UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Security for constrained IoT devices

Sérgio Daniel Rocha Pinto

Mestrado em Segurança Informática

Dissertação orientada por:
Prof. Doutor Francisco Cipriano da Cunha Martins
Prof. Doutora Maria Dulce Pedroso Domingos

2020

Agradecimentos

Este foi um projeto cheio de desafios, que exigiu grande vontade de aprender sobre áreas com as quais não estava familiarizado. Por esse motivo tanto eu, como as pessoas mais próximas, abdicámos de muito tempo. Essas pessoas são a minha família (a minha mãe, o meu pai, as minhas irmãs e sobrinhos) e a minha namorada Diana, que de todas as pessoas foi a que teve de ter a maior paciência para me conseguir apoiar nesta altura em que eu via o tempo a passar a correr e muitas vezes não teve a minha disponibilidade que merecia ter tido. A todos eles agradeço o apoio, pois sem eles não teria sido possível. Quero agradecer, e muito, aos meus amigos por continuarem a estar sempre presentes, mesmo quando o maior ausente era eu, foram eles os responsáveis por continuar a ter uma mente sã. Aos meus colegas Raimundo Chipongue e José Fernandes por me encorajarem a fazer o curso e serem um grande apoio em todas as suas fases. Também agradecer às empresas onde estive durante este período, a Compta e a Altitude, por me terem dado chefias que me apoiaram e deram condições para eu não parar, nomeadamente ao Bruno Gil e à Sandra Conduto. Por último, aos maiores responsáveis por ter iniciado e terminado esta jornada, o meu muito obrigado à minha professora e orientadora Dulce Domingos pela paciência e por apoiar todo o meu trabalho e ao professor Francisco Martins por todo o conhecimento que me passou nas áreas onde não me sentia à vontade, bem como o companheirismo dos dois.

Para a Diana
Para a família
e para os amigos

Resumo

A evolução constante nas tecnologias de informação e microeletrónica originaram um grande avanço na integração de microcomponentes com outros sistemas. Foram desenvolvidas várias técnicas para recolher e processar dados sobre o ambiente de forma a que sejam interpretados por sistemas mais complexos.

A Internet das Coisas nasceu e cresceu com o objetivo de conectar objetos físicos, comunicando através de redes otimizadas. Os nós de redes IoT apresentam grandes restrições ao nível de recursos (energia, computação e memória). Estas limitações originaram o desenvolvimento de tecnologias especificamente para eles.

Um dos principais problemas nestes sistemas é a segurança. Os mecanismos existentes para as redes convencionais são difíceis ou impossíveis de implementar, porque, por exemplo, para processar algoritmos de cifra assimétrica, é necessária uma grande capacidade computacional e de memória, quando trabalhando com dispositivos limitados. Outro aspeto a ter em conta é que são dispositivos que se ligam por wireless, tornando-os mais acessíveis a atacantes, não só pelo tipo de comunicação mas também por estarem muitas vezes fisicamente mais acessíveis.

O objetivo deste trabalho foi desenvolver uma aplicação, utilizando tecnologias normalizadas, que comunique através de um canal seguro e que permita um desenvolvimento mais amigável, mantendo uma solução de baixo custo.

Como contributo, desenvolvemos uma aplicação que comunica por CoAP através de um canal seguro de comunicação implementado com DTLS. Do lado do servidor desenvolvemos a aplicação com Javascript por ser uma linguagem de mais fácil aprendizagem e corremos com NodeJS. O grande pilar deste trabalho foi o projeto Aquamote da Green By Web, que é um sistema de irrigação automática, feito com componentes de baixo custo.

Neste trabalho estudámos as tecnologias mais utilizadas em IoT, tentando estabelecer uma correspondência dos vários protocolos para camadas de rede convencionais. Para cada uma delas estudámos o seu modo de funcionamento, aplicabilidade, vulnerabilida-

des de segurança e mecanismos de proteção.

O nosso caso de estudo, o Aquamote da Green by Web, é uma solução que consiste num servidor (web) ao qual se ligam os dispositivos Aquamote. Os dispositivos Aquamote são compostos por sensores, atuadores e um módulo GPRS responsável pela comunicação com o exterior. Os sensores medem variáveis de ambiente como a temperatura e a pluviosidade. Estas informações são então enviadas para um servidor web que decide que ações os atuadores devem ter.

Em relação à segurança do Aquamote, existem 3 limitações que tentámos mitigar neste projeto. (a) a comunicação com o servidor web não é cifrada, por isso é mais vulnerável a ataques de análise de tráfego; (b) não existe garantia de autenticidade e integridade; (c) o transporte é feito com TCP, que tem uma carga adicional nos dispositivos, por exemplo, quando estão a estabelecer novas ligações, o que degrada a eficiência energética.

No início do projeto definimos alguns requisitos: (a) implementação de um canal Seguro ao nível de transporte e com autenticação mútua; (b) reduzir o consumo de energia, reduzindo o número e o tamanho de mensagens trocadas; (c) garantir uma solução de baixo custo, minimizando os custos com hardware e desenvolvendo uma solução de fácil manutenção.

Para isso, desenhamos uma aplicação composta por nós que se conectam através da internet a um servidor, utilizando UDP como meio de transporte, protegido pelo protocolo DTLS. A aplicação será uma aplicação cliente-servidor CoAP.

Tendo como base a solução Aquamote, fizemos testes utilizando o protocolo TCP, e conseguimos estabelecer comunicações com sucesso. No entanto, na implementação da variante para UDP, a comunicação não parecia estável, com a maior parte das mensagens a serem perdidas. Não conseguimos dissipar as causas com toda a certeza, mas a causa que consideramos mais provável está relacionada com limitações a nível do ISP, utilizando o módulo GPRS. Considerámos que se em testes em ambientes controlados tivemos esses problemas, num ambiente em que a cobertura seja menor, seria mais desafiante conseguir ter uma comunicação resiliente.

Decidimos então implementar uma solução baseada em Wi-fi. Para isso, utilizando uma placa Arduino Uno, interligada com um módulo Wifi ESP8266, criámos o mesmo cenário.

Nesta condição já não havia perdas de pacotes. Esta solução liga-se a um servidor na Internet, no entanto, precisa sempre de um ponto de acesso para se conseguir ligar, ao

contrário do GPRS que apenas necessita de cobertura para tal.

Seguindo os testes bem sucedidos passámos à implementação da camada DTLS. Para isso utilizámos uma biblioteca C, MbedTLS, que foi desenvolvida especificamente para dispositivos embebidos.

Integrámos o protocolo DTLS primeiro utilizando chaves pré-partilhadas e depois com certificados. Enquanto o primeiro garantia uma maior eficiência, o segundo garantia propriedades de segurança mais fortes.

No caso da chave pré-partilhada, configuramos uma palavra-passe do lado do servidor e do lado do nó. Otimizámos também algumas especificações, como, por exemplo o "message autentication code" usar 8 bytes em vez de 16. Como algoritmo de cifra para as mensagens trocadas utilizamos o AES256.

Tentámos então a utilização de certificados no estabelecimento da chave de sessão DTLS. Devido à limitação computacional e de memória, não foi possível fazer testes bem sucedidos com os certificados.

Como solução para as nossas experiências, adotamos a solução com chaves pré-partilhadas.

Para termos uma aplicação em CoAP, integrámos uma biblioteca C chamada "coap-simple" e, para cada mensagem a enviar para o servidor, primeiro era criado o pacote CoAP, depois utilizada a função de MbedTLS para enviar por canal seguro. Para isso alterámos código da biblioteca MbedTLS para utilizar o socket instanciado pelo ESP8266. Criámos ainda uma função invocada no "loop" do Arduino que gera um pacote CoAP e envia por DTLS.

Do lado do servidor, utilizando NodeJS, integrámos duas bibliotecas (node-coap e node-dtls) e fizemos os desenvolvimentos necessários para, nativamente, conseguirmos ter um servidor CoAP que comunicasse por DTLS, nomeadamente, criámos uma classe que ficava responsável por criar o socket de comunicação DTLS e após isso tratava todos os pedidos CoAP.

O servidor recebia a mensagem por DTLS e fazia a conversão da mensagem para um objeto COAP, a partir daí, seguindo o protocolo, processava o seu conteúdo e respondia novamente com um objeto CoAP encapsulado num pacote DTLS.

Para comparar as otimizações da nossa solução, fizemos algumas experiências e reco-

lhemos dados como o número de mensagens trocadas e o tamanho das mesmas.

Em sumário, os resultados que obtivemos foram: (a) a utilização de protocolos baseados em UDP otimizam a comunicação dos vários componentes, reduzindo o número e tamanho de mensagens; (b) a camada de segurança, sendo cada vez mais fundamental, introduz alguma carga no Sistema; (c) o protocolo CoAP não introduz uma grande carga adicional comparando com o envio de mensagens simples por UDP; (d) o protocolo CoAP permite ter pedidos muito mais pequenos que os pedidos HTTP.

Com este protótipo, conseguimos ter uma solução que cumpre os objetivos que nos propusemos:

Uma solução composta por componentes low-cost, que comunica por um canal seguro. Reduzimos o número e tamanho de mensagens derivado de usarmos CoAP sobre UDP e conseguimos desenvolver uma solução baseada em tecnologias de fácil aprendizagem, como o NodeJS.

No entanto, existem alguns pontos que consideramos uma limitação desta solução. Se por um lado o facto de não usarmos certificados não significa que não possamos ter autenticação mútua, por outro lado, a utilização de chaves pré partilhadas permite que um agente malicioso consiga atuar como fidedigno se conseguir roubar essa chave. Também não conseguimos garantir segurança futura perfeita, pois os algoritmos de criptografia utilizados não são baseados no Diffie-Hellman.

De forma a melhorar a segurança da solução, sem necessitar aumentar a carga sobre o sistema, poderíamos ter 2 abordagens. Troca de chaves dos elementos periodicamente (manual ou remotamente — onde o servidor poderia iniciar um processo de troca de chaves) ou então implementar um mecanismo de chave individual onde cada elemento partilha uma única chave com o servidor.

Este trabalho foi apresentado num poster na International Conference in Engineering Applications, nos Açores em 2019

Palavras-chave: Redes de sensores sem fios, Internet das coisas, Segurança, CoAP, DTLS

Abstract

In the recent past the Internet of Things has been the target of a great evolution, both in terms of applicability and of use. Society increasingly wants to use and massify the IoT to obtain information and act in the environment, for example, to remotely control an irrigation system.

The reduction in the cost of devices and the constant evolution of personal mobile devices has largely contributed to their spread. However, its implementation is carried out in adverse environments and outside the typical information systems. The devices are, as a rule, limited in terms of resources, both computation and memory.

The applicability to the IoT of the security techniques already known to conventional systems has therefore to be adapted, because it does not take into account the characteristics of the resources of the devices and require additional load when exchanging messages between these system elements. In addition, the development of applications is difficult because there is not yet developed tools and standards as there are for the traditional HTTPS or TLS when considering conventional systems.

In this work, we intend to present a prototype of a low-cost solution (compared to existing equivalent solutions) that uses a secure communication channel based on standard protocols. An application is also developed based on technologies more familiar to programmers, similar to traditional Web development. We took into account the ”Green By Web” project as a case study.

We have concluded that it is possible to have a secure communication, using UDP/DTLS over the CoAP protocol. With this approach we optimized the number of exchanged messages between the client and the server to be up to 8 times less and their size to be up to 10%, comparing against applications that use TCP/TLS connections, such as web applications that use HTTPS. This allows the energy spent by the low-cost components to be lower and increases their battery lifetime.

Keywords: Wireless Sensor Networks, Internet of things, Security, CoAP, DTLS

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The constant evolution in information technology and microelectronics have led to major advances in the integration of micro components, such as sensors and other information systems. Significant research in these areas has resulted in the development of various techniques for collecting and processing environmental data for further interpretation using more complex systems.

The use of these components tend to increase as they become cheaper every day and their applicability grows in all areas of development. The Internet of Things (IoT) was developed with the aim of connecting physical objects and making them communicate through specific and optimized networks, taking into account the IoT devices' limitations, like energy resources.

The communication between the devices and the Internet is often necessary to transmit information collected at a given physical location. This connection has to be taken into account in the implementation and it may use wireless sensor networks (WSN). WSN were developed with the aim of being able to monitor environment characteristics in several locations simultaneously, thus making it possible to interpret them in an aggregated way.

1.2 Problem

IoT nodes present several resource restrictions at the level of resources (e.g., energy, computing and memory). These limitations triggered communication technologies specifically tailored to them. In the simplest scenario, each device can be directly connected to the Internet, using mobile technologies. However, other proprietary technologies such as SIGFOX [43] or Zigbee [3] may be used. Devices can even access internet wireless

routers using IEEE 802.11.

One of the big issues in the IoT is their security. Because of their limited nature these devices can not use all the security mechanisms that exist for conventional networks. For instance, asymmetric cryptography, which guarantees a high level of confidentiality, requires computational and energy processing that is not feasible for these devices. Due to the fact that the IoT devices do not communicate through wired networks, all the information is transmitted by radio frequency and thus easily accessible to an attacker.

Authenticity of information is also a very critical factor. An attacker can pretend to be a trusted actor in the network by relaying valid messages. The integration of new devices and the possible overlapping of several wireless networks also raises a sensitive issue.

Also, the development of applications for these devices, client or server side, is difficult to achieve, because nowadays it uses programming languages that require a more complex implementation. The C and C++ languages and their variants are the most common programming languages used, and they have a big learning curve. Integrate them in a system that needs to be secure turns out to be very challenging.

1.3 Objectives

This work aims at building an IoT application that communicates thru a secure channel, using standard protocols that turn its development easier and keep it low-cost.

1.4 Contribution

We built a prototype of a client-server system that communicates using DTLS. It uses NodeJS to build a CoAP server, providing an easier way to develop a REST application. Also it only uses low-cost components. The prototype was based on a Green By Web project, which is an irrigation system already in use. However, our prototype does not depend on any applicability.

1.5 Structure of the document

This document begins by presenting an analysis chapter on IoT and WSN solutions, describing their characteristics and the protocols and standards most used, as well as their state of the art. Next, we present a chapter that addresses the design of a solution for a prototype and the procedures for its development, which optimizes the security attributes of our case study. Finally we present the results of experiments on the same prototype,

highlighting the main differences between this development and the solution presented as case study.

Chapter 2

IoT Architecture

The Internet of Things (IoT) allows multiple physical objects to sense and talk with each other in order to share information and coordinate actions. The main challenge on development of such devices is related to their availability due to their resources limitations. Although there is some visible evolution in the IoT, it is important that the research focus in the creation of standards, in order to increase the community developing the same solutions and ensuring the interoperability between them and the conventional networks [2, 42].

2.1 The Internet of Things

While the first and second generations of the Internet were categorized by the connection of people to the Internet, with personal computers and mobile devices, respectively, the third generation aims to connect as well things to the Internet. These things belong to the physical world and have the capability to interact with each other and with other entities, having their identity in the virtual world. [2]

The IoT is deployed under different domains, being Wireless Sensor Networks (WSN) widely used when different network typologies and multi hop communication between low-cost and low-power devices (sensor nodes) are needed. In this sense, WSN are considered an integral part of the IoT paradigm of connecting everyday things (Figure 2.1). While WSN must assures the communication between each component, the IoT layer is responsible for providing IP-based communication, with internet connection as a requirement. [2]

More details about WSN are presented in the next section.

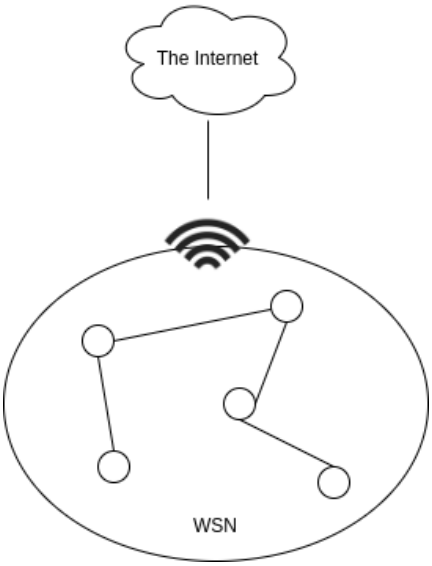

Figure 2.1: The IoT and WSN

2.2 Wireless Sensor Networks

Wireless Sensor Networks (WSNs) are networks composed by one or more gateways (also known as base stations) and a much larger number of nodes dispersed on a physical environment. They are typically used to ensure the communication for the IoT. Through sensors, the nodes measure information about the surrounding environment and send it to the base stations, which treats it and acts according to the implemented solution (Figure 2.2). A node can have sensors and actuators. The first are responsible for monitoring the environment, the second act on the environment. [40].

