reduced, which also reduces significantly the total current consumption. But as the purpose of this thesis is to make just a comparison and the CPU extra consumption contributes in the same way to the energy consumption of both protocols, we consider the results valid for the evaluation.

![Figure 5.4: Accumulated energy consumed over time by the client node in a Queue Mode scenario with sleeping time 60s and awake time 5s.](image)

In order to calculate the average energy consumption of the node, a linear regression of the accumulated energy consumption can be taken. This linear regression needs to discard the initial state where the client is connecting to the network. The linear regression can be used because the energy consumption follows a periodic cycle with a fixed tendency that continues in the future. With this linear regression, an accurate estimation of the total energy consumed in a future instant can be extracted. Of course this approximation would have a small error, positive or negative depending on the time instant, as a result of the steps produced when the client is awake. But this error is small enough to be considered acceptable, in the order of 50 mA as in Figure 5.4, less than 0.02 mAh. Finally, the slope of this linear regression gives an average current...
consumption in the client node, which can be used to calculate the battery lifetime if the effective capacity of the battery is known, as Section 5.7 shows.

There are some cases where the update message is delayed, because the network is more congested or simply because it is lost and the client needs to retransmit it. This effect causes the client to stay awake for a longer time waiting for the response of this packet, and the linear regression would not consider the energy consumed in this phase. But the total energy consumed in those situations, as they do not occur too often, does not have a high impact in the calculations of the battery lifetime, which would be slightly smaller than the calculated one.

Comparing all the scenarios run with Queue Mode, the most important difference between them is the radio duty cycles that are obtained, because they depend directly on the sleeping time and the client awake time. Table 5.3 include these duty cycles for every combination of the parameters used. The listening radio duty cycle, which the key part of the energy consumption and the one that Queue Mode wants to reduce, is notably affected by these parameters. For 10s of sleeping time and 5s of awake time, the duty cycle is 35.38% whereas for 5min and 1s it is reduced to 0.49%. Also the results of this table show that the dynamic adaption is useful to reduce this duty cycle. In the 10s scenario it can reduce the duty cycle from 35.38% to 5.84%, in the 60s from 8.3% to 0.93% and in the 5min from 1.8% to 0.22%. So this adaptation is appropriate to reduce the energy consumption, but it is necessary to check if other metrics are affected negatively by it.

Table 5.3: Transmission and reception duty cycles (in %) in the Queue Mode scenarios.

<table>
<thead>
<tr>
<th></th>
<th>1 Hop</th>
<th>2 Hops</th>
<th>3 Hops</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Duty RX</td>
<td>Duty TX</td>
<td>Duty RX</td>
</tr>
<tr>
<td>10 s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5s</td>
<td>35.38</td>
<td>0.13</td>
<td>34.9</td>
</tr>
<tr>
<td>1s</td>
<td>12.9</td>
<td>0.17</td>
<td>13</td>
</tr>
<tr>
<td>Dynamic</td>
<td>5.84</td>
<td>0.18</td>
<td>6.17</td>
</tr>
<tr>
<td>60 s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5s</td>
<td>8.3</td>
<td>0.34</td>
<td>8.34</td>
</tr>
<tr>
<td>1s</td>
<td>2.35</td>
<td>0.03</td>
<td>2.33</td>
</tr>
<tr>
<td>Dynamic</td>
<td>1.03</td>
<td>0.03</td>
<td>0.99</td>
</tr>
<tr>
<td>5 min</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5s</td>
<td>1.8</td>
<td>0.01</td>
<td>1.8</td>
</tr>
<tr>
<td>1s</td>
<td>0.49</td>
<td>0.01</td>
<td>0.51</td>
</tr>
<tr>
<td>Dynamic</td>
<td>0.22</td>
<td>0.01</td>
<td>0.23</td>
</tr>
</tbody>
</table>

The transmission radio duty cycle always has a low value because this radio mode is only used when the client needs to send a response to the server. But Table 5.3 shows that this value increases when the sleeping time is reduced, because the number of packets per time transmitted by the server increases too. For 10s the client sends five responses every ten seconds, but for 5min the five responses are only sent every five minutes, so the ratio is lower. Despite this fact, the difference between these duty cycles has a small impact in the total energy consumption.

If the number of hops is increased, the listening duty cycles are also slightly increased, because the RTT of the packets is higher, which means that the time between two consecutive requests is also higher. This increases the total time in which
the client is awake, but again with a low impact in the total energy consumption. Figure 5.5 shows the energy consumption in a scenario with the same parameters (60 and 5 seconds) but two hops, to illustrate that there are no considerable differences with the one hop scenario, shown in Figure 5.4. The initial phase where the client connects to the network is longer in this case due to the increment in the hop distance, but the difference in energy consumption is just around 0.08 mAh, therefore it has not a high impact in the battery lifetime.