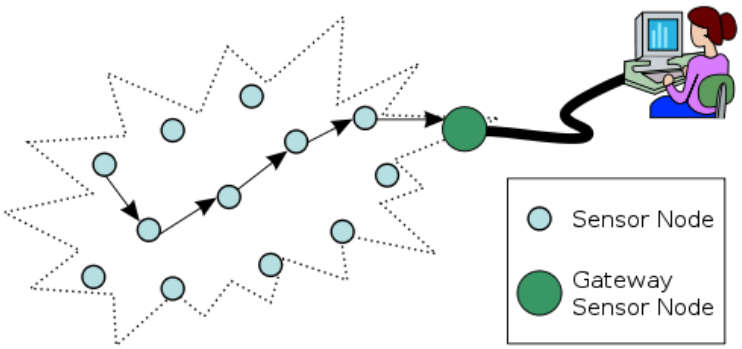

Figure 2.2: Wireless sensor network, extracted from [10]

WSNs can have different designs, each with its own specifications. Due to the adverse conditions of the environment where nodes are installed, they provide infrastructures that should be tolerant to message delays and losses. Because of short-life power source nodes, the transmitted or received data should use little bandwidth and thus optimize the power

used in the communication [40].

These types of networks are applied in systems that require constant environment monitoring. This means that they may be static (e.g., temperature sensors in a given space) or dynamic (e.g., speed sensor in a vehicle). The nodes comprising it may be fixed (where the number of nodes is defined when designing the solution) or dynamic (new nodes can be added to the solution) [40, 51]. Some WSN applications include:

- Chemical detection;
- Health care services;
- Emergency response;
- Survival missions;
- Movements of vehicles;
- Volcanic activity monitoring;
- Weather forecasts.

2.2.1 Sensor node

A sensor node consists of an integration of the following elements:

- An energy-saving sensing device;
- An analog to digital signal conversion circuit;
- A processor for small computations such as collect data aggregation before transmission to the base station;
- Wireless communication device - for transferring data to the base station;
- Power module.

In some networks, sensor nodes can communicate with one another to collaborate in collecting data, aggregating and sending to the base station, in order to reduce the energy consumption in these operations, and to increase their battery lifetime [40, 51].

2.2.2 Base station

Base stations act as gateways between the WSN and other networks or devices. In some networks, the base station is physically close to the nodes, and use low energy consumption communication protocols.

In other networks, base stations are significantly displaced from their nodes, and they require more energy-intensive protocols to transmit [49].

2.2.3 Security challenges

Compared to other network types, WSNs present specific challenges due to their vulnerabilities. Nodes have limited resources (lower power capacity, because they need batteries or other limited portable power supply mechanisms, little amount of memory and processing power), and in various types of applications are physically unprotected, usually communicating only by radio [40]. Vulnerabilities can be exploited in attacks that compromise one or more security properties.

Availability is probably the most critical point of WSN nodes. Attacks take advantage of devices having limited power and communicating by radio technologies causing them to be threatened by network jamming attacks. The attacks make use of additional devices that transmit signals on the same frequency to cause collisions, delaying and increasing power consumption, which can cause degradation or denial of service [47].

Sending false messages is also a threat to availability. These attacks include sending requests for new connection establishments, false retransmission requests and removing control messages. This leads to increased traffic and congestion and additional processing at the receiving nodes.

Using radio-based technologies also promotes easier access to the communication medium and makes it possible to compromise the confidentiality of the messages, making it susceptible to traffic analysis attacks (Eavesdropping). Communication to and from each node can be monitored by any device capable of capturing information transmitted over wireless. Also, it is possible to gather the localization of nodes using signal strength analysis techniques. Unauthorized physical access to data can provide attackers with valuable information to tamper with data on any network element (third party misappropriation). Depending on the type of application, many times, the physical location of the WSN nodes is not protected. First, because they are normal networks with large numbers of nodes; second, because they are geographically distant from each other, and, lastly, because many of the applications are intended to monitor the physical environment of public

spaces and, therefore, the physical protection of each would be very costly or not feasible at all [39, 45, 17].

Data integrity and authenticity is also threatened because of weak physical protection, which makes keys more easily to be stolen and used in attacks such as sending false messages, replacing trusted network nodes or changing nodes initial behavior.

The collection, processing and transmission of data should be minimized so that the power consumption is also minimal. However, security mechanisms should be considered, so, many times, more fragile encryption algorithms are used [27, 32].

Attacks in WSNs are categorized in two types: Passive attacks, those that do not adulterate the transmitted information. Eavesdropping is an example of a passive attack. Active attacks, where attackers can listen to and modify the transmitted data.

2.3 Standardized Protocols

In this section we present the main used protocols in the IoT and WSN. The existing standardized protocols for the TCP/IP stack are not usable by IoT devices, because they require much more computation power than those that the IoT devices should implement. Although there are some proprietary protocols for IoT devices, like Zigbee, supported by vendors, some entities like the World Wide Web Consortium (W3C), the Internet Engineering Task Force (IETF), the EPCglobal, the Institute of Electrical and Electronics Engineers (IEEE) and the European Telecommunications Standards Institute (ETSI) have developed specific standards for each layer of the IoT stack [2, 9]. Figure 2.3 presents some IoT protocols organized according to their layer. The next sections detail each layer.

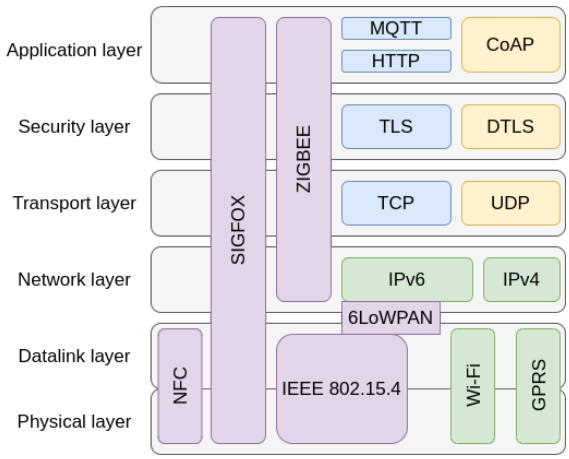

Figure 2.3: IoT Protocols

2.3.1 Physical and data link layer

This section presents the protocols used in physical and data link layers. We focus this study on NFC, IEEE 802.15.4, GPRS and IEEE 802.11 protocols as they are the most widely used nowadays.

NFC

The Near Field Communication (NFC) is a wireless communication technology with a 10 cm range. It defines two types of devices: active and passive. Active devices are those that have a power supply, while passive devices are the others [18].

Communication are base on a message/reply flow. A *initiator* device generates a Radio frequency (RF) field and sends a message to the *target* device that replies with some data. A passive device can only be a target device, while an active device can be a target or an initiator. An initiator can send messages to multiple targets [18].

Like other technologies, NFC is subject to many threats. One of them is the eavesdropping, although it is a short range communication system, some specific characteristics, like the quality of the attacker's receiver and the power sent by the NFC devices may allow its success. Data corruption and modification may be possible, since the attacker can listen and modify the transmitted data. Data insertion can be achieved transmitting additional data messages in the communication. Although it is very difficult, man-in-the-middle-Attacks are a threat to NFC too [18].

There is some defense capabilities defined to avoid the threats [18].

- Data corruption and modification - NFC devices can check the RF field while sending and can stop data transmitting if an attack is detected.
- Data Insertion - The answering device may continuously listen the point of transmission and detect attackers that join to connection.
- Man-in-the-middle-attacks - It is very difficult to achieve this attack. However, using active-passive communication may reduce even more the probabilities of this attack. Also, the active party can detect a potential attacker if is listening to the RF field.
- Secure Channel for NFC - Using a secure channel is the best approach against all the threats. As the man-in-the-middle attack is a very limited threat, a standard Diffie-Hellman protocol, with no authentication, may be used to agree on a shared key that should be use to derive a symmetric key, which is used to provide confidentiality, integrity to the transmitted data.

IEEE 802.15.4

IEEE 802.15.4 is a wireless communication standard that defines Physical and Media Access Control layers. It is similar to Bluetooth, but is more focus on the communication between resource limited devices. It uses 250 Kbit/s speed communications reaching up to 10 meters and supports 16 and 64 bit addressing. While the first decreases the messages size (as the address size is smaller, so is the message), the second allows a much larger number of connected devices [14].

Protocols like Zigbee and WirelessHART extends this standard to implement upper layers.

The physical part of the standard aims to achieve reliability by transforming the transmitted data to occupy more bandwidth at lower frequencies. Each frame of sent data occupies at most 128 bytes. The packets are small in order to avoid failures in low-energy wireless communications.

There are four types of frames in this standard: data frames, acknowledgment frames, beacon frames, and MAC command frames.

Communication security is managed in the MAC layer, using symmetric cryptography in the hardware. The algorithm applied is the AES with different types of operation:

- AES-CBC-MAC - to provide authenticity with a Message Authentication Code (MAC);
- AES-CTR - to provide confidentiality;
- AES-CCM - to provide both confidentiality and authenticity.

It use keys of 128 bits and may have a MAC of 32, 64 or 128 bits. This allows to have multiple levels of security, including none.

The keys used by the standard are implicitly known by the communication parties or determined from the information in the *Key Source* and *Key Index* fields. The first indicate which group key is the originator and the second identifies the specific source. Their management is not specified in the standard, and may be implemented in the upper layers.

IEEE 802.15.4 does not have an appropriate key model. The key management is applied according to each application nature. Using traversal key in Access Control Lists allows nonce to be reused and an attacker may be able to recover plain texts from ciphers.

Figure 2.4 shows how the data frame in the standard is built. To enable security, the

Security Enabled Bit in the Frame Control field should be set.

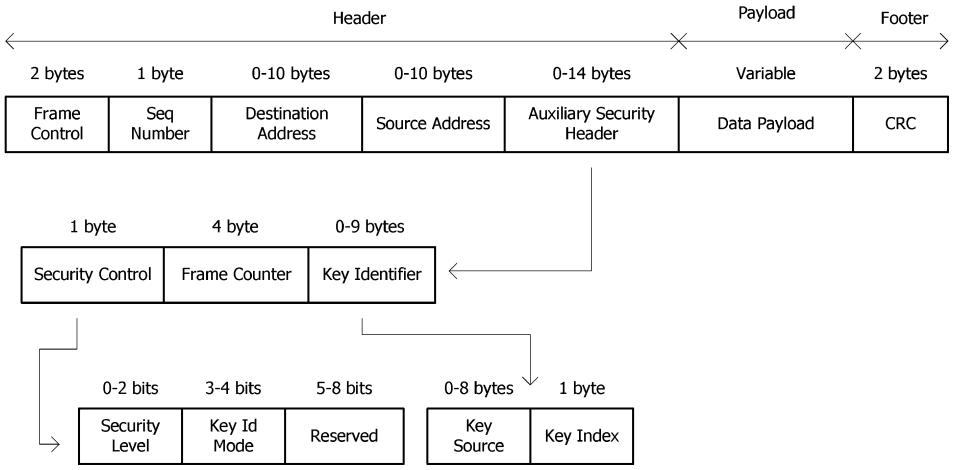

Figure 2.4: IEEE 802.15.4 data frame composition, extracted from [14]

The Auxiliary Security Header gives more information about how the frame is protected. It is composed by the Security Control, Frame Counter and Key Identifier fields. The Frame Counter field enables the message numeration (coordinated by the communication parties) and avoid Message Replay Attacks.

Depending on the protection applied, the payload is created accordingly to figure 2.5.

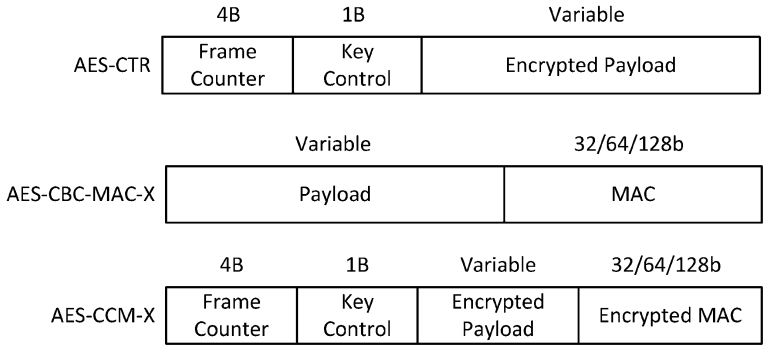

Figure 2.5: IEEE 802.15.4 protected payload, extracted from [14]

GPRS

General Packet Radio Service (GPRS) is a technology based on wireless packet communication and allows transfer rates far superior to predecessor technologies (GSM). The GPRS enables mobile operators to provide new types of services, such as video calls, with quality. The devices connect to a GPRS antenna near their locations, which forwards

their messages to the Internet using GPRS gateways (Figure 2.6). Many WSNs are implemented using this technology because of the large coverage and because it is a technology that only uses the resources in the reception and sending of data by radio [48].

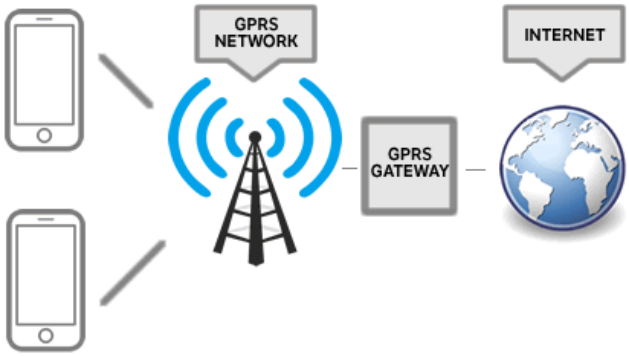

Figure 2.6: GPRS architecture, extracted from [35]

Natively, GPRS includes some mechanisms in order to increase the level of network security. The main security objectives are to protect the network against unauthorized access and to protect the privacy of users. It includes the following components:

- Subscriber Identity Module (SIM) - Implemented by a unequivocal card containing an identifier for each device (IMSI). It is protected by a 4-digit PIN. It has a secret authentication key and an algorithm that generates encryption keys;
- Subscriber Identity Confidentiality - IMSI protection mechanisms when signaling messages are transmitted;
- Subscriber Identity Authentication - Protects user from fraudulent actions;
- Data and Signaling Protection - Uses GPRS Encryption Algorithm (GEA) - symmetric flow cipher algorithm - to protect data confidentiality;
- GPRS Backbone - GPRS data protection mechanisms from mobile operators.

Some weaknesses in GPRS security are related to:

- The commitment of IMSI, given that in certain situations it can circulate in the network;
- The inability of the authentication mechanism to authenticate the network;
- Ability to reuse authentication parameters;
- Possibility of suppressing the cipher on the network or of changing the parameters of the cipher algorithms;
- Lack of efficacy in the security mechanisms in the GPRS backbone network.

IEEE 802.11 (WiFi)

The IEEE 802.11 is a set of protocols, both for physical layer and media access control, for implementing a wireless local area network (WLAN). It is widely used at homes, offices or public places as hot spots to multiple devices connect to the Internet, also known as WiFi. These networks do not have physical wired connection between senders and receivers, so they operate using radio frequency technology (RF).

WLAN are increasingly being used due to the fact that they are easier and cheaper to implement, reducing the complexity comparing to traditional LAN. They need an Access Point (AP) that broadcasts a wireless signal propagating their identification (Service Set Identifiers - SSID) which devices can detect and tune in. The devices that want to connect must have wireless network adapters (Figure 2.7).

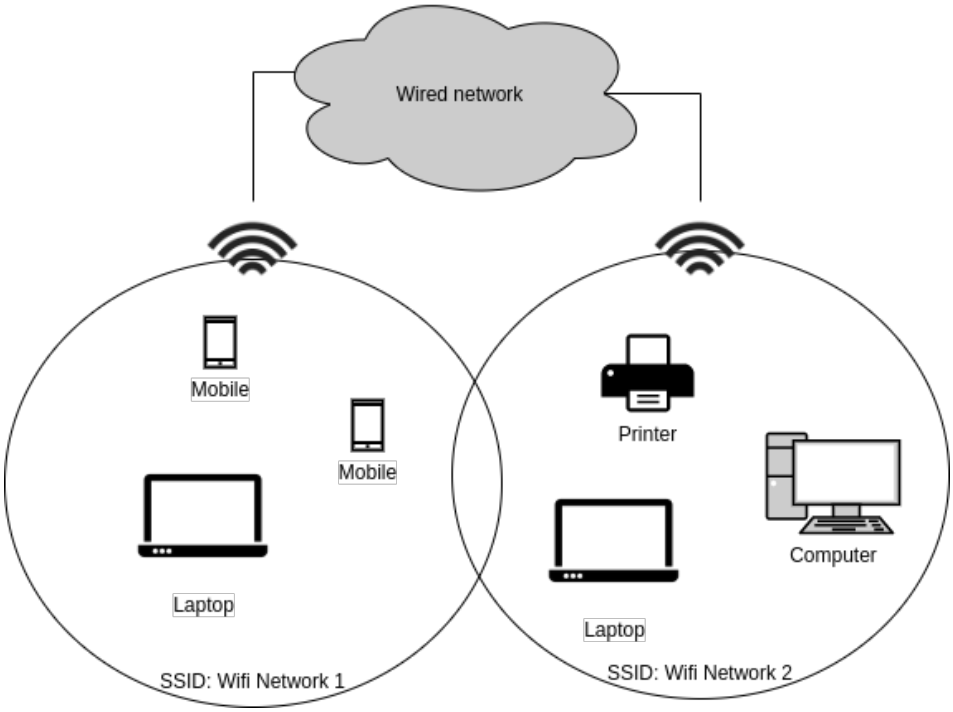

Figure 2.7: WiFi network architecture

Many specifications were developed according to the evolution of the technology and to support best transmission rates and a better coverage. It started with IEEE 802.11 that had transmission rates of 1 and 2 Mbps. IEEE 802.11b allows 11 Mpbs, while 802.11a/g can assure rates up to 54 Mpbs. Nowadays, IEEE 802.11n offers the maximum of 600 Mpbs, using 5GHz and 54Mbps using 2.4 GHz, while IEEE 802.11ac supports up to 1300 Mpbs. At the beginning, the velocity of the communication was a concern compared to wired networks. However, these issues tend do disappear as the technology supports in-

creasing velocity.

It is also possible to use WiFi to connect devices in a non structured way, that means, without an access point, allowing the devices to connect in a peer to peer communication mode.

To protect the network there are protocols to establish a connection that ensure security:

- Wired Equivalent Privacy (WEP) - This protocol was primarily develop to give the same security level than in wired connections. It was abandoned because it was very vulnerable to attacks.
- WiFi protected Access (WPA) - It uses a pre shared key and a Temporal Key Integrity Protocol (TKIP) for encryption and it was an improvement over WEP. WPA enterprise allows using server authentication to generate keys and certificates.
- WiFi Protected Access version 2 (WPA2) - The most significant change comparing to version 1 was that it uses AES for encryption that is considered sufficient to protect communication.

Although it is a wireless technology, WiFi allows the coverage of big areas, being widely used to supply internet access in many public places and enterprises. One single Access Point can cover up to 200 meters in an open area.

2.3.2 Network layer

Along with the creation of billions of devices connected through the Internet, the IPv4 standard became insufficient to address all these devices. For the TCP/IP stack, IPv6 was born. However, IPv6 packets are large to be used in 802.15.4 frames [14].

Using fragmentation, reassembly and header compression, the 6LowPAN adaptation layer assures that an IPv6 packets can be carried efficiently in small link layer frames.

This specification does not define any security mechanisms, despite there is known vulnerabilities, like forging or accidentally duplication of interface addresses, compromising the uniqueness of the devices.

2.3.3 Transport layer - TCP and UDP

User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) are the two main protocols used in transport layer that is responsible for delivery data from the sender to the receiver. UDP is the main standard in the IoT, because it provides best-effort and

less overhead, while TCP provides connection management and a more reliable connection between hosts [8].

UDP sends datagrams in IPv4 or IPv6 packets. A datagram is a basic transfer unit that has all the information for a message to be routed from the source to the destination, without the guarantee that the messages arrive in the same order that were sent. UDP does not guarantee as well packet delivery, but it is faster and its packets are smaller compared to TCP. Figures 2.8 and 2.9 show the differences between a UDP and TCP message.

TCP SEGMENT HEADER FORMAT												
Bit #	0	7	8	15	16	23	24	31				
0	SOURCE PORT						DESTINATION PORT					
32	SEQUENCE NUMBER											
64	ACKNOWLEDGMENT NUMBER											
96	DATA OFFSET		RES		FLAGS		WINDOW SIZE					
128	HEADER AND DATA CHECKSUM						URGENT POINTER					
160...	OPTIONS											

Figure 2.8: TCP header segment format

UDP is a protocol that is not connection-oriented. Unlike TCP (figure 2.10) there is no

UDP SEGMENT HEADER FORMAT												
Bit #	0	7	8	15	16	23	24	31				
0	SOURCE PORT						DESTINATION PORT					
32	LENGTH						HEADER AND DATA CHECKSUM					

Figure 2.9: UDP header segment format

notion of liveness of the other party, the only information exchanged between the parties is the message itself, as referred in figure 2.11. With UDP, when guarantees are required, some control flows like timeouts, retransmissions and acknowledgments, must be implemented for that application.

Because packets are easily forged, source port and address are insufficient for authentication. To guarantee security, UDP uses DTLS (detailed in section 2.3.4).

Fragmentation

The Maximum Transmission Unit (MTU) defines the maximum size that one communication layer can support. Packets that are larger than the MTU must be fragmented into smaller packets during the transmission.

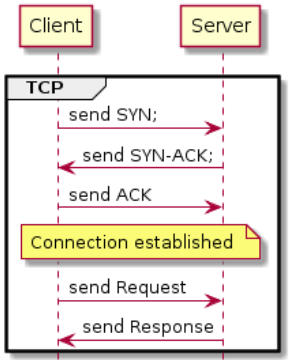

Figure 2.10: TCP message flow

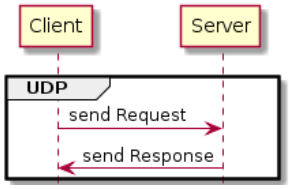

Figure 2.11: UDP message flow

Normally, the MTU is established to 1500 bytes, so every packet bigger than this, are fragmented. In TCP, because of its nature, that is not a problem; in UDP, splitting a message increases the risk of not delivery or deliver the fragmented messages out of order.

In theory, the size field sets a limit of 65,535 Kb for a UDP datagram. However, to increase the reliability of the communication, one packet is safe if it is not bigger than 1500 bytes. That introduces a challenge in constrained environments because when communicating in UDP the peers must be able to reassemble the messages when they are bigger than the MTU and are fragmented in the communication, introducing additional overhead. When the biggest concern is to optimize the consumed energy, fragmentation and reassemble should be avoided.

2.3.4 Secure transport layer - TLS and DTLS

The main goal of TLS is to provide confidentiality between two parties in a network (usually using TCP).

It runs between the application and transport layer, so it can be used by any application as a transport standard. However, applications must specify how to add security with TLS, namely, the cryptography algorithms to be used, the key sizes and the authentication mechanism.

TLS

Transport Layer Security (TLS) is the successor of SSL. Originally, in 1994, Netscape created a way to secure communications between clients and servers on the web. Version 1.0 was never released because of its serious security flaws. So, the first public release of SSL was 2.0 in 1995, and the final version (3.0) in 1996.

In 2011, SSL 2.0 was considered deprecated because it had known deficiencies [20], namely:

- Message authentication using MD5;
- Handshake was vulnerable to man-in-the-middle attacks, allowing the use of weak cipher picking;
- Message integrity and encryption use the same key;
- Session can be easily terminated by a man-in-the-middle with a TCP FIN message.

In 2015, SSL 3.0 was also considered deprecated because of the following vulnerabilities [21]:

- Non-deterministic padding used in Cipher Block Chaining allows plain text recovery (POODLE attack);
- Man-in-the-middle attacks in key renegotiation or session resumption;
- Use of weak cryptography algorithms such as SHA-1 and MD5.

SSL 3.0 was also unable to take advantage of new developed features for TLS, such as:

- Authenticated Encryption with Additional Data (AEAD);
- Elliptic Curve Diffie-Hellman (ECDH) and Digital Signature Algorithm (ECDSA);
- Stateless session tickets;
- A datagram mode of operation, DTLS;
- Application-layer protocol negotiation.

TLS was first introduced in 1999 as an upgrade to SSLv3. Despite not having significant differences, it did not allow for interoperability.

TLS 1.1 was a minor update released in April 2006. Some differences in this version included protections against Cipher Block Chaining (CBC) attacks.

TLS 1.2 was released in August 2008. Changes included adding cipher-suite-specified pseudo random functions (PRFs), adding AES cipher suites and removing IDEA and DES cipher suites.

The current version of TLS, TLS 1.3, was released in August 2018 [23] and has improved security and speed. It includes the following enhancements:

- The list of supported symmetric encryption algorithms has been pruned of all considered legacy algorithms. All the remaining algorithms are Authenticated Encryption with Associated Data (AEAD) algorithms;
- Added the zero-RTT (0-RTT) mode, saving a round-trip at connection setup for some application data at the cost of certain security properties. (This may be very useful in systems that need to optimize messages exchange, as it only needs 4 messages to establish a session) (figure 2.12);
- Static RSA cipher suites have been removed; all public-key based key exchange mechanisms now provide forward secrecy, like ephemeral Diffie-Hellman based protocols;
- All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows for various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection;
- The handshake state machine has been restructured to be more consistent and to remove superfluous messages;
- Elliptic curve algorithms are now in the base spec, and new signature algorithms, such as EdDSA, are included. TLS 1.3 removed point format negotiation in favor of a single point format for each curve. The use of elliptic curve algorithms ensures similar public key cryptography security level at a lesser cost. The strength of the cipher with 160 bit keys is equivalent to 1024-bit keys in the RSA protocol, has a smaller number of calculations in private key usage, uses little space to store keys and cipher parameters require little bandwidth, which it is also an advantage for systems composed of limited resources devices.

The protocol works as follows:

- For each session, a session key is agreed, in the TLS handshake phase (figure 2.14);
- Every message sent/received in the session is encrypted/decrypted with the session key, using a symmetric algorithm (e.g., AES);
- Every message has a MAC that is used to guarantee integrity of the payload (figure 2.13).

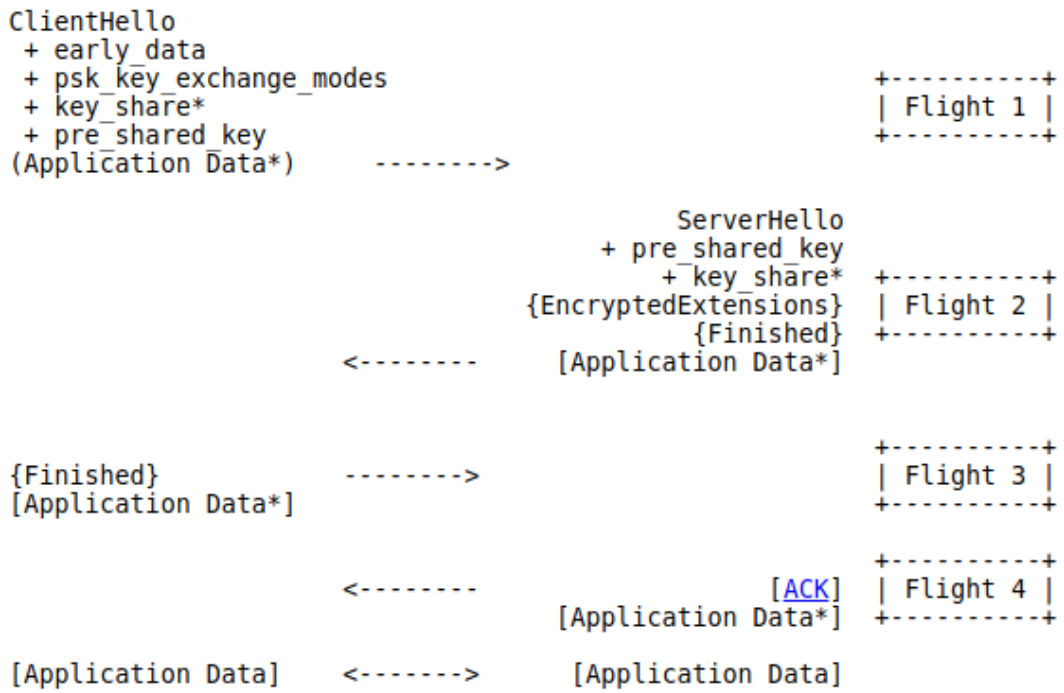

Figure 2.12: 0-RTT handshake (Extracted from [26])

Byte	+0	+1	+2	+3
0	Content type			
1..4	Version		Length	
5..n	*Payload*			
n..m	MAC			
m..p	Padding (block ciphers only)			

Figure 2.13: TLS message, extracted from [37]

The TLS handshake involves a series of steps through which both the parties validate each other and start communicating through the secure TLS tunnel. In this phase both parties also agree in the session key and the symmetric algorithm to be used in the session.

The peer's identity can be authenticated using public key cryptography. This step is optional and can be achieved with trusted certificates. Normally, at least one peer requires the authentication.

The negotiation of the session key is unavailable to eavesdroppers. If public key cryptography is not used, the alternative is to use pre shared keys, normally already installed in participating devices.

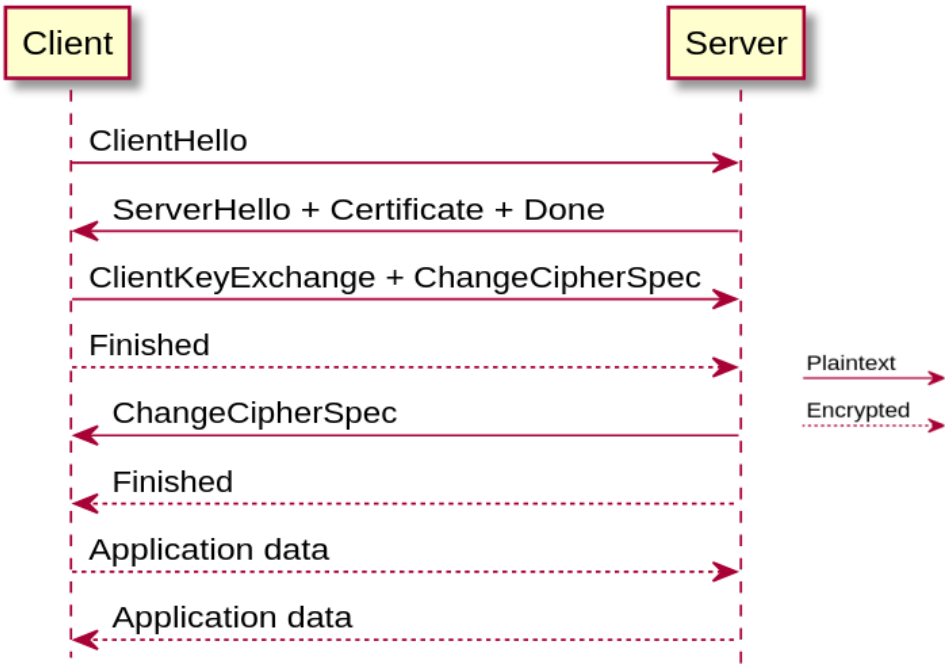

Figure 2.14: TLS 1.2 Handshake

DTLS

Datagram Transport Layer Security (DTLS) is the equivalent of TLS in TCP but for UDP communication. DTLS features full encryption and authentication. However, it does not include delivery guarantee nor in-order delivery of data.

It includes an implementation of TLS-like handshake, modified to work with datagrams (figure 2.15), that is, it handles the problems of packet loss and packet reordering for that negotiation. After the key is exchanged between the peers, DTLS does not handle those concerns anymore and have the same features and the same issues of a UDP message.

2.3.5 Application layer

This section presents some protocols that are most relevant in the IoT. The study refers to HTTP and REST, CoAP and MQTT.

HTTP

The Hypertext Transfer Protocol (HTTP or HTTPS when protected by TLS) is an application-level protocol that works as a synchronized request/response protocol in a client/server model and uses TCP connections (figure 2.16).

The client send a request to the server, defining the method and the resource (URI) that wants to access, followed by a MIME message containing the payload of the request.