![Figure 5.5: Accumulated energy consumed over time by the client node in a Queue Mode scenario with sleeping time 60s and awake time 5s and 2 hops.](image)

With respect to the RTT metric, it is always on the order of tens of milliseconds, because when the request is done, the client is awake with the radio in listening mode, so there are no high delays between the transmission and reception of packets. This metric is not affected by the sleeping or awake times. Figure 5.6 shows the mean RTT calculated for a Queue Mode scenario with different number of hops. For this calculation, packet retransmissions are not taken into account, because this would notably increase the RTT due to more than one packet is needed between the transmission of a request and the reception of its response. This would make the metric confusing. The figure shows the mean RTT and the standard deviation of this calculation. It can be concluded that the mean RTT is only increased by a few milliseconds when more hops are placed between the server and the client. This means that the actual transmission of packets takes less time compared to the time required to process the request or response in the client or in the server. The standard deviation is on the order of a few milliseconds, but it increases when the number of hops increases. This means that with three hops the RTT metric can have a value up to 90 ms, whereas with one hop it only reaches 74 ms. This fact indicates that even though the average value is almost equal, the number of hops affect the RTT metric.
Finally, the obtained PRR is always above 96% in all the evaluated scenarios. There is not any correlation between the configuration of the scenario parameters and the PRR results. Also the dynamic adaptation does not affect this metric, so the selected margin is safe enough for the network conditions of these scenarios. Figure 5.7 illustrates the number of retransmissions that the server does until it gets the acknowledgement response for each packet. It shows that for 48 packets there are no retransmissions, and for one packet there is one retransmission, so one packet out of 49 is lost.

Figure 5.6: Mean round trip time in a Queue Mode scenario for different number of hops.

Figure 5.7: Number of retransmissions performed for each packet in a Queue Mode scenario.
5.5. Results

5.5.1.2 Randomized requests scenario

In the case of the randomized scenarios, there are no differences in the energy consumption curve. The client node is still waking up and going to sleep in a periodic way, so the consumed current over time is the same as before. There are special cases where 10 requests are sent to the client, increasing the total amount of time that it is awake. This is produced when there are five packets in the queue, and the random time offset calculated by the server in the next iteration is smaller than the client awake time. In this case, the new five requests plus the five that where queued are sent. But this only happens in a few cases and on average it does not affect the total energy consumption. Figure 5.8 shows the energy consumption in a randomized scenario with 60 s as sleeping time and 5 s as awake time. It can be seen that the energy consumption is the same as in the synchronized scenario shown in Figure 5.4.

![Figure 5.8: Accumulated energy consumed over time by the client node in a Queue Mode randomized scenario with sleeping time 60 s and awake time 5 s.](image-url)

The RTT metric is the one that is clearly affected by the randomized scenario. Considering the case in which the requests need to be put in the queue, the RTT can have a value up to the sleeping time of the client, without considering any retransmissions. Figure 5.9 shows the mean RTT for a scenario with 1 hop, 5 s as client awake time, and sleeping times of 10 s and 60 s. The conclusion is that these values increase a lot compared to the synchronized scenarios due to the queueing of requests.

The PRR metric is not affected by the randomness, because the transmission of packets is done in the same way as in the synchronized scenarios, it does not matter if the packets are previously put in the queue or not.

Finally, the conclusion made from the randomized scenarios is that they should not
be used for applications in which the server requires data from the client at arbitrary time instants and with a high immediacy. Queue Mode is useful if the server and they client are synchronized and the data is required at fixed and periodic time instants, when the client wakes up. Also it could be used if the data is not critical and the server can wait for the requests until the client wakes up. The implementation done in this thesis allows the server to change the sleeping time of the client by a write request in the queue object (Section 3.3), so it can be changed at any point and does not have to be fixed all the time. This adds more reasons to use the Queue Mode in a synchronized scenario where the sleep time of the client and period in which the server requests data are the same, as this synchronized period can be changed.

### 5.5.2 TSCH Scenarios

After knowing the results obtained from the Queue Mode scenarios, this section shows the results obtained with TSCH. Both types of scenarios are shown: synchronized and randomized, and again the results and differences between them are explained.

#### 5.5.2.1 Synchronized requests scenario

TSCH is a way of radio duty cycling based on dividing the time into slots that are grouped in a slotframe and repeated periodically. This period is a lot smaller that the one used with Queue Mode, in the range of milliseconds to hundreds of milliseconds. This makes the energy consumption almost constant, with smaller peaks that are produced when the client receives and transmits a packets. Figure 5.10 shows the current consumed by a node using TSCH with the minimal schedule, and 60s as sending period for the server. The same procedure as in Section 5.5.1.1 is used to determine the current consumption. It has a high consumption when the node is
scanning for TSCH beacons in order to connect to the border router, and then once the schedule is established, the consumption is almost constant with small peaks when packets are received (vertical lines) and therefore responses are sent.

Figure 5.10: Current consumed over time by the client node in a TSCH scenario with minimal schedule and a sending period of 60 s.

When communicating using the TSCH protocol, the schedule used plays an important role in energy consumption because it determines the radio duty cycle. The listening duty cycle is reduced while the slotframe length uses in the communication increases. Therefore, the highest listening duty cycles are obtained using the minimal schedule (length 7), and the smallest using Orchestra with a length 71. Orchestra with its default configuration is at a middle point, with a length of 17. For the transmission duty cycle, the schedule does not have an effect, but the sending period of the server does. For smaller periods, the number of requests that need to be responded increases, and therefore the transmission duty cycle is increased.

Table 5.4 includes the transmission and listening radio duty cycles for all the tested scenarios, with the three schedules and with the three different sending periods for the server. The table shows that for the minimal configuration the listening duty cycles are around 3.5%, whereas with Orchestra-17 it is around 2.2% and with Orchestra-71 it is reduced to around 1.1%. So none of the configurations are able to reach as low listening duty cycles as with Queue Mode (Table 5.3). For the transmission duty cycles, they are reduced while the sending period is increased. It can also be seen that all the duty cycles are slightly augmented when the number of hops increases.

If the energy consumed is integrated, the accumulated energy over time can be calculated and the linear regression can be applied again to have an average current consumption. Figure 5.11 illustrates this energy consumption, from the same scenario as Figure 5.10 (60 s and Orchestra-17). As the energy consumption over time is almost
Table 5.4: Transmission and reception duty cycles (in %) in the TSCH scenarios.

<table>
<thead>
<tr>
<th></th>
<th>1 Hop</th>
<th>2 Hops</th>
<th>3 Hops</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Duty RX</td>
<td>Duty TX</td>
<td>Duty RX</td>
</tr>
<tr>
<td>10 s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minimal</td>
<td>3.58</td>
<td>0.24</td>
<td>3.63</td>
</tr>
<tr>
<td>Orchestra</td>
<td>2.43</td>
<td>0.25</td>
<td>2.52</td>
</tr>
<tr>
<td>Orchestra-71</td>
<td>1.36</td>
<td>0.24</td>
<td>1.42</td>
</tr>
<tr>
<td>60 s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minimal</td>
<td>3.49</td>
<td>0.06</td>
<td>3.52</td>
</tr>
<tr>
<td>Orchestra</td>
<td>2.22</td>
<td>0.05</td>
<td>2.24</td>
</tr>
<tr>
<td>Orchestra-71</td>
<td>1.18</td>
<td>0.05</td>
<td>1.2</td>
</tr>
<tr>
<td>5 min</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minimal</td>
<td>3.47</td>
<td>0.02</td>
<td>3.49</td>
</tr>
<tr>
<td>Orchestra</td>
<td>2.2</td>
<td>0.02</td>
<td>2.21</td>
</tr>
<tr>
<td>Orchestra-71</td>
<td>1.15</td>
<td>0.02</td>
<td>1.15</td>
</tr>
</tbody>
</table>

constant, the accumulated energy curve is linear and does not contain steps as in the Queue Mode scenarios. Therefore, the linear regression used as an approximation is more accurate for these scenarios. There is again an initial state with a higher slope where the client is connecting to internet with the radio turned on all the time. The slope of the linear regressions that represents the average current consumption is proportionally affected by the radio duty cycles, and therefore by the schedules and periodicity used. Section 5.6 shows these values when comparing them with the average current consumption of the Queue Mode.

Figure 5.11: Total energy consumed by the client node in a TSCH scenario with the minimal schedule and a sending period of 60 s.

The RTT metric is affected by both parameters: the slotframe length and the number of hops. The slotframe length determines the distance between the transmission and reception slots used to communicate between the parents and their children in the network topology. This distance is the time that the packet waits in the node from when it is received until it can be sent. So when the length increases,
the distance also increases, causing the transmission time for the packet to become higher. This transmission time is also increased with the number of hops, again due to the waiting time that the packets have in the additional nodes.