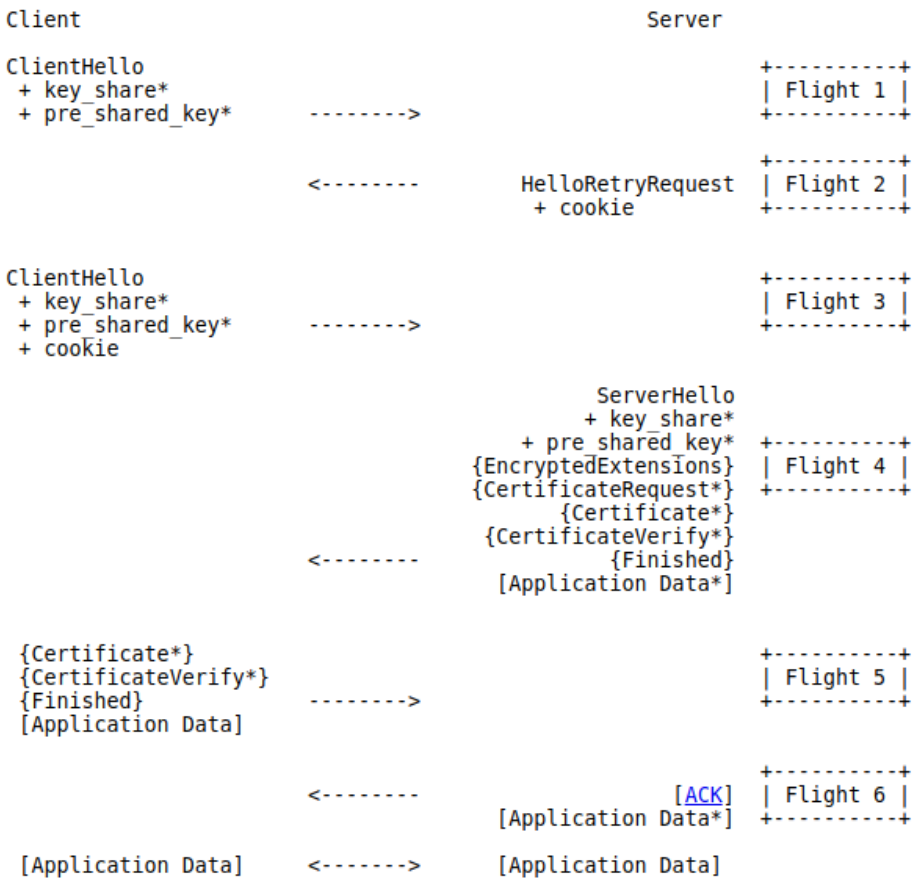

Figure 2.15: Message flights for a full DTLS 1.3 Handshake (with cookie exchange) (Extracted from [26])

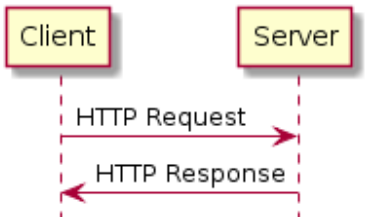

Figure 2.16: HTTP Protocol

The server responds with a status line and a success or error code followed by a MIME message containing the response payload, that, in case of success, contains the resource requested.

The HTTP defines a set of request methods accountable to designate the action to be taken on the requested resource [22]

- GET - Is the method used by HTTP requests to retrieve information and must not modify any object. The URI identifies the resource to get information from. This method allows the variant ”conditional GET” if the request has a flag, in order to

optimize in transferring information, and allow using cache in the clients side if server has not been updated since last similar request.

- POST - The server takes the action on POST requests according to the URI. It is widely used to provide form data about some resource to a server. The action performed by the POST method might not result in a resource that can be identified by a URI. The responses to POST requests are not cacheable.

- PUT - This method is used to store or update a resource identified by the URI. The response to these requests depends on the previous existence of the resource in the server.

- DELETE - The DELETE method is used to delete the respective resource from the server.

Representational State Transfer (REST)

Representational State Transfer (REST) is a software architectural style that defines constraints to be used in Web services communication. This approach provides the capability to easily communicate between clients and servers using standards that optimize the messages payload and processing load. Among those constraints there are:

- Client - Server architecture: By separating the user interface concerns from the data storage concerns, the portability of the user interface across multiple platforms is improved;

- Stateless communication: This constraint ensures that every request from client to server must contains the information necessary to understand the request and do not have any context stored at server side;

- Cache: Every data contained in a response must be cacheable or non-cacheable. When a response is cacheable the client reuse it in later and equivalent requests.

- Uniform interface: The uniform interface simplifies and decouples the architecture, which enables each part to evolve independently. The four guiding principles of this interface are:

  - Identification of resources: Individual resources are identified in requests, for example, using URIs in web-based REST systems.
  - Manipulation of resources through representations: When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource.

  - Self-descriptive messages: Each message includes enough information to describe how to process the message.
  - Hypermedia as the engine of application state: Clients make state transitions only through actions that are dynamically identified within hypermedia by the server.

- Layered system: Layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot ”see” beyond the immediate layer with which they are interacting.

The elements transferred in a REST system are representation of resources and have the following characteristics:

- Resource: is the key abstraction of information in REST. Any element that exists in a system is a resource (e.g., a document, an image);
- Resource identifier: Key to identify the resource. In HTTP, we use URIs;
- Resource representation: State of some resource at any particular timestamp. Consists in data, metadata and hypermedia links;
- Media type: The data format of a resource representation.

It is very common to see REST to be used along with HTTP, for example for CRUD (Create, Read, Update and Delete) applications. One example of a simple CRUD, HTTP, REST application endpoints is described below:

- URL: http://domain.com/api/entries
  - HTTP Method: GET
  - Result: Returns all entries in application/JSON format
- URL: http://domain.com/api/entries
  - HTTP Method: POST
  - POST body: JSON string
  - Result: Returns a response with code 201 and the URI of the created resource.
- URL: http://domain.com/api/entries/id
  - HTTP Method: GET
  - Result: Return entry with id ”id” in application/JSON format
- URL: http://domain.com/api/entries/id

  - HTTP Method: PUT
  - PUT body: JSON string
  - Result: Updates entry with id ”id”

- URL: http://domain.com/api/entries/id
  - HTTP Method: DELETE
  - DELETE body: empty
  - Result: Deletes entry with id ”id”

CoAP

Constrained Application Protocol (CoAP) is a web-transfer protocol for use on limited networks, such as WSNs. As in most web APIs, it depends on the Representational State Transfer (REST) architecture and applies HTTP operations such as GET, PUT, POST, and DELETE to resources identified by a URI. It allows integration with various types of data representation (for example, XML and JSON). It is designed to work with micro controllers, which have a limited RAM (10 kB), and to use the minimum of resources [25].

Unlike HTTP, CoAP deals with messages asynchronously over a datagram-oriented transport such as UDP. However, the protocol gives the ability to support reliability using an abstract messages layer that provides 4 types of messages: Confirmable, Non-confirmable, Acknowledgment, Reset [25].

The following features are also included in the protocol:

- Low header overhead and parsing complexity;
- URI and Content-type support (as in HTTP);
- Simple proxy and caching capabilities;
- A stateless HTTP mapping, allowing proxies to be built providing access to CoAP resources via HTTP in a uniform way or for HTTP simple interfaces to be realized alternatively over CoAP.

The protocol natively provides, and by default, security through DTLS, with parameters equivalent to 3072 bit RSA keys, without compromising the power constraints of the nodes. However, it continues to be the target of some attacks, namely:

- Bugs processing requests, due to application complexity;
- In the presence of proxies, the network becomes vulnerable to man-in-the-middle-attacks;

- Risk of amplification - The response packet is usually of larger size than the requests. An attacker can then add some payload to some packets and thereby cause a denial of service;

- Because there is no handshake phase in the UDP protocol, it is possible for an attacker to perform IP spoofing attacks.

Despite using UDP as the protocol in the transport layer, it is possible, natively, to have reliability in the exchanged messages.
Each message can be confirmable (CON) or non-confirmable (NON). The first one requires the server to answer with an Acknowledge message when it receives the request; and the client to send an Acknowledge when it receives the response. If the server can provide a response rapidly (Piggybacked Response), the ACK may have the response in the same message. However, if the server is not able to respond immediately, an empty ACK must be returned before the response (figure 2.17).

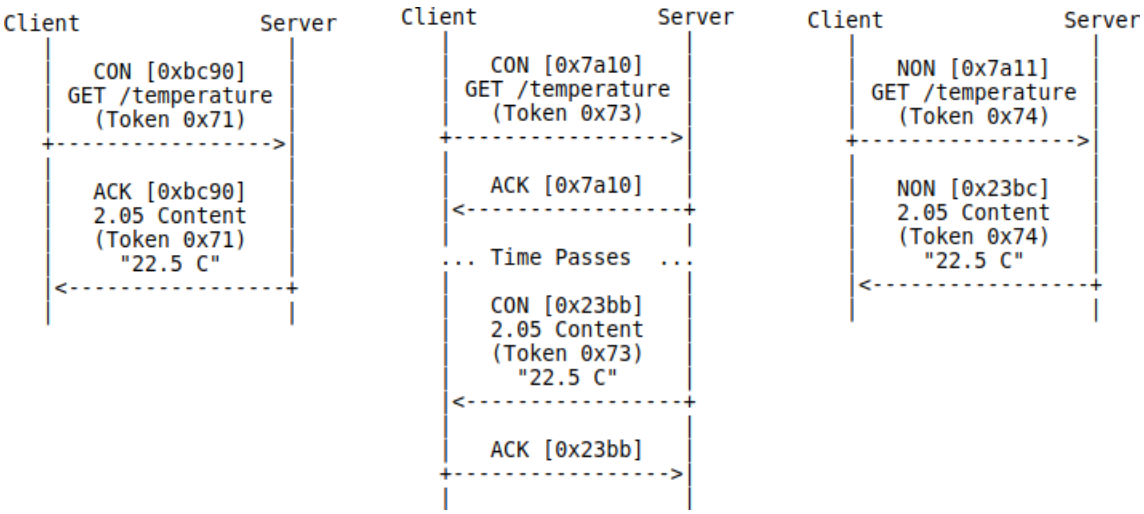

Figure 2.17: CoAP CON message (left), CON message with delay (center) and NON message (right), extracted from [25]

The message header format shows that the messages are optimized for constrained devices, allowing to reduce energy in sending/receiving and handling them. They are encoded in a simple binary format. The message format starts with a fixed-size 4-byte header. This is followed by a variable-length Token value, which can be between 0 and 8 bytes long. Following the Token value comes a sequence of zero or more CoAP Options in Type-Length-Value (TLV) format, optionally followed by a payload that takes up the rest of the datagram (figure 2.18).

The fields in the header are defined as follows:

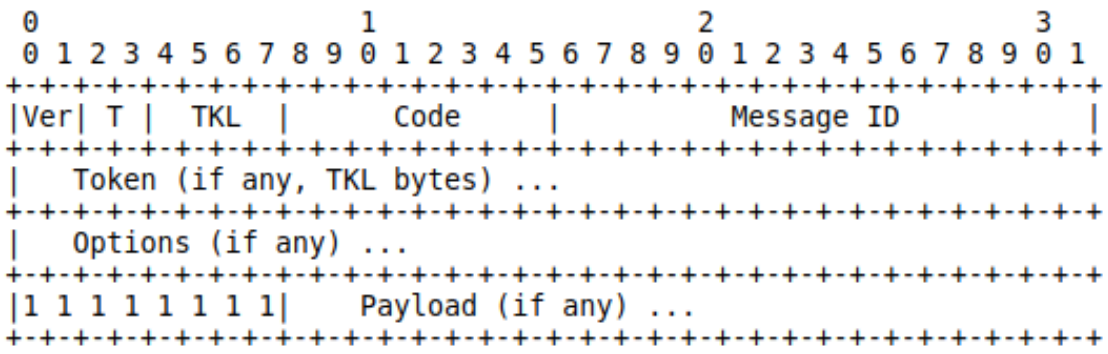

Figure 2.18: CoAP message format, extracted from [25]

- Version (Ver): 2-bit unsigned integer. Indicates the CoAP version number;
- Type (T): 2-bit unsigned integer. Indicates if this message is of type Confirmable (0), Non-confirmable (1), Acknowledgment (2), or Reset (3);
- Token Length (TKL): 4-bit unsigned integer. Indicates the length of the variable-length Token field (0-8 bytes);
- Code: 8-bit unsigned integer, split into a 3-bit class (most significant bits) and a 5-bit detail (the least significant bits). The class can indicate a request (0), a success response (2), a client error response (4), or a server error response (5). Code 0.00 indicates an Empty message. In case of a request, the Code field indicates the Request Method; in case of a response, a Response Code;
- Message ID: 16-bit unsigned integer in network byte order. Used to detect message duplication and to match messages of type Acknowledgment/Reset to messages of type Confirmable/Non-confirmable.

The header is followed by the Token value, which may be 0 to 8 bytes, as given by the Token Length field. The Token value is used to correlate requests and responses.

Header and Token are followed by zero or more Options. An Option can be followed by the end of the message, by another option, or by the Payload Marker and the payload.

MQTT

Message Queue Telemetry Transport is a publish-subscribe messaging protocol. It provides minimal battery loss and reduced bandwidth. Like other protocols with pub-sub architecture, it is composed by:

- Publisher - the device that publishes data;

- Subscriber - the devices that wants to receive data from a specific object;
- Broker - the intermediary that forwards information from publishers to subscribers;

2.3.6 Transversal protocols

This section presents the transversal protocols, that means, technologies that implement all the network stack from physical layer to application; SIGFOX and ZigBee are detailed here as they are widely used nowadays in the IoT.

SIGFOX

SIGFOX is a company that has implemented ultra-narrowband (UNB) technology to connect remote devices (figure 2.19). The company has several partners around the world who help in the deployment of antennas to increase network coverage [43]. The strengths of this technology are:

- Simplicity - no configuration required, active connection or signaling;
- Autonomy - Low power nodes' consumption;
- Reduced size of messages;
- A technology that works for both short-range and long-range implementation.

The mode of interaction with the devices is through a SIGFOX management tool, via REST API or via "Callbacks", HTTP/HTTPS communication mechanism.

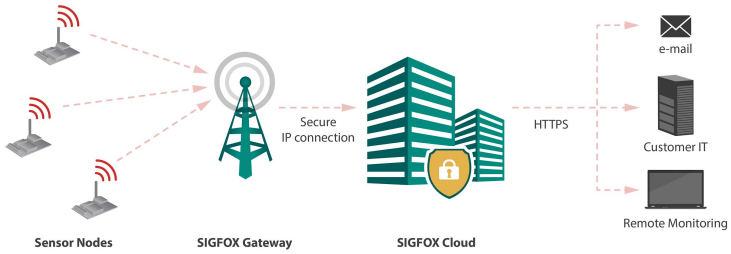

Figure 2.19: SIGFOX Architecture, extracted from [30]

All exchanged messages on the network are signed with information about the device and the message itself. The key for each device is provided prior to its implementation. It is tolerant to spoofing, replay, and message alteration attacks.

Confidentiality is not guaranteed by the SIGFOX communication, and some encryption algorithm must be used at the application level to protect.

Zigbee

The Zigbee protocol is based on the IEEE 802.15.4 standard, but with specific characteristics to increase the lifetime of the nodes. It allows for networks with up to 65000 nodes and fast implementation. It is designed for short distance communication [53].

Natively the protocol provides the following security measures

- Symmetric encryption;
- Message authentication;
- Integrity protection; and
- Anti-replay protection.

Security is based on symmetric encryption keys. Two key types are used to protect network communication.

- A 128-bit network key, shared by all available devices, used in the cipher of the broadcast communication;
- A 128-bit connection key, shared between two devices, used to protect communication between them.

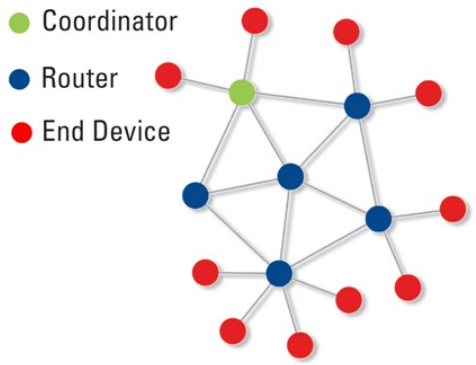

Figure 2.20: Zigbee architecture example, extracted from [19]

The Zigbee protocol (figure 2.20) consists of 4 layers: Physics and Medium Access Control (MAC), in common with IEEE 802.15.4, Network, and Application Protocol Zigbee.

The data transmitted between layers is reliable due to its ”Open trust” model.

At the network level, all packets are protected by AES-128 (confidentiality) and CBC-MAC (integrity) algorithms.

The application layer checks if the information to be sent will be protected by the Network layer, and if it is not, it protects the necessary packets by using the encryption key.

Key management: Cipher keys can be (a) preinstalled on devices; (b) transferred; and (c) derived from other keys (requires preinstalled keys).

When a new, unconfigured device joins the network, there is a risk that the keys are transmitted unprotected, however the time span is very short. To succeed an attacker may use congestion techniques to force a device to do factory reset and try to reenter the network. Physical access to devices is also a threat to the confidentiality of keys.