Figure 5.12 shows the mean RTT obtained for the three different schedules, each one for 1 to 3 hops. It can be seen that the round trip times are increased by the two parameters already mentioned. With TSCH there is clearly a trade off between the radio duty cycles and the RTT. To achieve low duty cycles, the schedule needs to be longer and therefore the RTT increases.

One remarkable aspect is that the increments of the RTT are not as trivial to estimate as one could expect. As an example, incrementing the number of hops by one or doubling the slotframe length does not necessarily lead to a doubled RTT. This effect is provoked by the position in which the transmission and reception slots are located inside the slotframe, which are determined using the receiver ID and the length of the slotframe. If one hop is added but the position of the transmission slot of the new parent is closer to the reception slot of its child, then the delay in this hop is smaller. Also if the total length of the slotframe is increased but the calculated positions are closer, the increment in the RTT would not be as big as the increment in the length.
Figure 5.12 (c) shows that for three hops scenarios, the mean RTT is higher than 2 seconds. This means that the retransmission timeout of the server needs to be increased, otherwise retransmissions happen when the packet is not actually lost. Therefore this parameter needs to be taken into account when developing an IoT application using TSCH with low radio duty cycles.

Finally, the PRR is also close to 100% in most of the cases. But it can be seen how the PRR is reduced for longer slotframe lengths, which means that more packets need to be retransmitted. Figure 5.13 shows the number of retransmissions performed in a scenario with Orchestra-71 and 60s as the sending period. In this case seven packets are retransmitted.

![Figure 5.13: Number of retransmissions performed for each packet in a TSCH scenario with Orchestra-71.](image)

5.5.2.2 Randomized requests scenario

In this case, there are no differences between the synchronized and the randomized scenario. The performance is the same with respect to all the metrics. This is because TSCH schedules have a short period, so it does not matter at which instant the server sends the requests. On average, the energy consumption, RTT, and PRR are the same. This makes TSCH suitable to use in those scenarios where the server requests data at random instants and it requires a high immediacy, where Queue Mode cannot be used.

5.6 Comparison between Queue Mode and TSCH

After seeing the results of each protocol independently, this section makes a comparison between them with respect to the four metrics. As a result of this comparison, conclusions about which protocol is more suitable to use in each situation can be drawn.
5.6. Comparison between Queue Mode and TSCH

5.6.1 Energy Consumption

For the energy consumption metric, the average current consumption extracted using the slope of the linear regression can be used, to see which protocol requires more current and therefore would have a lower battery lifetime. Four aspects contribute to the average current consumption, the radio in listening and transmission mode, and the CPU in active and low power mode. Usually the radio has a higher influence, but as a result of this evaluation, it is also proved that in some situations, even with a higher radio duty cycle, if the active CPU duty cycle is much lower, the node consumes less energy.

5.6.1.1 Sending period 10 s

With a sending period of 10 seconds, the main conclusion is that the consumption is always lower using the TSCH protocol, even when the dynamic adaptation of the client awake time is enabled. For longer awake time, the listening duty cycles are too high (around 35% with 5 s and around 13% with 1 s), so this cannot be used in practice. Figure 5.14 shows a comparison between the Queue Mode running with dynamic adaptation, which reaches around 5% of listening duty cycle, against the three different TSCH schedules. The graphics illustrate the scenario with one hop, but there are no significant differences in energy consumption when the number of hops is increased, as Section 5.5.1 explains. The slope, which is the mean current consumption when the network varies a lot when using TSCH, due to the random channel hopping that the nodes perform until they get a valid beacon from another node. The energy consumed in this phase is negligible compared to the total energy consumed in the node.

The conclusion drawn from these experiments is that for short sending periods, which represent a demanding scenario where the server requests data often, TSCH is a better choice with respect to energy efficiency.

Figure 5.14: Energy comparison for sending period 10 s between Queue Mode with dynamic adaptation and TSCH.
5.6.1.2 Sending period 60 s

In this case, with Queue Mode and 5 s for the client awake time the listening duty cycle obtained is around 8%, which is again too high for low power consumption purposes. But when the client awake time is reduced to 1 s, the average current consumption with Queue Mode is lower than with TSCH. The listening duty cycle is reduced to 2.35% in this case. Figure 5.15 shows a comparison between this Queue Mode configuration, and the three different TSCH schedules. It shows that the average current is lower for the Queue Mode in all the cases.

![Energy comparison for sending period 60 s between Queue Mode with 1 s client awake time and TSCH.](image)

It is important to remark that for Orchestra-71, the listening duty cycle is lower than with QueueMode-1s, but still the energy consumption is a bit higher. This is due to the active CPU duty cycle, which is around 1% in Queue Mode and 3% in TSCH. This gives another important conclusion: in order to run the TSCH protocol more CPU resources are needed, and therefore even with lower radio duty cycles, the consumption can be above the Queue Mode.

If the dynamic adaptation is enabled, the slope is reduce to 0.5 mA, which is even more efficient than the ones achieved with TSCH. Therefore, the experiments with 60 s show that Queue Mode is better in terms of energy consumption, but the other metrics need to be checked to conclude if it is a more suitable option than TSCH or not (Sections 5.6.2 and 5.6.3).

5.6.1.3 Sending period 5 min

Finally, as it is expected, with a sending period of 5 minutes Queue Mode is even more efficient than TSCH with any of the schedules. Even with a 5 s client awake time, the listening duty cycle is 1.8%, which achieves an average current of 0.575 mA. With the dynamic adaptation, this duty cycle is reduced to 0.22% and the current consumption is around 0.26 mA, which is the lowest one in all the experiments. Figure 5.16 shows the comparison between the Queue Mode with dynamic adaptation and the three TSCH schedules, to illustrate the improvement in energy efficiency that it produces.
5.6. Comparison between Queue Mode and TSCH

Figure 5.16: Energy comparison for sending period 5 min between Queue Mode with dynamic adaptation and TSCH.

Again the conclusion is that Queue Mode is more energy efficient in this situation, but the other metrics need to be checked to see if they are affected in a bad way.

5.6.2 Round Trip Time

Once the energy consumption is compared, the other metrics need to be checked in order to see if the energy efficiency affects them. This section compares the RTT. As previous sections show, in scenarios with Queue Mode this metric is almost independent from the scenario parameters. Changing the sleeping or awake times does not vary the RTT, and adding more hops slightly increases it, but not as much as it could be expected, because most of the time is consumed in processing the packets. On the other hand, with TSCH the RTT is affected significantly by both the number of hops and the schedule length.

Figure 5.17 shows a bar graph that compares the mean RTT obtained in all the scenarios, for different numbers of hops. The sending period is 10 s in all of them, because there is a larger number of packets sent in these scenarios, so the mean value and the standard deviation get more accurate. The figure illustrates how for Queue Mode the value is almost constant for each configuration, but for TSCH it increases with the slotframe length and the number of hops. This means that when using Queue Mode, low radio cycles can be achieved without affecting the RTT, whereas for TSCH lowering these duty cycles translates in a significant increment in the RTT value. Section 5.6.1 shows that Queue Mode has a better energy efficiency for the sleeping times of 60 s and 5 min, and now this section presents that Queue Mode has a better performance in relation to the RTT metric. These obtained results translate in that for these situations, Queue Mode is a more suitable option. TSCH should only be used for short sleeping times, like the 10 s used in the scenarios, where Queue Mode is not able to achieve a proper energy efficiency.
CHAPTER 5. EVALUATION

Figure 5.17: Round trip time comparison between all Queue Mode and TSCH scenarios, with different numbers of hops.

5.6.3 Packet Reception Rate

Finally, the PRR metric is evaluated. Figure 5.18 shows a comparison between the PRR values obtained in all the scenarios, again in one bar graph for each number of hops used. The sending period is again 10s because more packets sent and therefore better results can be extracted. The value is always high, but these graphs show that it is reduced in the TSCH scenarios while the slotframe length increases. This gives a similar conclusion than with the RTT metric, to achieve low radio duty cycles with TSCH, the PRR is adversely affected, whereas for Queue Mode it is independent. This conclusion gives another reason for the suitable use of Queue Mode.
5.6. Comparison between Queue Mode and TSCH

![Bar chart](image1)

(a) 1 hop.

![Bar chart](image2)

(b) 2 hops.

![Bar chart](image3)

(c) 3 hops.

Figure 5.18: Packet Reception Rate comparison between all Queue Mode and TSCH scenarios, with different number of hops.

### 5.6.4 Memory footprint

This section evaluates the memory footprint that the implementation of the Queue Mode adds to the LwM2M modules, and also compares the memory footprint for the whole binary file that runs in the Firefly platform when using Queue Mode or TSCH.

Chapter 3 explains that the software modules that are modified in the LwM2M implementation inside Contiki-NG are the `lwm2m-rd-client` and the `lwm2m-engine`. Also, the `lwm2m-notification-list` is added to store notifications when the client wakes up (Section 3.2.1 and the `lwm2m-q-object` module is included to implement this object that manages the Queue Mode operation.