2.4 CoAP state of the art

This section reviews related work about CoAP, as we used this protocol to design and implement our solution (detailed in chapter 3). When analyzing the standardized protocols both for WSN and IoT, we began to understand that probably CoAP would be the best technology to adopt when converting IoT communication to UDP and provide an easy to develop application. So we dig into other CoAP implementations research. For that, the key points were (a) if it was already as a RFC, (b) if it used low-consumption security mechanism for the communication and (c) how we could use the best approaches in our prototype.

In [5], S Arvind, et al., study the security vulnerabilities and types of attacks to CoAP, carrying out a very similar scenario as we tend to build in the next chapter. They use a temperature sensor integrated with low-power WiFi chip to send series of outputs in a WiFi network. Also they use a network simulator to do penetration testing to a simple CoAP application. They were able to implement a malicious proxy and see clear text communication, concluding that end-to-end encryption should be implemented so that the application become secure.

Iglesias-Urkia, Markel, et al. [50] gave us the vision about how we could implement end-to-end encryption. This allowed us to study DTLS communication with 3 approaches: without authentication; with server authentication; and with client and server authentication. Also, the article provides a study about the mainly used technologies when implementing CoAP, which showed that CoAP is available using javascript (using NodeJS) or python, two programming languages that are usually adopted to build web applications. However, the python solution is not interoperable because it is based on an old CoAP specification and also do not have any security layer to establish end-to-end encryption

using DTLS. Despite having successful results, the article is based on results of a solution built on top of Raspberry Pi acting as WSN nodes, which is not the typical resources that are suitable to deploy in this kind of network.

Granjal, Jorge, et al., [13] addresses the limitations the current solutions have for end-to-end security in the IoT. They compare two approaches for CoAP: encryption at transport layer, using DTLS or object security, that is, encrypting message content. Both of them guarantee good results in application lifetime, however, DTLS session was implemented without public-key cryptography.

Dizdarević, Jasenka, et al., [12] presents many protocols and their best characteristics for implementation in IoT devices, concluding that CoAP is the best suitable regarding power saving. Also, they show two approaches when developing CoAP applications, using CoAP to handle all the server side requests, or, making CoAP server acting like a gateway to HTTP requests over internet. The first one allows a simpler architecture model, the second one allows a bigger scalability and the developing of more complex server application, being CoAP servers responsible for translating and forwarding CoAP nodes requests to the HTTP servers.

2.5 Summary

As a summary, the networks that we have described in this chapter hold a lot of challenges regarding the nature of their components. Usually, the deployed devices in a WSN are extremely limited in terms of memory, processing and energy, which introduces many challenges when developing a solution. Also, the communication between the nodes require a lot of power consumption. The wireless network, due to many factors, such as radio frequency noise, may be vulnerable to loss of messages.

Encryption techniques performed at the level of the nodes introduce a large load on them. This is because public key cryptography, such as RSA, requires a lot of computational power. Its applicability is difficult in this type of systems because, although it guarantees an ideal level of confidentiality, they greatly limit the lifetime of the nodes. Algorithms based on elliptical curves, as mentioned in the TLS section, can be an advantage when used in these solutions.

One solution is to use symmetric encryption (TinyOS's TinySec uses the AES algorithm), however this technique introduces some key management and distribution issues [31, 29, 28, 7, 50]. Zigbee uses symmetric keys and has two approaches to managing them, as mentioned in its section. Zhu, Sencun, et al., [52] present four alternatives to

manage:

- *Global Key* - All nodes have a single key. If this key is discovered all communication is compromised;
- *Pair-wise key node* - Each node shares a key with its neighbors. If a key is discovered, only the communication between these two nodes is compromised;
- *Pair-wise key network* - Each group shares a key. The main nodes of all clusters share another key to communicate between them or use the ”Pair-wise key node”. It requires large processing of the main nodes because they are forced to encrypt and decrypt information in all messages. Each cluster member is the main node at a time;
- *Individual key* - Each node has a key shared only with the collection station.

Bekara, Chakib, et al., [6], also suggest the collection station to periodically send to all nodes a new encryption key. Sensor nodes that do not have the new keys are rejected. This technique limits the substituting and reprogramming attacks on nodes. It can only be applied on closed networks, as new nodes are not accepted. Nodes that lose communication regarding key renewal are also unable to communicate.

In the next chapter we describe how we came to a low-cost solution that take the secure communication into account, using already existent protocols.

Chapter 3

The Project

In this chapter we describe our prototype of an IoT system that uses the protocols and standards presented in the previous chapter. Here we describe the requirements' analysis, and the architecture and design of the solution, and present the development report.

We start the chapter by presenting our case study that is the basis for this project.

3.1 Case Study: Green by Web

Green By Web is an automated irrigation solution that is currently in production in a few organizations to handle the management of their gardens' irrigation program.

The solution consists of a web server and Aquamote devices that connect to it. The Aquamote device is composed by sensors, actuators, a GPRS module, and a microcontroller. The sensors are responsible for measuring environment variables, like temperature and rain fall. These data is then sent via GPRS to a web server that decides what actions should the actuators take. Those actions are once again sent by GPRS to the Aquamote device. The actuators are responsible for starting and stopping the irrigation accordingly.

3.1.1 Aquamote

Aquamote has a custom-made microcontroller board, used in network nodes to be deployed in the environment. It comprises an ATMEL microcontroller, with 64 Kb of flash memory and 4 Kb of SRAM.

As in other similar devices, this module was build for a specific purpose, connecting sensors and actuators that compose the embedded subsystems.

The communication is made via GPRS, using a SIM800L module that is connected to the microcontroller.

Microcontrollers are responsible for handling data flow and processing. These microcontrollers do not use the typical von Neumann architecture, where the program instructions are stored in same memory space as data. It follows the Harvard architecture that stores machine instructions and data in separate memory units: flash memory (non-volatile memory) to store program instructions; and SRAM (volatile memory) to store data. This approach requires more awareness of the physical layer when affecting and accessing variables.

Microcontrollers use on-chip embedded Flash memory to store programs. This allows for a shorter start-up period. However, the available memory cannot be increased. This may be a limiting factor compared to microprocessors (which have external memory) because the maximum memory in a microcontroller on the market have 2 Mbytes of program memory.

Microcontrollers provide digital and analog input/output interfaces that can be connected to other electronic components.

The SIM800L connects to the microcontroller on RS-232 port and allows sending HTTP requests using firmware libraries. More information about connection and handling data flows is available in original work [34].

Limitations

The Green By Web solution has three limitations that we address within this project, namely:

- The communication with the web server is not encrypted and, therefore, it is not confidential and a vulnerable target to traffic analysis attacks.
- There is no assurance of data integrity and data authenticity.
- The transport protocol used is TCP, which has the overhead of the messages to establish the connection and assure reliability, and, therefore, degrades energy efficiency.

3.2 Requirements' analysis

We started by analyzing the communication between Aquamote [34] and the servers. The communications use TCP without confidentiality and integrity. On the top of this, TCP communication require more energy than UDP to exchange messages, and, as seen before, energy consumption must be minimized in this type of devices in order to increase

their availability.

For this prototype we want to ensure that all participants communicate in a secure way, i.e., communication must be confidential, with a guarantee of integrity and mutual authentication of the data.

Additionally, we want to reduce energy consumption by decreasing the size and number of messages exchanged between IoT devices and the server.

It is also important assure a low-cost solution, minimizing the hardware costs and providing an easy to maintain and develop solution.

The following section discusses the architecture and design of the solution we propose.

3.3 Architecture and Design

We implement mechanisms that ensure security using non-proprietary and standard solutions, maintaining a low-cost components for its devices.

Figure 3.1 presents the proposed system architecture. As a means of transport we intend to use UDP because it is a less demanding protocol when it comes to message transmission compared to TCP, having less and smaller packets to be exchanged to communicate a message. The SIM800L module provides the UDP protocol, but need to create the se-

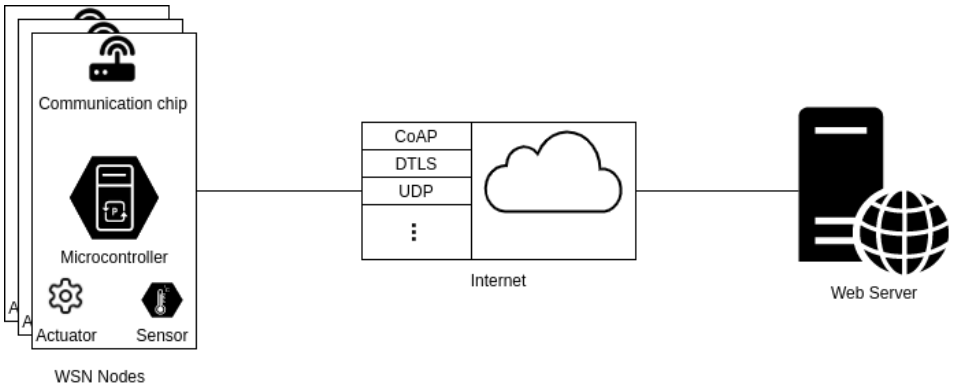

Figure 3.1: Prototype architecture

curity layer that guarantees confidentiality, integrity and authenticity. In this case, we use DTLS to implement a secure communication channel.

Also, we use CoAP at the application level as it is an optimized technology for these systems. At this point we also implemented a CoAP server that communicates via DTLS,

using NodeJS.

With these options we assemble a prototype that ensures secure communication with mutual authentication, confidentiality and integrity using DTLS. The UDP allow us to reduce the number and size of messages and therefore optimize energy efficiency. On the server side we choose a technology that implements the CoAP server in a way that is easy to develop and maintain (NodeJS). All devices are low-cost, based on the Green By Web solution.

3.4 Implementation: Aquamote scenario

We configurated an environment for this project. As we pursue a generic solution, we build a simulated web server that responds to generic requests from nodes, eliminating all the business rules of the automated irrigation system (figure 3.2).

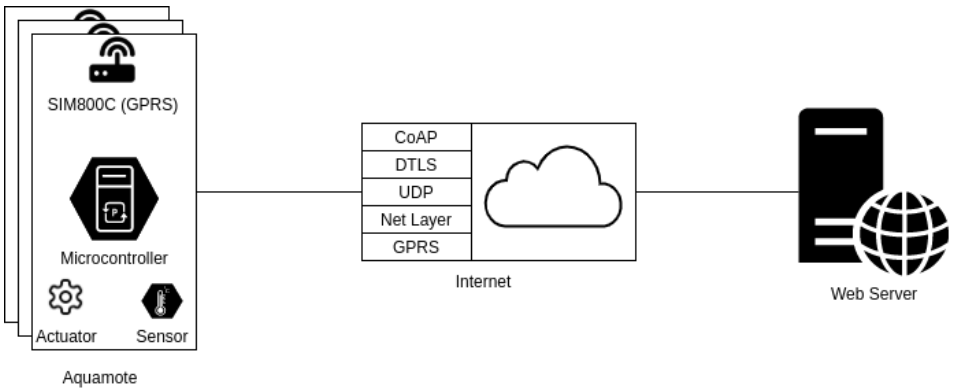

Figure 3.2: Scenario - simulated Aquamote architecture

For this approach, we simplified the communication logic to allow sending and receiving generic TCP messages to a TCP server. The program installed in the node sends a message with the payload ”12345”. We verified using wireshark that the server successfully receives the message and sends a response.

We changed the communication protocol. First, we built a UDP server and then, we changed the Aquamote node to communicate using UDP. Both TCP and UDP servers were simulated on a Raspberry Pi device, connected to the Internet with a public IP, providing a functional communication layer to be used by applications and security protocols. Our test verified that a single message was successfully sent from the developed node to the server.

After having 2 functional prototypes, we ran massive tests. Comparing TCP and UDP

communication showed that the number of exchanged packets is eight times smaller in UDP than in TCP. That is explained by the need of TCP to guarantee messages delivery, being a connection-oriented protocol. To accomplish that, it must perform a three-way handshake [24] and that requires more exchanged packets between endpoints.

This completes one of our goals: reduce the exchanged messages. However, after running the 2 programs 100 times each we observed a problem: the TCP program successfully delivers 100 messages; on the contrary, the UDP program successfully delivers only 30 messages. An analysis revealed the UDP packets were getting lost over the Internet. The server was receiving UDP messages intermittently. That was confirmed by making a server side network capture. It was expected that some losses could occur using UDP protocol, but this order of magnitude made this scenario not viable.

Taking into account that this scenario, using GPRS, raised some communication problems, we decided to create a new scenario based on Arduino nodes and using WiFi. This approach is described in the next section.

3.5 Implementation: Arduino node scenario

This section describes our second approach. First we describe the new low-cost components used (Arduino UNO and ESP 8266) and then, our solution.

Arduino Uno [4], illustrated in figure 3.3, is an open-source microcontroller board based on ATmega328P. It has 14 digital input/output pins and 6 analog inputs that provide capability to connect with external modules. It is widely used, so there are many references on the Internet to help to start using Arduino.

Arduino has two main functions that are triggered when the module boots:

- ”setup” - function that is used to initialize objects and only runs one time;
- ”loop” - function that is always running while the module is up. When it completes an iteration, it starts again.

ESP8266 [1] is a low-cost WiFi chip that implements the full TCP/IP stack and microcontroller capability. This module allows microcontrollers to connect to a WiFi networks, as a client or as an access-point, using AT commands set [44]. In this project we use ESP-01, illustrated in Figure 3.4.

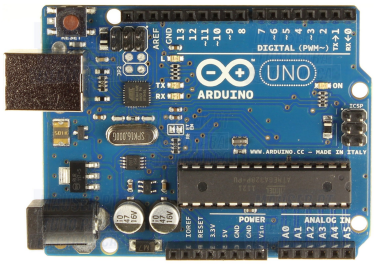

Figure 3.3: Arduino UNO

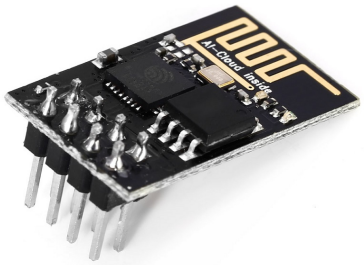

Figure 3.4: ESP8266-01

Building the scenario: Using a ESP8266-01 module connected to an Arduino Uno, we changed the architecture of the solution, making it communicate with the server using WiFi.

We decided to change the board to Arduino Uno because, as Aquamote was custom-made, it required much more effort to adapt the new communication layer to it. We could use only the ESP8266 that has a programmable microcontroller, but it would require much more effort to build and deploy the program. Also, it would become very limited in terms of scalability because it offers few input/output interfaces to connect subsystems than these solutions normaly do.

We repeated the tests and there was not packages lost neither in TCP nor in UDP. So, we had a workable scenario (figure 3.5). At this point, two of the goals were achieved: reduce exchanged messages between the endpoint and the server and use low cost modules. However, it requires a WiFi connection between the ESP8266-01 module and the server/gateway.

3.5.1 DTLS

At this point we have a functional prototype communicating by UDP. For implementing DTLS, we selected Mbed TLS, a C library used to integrate embedded applications with

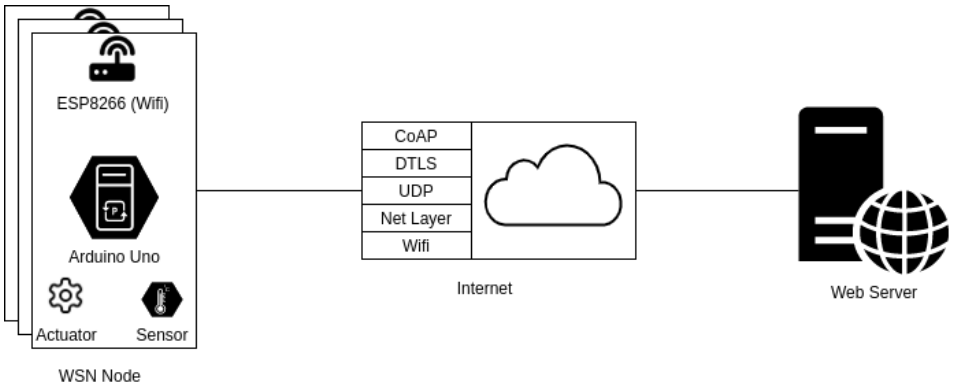

Figure 3.5: Second scenario - ESP8266 connected to hotspot

TLS and DTLS communication, providing blocks and functions that handle secure communication, cryptography and key management [33]. First, we implemented a library using pre-shared keys to establish session key.

Pre-shared keys

We were limited by the module capacity, not only to minimize the required power, but also because the module has very limited ROM and RAM. So, we compiled the solution and programmed the node using configurations that optimize the processing load and memory usage. The configuration used had the following features:

- no bignum, no PK, no X509;
- fully modern and secure (provided the pre-shared keys have high entropy);
- very low record overhead with CCM-8;
- optimized for low RAM usage.

The pre-shared key is defined with an identity and a password.

With this experiment, we were able to exchange DTLS messages between the server and the sensor (figure 3.6). Using Wireshark [46] it is possible to view all the negotiation phase from the DTLS protocol. The application message is encrypted with AES 256 algorithm. The cipher suite used here was ”TLS PSK WITH AES 256 CCM 8”.

Certificates

Next we use the cipher suites that use certificates to generate a session key. However, the module did not have enough memory to handle the complete certificate. In the experiments that were made, the sensor module crashed every time in the handshake phase of

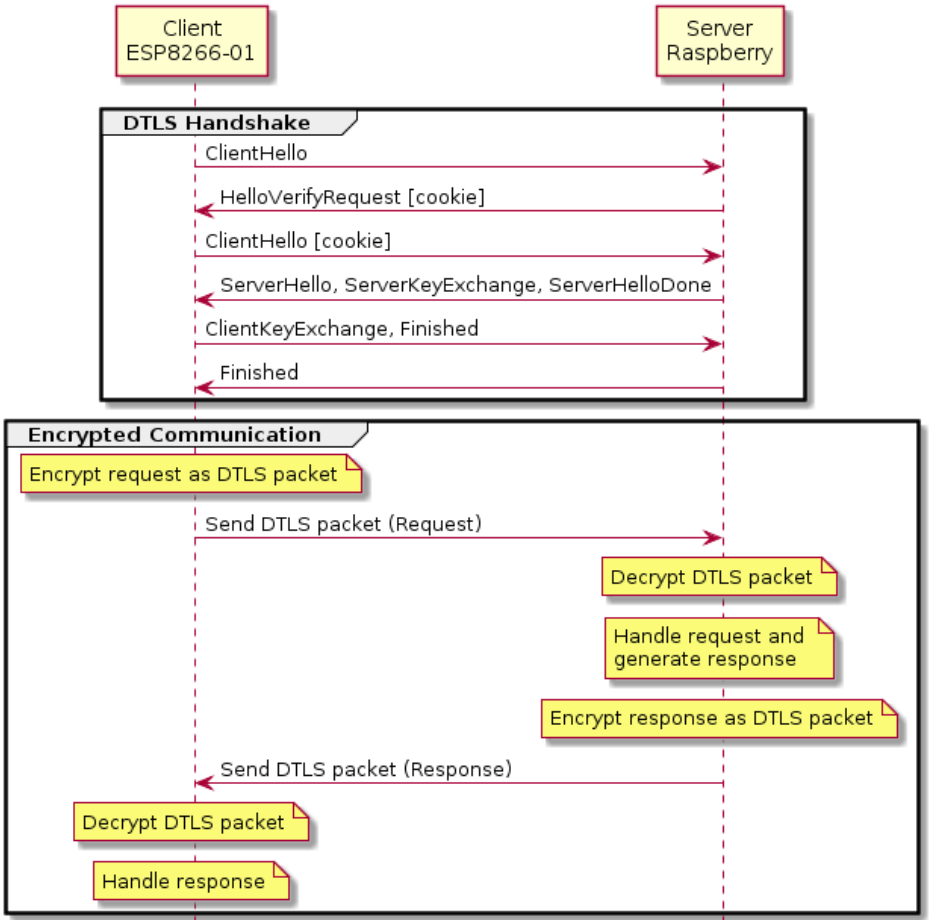

Figure 3.6: Second scenario with Pre Shared Keys

the DTLS protocol, when reading the server certificate. Because the major issue was the lack of memory, the chosen cipher suites did not use RSA (that has 2048 bit public key) and use ECDSA instead [36]. The following cipher suites were used in these experiments:

- TLS ECDHE ECDSA with AES 256 GCM SHA384,
- TLS ECDHE ECDSA with AES 128 GCM SHA256

Analyzing the program behaviour at the sensor, the memory was insufficient even so. Trying the same configuration in an ordinary computer showed that the handshake is successful when there is no memory restrictions.

We tried to optimize the code at the sensor, when reading the certificate, but with no success. We had errors regarding the internal timer of the ESP module, which was caused WDT Reset [15]. The code was changed in order to assure that the extra time consumption was expected (because of handling the certificate), however, when the WDT Reset was solved, we experimented stack overflow errors (figure 3.7).

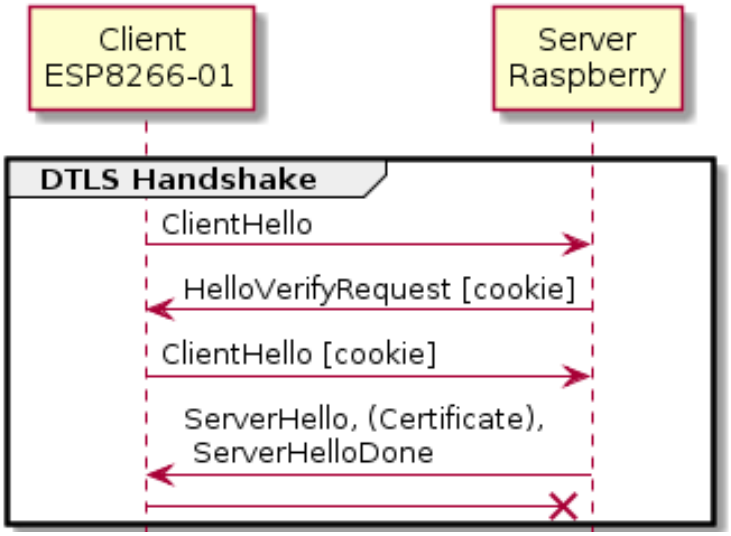

Figure 3.7: Second scenario using certificates

We dropped the implementation of cipher suites with certificates because of these constraints and continue the development of the prototype using cipher suites with pre-shared keys.

CoAP

To finalize our client development, we built a CoAP client application that communicate using the TLS layer developed before.

For that, we created a ”DTLSClient” class. It uses the development made before to send and receive requests and is integrated with a ”coap-simple” C library [38]. The ”DTLSClient” library generates the CoAP message and sends the content to the development that is responsible for using DTLS to send it to the server. This class provides function ”sendDtlsMsg” that is called in the Arduino ”loop” function.

```
int DTLSClient::sendDtlsMsg( char * msg )
```

In every ”loop” Arduino calls ”sendDtlsMsg” with ”time” as the first argument. It generates a CoAP message ”coap://server-address/time” and sends it over DTLS to the server.

3.6 Implementation: The CoAP Server

Regarding the server development, our main goal was to build a platform that would provide an easy way to develop secure CoAP applications. Other implementations that achieve this requirement are developed in C language.

Nowadays, developers tend to build their applications using high level programming lan-

guages. Knowing that, and because of the similarities of CoAP and HTTP, we decided to develop our application using a highly used programming language for web development - Javascript running on top of NodeJS. The main reason we choose NodeJS is because it enables fast performance comparing to the other alternative we considered - Python [16].

We have installed a NodeJS instance in a Raspberry running Raspbian OS with the following modules:

- mcollina/node-coap [11]
- Rantanen/node-dtls [41]

We also installed the node-coap-client module to develop a simple client application for testing.

We developed a simple server application, using just DTLS module to guarantee that encrypted messages between the node and the sensor were correctly exchanged. After handling all the node constraints, we finally could communicate using DTLS.

Next we integrated node-coap with node-dtls. By default, both modules create objects with listen ports to handle messages. However, we wanted to listen CoAP encrypted messages in a single port. So we create a DTLS object to receive the incoming traffic and a CoAP parser, using coap-request object. The communication flow between the node and the server was as in the figure 3.8.

With this approach the server could be used for two different purposes, depending on the solution. We can use the server as a typical web server to handle and respond to requests. Or, it might be used as a middleware, that can convert simple CoAP requests from the sensor to more complex messages, taking care of the additional processing load at client side. That would depend on the developed CoAP application at server side.

3.7 Solution Analysis

This sections makes a critical analysis of the developed solution.

3.7.1 Limitations

Both pre-shared keys and the use of certificates allow us to establish mutual authentication in communication. However, a malicious agent, by improperly accessing a pre-shared key, is able to act like any of the nodes or server in a way that seems trustworthy. If a certificate's private key is used by a malicious agent, it can only impersonate the agent

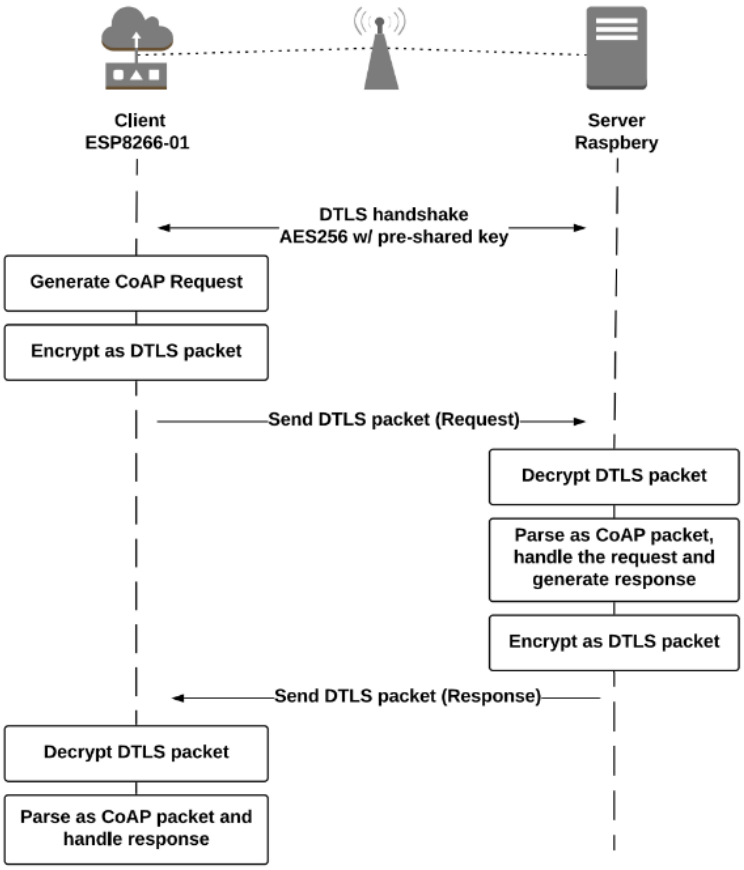

Figure 3.8: Client-Server CoAP application communication with DTLS

from whom the key was stolen.

We are not able to guarantee forward secrecy because the algorithms used with pre-shared key are not based in diffie-hellman.

Changing Pre-shared key

In order to increase the security, decreasing the risk of disclosing the pre-shared key, the server should periodically change the pre-shared key. That could be done using one of the following approaches:

- Manually - The application would be deployed in the server and in the clients with new pre-shared key. That would be very difficult to maintain, especially if it has a big number of connected clients;
- Remotely - The server could start a process to change pre-shared keys, using the previous available key. Regarding that, the server could send the new shared key to the clients, that would update it and initiate a new DTLS communication with the most recent key.

Individual key

As an improvement of these solution, it is easy to implement a mechanism of ”Individual Key”, where each node shares a different key with the server. For these, each node is deployed with its pre-shared key with the server. So, if one key is disclosed, only the communication with one node is compromised. This mechanism together with the previous one, guarantees much more security in the communication and does not increase the power consumed too much, as the server is basically responsible for all the work.

3.7.2 Costs

One of the prototype goals was to be low-cost and to provide a good level of security. In this section we detail the costs of the components.

Arduino Uno, being and open source board, can be produced following its data sheet. We have bought one already built unit for 20 euros.

ESP8266-01 is available in sites like Ebay, Amazon and AliExpress. We bought a unit from AliExpress for 2 euros.

Finally, for our server/middleware solution we bought a Raspberry Pi that costs 27,50 euros.

So, we built this prototype (with only 1 client node) for 49,5 euros. Each additional client node would have a cost of more 22 euros. To be implemented with a specific applicability it would require also the sensors and actuators.

3.8 Conclusions

This chapter presented the proposed solution to ensure low-cost security in systems using IoT devices.

Considering that it was not possible to use the initial Green By Web system to implement the proposal, it was decided to create another scenario, based also on low-cost components. With this approach it was possible to implement a secure communication channel, even using components with less processing power than those used by our case study. Taking into account the evolution of the project, it would be possible to implement the same secure communication channel in the original solution, if we used a WiFi module instead of the GPRS module. Connection problems using GPRS are complex to analyse because the Internet connection is made directly by the telecom provider.

The following chapter presents the comparative results of the tests performed on this proposal.

Chapter 4

Results

This chapter presents the results of the tests performed on our prototype focusing on the protocol differences between our project and our case study.

4.1 Testing Scenario

First, we compare the usage of TCP with UDP to give an overview of the benefits of each communication protocol. Then, we compare TLS with DTLS. In the next section we compare the additional load needed to send UDP messages over a secure channel, and, finally, we give an overview of the results from HTTPs and CoAPs.

We have deployed our CoAP server in a Raspberry Pi, with a CPU clock of 700 Mhz, 512 Mb of RAM, running a Raspbian OS in an SD card of 4 Gb. The client was deployed in an Arduino Uno, which has 32 Kb of flash memory (used to store the compiled program), 2Kb of RAM and a clock speed of 16 Mhz, connected to an ESP8266-01 that is responsible for connecting to the WiFi network. Raspberry Pi was connected in a different LAN of the client with NAT enabled to be accessible from the Internet.

In this type of applications it makes sense to have messages with minimal requests. The first test was to compare these same requests with little contents in TCP and UDP applications to then be able to compare with the active security layer (DTLS and TLS), and, finally, with the application layer (HTTP and COAP) (table 4.1). The scenario is to send the message ”time” from the client to the server. For that, we make a network capture for each experience, in which we analyze the number of messages exchanged between

Compare		**Section**
UDP	TCP	Section 4.2
DTLS	TLS	Section 4.3
COAP (UDP)	HTTP/COAP (TCP)	Section 4.5

Table 4.1: Proposed scenarios

client and server and the amount of data. Also, we observed the real-time energy using an ammeter, but because it has lack of precision, we do not show the values on our results.

4.2 TCP vs UDP

We compared the traffic between the peers for both TCP and UDP connections. We could check that a TCP connection exchanges 8 packets to establish and delete a connection(figure 4.1), with the total length of 576 bytes (figure 4.2), giving a maximum packet size of 82 bytes (figure 4.3).

Using UDP connection, only 1 packet was sent for the same message (figure 4.1), with a total length of 58 bytes (graph of figure 4.2). That represents 10 percent of the TCP exchanged data. It is a very significant improvement to constrained devices, assuming that we do not need a connection-oriented communication and would help to significantly decrease the power consumption when exchanging messages.

Also, using the ammeter we observed that the peak of energy was much higher using TCP, but we can not quantify that value because of the lack of precision of the meter device.

With that result we confirm that UDP can optimize the number and size of packets exchanged comparing to TCP.

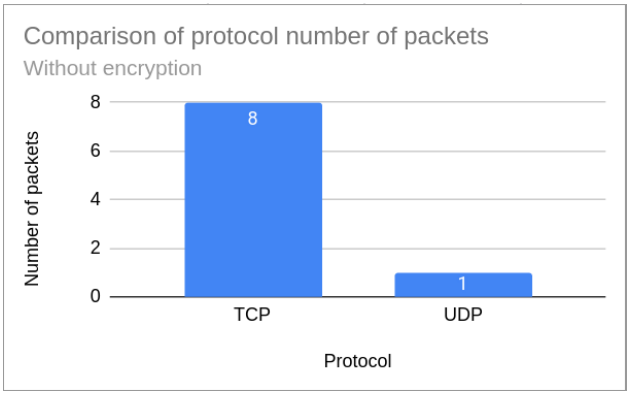

Figure 4.1: Comparing the number of packets between TCP and UDP

4.3 TLS vs DTLS

After comparing TCP with UDP, we introduced a secure communication layer. To do this, as explained in the previous chapter, we implemented the mbedtls library in arduino, both for TCP and UDP. So we remade message exchange using TLS and DTLS, respectively,

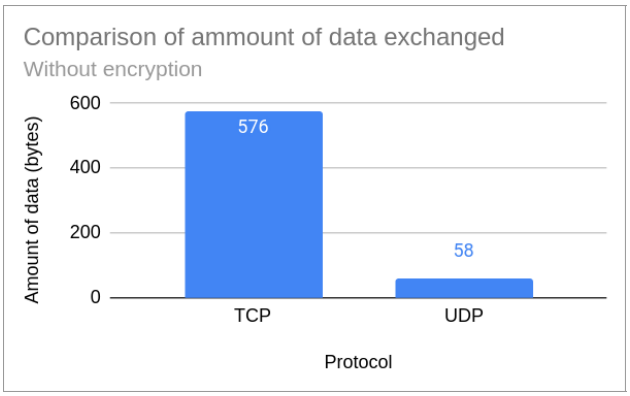

Figure 4.2: Comparing the amount of data exchanged between TCP and UDP

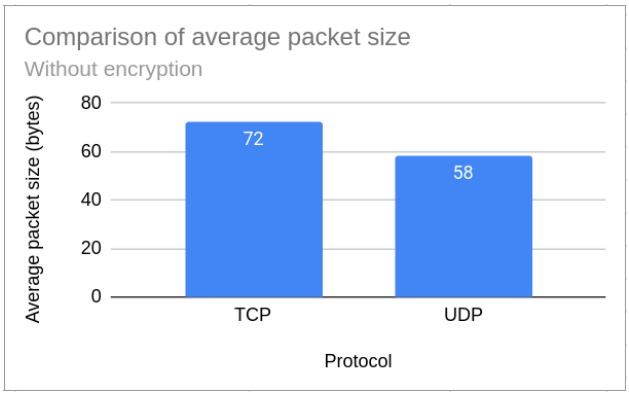

Figure 4.3: Comparing the maximum packet size between TCP and UDP

using a pre-shared key. We used the ”MBEDTLS TLS PSK with AES 256 CCM 8” cipher set.