Table 5.5 shows the memory footprint of these modules, with Queue Mode enabled or disabled, with the goal of calculate how much memory usage does the Queue Mode implementation add. For the `lwm2m-rd-client`, the table shows that the ROM memory is increased by 364 bytes due to the code added to implement the state machine. No additional RAM is needed for this module. In the `lwm2m-engine`, the ROM is also increased but with a lower value — 300 bytes — because less lines of code are added.
The engine with Queue Mode needs 8 bytes of additional RAM (uninitialized). The \texttt{lwm2m-notification-list}, which is used inside the engine to store the notifications, adds 739 bytes of ROM and 146 bytes of RAM, setting the length of the queue to store notifications as 10. This length could be reduced in order to save RAM memory. Finally, the \texttt{lwm2m-q-object} adds 572 bytes of ROM and 219 bytes of RAM, but this module is not mandatory for the specification, it is only used for concrete purposes in this thesis to have a better operation of the Queue Mode (Section 3.3). Therefore, the total ROM memory added by the implementation is 1985 bytes, and the total RAM memory is 374 bytes.

Table 5.5: Memory footprint of the different LwM2M software modules with Queue Mode enabled and disabled.

<table>
<thead>
<tr>
<th>Module</th>
<th>Q-Mode disabled</th>
<th>Q-Mode enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{lwm2m-rd-client}</td>
<td>1806</td>
<td>2170</td>
</tr>
<tr>
<td>\texttt{lwm2m-engine}</td>
<td>4109</td>
<td>4409</td>
</tr>
<tr>
<td>\texttt{lwm2m-notification-list}</td>
<td>739</td>
<td>980</td>
</tr>
<tr>
<td>\texttt{lwm2m-q-object}</td>
<td>572</td>
<td>219</td>
</tr>
</tbody>
</table>

Now the memory footprints of the whole system with either Queue Mode or TSCH can be compared to see which implementation has a larger one. Table 5.6 shows these footprints. The total amount of memory used is larger when running TSCH both with and without Orchestra than when running Queue Mode, with CSMA as MAC layer. TSCH minimal configuration requires 9934 bytes more of ROM memory and 2868 bytes of RAM memory, while TSCH with Orchestra requires 10813 bytes of ROM and 3004 bytes of RAM.

Table 5.6: Total system’s memory footprint when using Queue Mode or TSCH.

<table>
<thead>
<tr>
<th></th>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue Mode</td>
<td>66082</td>
<td>2853</td>
<td>13696</td>
<td>82631</td>
</tr>
<tr>
<td>TSCH-minimal</td>
<td>76016</td>
<td>1825</td>
<td>17592</td>
<td>95433</td>
</tr>
<tr>
<td>TSCH-orchestra</td>
<td>76895</td>
<td>1909</td>
<td>17644</td>
<td>96448</td>
</tr>
</tbody>
</table>

The conclusion is that the Queue Mode implementation has a lower memory footprint than the TSCH one, which makes it more easy to fit in the memory-constrained devices.

5.7 Battery lifetime calculations

For the energy consumption metric, the average current consumed by the client node has been compared, which is calculated taking the slope of the linear regression applied to the energy accumulated curve. But for IoT devices, a more clear metric is the actual battery lifetime that the node would have when operating in fixed conditions. This gives a better picture of the energy efficiency, and can be used in the comparison of the two protocols.
In order to do that, the value of the battery capacity is needed (in mAh). With this value and the average current consumption that is extracted from the experiments, an approximate lifetime can be calculated. When the Firefly platforms are used in a real deployment, they are usually powered using a couple of AA alkaline batteries, that give the necessary voltage. It is difficult to find a value for the battery capacity, because it depends on many factors, like the environmental conditions, the extracted current, and the way the current is extracted (continuously or in steps). Also while the capacity decreases, the voltage also decreases and the battery becomes useless even though it still contains energy. Therefore, what is usually done in practice is take the nominal value of the capacity, and add a discharge safety as a percentage to consider all these effects. A typical value for the capacity of alkaline AA batteries is 2600 mAh [39], and taking a discharge safety of 20%, this gives a capacity of 2000 mAh to use in the calculations for this thesis. This would give just an approximation of the actual lifetime that the node would have in a real deployment, but is still useful in the comparison between Queue Mode and TSCH.

It is important to consider that this battery lifetime would be much lower that the one achieved by the client node in a real deployment, due to the energy reading that it needs to perform. In a real deployment, the node would be in low power mode the whole sleeping time, whereas in the experiments it needs to switch to active mode often to calculate and print the times measured with Energest (Section 2.6). The active CPU duty cycles in sleeping mode in these cases are around 1%, which means that the current consumed is 0.2 mA. This current is much higher than the 1.3 µA of the LPM 2 (Table 5.2). Again, the purpose of this section is just to give a comparison between both protocols with an approximate battery lifetime, so this is considered as a good enough approximation, because it affects both protocols in the same way.