In the experiment we went using TLS, for one message, 18 network packets were exchanged (figure 4.4) with a total of 1798 bytes (figure 4.5), the maximum packet size is 370 bytes (figure 4.6). Using DTLS, the number of packets was 8 (figure 4.4), with a total size of 798 bytes (figure 4.5) (maximum packet size is 145 bytes - figure 4.6).

This difference is also explained by the fact that in the TLS connection there is always an additional ”Acknowledge” packet for each message sent, which is not the case with DTLS.

These results further reinforce the idea that using UDP-based technologies we can optimize the power consumed of the devices.
The type of application we implemented in this prototype aims at sending a message and processing it, and putting the node in standby after that. However, for this study it also makes sense to evaluate the difference in packets and the corresponding size if a

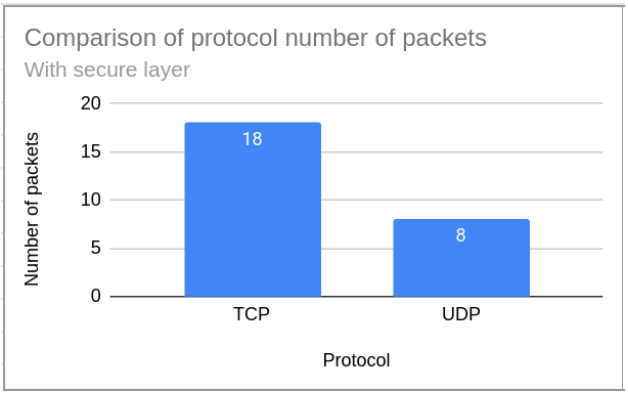

Figure 4.4: Comparing the numbers of packets between TLS and DTLS

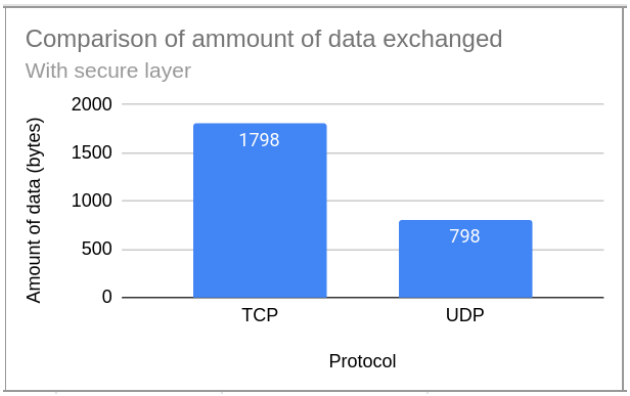

Figure 4.5: Comparing the data exchanged between TLS and DTLS

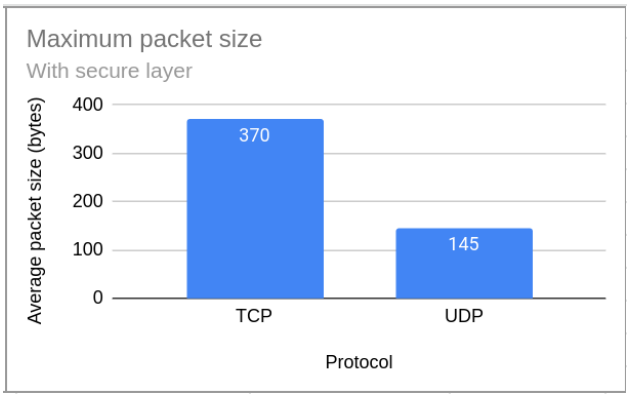

Figure 4.6: Comparing the maximum packet size between TLS and DTLS

DTLS session remains active for several message exchanges. This is because TLS and DTLS need a trading key protocol, which is only needed in the first message, and the key is reused in the subsequent messages. The results were as expected, showing a higher growth trend line for TLS communications, as can be seen in figures 4.8 and 4.7

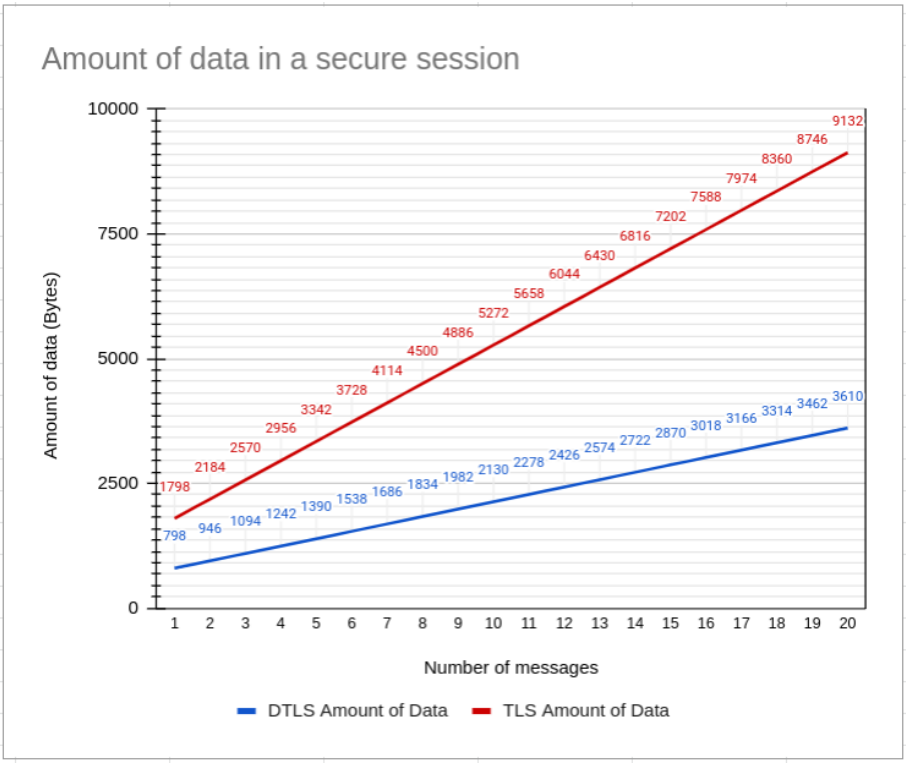

Figure 4.7: Comparing the data exchanged in a TLS/DTLS session

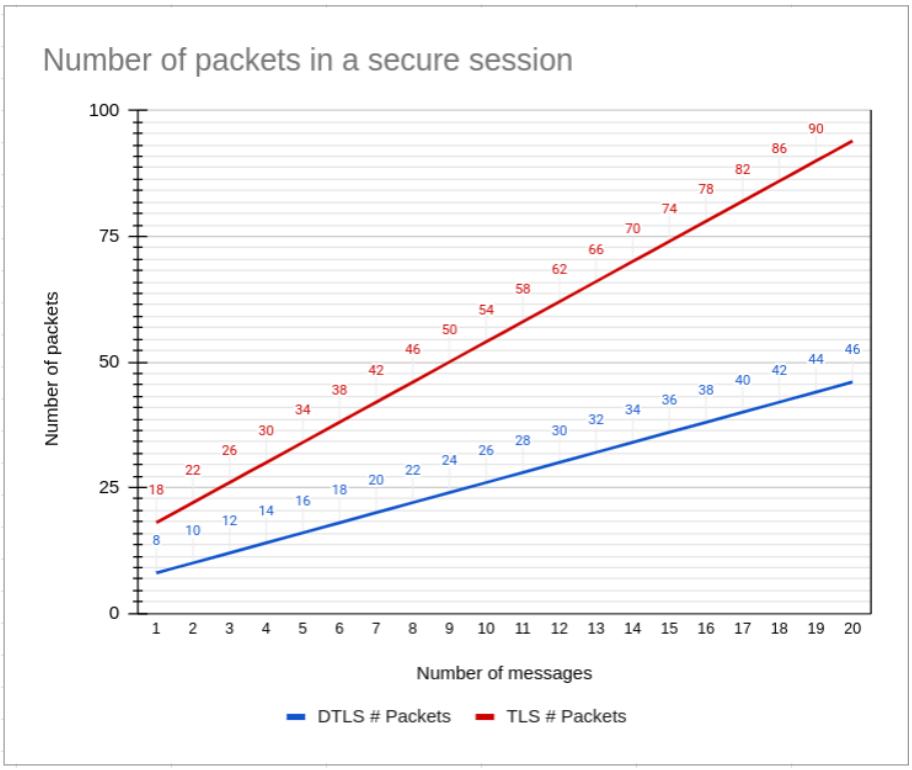

Figure 4.8: Comparing the number of packets in a TLS/DTLS session

4.4 UDP vs DTLS

The UDP security layer introduces an additional load on the system, both in terms of messages exchanged and their size. Also, the computational power to encrypt and decrypt all communication is greater.

However, considering today's evolving threats, it is important to bear in mind that these disadvantages are necessary to ensure that information systems are secure and guarantee protection to possible attacks.

Whereas on a simple UDP connection, only one 54-byte packet is needed to transmit a message. The same message, using DTLS, requires 7 more packets (figure 4.9), (communication to establish the session key). Regarding the size, and excluding the key handshake, the DTLS message took 74 bytes and 58 bytes for the UDP connection (figures 4.10 and 4.11). Handshake messages total 650 bytes.

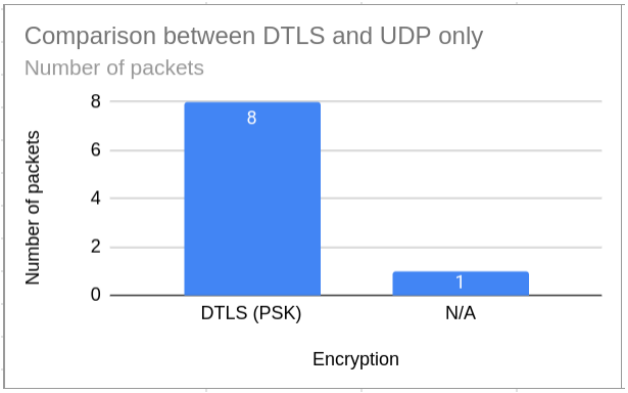

Figure 4.9: Comparing the number of packets - DTLS vs UDP

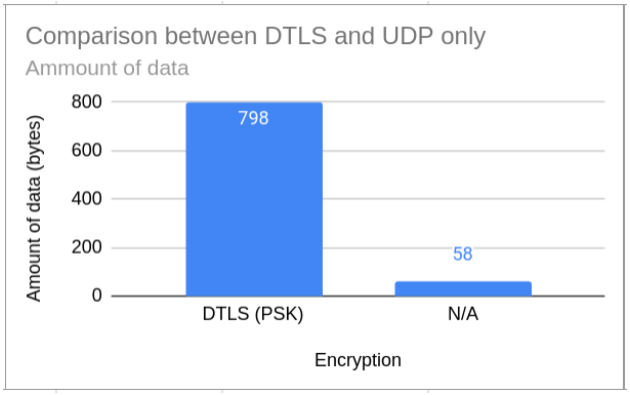

Figure 4.10: Comparing the amount of data - DTLS vs UDP

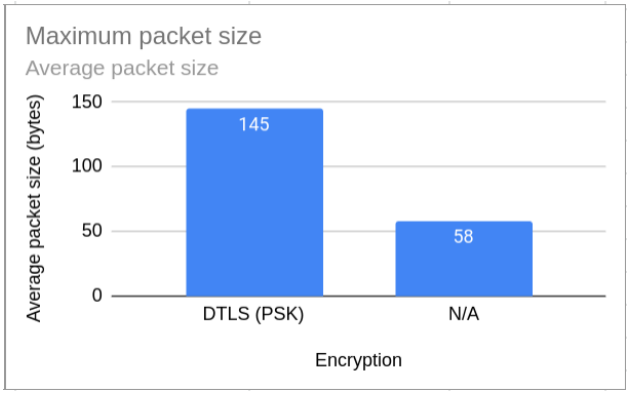

Figure 4.11: Comparing the maximum packet size - DTLS vs UDP

4.5 HTTPs vs CoAPs

As a last analysis, we compared the results obtained between CoAP systems on DTLS and HTTPS. As expected, HTTPS has more data exchanged for a message than CoAP. By its specifications, HTTP messages are much larger than COAP's, taking into account the size of the headers and the information present in them (figures 4.12 and 4.13).

It is worth mentioning that although COAP's objective is to be use over UDP transport, there are situations where it can make sense to use TCP as the transport layer. Experience is repeated with this solution. So, repeating the experiment, now with CoAP in TLS and DTLS, we obtained results similar to the previous section (figures 4.12, 4.13, 4.14), revealing that CoAP by itself, for the same type of messages does not introduce a big load on either the number of messages or their size. There is a greater computational power to generate and parse the exchanged CoAP packages, though.

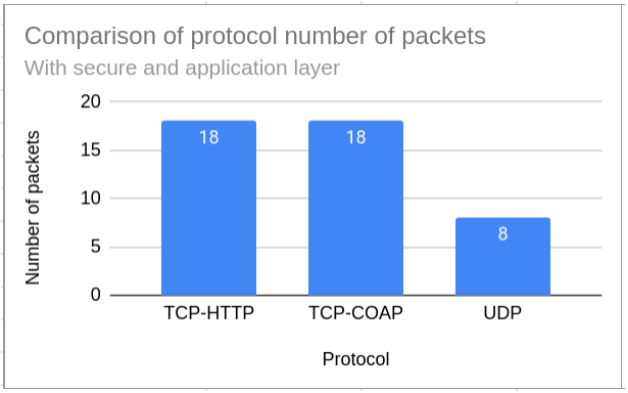

Figure 4.12: Comparing the number of packets - HTTP vs COAP (TCP and UDP)

We made an analysis regarding multiple messages in the same DTLS/TLS sessions, always using CoAP based applications. The conclusions were very similar to the previous section, taking into account that TCP messages need to exchange more packets than UDP

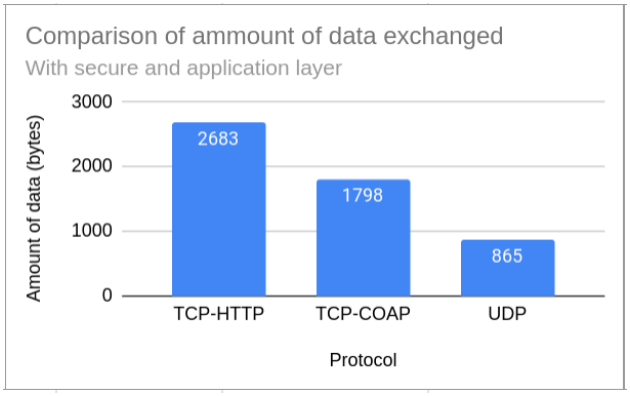

Figure 4.13: Comparing amount of data - HTTP vs COAP(TCP and UDP)

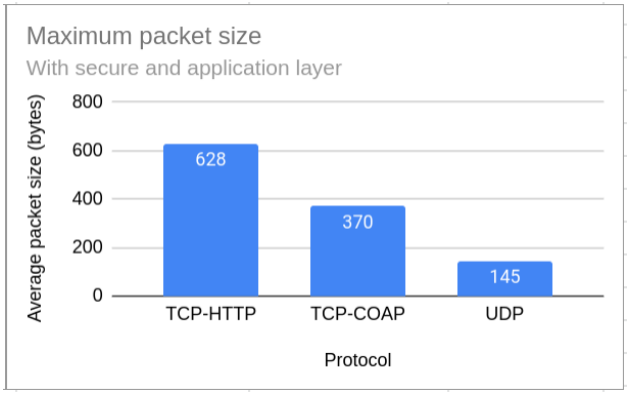

Figure 4.14: Comparing maximum packet size between HTTP and COAP(TCP and UDP)

for a single message (figures 4.15 and 4.16).