Figure 5.19 shows a comparison between the lifetime achieved in the different scenarios, for the three selected sending periods. This graphs show the results of the 1 hop configurations, but it has been showed in previous sections that the number of hops does not affect significantly in the energy consumption.

From these graphs similar conclusions can be drawn. For a sending period of 10 s, the Queue Mode is not suitable and TSCH has a clearly better efficiency. The approximate battery lifetime is low in all the cases because this scenario is very demanding and the nodes cannot use low radio duty cycles. For a sending period of 60 s, Queue Mode has a higher lifetime except when using the client awake time of 5 seconds. The dynamic adaptation achieves a lifetime of almost half a year, which could be a suitable value considering the CPU behaviour that has been explained. Finally, for the sending period of 5 min the dominance of the Queue Mode is more notable, achieving a lifetime of 305 days with the dynamic adaptation enabled. It can be seen along all the graphs that the lifetime achieved with TSCH with each schedule is almost constant, and only slightly increased when lowering the sending period because there are less packets to receive and send.

In a real deployment, where the CPU is actually in low power mode all the sleeping time of the node achieves, the lifetimes achieved would be much higher. With Queue Mode, the total active CPU duty cycle in sleeping mode is around 1%, which translates in a current consumption of 0.2 mA for this active mode, plus 1.287 µA from the low
Figure 5.19: Battery lifetime comparison between all Queue Mode and TSCH scenarios, for different sending periods.

5.8 Alternative scenarios

In this section some additional scenarios are evaluated, to gain more understanding about the LwM2M’s performance. First, a monitoring scenario is tested, which constitutes a typical one used in IoT environments. In this scenario, the sensor nodes take data from the environment where they are placed, like a room or a mechanical
structure, and send it to the server in a periodic way or when the values change significantly. Therefore, the server does not need to send any request. On the other hand, an scenario with the default value for the client awake time recommended by the OMA specification is used, to show the consequences that it has.

5.8.1 Monitoring scenario

This scenario is used as an example of a usual IoT application where the server does not make requests to the client to perform actions, but is only interested to retrieve data from it. One typical example are the applications for monitoring environmental parameters, like the temperature of a room or the state of a bridge. In this case, the server performs an observe request for the resources it is interested in, and then the client sends a notification periodically or when the value changes.

To simulate this kind of application, the Leshan server performs an observe request to two different resources of the client using Queue Mode, and then the client sends a notification every time it wakes up. Figure 5.20 shows the accumulated energy curve that is obtained when the sleeping time is set to 60 seconds and the dynamic adaptation is enabled. The listening duty cycle has a value of around 1%, but the transmission duty cycle is reduced notably compared to the scenarios where five requests are made (from 0.33% to 0.03%). This translates to an average current consumption of 0.29 mA, close to the cases with 5 minutes of sleeping time in the previous scenarios. This would achieve a lifetime of 287 days, again considering the this is just an approximation because the CPU is active much more time than in a real deployment.

![Energy Consumption](image)

Figure 5.20: Accumulated energy consumed over time by the client node in a monitoring scenario using Queue Mode with dynamic adaptation and sleeping time 60 s.

If TSCH with Orchestra-17 is used in the same conditions, which means sending two
notifications every 60 seconds, the average current consumption is 1.1 mA, almost the same as in the previous scenario with 5 requests. Figure 5.21 shows the accumulated energy curve in this case. This only achieves a lifetime of 75.75 days.

![Accumulated energy consumed over time by the client node in a monitoring scenario using TSCH with Orchestra-17 and notification period 60s.](image)

For the RTT and PRR, the values are similar to the previous scenarios, because the transmission and reception conditions do not change. Then, the conclusion that can be derived is that in this kind of scenarios, Queue Mode is more energy efficient and a more suitable option than TSCH.

### 5.8.2 Scenario with default values recommended by OMA specification

This section aims to show the results that are obtained if the client awake time recommended by the OMA LwM2M specification is used, which is 93 seconds inherited from the CoAP protocol. The sleeping times of 10 s and 60 s does not make sense in this case, because then the client would be longer time in the awake state than sleeping, so no energy would be saved.