4.6 Results analysis

According to the existing standards, CoAP, using a security layer over UDP (DTLS), is the best solution considering the requirements to which we have set ourselves. The main goals were:

- To reduce the power consumption and, consequently, to increase the useful life of the nodes. Using UDP allows us to decrease the number and size of the messages and, consequently, to increase batteries lifetime;

- To have confidential communication between sensors (clients) and servers, using DTLS protocol to protect the traffic;

- We are able to keep the components low cost, which allows us to increase the number of devices we have in the system without a large investment;

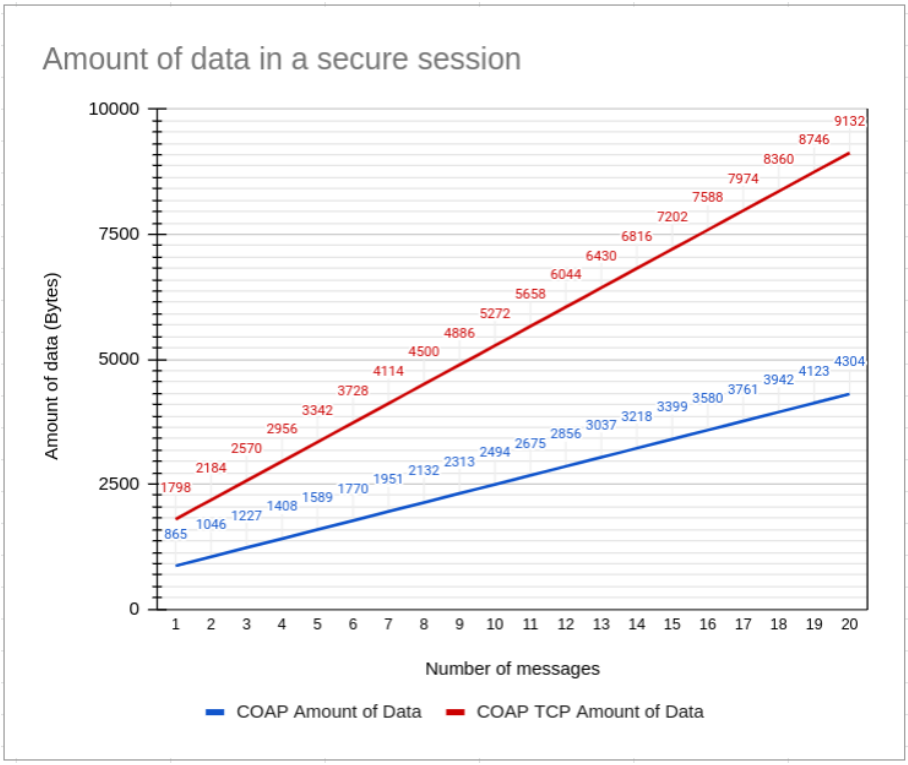

Figure 4.15: Comparing data exchanged in a TLS/DTLS session

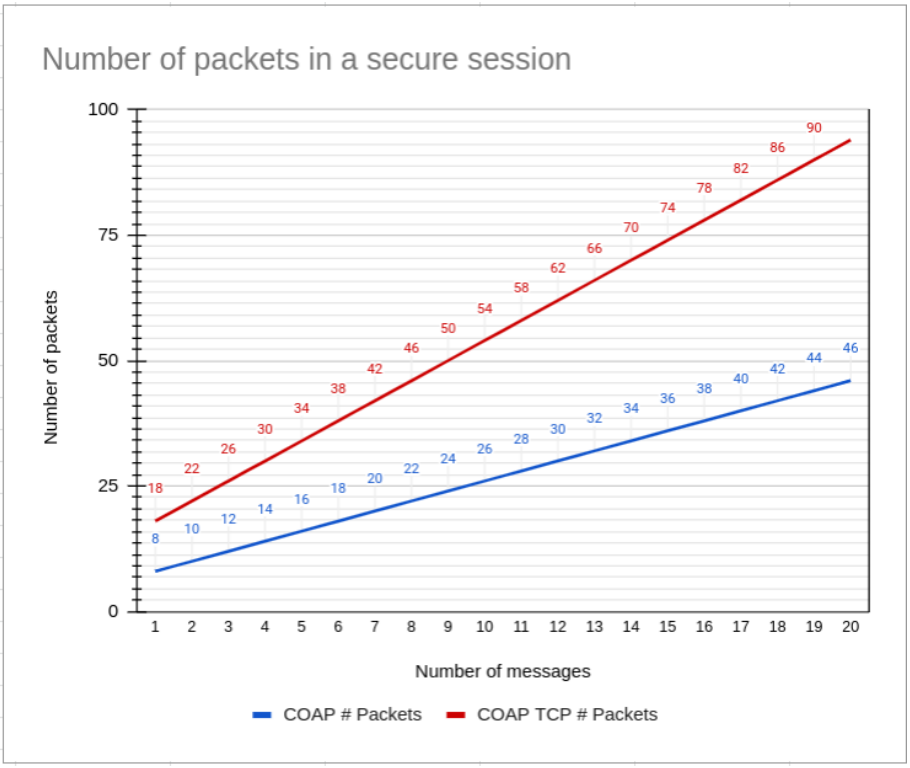

Figure 4.16: Comparing the number of packets in a TLS/DTLS session

- We do not introduce a large amount of unnecessary information to this type of systems;
- It allows us to create back-end applications in an easier way and with more mature technology and great support on the Internet, namely NodeJS.

Following, we present a table (4.2) with the summarized results of the experiments.

Experiment	**Protocol**	**Encryption**	**Application**	**# Packets**	**Amount of data (bytes)**	**Maximum Size of a packet (bytes)**
TCP x UDP	TCP	N/A	N/A	8	576	82
	UDP	N/A	N/A	1	58	58
TLS x DTLS	TCP	TLS (PSK)	N/A	18	1798	370
	UDP	DTLS (PSK)	N/A	8	798	145
TLS x DTLS (COAP-HTTP)	TCP	TLS (PSK)	HTTP	18	2683	628
	TCP	TLS (PSK)	COAP	18	1798	370
	UDP	DTLS (PSK)	COAP	8	865	145
UDP x DTLS	UDP	DTLS (PSK)	N/A	8	798	145
	UDP	N/A	N/A	1	58	58

Table 4.2: Summarized results

Chapter 5

Conclusion

This project was designed to study the IoT in general and their security in particular, with focus on extremely constrained and low-cost devices. For this, we made a survey of the state of security, analyzing the known attacks and defense techniques for a common set of used protocols. For the existing implementation technologies, we studied their own security features.

Then, we started building a prototype that take advantage of the best of IoT technologies and standards and optimizes the security of a constrained system, increasing the lifetime of the nodes' batteries and allowing to have a communication that guarantees confidentiality, integrity and authenticity. During the development we encountered some problems regarding the limitations, both in terms of processing and memory, of the devices. Also, the fact that we have a system that communicates by UDP brought challenges in what concerns the transmission of packets, making it impossible to use GPRS. At the same time we developed a CoAP server that reply messages from nodes using a security layer (DTLS), using a widely used technology, NodeJS. The devices' components are low-cost, which allow us to assemble a workable prototype with low investment.

The results collected prove that there is an optimization both in terms of the number of exchanged packets, and their size, if we adopt a UDP-based solution, but there is still an additional burden by introducing the DTLS layer. The fact that we use the CoAP protocol also makes web requests much smaller compared to the most used HTTP protocol.

There is still potential for this area and this specific solution to evolve, as enriching the solutions proposed here with even more robust techniques, namely in key management and the use of stronger encryption algorithms.

Bibliography

[1] AI-Thinker. Esp-01 wifi module. http://www.microchip.ua/wireless/esp01.pdf.

[2] Al-Fuqaha, Guizani, Mohammadi, Aledhari, and Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376, Fourthquarter 2015.

[3] Zigbee Alliance. Zigbee alliance. http://www.zigbee.org/, 2017. Accessed: 2017-03-04.

[4] Arduino. Arduino uno rev3. https://store.arduino.cc/arduino-uno-rev3.

[5] S. Arvind and V. A. Narayanan. An overview of security in coap: Attack and analysis. In *2019 5th International Conference on Advanced Computing Communication Systems (ICACCS)*, pages 655–660, 2019.

[6] Chakib Bekara and Maryline Laurent-Maknavicius. A new resilient key management protocol for wireless sensor networks. In *IFIP International Workshop on Information Security Theory and Practices*, pages 14–26. Springer, 2007.

[7] Seyit A Camtepe and Bülent Yener. Key distribution mechanisms for wireless sensor networks: a survey. *Rensselaer Polytechnic Institute, Troy, New York, Technical Report*, pages 05–07, 2005.

[8] CardinalPeak. Using udp in internet-of-things devices. https://cardinalpeak.com/blog/using-udp-in-internet-of-things-devices/, 2014. Accessed: 2017-03-04.

[9] Aniruddha Chakrabarti. Emerging open and standard protocol stack for iot. https://www.linkedin.com/pulse/emerging-open-standard-protocol-stack-iot-aniruddha-chakrabarti, 2015. Accessed: 2015-02-05.

[10] Menaka Chandra, Ankit Naik, and Chayya Chandra. Study of wireless sensor networks security issues and attacks.

[11] Matteo Collina. node-coap. https://github.com/mcollina/node-coap.

[12] Jasenka Dizdarević, Francisco Carpio, Admela Jukan, and Xavi Masip-Bruin. A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration. *ACM Comput. Surv.*, 51(6), January 2019.

[13] Jorge Granjal and Edmundo Monteiro. Adaptable end-to-end security for mobile iot sensing applications. pages 20–25, 11 2017.

[14] Jorge Granjal, Edmundo Monteiro, and Jorge Sá Silva. Security for the internet of things: a survey of existing protocols and open research issues. *IEEE Communications Surveys & Tutorials*, 17(3):1294–1312, 2015.

[15] Ivan Grokhotkov. My esp crashes running some code. how to troubleshoot it? https://arduino-esp8266.readthedocs.io/en/latest/faq/a02-my-esp-crashes.html#watchdog.

[16] Ivan Grokhotkov. Python vs node.js: Which programming language to choose? https://hackernoon.com/python-vs-nodejs-which-programming-language-to-choose-98721d6526f2.

[17] Carl Hartung, James Balasalle, and Richard Han. Node compromise in sensor networks: The need for secure systems. *Department of Computer Science University of Colorado at Boulder*, 2005.

[18] Ernst Haselsteiner and Klemens Breitfuß. Security in near field communication (nfc). In *Workshop on RFID security*, pages 12–14, 2006.

[19] Texas Instruments Iboun Taimiya Sylla. To zigbee or not to zigbee? factors to consider when selecting zigbee technology. http://sensornet-cyeh.blogspot.com/2009/03/to-zigbee-or-not-to-zigbee-factors-to.html, 2009. Accessed: 2017-09-12.

[20] IETF. Rfc 6176 - prohibiting secure sockets layer (ssl) version 2.0. https://tools.ietf.org/html/rfc6176.

[21] IETF. Rfc 7568 - deprecating secure sockets layer version 3.0. https://tools.ietf.org/html/rfc7568.

[22] IETF. Rfc 8446 - hypertext transfer protocol – http/1.1. https://tools.ietf.org/html/rfc2616.

[23] IETF. Rfc 8446 - the transport layer security (tls) protocol version 1.3. https://tools.ietf.org/html/rfc8446.

[24] IETF. Rfc 793 - transmission control protocol. https://tools.ietf.org/html/rfc793, 1981. Accessed: 2019-05-09.

[25] IETF. Rfc 7252 - the constrained application protocol (coap). https://tools.ietf.org/html/rfc7252, 2016. Accessed: 2016-12-07.

[26] IETF. The datagram transport layer security (dtls) protocol version 1.3. https://tools.ietf.org/id/draft-ietf-tls-dtls13-34.html, 2019. Accessed: 2020-03-23.

[27] Ricardo Mendão Silva e Fernando Boavida Jorge Sá Silva. Redes de sensores sem fios. In *Redes de Sensores sem Fios*. FCA, 2016.

[28] Chris Karlof, Naveen Sastry, and David Wagner. Tinysec: a link layer security architecture for wireless sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 162–175. ACM, 2004.

[29] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[30] Libelium. Libelium sensors connect with sigfox for smart cities and the iot. https://www.libelium.com/libeliumworld/sigfox-connectivity-waspmote-868mhz-europe-900mhz-us-long-range/, 2016. Accessed: 2016-12-07.

[31] An Liu and Peng Ning. Tinyecc: A configurable library for elliptic curve cryptography in wireless sensor networks. In *Information Processing in Sensor Networks, 2008. IPSN'08. International Conference on*, pages 245–256. IEEE, 2008.

[32] David Martins and Hervé Guyennet. Wireless sensor network attacks and security mechanisms: A short survey. In *Network-Based Information Systems (NBiS), 2010 13th International Conference on*, pages 313–320. IEEE, 2010.

[33] ARM mbed. Mbed tls. https://tls.mbed.org/.

[34] Carlos Mão de Ferro. Sistema de rega inteligente. http://hdl.handle.net/10451/10118, 2013. Accessed: 2017-03-04.

[35] How To Tech Naija. Highlighting the differences between gprs, edge, 3g, hsdpa, hspa+ and 4g lte. https://howtotechnaija.com/differences-between-gprs-edge-3g-hsdpa-hspa-4glte/, 2018. Accessed: 2018-10-04.

[36] Nick Naziridis. Comparing ecdsa vs rsa. https://www.ssl.com/article/comparing-ecdsa-vs-rsa/.

[37] High Performance Browser Networking. Transport layer security (tls). https://hpbn.co/transport-layer-security-tls/, 2017. Accessed: 2017-11-14.

[38] Hirotaka Niisato. Coap-simple-library. https://github.com/hirotakaster/CoAP-simple-library.

[39] Bryan Parno, Adrian Perrig, and Virgil Gligor. Distributed detection of node replication attacks in sensor networks. In *2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 49–63. IEEE, 2005.

[40] Vidyasagar Potdar, Atif Sharif, and Elizabeth Chang. Wireless sensor networks: A survey. In *Advanced Information Networking and Applications Workshops, 2009. WAINA'09. International Conference on*, pages 636–641. IEEE, 2009.

[41] Mikko Rantanen. node-dtls. https://github.com/Rantanen/node-dtls.

[42] Pallavi Sethi and Smruti R. Sarangi. Internet of things: Architectures, protocols, and applications. *J. Electrical and Computer Engineering*, 2017:9324035:1–9324035:25, 2017.

[43] SIGFOX. Sigfox developer portal. http://makers.sigfox.com/about/, 2016. Accessed: 2016-12-05.

[44] Espressif Systems. At instruction set. https://www.espressif.com/sites/default/files/documentation/4a-esp8266_at_instruction_set_en.pdf.

[45] Xun Wang, Wenjun Gu, Kurt Schosek, Sriram Chellappan, and Dong Xuan. Sensor network configuration under physical attacks. In *Networking and Mobile Computing*, pages 23–32. Springer, 2005.

[46] Wireshark. Wireshark. https://www.wireshark.org/docs/man-pages/.

[47] Anthony D Wood and John A Stankovic. Denial of service in sensor networks. *computer*, 35(10):54–62, 2002.

[48] Christos Xenakis. Security measures and weaknesses of the gprs security architecture. *IJ Network Security*, 6(2):158–169, 2008.

[49] Shu Yinbiao, Kang Lee, Fan Jianbin, Peter Lanctot, et al. Internet of things: wireless sensor networks. *White Paper, International Electrotechnical Commission, http://www. iec. ch*, 2014.

[50] Zheng Yue-Feng, Chen Zhuo-Ran, Han Jia-Yu, and Li Zheng. A novel based-node level security strategy in wireless sensor network. In *2012 International Conference on Information Management, Innovation Management and Industrial Engineering*, volume 1, pages 507–510. IEEE, 2012.

[51] Morteza M Zanjireh and Hadi Larijani. A survey on centralised and distributed clustering routing algorithms for wsns. In *2015 IEEE 81st Vehicular Technology Conference (VTC Spring)*, pages 1–6. IEEE, 2015.

[52] Sencun Zhu, Sanjeev Setia, and Sushil Jajodia. Leap+: Efficient security mechanisms for large-scale distributed sensor networks. *ACM Transactions on Sensor Networks (TOSN)*, 2(4):500–528, 2006.

[53] Tobias Zillner and S Strobl. Zigbee exploited: The good the bad and the ugly, 2015.