Figure 5.22 shows the accumulated energy obtained using 5 min as sleeping time. In this case, the listening duty cycle is 23.7%, which translates into an average current consumption of 5.056 mA. This value is very high, and close to the previous scenarios where the sleeping time is 10 s, where it is shown that TSCH is more efficient. It can be seen that in the 93 s that the client is awake, the energy consumed is around 2000 mAs (0.56 mAh), which highlights that this awake time is too high for low power consumption purposes. With the calculated average current, the expected battery lifetime when using the same approximation as in Section 5.7 is only 16.5 days.
Therefore, the sleeping time needs to be increased in order to achieve a low energy consumption even with this client awake time. If it is set to 20 minutes, the achieved listening duty cycle is 8.29%, which is close to the scenario with 60 s for sleeping and 5 s for awake time. The average current consumption obtained is 1.8 mA, which gives a battery lifetime of 46.29 days, which is still low.

These experiments show how the 93 seconds is a too high value for low power consumption in IoT devices. To achieve a radio duty cycle similar to the ones obtained with the dynamic adaptation (around 1%), the sleeping time needs to be over 2 hours, which is not suitable for most of the applications. But as this parameter is only a recommendation, it can be reduced a lot in order to achieve a high energy efficiency, without having a bad effect in the performance of the protocol, as it has been shown in this thesis.
Chapter 6

Conclusions and Future Work

The main goal of this Master’s thesis was to evaluate the performance of the LwM2M application protocol using the low power strategies TSCH and Queue Mode. In order to achieve this goal, the Queue Mode has been implemented at both sides of the communication, the server inside Eclipse Leshan and the client inside Contiki-NG. As a result of the evaluation part of the thesis, the operation of both implementations was tested, and we can conclude that it is correct and can be used in future projects. Moreover, a reduced version of the server software has been accepted and integrated as a pull request in the Leshan GitHub repository [8], showing the success of the implementation part of this thesis.

For the evaluation, four different metrics have been used (energy consumption, round trip time, packet reception rate, and memory footprint) and different scenarios have been tested, in order to compare both strategies. The conducted experiments allow to draw the general conclusion that can be used as a guideline to select the most suitable low power strategy in each situation. TSCH is a better option in demanding scenarios where the server needs to communicate very often and therefore the sleeping times need to be low, which translates in a bad energy efficiency using Queue Mode. Also, in the cases where the server does not request data at fixed time instants and requires this data with a high immediacy, Queue Mode is not suitable because the RTT can become too high, so TSCH should be used in these cases too. But in less demanding scenarios where the sleeping time can be higher and the server requires data at fixed time instants, Queue Mode has been shown to be a better option in terms of all the measured metrics. The energy consumption is lower and, in addition, the RTT, the number of lost packets, and the memory footprint are also lower. The battery lifetimes that can be achieved with the Queue Mode are significantly higher. Therefore, Queue Mode is a much more suitable option in this kind of scenarios. Finally for monitoring applications where the client just needs to send data in form of notifications, Queue Mode achieves an even better energy efficiency than TSCH because the awake times can be reduced more.

For a correct operation, the Queue Mode requires that the client and the server are synchronized, which means that the communication is done always when the client is awake. The implementation done in this thesis favours this use case, because the
sleeping time does not need to be fixed since the starting point. The server can change the sleeping time at any moment with a write request to the client’s Queue Object, for example if in concrete situations the server needs to communicate more often, it can reduce the sleeping time of the client with a write request, and then set it back to the higher one. This makes easy that the client and the server are synchronized and adds another reason to use Queue Mode in real IoT scenarios.

The main disadvantage of Queue Mode is that it can only be used in the leaf nodes of the network topology, and the router nodes need to keep the radio always on, using CSMA, in order to route the packets. This means that these router devices cannot use a battery as the power supply, because the energy would be drained quickly. This reduces the number of applications where Queue Mode can be used at the moment, and a way to reduce the radio duty cycle also in the router nodes using Queue Mode should be investigated.

As future work, regarding the implementation of the Queue Mode, another way to announce the client awake time could be done. In the current implementation the server needs to do a read request every time, which is not really efficient. A possible solution could be to include this time as a parameter inside the update message that the client sends to inform that it has woken up. In addition, a way of reducing the power consumption in the router nodes when using the Queue Mode could be investigated, as it was explained before, to increase the number of applications where Queue Mode can be used.

With respect to the evaluation, the performed experiments are done considering a network with a low congestion and interferences because only a few number of nodes are used, and with the server running locally. Experiments could be conducted in order to see how the metrics are affected in such cases, for example when there are other nodes in the network sending different kinds of traffic, or when the server is running in the cloud with a larger amount of clients registered. These experiments can be useful for a more comprehensive evaluation in other kind of scenarios.
References


